
C++11
ЯЗЫК C++
программирования
Четвертое издание
Бьерн Страуструп
создатель C++

Язык программирования C++
Четвертое издание

Современный стандарт C++11 позволяет программистам воплощать свои идеи точнее и лаконичнее, создавая при этом более производительный код. В новой редакции Бьерн Страуструп представляет основательно переработанный и дополненный вариант своего фундаментального руководства всем, кто заинтересован в повышении эффективности использовании C++

Книга **Язык программирования C++, 4-е издание** с высокой степенью детализации и надлежащими обобщениями рассматривает все аспекты языка: языковой синтаксис, основополагающие абстракции, применение стандартных библиотек и ключевые приемы разработки. Книга насыщена многочисленными компактными примерами кода на «чистом C++11», тщательно отобранными автором в качестве лучших иллюстраций практического использования. Автор дает множество перекрестных ссылок, как в рамках книги так и ссылок на ISO стандарт языка, обеспечивая возможность более глубокого погружения в предмет

Круг предложенных в книге новшеств стандарта C++11 включает

- поддержку параллельных вычислений
- регулярные выражения, указатели для управления ресурсами, API случайных чисел, улучшенные контейнеры
- универсальную инициализацию и расширенные возможности списков инициализации, упрощенные операторы for, перемещающую семантику, поддержку Unicode
- лямбда-выражения, обобщенные константные выражения, контроль над функциями, объявленные по умолчанию в классах, вариативные шаблоны, псевдонимы шаблонов, определяемые пользователем литералы
- вопросы совместимости с ранними версиями языка

Книга демонстрирует поистине энциклопедический охват, уделяя пристальное внимание

- базовым концепциям, в частности, рассматривая типизацию, объекты, области видимости, организацию данных, основы организации вычислений, и т.п.
- модульности разрабатываемого кода в контексте разбиения на пространства имен, файлы (исходного кода) и в связи с обработкой исключений
- абстракциям C++, включая концепции классов, их иерархии, применения шаблонов для соединения на практике возможностей традиционного, объектно-ориентированного и обобщенного программирования
- компонентам стандартной библиотеки: контейнерам, алгоритмам, итераторам, контейнерам пар, строкам, потоковому вводу/выводу, средствам работы с локалями, числовым константам и т.п.
- детальному рассмотрению базовой модели использования памяти в C++

Четвертое издание поможет программистам быстро и доступно перейти от стандарта C++98 и любых других языков программирования к C++11. Книга предоставляет множество сведений и приемов, полезность которых не смогут отрицать даже искушенные разработчики на C++11

С этой книгой освоить C++ будет несложно



Бьерн Страуструп — автор идеи (C++)-расширений к языку C, первым реализовавшим их на практике, автор книг «Программирование. Принципы и практика использования C++» и «Язык программирования C++», многих популярных и академических публикаций. Доктор философских наук, заслуженный профессор Техасского Университета A&M, руководитель кафедры компьютерных наук Инженерного Колледжа, член Национальной Инженерной Академии США, научный сотрудник института IEEE и ассоциации ACM (Association for Computing Machinery). Бьерн Страуструп активно участвует в рабочей группе ISO по Стандартизации C++

informit.com/aw
stroustrup.com





ЯЗЫК C++ программирования

Четвертое издание

Сканирование и обработка: MelkorBSD, 02.01.2026

The C++ Programming Language

Fourth Edition

Bjarne Stroustrup

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Totonto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

ЯЗЫК
программирования
C++
Четвертое издание

Бьерн Страуструп

перевод с английского и редакция
Н.Н. Мартынова
А.Н. Мартынова



Москва
Издательство БИНОМ
2024

УДК 004.43
ББК 32.973.26-018.1
С 83

Бьерн Страуструп

С 83 Язык программирования C++. 4-е издание. Пер. с англ. — М.: БИНОМ-МЕД, 2024 г. . — 1216 с.: ил.

ISBN 978-5-6045724-6-7

Книга представляет стандарт ISO/IEC 14882-2011 (называемый в обиходе C++11) и призвана помочь программистам быстро и доступно перейти от стандарта C++98 и любых других языков программирования к C++11. Традиционно, как и в предыдущих изданиях, автор демонстрирует свой «фирменный стиль» — эффективные подходы к решению разнообразных задач проектирования и программирования, демонстрация навыков программирования на С-совместимом ядре C++, объектно-ориентированный подход к созданию программных продуктов.

Четвертое издание «Язык программирования C++» предлагает содержательный пакет новшеств и ярко выраженную направленность на изучение техники написания качественного программного кода.

Книга адресована программистам, использующим в своей повседневной работе C++. Она также будет полезна преподавателям, студента и всем, кто хочет ознакомиться с описанием языка «из первых рук».

Дизайн обложки *М.В.Пясецкой*

Authorized translation from the English language edition, entitled «C++ PROGRAMMING LANGUAGE, THE, 4th Edition», ISBN 0321563840; by STROUSTRUP, BJARNE; published by Pearson Education, Inc, publishing as Addison-Wesley Professional.
Copyright © 2013 by Pearson Education, Inc

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieal system, without permission from Pearson Education, Inc. Russian language edition published by BINOM Publishers.
Copyright © 2014 Издательство БИНОМ

ISBN 0-321-56384-0 (англ.)

ISBN 978-5-6045724-6-7 (рус.)

Издание книги книги Бьерна Страуструпа
«Язык программирования C++, 4-е издание» на русском языке
посвящено памяти нашего товарища — Николая Николаевича Мартынова,
великолепного автора, переводчика, редактора.
Он не дожил до выхода переведенной им книги в свет,
но его вклад в развитие «программистской среды»
в России огромен и измеряется десятками тысяч книг,
в издании которых он принял участие в разном качестве.

Издательство БИНОМ

Предисловие к изданию на русском языке

Книга Бьерна Страуструпа «Язык программирования C++, 3-е издание» была переведена на русский язык и издана в РФ в самом начале века. В течение десяти лет она неизменно пользовалась большим успехом у читателей, да и в любой стране мира, где работают программисты, книга от создателя языка C++ заслуженно считается классикой жанра.

В 2010 на отечественном рынке появилась книга Б. Страуструпа «Язык программирования C++, специальное издание», в которой текст перевода подвергся значительной ревизии и корректировке, были исправлены некоторые допущенные ошибки перевода, подвергнутого экспертизе хорошо знакомых с данной предметной областью специалистов.

Кроме того, были учтены найденные за несколько лет ошибки самого автора, обнаруженные им самим, его читателями и почитателями, опытными и квалифицированными профессионалами. Следует отметить, что все высказанные замечания оперативно отражались на специально созданном сайте поддержки книги http://www.research.att.com/~bs/3rd_errata.html. Наконец, поверглась уточнению сложившаяся система русскоязычной терминологии, постоянно отражавшаяся в разного рода учебной литературе.

И вот пришла пора изданию на русском языке новой книги Бьерна Страуструпа «Язык программирования C++, 4-е издание», представляющей стандарт ISO/IEC 14882-2011 (называемый в обиходе C++11) и призванной помочь программистам быстро и доступно перейти от стандарта C++98 и любых других языков программирования к C++11. Говоря в рамках Предисловия вкратце, круг предложенных новшеств включает поддержку параллельных вычислений; регулярные выражения, указатели для управления ресурсами, API случайных чисел, улучшенные контейнеры; универсальную инициализацию и расширенные возможности списков инициализации, упрощенные операторы for, перемещающую семантику, поддержку Unicode; лямбда-выражения, обобщенные

константные выражения, контроль над функциями, объявленные по умолчанию в классах, вариативные шаблоны, псевдонимы шаблонов, определяемые пользователем литералы; вопросы совместимости с ранними версиями языка.

Возможно, наиболее значимой качественной характеристикой нового издания книги Б. Страуструпа является ее ярко выраженная «заточенность» на изучение техники написания качественного кода.

Пусть попадет эта книга в руки всем тем, кто хочет приобщиться к C++!

Н.Н. Мартынов

Краткое содержание

Содержание 9
Предисловие 31
Предисловие автора к третьему изданию на русском языке 34
Предисловие к третьему изданию 37
Предисловие ко второму изданию 39
Предисловие к первому изданию 41

Часть I. Введение
1. Обращение к читателю. 45
2. Обзор языка C++: Основы 81
3. Обзор языка C++: Механизмы абстракции 101
4. Обзор языка C++: Контейнеры и Алгоритмы 127
5. Обзор языка C++: Параллелизм и Утилиты. 150

Часть II. Базовые средства
6. Типы и объявления 173
7. Указатели, массивы и ссылки 207
8. Структуры, объединения и перечисления 235
9. Операторы 256
10. Выражения 270
11. Операции 300
12. Функции. 330
13. Обработка исключений 365
14. Пространства имен. 408
15. Исходные файлы и программы 434

Часть III. Механизмы абстракции
16. Классы 461
17. Конструирование, очистка, копирование и перемещение 490
18. Перегрузка операций 533
19. Специальные операции 553
20. Производные классы 578
21. Классовые иерархии. 610
22. RTTI (идентификация типа на этапе выполнения) 636
23. Шаблоны 657
24. Обобщенное программирование. 689
25. Специализации 709

26. Конкретизация. 726
27. Шаблоны и иерархии 743
28. Метапрограммирование 761
29. Проект для работы с матрицами 804

Часть IV. Обзор стандартной библиотеки
30. Обзор стандартной библиотеки. 833
31. STL контейнеры 857
32. STL алгоритмы 896
33. STL итераторы. 920
34. Память и ресурсы 938
35. Прочие компоненты библиотеки. 970
36. Строки 991
37. Регулярные выражения 1006
38. Потоки ввода/вывода 1025
39. Локализация 1058
40. Численные расчеты 1103
41. Параллелизм 1132
42. Потоки и задачи 1148
43. Стандартная библиотека C. 1188
44. Совместимость 1201

Содержание

Предисловие 31
Предисловие автора к третьему изданию на русском языке 34
Предисловие к третьему изданию 37
Предисловие ко второму изданию 39
Предисловие к первому изданию 41

Часть 1. Введение
1. Обращение к читателю 45
1.1. Структура книги 45
1.1.1. Введение 46
1.1.2. Базовые средства 46
1.1.3. Механизмы абстракции 47
1.1.4. Стандартная библиотека 48
1.1.5. Примеры и ссылки 49
1.2. Проект C++ 51
1.2.1. Стили программирования 53
1.2.2. Проверка типов 56
1.2.3. Совместимость с C 57
1.2.4. Язык, библиотеки и операционные системы 58
1.3. Изучение C++ 60
1.3.1. Программирование на C++ 62
1.3.2. Советы C++ — программистам 62
1.3.3. Советы C-программистам 63
1.3.4. Советы Java-программистам 64
1.4. История C++ 65
1.4.1. Хронология 66
1.4.2. Ранний период 67
1.4.2.1. Средства языка и средства библиотеки 67
1.4.3. Стандарт 1998 69
1.4.3.1. Средства языка 69
1.4.3.2. Стандартная библиотека 70
1.4.4. Стандарт 2011 71
1.4.4.1. Средства языка 71
1.4.4.2. Стандартная библиотека 73
1.4.5. Где используется C++? 73
1.5. Советы 75
1.6. Библиография 76

2. Обзор языка C++: Основы 81
2.1. Введение 81

2.2. Основы 82
2.2.1. Привет, Мир! 83
2.2.2. Типы, переменные и арифметические операции 84
2.2.3. Константы 86
2.2.4. Операторы выбора и циклы 87
2.2.5. Указатели, массивы и циклы 88
2.3. Пользовательские типы 90
2.3.1. Структуры 90
2.3.2. Классы 92
2.3.3. Перечисления (enumerations) 93
2.4. Модульность 94
2.4.1. Раздельная компиляция 95
2.4.2. Пространства имен 96
2.4.3. Обработка ошибок 97
2.4.3.1. Исключения (exceptions) 98
2.4.3.2. Инварианты 98
2.4.3.3. Статические утверждения (static assertions) 100
2.5. Заключение 100
2.6. Советы 100

3. Обзор языка C++: Механизмы абстракции 101
3.1. Введение 101
3.2. Классы 102
3.2.1. Конкретные типы 102
3.2.1.1. Арифметический тип 103
3.2.1.2. Контейнеры 104
3.2.1.3. Инициализация контейнеров 106
3.2.2. Абстрактные типы 107
3.2.3. Виртуальные функции 109
3.2.4. Иерархии классов 110
3.3. Копирование и перемещение 114
3.3.1. Копирование контейнеров 114
3.3.2. Перемещение контейнеров 116
3.3.3. Управление ресурсами 118
3.3.4. Подавление операций 118
3.4. Шаблоны 119
3.4.1. Параметризуемые типы 119
3.4.2. Шаблоны функций 121
3.4.3. Функциональные объекты (объекты-функции) 121
3.4.4. Шаблоны с переменным числом параметров 123
3.4.5. Синонимы 124
3.5. Советы 125

4. Обзор языка C++: Контейнеры и Алгоритмы 127
4.1. Библиотеки 127
4.1.1. Обзор стандартной библиотеки 128
4.1.2. Заголовочные файлы стандартной библиотеки и ее пространство имен 129
4.2. Строки 130
4.3. Потоковый ввод/вывод 131
4.3.1. Вывод 131

4.3.2. Ввод . . . 132
4.3.3. Ввод/вывод пользовательских типов . . . 133
4.4. Контейнеры . . . 135
4.4.1. Класс vector . . . 135
4.4.1.1. Элементы . . . 136
4.4.1.2. Проверка диапазона . . . 137
4.4.2. Контейнер list . . . 138
4.4.3. Контейнер map . . . 139
4.4.4. Контейнер unordered_map . . . 140
4.4.5. Обзор контейнеров . . . 140
4.5. Алгоритмы . . . 141
4.5.1. Использование итераторов . . . 143
4.5.2. Типы итераторов . . . 145
4.5.3. Итераторы потоков . . . 145
4.5.4. Предикаты . . . 147
4.5.5. Обзор алгоритмов . . . 148
4.5.6. Алгоритмы для контейнеров . . . 148
4.6. Советы . . . 149

5. Обзор языка C++: Параллелизм и Утилиты . . . 150
5.1. Введение . . . 150
5.2. Управление ресурсами . . . 150
5.2.1. Интеллектуальные указатели unique_ptr и shared_ptr . . . 151
5.3. Параллелизм . . . 153
5.3.1. Задачи и потоки . . . 154
5.3.2. Передача аргументов . . . 155
5.3.3. Возврат результатов . . . 155
5.3.4. Разделяемые данные . . . 156
5.3.4.1. Ожидание наступления событий . . . 157
5.3.5. Взаимодействие задач . . . 158
5.3.5.1. future и promise . . . 159
5.3.5.2. packaged_task . . . 160
5.3.5.3. async() . . . 161
5.4. Утилиты — небольшие полезные компоненты библиотеки . . . 161
5.4.1. Время . . . 162
5.4.2. Функции для работы с типами . . . 162
5.4.2.1. iterator_traits . . . 162
5.4.2.2. Предикаты типов . . . 164
5.4.3. pair и tuple . . . 164
5.5. Регулярные выражения . . . 166
5.6. Математика . . . 166
5.6.1. Математические функции и алгоритмы . . . 166
5.6.2. Комплексные числа . . . 167
5.6.3. Случайные числа . . . 167
5.6.4. Векторная арифметика . . . 169
5.6.5. Числовые пределы . . . 169
5.7. Советы . . . 169

Часть 2. Базовые средства

6. Типы и объявления . . . 173
6.1. Стандарт ISO C++ . . . 173

6.1.1. Реализации . . . 174
6.1.2. Базовый набор символов исходного кода программ . . . 175
6.2. Типы . . . 175
6.2.1. Фундаментальные типы . . . 176
6.2.2. Логический тип . . . 177
6.2.3. Символьные типы . . . 178
6.2.3.1. Знаковые и беззнаковые символьные типы . . . 180
6.2.3.2. Символьные литералы . . . 181
6.2.4. Целые типы . . . 182
6.2.4.1. Целые литералы . . . 183
6.2.4.2. Типы целых литералов . . . 183
6.2.5. Типы чисел с плавающей запятой . . . 184
6.2.5.1. Литералы с плавающей запятой . . . 184
6.2.6. Префиксы и суффиксы . . . 185
6.2.7. void . . . 186
6.2.8. Размеры . . . 186
6.2.9. Выравнивание . . . 188
6.3. Объявления . . . 189
6.3.1. Структура объявлений . . . 191
6.3.2. Объявление нескольких имен . . . 192
6.3.3. Имена . . . 192
6.3.3.1. Ключевые слова . . . 194
6.3.4. Область видимости . . . 194
6.3.5. Инициализация . . . 196
6.3.5.1. Отсутствие инициализаторов . . . 198
6.3.5.2 Списки инициализации . . . 199
6.3.6. Выведение типа: auto и decltype() . . . 200
6.3.6.1 Спецификатор типа auto . . . 200
6.3.6.2 auto и {}-списки . . . 201
6.3.6.3 Спецификатор decltype() . . . 202
6.4. Объекты и значения . . . 202
6.4. lvalue и rvalue . . . 203
6.4.2. Время жизни объектов . . . 204
6.5. Синонимы типов . . . 204
6.6. Советы . . . 205

7. Указатели, массивы и ссылки . . . 207
7.1. Введение . . . 207
7.2. Указатели . . . 207
7.2.1. void* . . . 208
7.2.2. nullptr . . . 209
7.3. Массивы . . . 210
7.3.1. Инициализаторы массивов . . . 211
7.3.2. Строковые литералы . . . 212
7.3.2.1. «Сырые» символьные строки . . . 213
7.3.2.2. Широкие (многобайтовые) символы . . . 215
7.4. Указатели на массивы . . . 216
7.4.1. Навигация по массивам . . . 217
7.4.2. Многомерные массивы . . . 219
7.4.3. Передача массивов . . . 219

7.5. Указатели и модификатор const ... 221
7.6. Указатели и владение объектами ... 223
7.7. Ссылки ... 224
7.7.1. lvalue-ссылки ... 225
7.7.2. rvalue-ссылки ... 228
7.7.3. Ссылки на ссылки ... 231
7.7.4. Указатели и ссылки ... 231
7.8. Советы ... 233

8. Структуры, объединения и перечисления ... 235
8.1. Введение ... 235
8.2. Структуры ... 235
8.2.1. Раскладка структурных объектов в памяти ... 237
8.2.2. Имена структур ... 238
8.2.3. Структуры и классы ... 239
8.2.4. Структуры и массивы ... 240
8.2.5. Эквивалентность типов ... 242
8.2.6. POD (Plain Old Data) ... 243
8.2.7. Битовые поля ... 245
8.3. Объединения ... 246
8.3.1. Объединения и классы ... 247
8.3.2. Анонимные объединения ... 248
8.4. Перечисления ... 250
8.4.1. enum class ... 251
8.4.2. «Простой» enum ... 253
8.4.3. Безымянные перечисления ... 255
8.5. Советы ... 255

9. Операторы ... 256
9.1. Введение ... 256
9.2. Сводка операторов ... 256
9.3. Объявления как операторы ... 257
9.4. Операторы выбора ... 258
9.4.1. Оператор if ... 259
9.4.2. Оператор switch ... 260
9.4.2.1. Объявления в ветвях **switch**-оператора ... 262
9.4.3. Объявления в условиях ... 262
9.5. Операторы итераций ... 263
9.5.1. Оператор range-for ... 263
9.5.2. Оператор for ... 264
9.5.3. Оператор while ... 265
9.5.4. Оператор do ... 266
9.5.5. Выход из цикла ... 266
9.6. Оператор goto ... 267
9.7. Комментарии и отступы ... 268
9.8. Советы ... 269

10. Выражения ... 270
10.1. Введение ... 270
10.2. Проект «Настольный калькулятор» ... 270
10.2.1. Парсер ... 271

10.2.2. Ввод ... 275
10.2.3. Низкоуровневый ввод ... 278
10.2.4. Обработка ошибок ... 279
10.2.5. Главная программа ... 280
10.2.6. Заголовочные файлы ... 281
10.2.7. Аргументы командной строки ... 281
10.2.8. Замечание по стилю ... 282
10.3. Сводка сведений по операциям ... 283
10.3.1. Результаты ... 286
10.3.2. Порядок вычислений ... 287
10.3.3. Приоритеты операций ... 288
10.3.4. Временные объекты ... 289
10.4. Константные выражения ... 290
10.4.1. Символические константы ... 292
10.4.2. Ключевое слово const ... 293
10.4.3. Литеральные типы ... 293
10.4.4. Аргументы-ссылки ... 294
10.4.5. Адресные константные выражения ... 295
10.5. Неявное приведение типов ... 295
10.5.1. Продвижения ... 295
10.5.2. Преобразования ... 296
10.5.2.1. Интегральные преобразования ... 296
10.5.2.2. Преобразования типов с плавающей запятой ... 296
10.5.2.3. Преобразования указателей и ссылок ... 297
10.5.2.4. Преобразование указателей на члены классов ... 297
10.5.2.5. Преобразования для типа bool ... 297
10.5.2.6. Преобразования интегральных типов и типов с плавающей запятой ... 298
10.5.3. Обычные арифметические преобразования ... 298
10.6. Советы ... 299

11. Операции ... 300
11.1. Отдельные операции ... 300
11.1.1. Логические операции ... 300
11.1.2. Побитовые логические операции ... 301
11.1.3. Условные выражения ... 302
11.1.4. Инкремент и декремент ... 302
11.2. Свободная память ... 304
11.2.1. Управление памятью ... 306
11.2.2. Массивы ... 308
11.2.3. Выделение памяти ... 309
11.2.4. Перегрузка операции new ... 310
11.2.4.1. nothrow версии операции new ... 312
11.3. Списки ... 312
11.3.1. Модель реализации ... 313
11.3.2. Квалифицированные списки ... 314
11.3.3. Неквалифицированные списки ... 315
11.4. Лямбда-выражения ... 316
11.4.1. Модель реализации ... 317
11.4.2. Альтернативы ... 318

11.4.3. Захват ... 319
11.4.3.1. Лямбды и время жизни ... 321
11.4.3.2. Имена из пространств имен ... 322
11.4.3.3. Лямбды и this ... 322
11.4.3.4. Лямбды с ключевым словом mutable ... 322
11.4.4. Вызов и возврат ... 323
11.4.5. Тип лямбда-выражений ... 323
11.5. Явное преобразование типов ... 324
11.5.1. Конструирование ... 325
11.5.2. Именованные преобразования ... 327
11.5.3. Преобразования в С-стиле ... 328
11.5.4. Функциональный стиль ... 329
11.6. Советы ... 329

12. Функции ... 330
12.1. Объявления функций ... 330
12.1.1. Почему функции? ... 331
12.1.2. Составные части объявления функции ... 331
12.1.3. Определение функции ... 332
12.1.4. Возврат значений ... 333
12.1.5. inline-функции ... 335
12.1.6. constexpr-функции ... 336
12.1.6.1. constexpr-функции и ссылки ... 337
12.1.6.2. Вычисления с условиями ... 338
12.1.7. [[noreturn]]-функции ... 338
12.1.8. Локальные переменные ... 338
12.2. Передача аргументов ... 339
12.2.1. Ссылочные аргументы ... 340
12.2.2. Аргументы-массивы ... 342
12.2.3. Аргументы-списки ... 344
12.2.4. Неопределенное количество аргументов ... 345
12.2.5. Аргументы по умолчанию ... 348
12.3. Перегрузка функций ... 349
12.3.1. Автоматическое разрешение перегрузки ... 350
12.3.2. Перегрузка и тип возврата ... 352
12.3.3. Перегрузка и область видимости ... 352
12.3.4. Разрешение перегрузки при множественных аргументах ... 353
12.3.5. Ручное разрешение перегрузки ... 353
12.4. Предусловия и постусловия ... 354
12.5. Указатель на функцию ... 355
12.6. Макросы ... 359
12.6.1. Условная компиляция ... 362
12.6.2. Предопределенные макросы ... 363
12.6.3. Прагмы ... 363
12.7. Советы ... 363

13. Обработка исключений ... 365
13.1. Обработка ошибок ... 365
13.1.1. Исключения ... 366
13.1.2. Традиционная обработка ошибок ... 368
13.1.3. Попытка кое-как дотянуть до конца ... 369

13.1.4. Альтернативный взгляд на исключения ... 369
13.1.4.1. Асинхронные события ... 370
13.1.4.2. Исключения, которые не являются ошибками ... 370
13.1.5. Когда нельзя использовать исключения ... 371
13.1.6. Иерархическая обработка ошибок ... 372
13.1.7. Исключения и эффективность ... 373
13.2. Гарантии исключений ... 375
13.3. Управление ресурсами ... 376
13.3.1. Завершающие конструкции ... 379
13.4. Инварианты ... 381
13.5. Генерация и перехват исключений ... 384
13.5.1. Генерация исключений ... 384
13.5.1.1. noexcept-функции ... 386
13.5.1.2. Операция noexcept ... 387
13.5.1.3. Спецификации исключений ... 388
13.5.2. Перехват исключений ... 388
13.5.2.1. Повторная генерация исключения ... 389
13.5.2.2. Перехват всех исключений ... 390
13.5.2.3. Множественные обработчики ... 391
13.5.2.4. Функции из try-блоков ... 391
13.5.2.5. Завершение ... 392
13.5.3. Исключения и потоки ... 394
13.6. Реализация класса vector ... 395
13.6.1. Простой vector ... 395
13.6.2. Явное представление памяти ... 399
13.6.3. Присваивание ... 401
13.6.4. Изменение размера ... 403
13.6.4.1. reserve() ... 403
13.6.4.2. resize() ... 404
13.6.4.3. push_back() ... 405
13.6.4.4. Финальные размышления ... 405
13.7. Советы ... 406

14. Пространства имен ... 408
14.1. Проблемы композиции ... 408
14.2. Пространства имен ... 409
14.2.1. Явная квалификация ... 411
14.2.2. using-объявления ... 412
14.2.3. using-директивы ... 412
14.2.4. Поиск на основе аргументов ... 414
14.2.5. Пространства имен открыты ... 416
14.3. Модуляризация и интерфейсы ... 417
14.3.1. Пространства имен как модули ... 418
14.3.2. Реализация ... 420
14.3.3. Интерфейсы и реализации ... 421
14.4. Композиция на основе пространств имен ... 423
14.4.1. Удобства против надежности ... 423
14.4.2. Синонимы для пространств имен ... 424
14.4.3. Композиция пространств имен ... 425
14.4.4. Композиция и выбор ... 426

14.4.5. Пространства имен и перегрузка....427
14.4.6. Версии....429
14.4.7. Вложенные пространства имен....430
14.4.8. Безымянные пространства имен....431
14.4.9. Заголовочные файлы языка C....432
14.5. Советы....433

15. Исходные файлы и программы....434
15.1. Раздельная компиляция....434
15.2. Компоновка....435
15.2.1. Имена, локальные в файлах....438
15.2.2. Заголовочные файлы....438
15.2.3. Правило одного определения....440
15.2.4. Заголовочные файлы стандартной библиотеки....442
15.2.5 Связывание с отличным от C++ кодом....443
15.2.6. Компоновка и указатели на функции....445
15.3. Применение заголовочных файлов....446
15.3.1. Единственный заголовочный файл....446
15.3.2. Организация программы со множеством заголовочных файлов....449
15.3.2.1. Другие модули калькулятора....451
15.3.2.2. Использование заголовочных файлов....453
15.3.3. «Стражи включения»....453
15.4. Программы....454
15.4.1. Инициализация нелокальных переменных....455
15.4.2. Инициализация и параллелизм....456
15.4.3. Завершение программы....456
15.5. Советы....457

Часть III. Механизмы абстракции

16. Классы....461
16.1. Введение....461
16.2. Основы классов....462
16.2.1. Функции-члены....463
16.2.2. Копирование по умолчанию....464
16.2.3. Контроль доступа....465
16.2.4. class и struct....466
16.2.5. Конструкторы....467
16.2.6. explicit-конструкторы....469
16.2.7. Внутриклассовая инициализация....471
16.2.8. Определение функций внутри класса....471
16.2.9. Изменяемость....472
16.2.9.1. Константные функции-члены....472
16.2.9.2. Физическая и логическая неизменяемость....473
16.2.9.3. mutable....474
16.2.9.4. Изменяемость посредством косвенного обращения....474
16.2.10. Ссылка на себя....475
16.2.11. Доступ к членам....477
16.2.12. Статические члены....478
16.2.13. Типы данных-члены класса....479
16.3. Конкретные классы....480
16.3.1. Функции-члены....483

16.3.2. Вспомогательные функции 485
16.3.3. Перегруженные операции 487
16.3.4. Роль конкретных классов 487
16.4. Советы 488

17. Конструирование, очистка, копирование и перемещение 490
17.1. Введение 490
17.2. Конструкторы и деструкторы 492
17.2.1. Конструкторы и инварианты 493
17.2.2. Деструкторы и ресурсы 494
17.2.3. Деструкторы членов и базового класса 495
17.2.4. Вызов конструкторов и деструкторов 496
17.2.5. Виртуальные деструкторы 497
17.3. Инициализация объектов класса 497
17.3.1. Инициализация в отсутствие конструкторов 497
17.3.2. Инициализация посредством конструкторов 499
17.3.2.1. Инициализация конструкторами 501
17.3.3. Умолчательные конструкторы 502
17.3.4. Конструкторы со списком инициализаторов 503
17.3.4.1. Разрешение неоднозначностей для конструкторов со списком инициализаторов 504
17.3.4.2. Использование initializer_list 505
17.3.4.3. Прямая инициализация и инициализация копированием 506
17.4. Инициализация членов и инициализация базового класса 507
17.4.1. Инициализация членов 508
17.4.1.1. Инициализация членов и присваивание 509
17.4.2. Инициализаторы базового класса 509
17.4.3. Делегирующие конструкторы 510
17.4.4. Внутриклассовые инициализаторы 511
17.4.5. Инициализация статических членов 513
17.5. Копирование и перемещение 514
17.5.1. Копирование 514
17.5.1.1. Опасайтесь умолчательных конструкторов 515
17.5.1.2. Копирование базовых классов 516
17.5.1.3. Смысл копирования 517
17.5.1.4. Срезка 519
17.5.2. Перемещение 520
17.6. Генерирование умолчательных операций 524
17.6.1. Явное требование генерирования умолчаний 524
17.6.2. Умолчательные операции 525
17.6.3. Применение умолчательных операций 526
17.6.3.1. Умолчательные конструкторы 526
17.6.3.2. Поддержка инвариантов 526
17.6.3.3. Ресурсные инварианты 527
17.6.3.4. Частичные инварианты 528
17.6.4. Объявление функций с =delete 530
17.7. Советы 531

18. Перегрузка операций 533
18.1. Введение 533
18.2. Функции определения операций 534

18.2.1. Бинарные и унарные операции 535
18.2.2. Стандартный смысл операций 536
18.2.3. Операции и пользовательские типы 537
18.2.4. Передача объектов 538
18.2.5. Операции в пространствах имен 539
18.3. Тип комплексных чисел 540
18.3.1. Операции-члены и операции не-члены 541
18.3.2. Смешанная арифметика 542
18.3.3. Приведение типов 543
18.3.3.1. Преобразования операндов 544
18.3.4. Литералы 545
18.3.5. Функции доступа 546
18.3.6. Вспомогательные функции 546
18.4. Приведение типа 547
18.4.1. Операции приведения типа 548
18.4.2. explicit-операции приведения 549
18.4.3. Неоднозначности 550
18.5. Советы 551

19. Специальные операции 553
19.1. Введение 553
19.2. Специальные операции 553
19.2.1. Индексирование 553
19.2.2. Вызов функции 554
19.2.3. Разыменование 556
19.2.4. Инкремент и декремент 558
19.2.5. Выделение памяти и ее освобождение 559
19.2.6. Пользовательские литералы 560
19.3. Класс строк 564
19.3.1. Важные операции 564
19.3.2. Доступ к символам 565
19.3.3. Представление 565
19.3.3.1. Вспомогательные функции 567
19.3.4. Функции-члены 568
19.3.5. Полезные функции 570
19.3.6. Использование класса String 572
19.4. Друзья 572
19.4.1. Поиск друзей 574
19.4.2. Друзья и функции-члены 575
19.5. Советы 576

20. Производные классы 578
20.1. Введение 578
20.2. Производные классы 579
20.2.1. Функции-члены 581
20.2.2. Конструкторы и деструкторы 582
20.3. Классовые иерархии 583
20.3.1. Поля типов 583
20.3.2. Виртуальные функции 585
20.3.3. Явная квалификация 588
20.3.4. Управление замещением 588

20.3.4.1. override . . . 589
20.3.4.2. final . . . 590
20.3.5. Использование членов базового класса . . . 592
20.3.5.1. Наследование конструкторов . . . 593
20.3.6. Ослабление типа возврата . . . 595
20.4. Абстрактные классы . . . 596
20.5. Управление доступом . . . 598
20.5.1. Защищенные члены . . . 601
20.5.1.1. Использование защищенных членов . . . 602
20.5.2. Доступ к базовым классам . . . 603
20.5.2.1. Множественное наследование и контроль доступа . . . 604
20.5.3. Контроль доступа и using-объявления . . . 604
20.6. Указатели на члены . . . 605
20.6.1. Указатели на функции-члены . . . 605
20.6.2. Указатели на члены-данные . . . 607
20.6.3. Члены базового и производного классов . . . 608
20.7. Советы . . . 608

21. Классовые иерархии . . . 610
21.1. Введение . . . 610
21.2. Дизайн классовых иерархий . . . 610
21.2.1. Наследование реализации . . . 611
21.2.1.1. Критика . . . 613
21.2.2. Наследование интерфейса . . . 614
21.2.3. Альтернативные реализации . . . 616
21.2.3.1. Критика . . . 618
21.2.4. Локализация создания объектов . . . 619
21.3. Множественное наследование . . . 620
21.3.1. Множественные интерфейсы . . . 620
21.3.2. Множественные классы реализации . . . 621
21.3.3. Разрешение неоднозначности . . . 623
21.3.4. Повторное использование базовых классов . . . 626
21.3.5. Виртуальные базовые классы . . . 627
21.3.5.1. Конструируем виртуальные базовые классы . . . 629
21.3.5.2. Вызов функции-члена виртуального базового класса . . . 630
21.3.6. Реплицирование или виртуальный базовый класс . . . 632
21.3.6.1. Замещение функций виртуального базового класса . . . 633
21.4. Советы . . . 635

22. RTTI (идентификация типа на этапе выполнения) . . . 636
22.1. Введение . . . 636
22.2. Навигация по классовой иерархии . . . 636
22.2.1. dynamic_cast . . . 638
22.2.1.1. Операция dynamic_cast для ссылок . . . 640
22.2.2. Множественное наследование . . . 641
22.2.3. static_cast и dynamic_cast . . . 642
22.2.4. Восстановление интерфейса . . . 643
22.3. Двойная диспетчеризация и «посетители» . . . 647
22.3.1. Двойная диспетчеризация . . . 647
22.3.2. «Посетители» . . . 649

22.4. Конструирование и уничтожение объектов 651
22.5. Идентификация типа 651
22.5.1. Расширенная информация о типе 653
22.6. Правильное и неправильное использование RTTI 654
22.7. Советы 656

23. Шаблоны 657
23.1. Введение 657
23.2. Простой шаблон строк 660
23.2.1. Определение шаблона 661
23.2.2. Конкретизация шаблона 663
23.3. Проверка типов 663
23.3.1. Эквивалентность типов 664
23.3.2. Обнаружение ошибок 665
23.4. Члены классовых шаблонов 666
23.4.1. Члены-данные 667
23.4.2. Функции-члены 667
23.4.3. Члены-синонимы типов 667
23.4.4. Статические члены 668
23.4.5. Члены-типы 668
23.4.6. Члены-шаблоны 669
23.4.6.1. Шаблоны и конструкторы 670
23.4.6.2. Шаблоны и ключевое слово virtual 670
23.4.6.3. Использование вложения 670
23.4.7. Друзья 673
23.5. Шаблоны функций 674
23.5.1. Аргументы функциональных шаблонов 676
23.5.2. Вывод аргументов для функциональных шаблонов 677
23.5.2.1. Вывод ссылок 678
23.5.3. Перегрузка шаблонных функций 679
23.5.3.1. Разрешение неоднозначности 681
23.5.3.2. Неудачная подстановка аргументов 681
23.5.3.3. Перегрузка и наследование 683
23.5.3.4. Перегрузка и невыводимые параметры 683
23.6. Синонимы для шаблонов 683
23.7. Организация исходного кода 684
23.7.1. Компоновка 686
23.8. Советы 687

24. Обобщенное программирование 689
24.1. Введение 689
24.2. Алгоритмы и обобщение 690
24.3. Концепции 694
24.3.1. Исследование концепций 694
24.3.2. Концепции и ограничения 697
24.4. Делаем концепции конкретными 698
24.4.1. Аксиомы 702
24.4.2. Многоаргументные концепции 703
24.4.3. Концепции-значения 704
24.4.4. Проверки ограничений 704

24.4.5. Проверка определения шаблона 706
24.5. Советы 708

25. Специализации 709
25.1. Введение 709
25.2. Параметры и аргументы шаблонов 710
25.2.1. Типы как аргументы 710
25.2.2. Величины как аргументы 711
25.2.3. Операции как аргументы 713
25.2.4. Шаблоны как аргументы 715
25.2.5. Аргументы шаблонов по умолчанию 716
25.2.5.1. Умолчательные аргументы шаблонных функций 717
25.3. Специализации 717
25.3.1. Специализации интерфейса 720
25.3.1.1. Специализации реализации 721
25.3.2. Первичный шаблон 721
25.3.3. Порядок специализаций 723
25.3.4. Специализации шаблонных функций 723
25.3.4.1. Специализации и перегрузки 723
25.3.4.2. Специализации, не являющиеся перегрузкой 724
25.4. Советы 725

26. Конкретизация 726
26.1. Введение 726
26.2. Конкретизация шаблона 727
26.2.1. Когда требуется конкретизация 727
26.2.2. Ручной контроль конкретизации 728
26.3. Связывание имен 730
26.3.1. Зависимые имена 731
26.3.2. Связывание в точке определения 733
26.3.3. Связывание в точке конкретизации 734
26.3.4. Множественные точки конкретизации 736
26.3.5. Шаблоны и пространства имен 737
26.3.6. Сверхагрессивный поиск на основе зависимости аргументов 738
26.3.7. Имена из базовых классов 740
26.4. Советы 742

27. Шаблоны и иерархии 743
27.1. Введение 743
27.2. Параметризация и иерархии 744
27.2.1. Генерируемые типы 746
27.2.2. Конверсия шаблонов 748
27.3. Иерархии классовых шаблонов 749
27.3.1. Шаблоны как интерфейсы 750
27.4. Параметры шаблона в качестве базовых классов 750
27.4.1. Композиция структур данных 751
27.4.2. Линеаризация классовых иерархий 754
27.5. Советы 759

28. Метапрограммирование 761
28.1. Введение 761

28.2. Функции типа....763
28.2.1. Синонимы типов....766
28.2.1.1. Когда использовать синонимы не следует....766
28.2.2. Предикаты типа....767
28.2.3. Выбор функции....769
28.2.4. Шаблоны _traits из стандартной библиотеки....769
28.3. Управляющие структуры....771
28.3.1. Выбор....771
28.3.1.1. Выбор из двух типов....771
28.3.1.2. Этап компиляции или этап выполнения....772
28.3.1.3. Выбор из нескольких типов....773
28.3.2. Итерация и рекурсия....774
28.3.2.1. Рекурсия с классами....775
28.3.3. Когда использовать метапрограммирование....775
28.4. Условное определение: Enable_if....776
28.4.1. Использование Enable_if....778
28.4.2. Реализация Enable_if....779
28.4.3. Enable_if и концепции....780
28.4.4. Дополнительные примеры с Enable_if....780
28.5. Список времени компиляции: Tuple....783
28.5.1. Простая функция вывода....785
28.5.2. Доступ к элементам....786
28.5.2.1 Константный Tuple....787
28.5.3. Функция make_tuple....788
28.6. Шаблоны с переменным числом параметров....789
28.6.1. Безопасная по типам функция printf()....789
28.6.2. Технические детали....791
28.6.3. Перенаправление....793
28.6.4. Шаблон tuple из стандартной библиотеки....794
28.7. Библиотека для работы с единицами измерений....797
28.7.1. Тип Unit....798
28.7.2. Тип Quantity....798
28.7.3. Литералы для Unit....800
28.7.4. Вспомогательные функции....801
28.8. Советы....802

29. Проект для работы с матрицами....804
29.1. Введение....804
29.1.1. Использование матриц....804
29.1.2. Требования к матрицам....806
29.2. Шаблон Matrix....807
29.2.1. Конструирование и присваивания....808
29.2.2. Индексирование и срезы....810
29.3. Арифметические операции с Matrix....812
29.3.1. Скалярные операции....813
29.3.2. Сложение....813
29.3.3. Умножение....814
29.4. Реализация Matrix....816
29.4.1. slice()....816
29.4.2. Matrix_slice....817

29.4.3. Matrix_ref . . . 818
29.4.4. Инициализация Matrix списком инициализаторов . . . 819
29.4.5. Доступ к Matrix . . . 821
29.4.6. Матрицы нулевой размерности . . . 823
29.5. Решение линейных уравнений . . . 824
29.5.1. Классический алгоритм Гаусса . . . 825
29.5.2. Более общее решение (выделение главного элемента) . . . 826
29.5.3. Тестирование . . . 826
29.5.4. Оптимизация кода . . . 827
29.6. Советы . . . 829

Часть IV. Обзор стандартной библиотеки

30. Обзор стандартной библиотеки . . . 833
30.1. Введение . . . 833
30.1.1. Средства стандартной библиотеки . . . 834
30.1.2. Ограничения дизайна . . . 835
30.1.3. Стиль описания . . . 836
30.2. Заголовочные файлы . . . 837
30.3. Поддержка языка . . . 840
30.3.1. Поддержка initializer_list . . . 841
30.3.2. Поддержка range-for . . . 841
30.4. Обработка ошибок . . . 842
30.4.1. Исключения . . . 842
30.4.1.1. Стандартная иерархия exception . . . 843
30.4.1.2. Продвижение (распространение) исключений . . . 844
30.4.1.3. terminate() . . . 846
30.4.2. Утверждения . . . 846
30.4.3. system_error . . . 847
30.4.3.1. Коды ошибок . . . 848
30.4.3.2. Категории ошибок . . . 849
30.4.3.3. Исключение system_error . . . 850
30.4.3.4. Потенциально переносимый error_condition . . . 851
30.4.3.5. Отображение кодов ошибок . . . 851
30.4.3.6. Коды ошибок errc . . . 853
30.4.3.7. Коды ошибок future_errc . . . 855
30.4.3.8. Коды ошибок io_errc . . . 855
30.5. Советы . . . 855

31. STL контейнеры . . . 857
31.1. Введение . . . 857
31.2. Обзор контейнеров . . . 857
31.2.1. Внутреннее представление . . . 860
31.2.2. Требования к элементам . . . 862
31.2.2.1. Сравнения . . . 863
31.2.2.2. Другие операции сравнения . . . 864
31.3. Обзор операций . . . 865
31.3.1. Типы, связанные с контейнерами . . . 868
31.3.2. Конструкторы, деструкторы и присваивания . . . 868
31.3.3. Размер и емкость . . . 870
31.3.4. Итераторы . . . 870
31.3.5. Доступ к элементам . . . 871

31.3.6. Стековые операции . . . 872
31.3.7. Операции, типичные для списков . . . 873
31.3.8. Прочие операции . . . 874
31.4. Контейнеры . . . 874
31.4.1. vector . . . 874
31.4.1.1. vector и его рост . . . 875
31.4.1.2. vector и проблема вложенности . . . 876
31.4.1.3. vector и массивы . . . 877
31.4.1.4. vector и string . . . 878
31.4.2. Списки . . . 878
31.4.3. Ассоциативные контейнеры . . . 881
31.4.3.1. Упорядоченные ассоциативные контейнеры . . . 881
31.4.3.2. Неупорядоченные ассоциативные контейнеры . . . 885
31.4.3.3. Конструирование unordered_map . . . 885
31.4.3.4. Хэш-функции и функции эквивалентости . . . 888
31.4.3.5. Фактор загрузки и корзины . . . 890
31.5. Адаптеры контейнеров . . . 891
31.5.1. stack . . . 891
31.5.2. queue . . . 893
31.5.3. priority_queue . . . 893
31.6. Советы . . . 894

32. STL алгоритмы . . . 896
32.1. Введение . . . 896
32.2. Алгоритмы . . . 896
32.2.1. Последовательности . . . 897
32.3. Аргументы политик . . . 898
32.3.1. Сложность . . . 900
32.4. Немодифицирующие алгоритмы . . . 900
32.4.1. for_each() . . . 901
32.4.2. Предикаты для последовательностей . . . 901
32.4.3. count() . . . 901
32.4.4. find() . . . 902
32.4.5. equal() и mismatch() . . . 902
32.4.6. search() . . . 903
32.5. Модифицирующие алгоритмы . . . 904
32.5.1. copy() . . . 904
32.5.2. unique() . . . 905
32.5.3. remove() и replace() . . . 907
32.5.4. rotate(), random_shuffle() и partition() . . . 907
32.5.5. Перестановки . . . 908
32.5.6. fill() . . . 909
32.5.7. swap() . . . 910
32.6. Сортировка и поиск . . . 910
32.6.1. Бинарный поиск . . . 913
32.6.2. merge() . . . 914
32.6.3. Алгоритмы для множеств . . . 915
32.6.4. Кучи . . . 916
32.6.5. lexicographical_compare() . . . 917
32.7. Минимумы и максимумы . . . 917
32.8. Советы . . . 918

33. STL итераторы 920
33.1. Введение 920
33.1.1. Модель итераторов 921
33.1.2. Категории итераторов 922
33.1.3. Характеристики итераторов 923
33.1.4. Операции над итераторами 925
33.2. Адаптеры итераторов 927
33.2.1. Обратные итераторы 927
33.2.2. Итераторы вставки 929
33.2.3. Перемещающий итератор 930
33.3. Функции доступа к диапазону 931
33.4. Функциональные объекты 932
33.5. Адаптеры функций 933
33.5.1. bind() 933
33.5.2. mem_fn() 935
33.5.3. function 935
33.6. Советы 937

34. Память и ресурсы 938
34.1. Введение 938
34.2. «Почти контейнеры» 938
34.2.1. array 939
34.2.2. bitset 942
34.2.2.1. Конструкторы 943
34.2.2.2. Операции над bitset 944
34.2.3. vector<bool> 946
34.2.4. Группировка значений 947
34.2.4.1. pair 947
34.2.4.2. tuple 948
34.3. Указатели для управления ресурсами 950
34.3.1. unique_ptr 950
34.3.2. shared_ptr 954
34.3.3. weak_ptr 957
34.4. Аллокаторы 959
34.4.1. Умолчательный аллокатор 960
34.4.2. Характеристики аллокаторов 961
34.4.3. Характеристики указателей 962
34.4.4. Вложенные аллокаторы 963
34.5. Интерфейс сборщика мусора 964
34.6. Неинициализированная память 967
34.6.1. Временные буферы 967
34.6.2. raw_storage_iterator 968
34.7. Советы 969

35. Прочие компоненты библиотеки 970
35.1. Введение 970
35.2. Время 970
35.2.1. duration 971
35.2.2. time_point 974
35.2.3. Часы 975
35.2.4. Характеристики времени 976

35.3. Рациональная арифметика времени компиляции . . . 977
35.4. Функции типа . . . 978
35.4.1. Характеристики типов . . . 978
35.4.2. Генераторы типа . . . 982
35.5. Небольшие утилиты . . . 987
35.5.1. move() и forward() . . . 987
35.5.2. swap() . . . 988
35.5.3. Операции сравнения . . . 988
35.5.4. Сравнение и хэширование type_info . . . 989
35.6. Советы . . . 990

36. Строки . . . 991
36.1. Введение . . . 991
36.2. Классификация символов . . . 991
36.2.1. Функции классификации . . . 991
36.2.2. Характеристики символов . . . 992
36.3. Строки . . . 994
36.3.1. string против строк в С-стиле . . . 994
36.3.2. Конструкторы . . . 996
36.3.3. Важнейшие операции . . . 997
36.3.4. Ввод/вывод . . . 998
36.3.5. Числовые преобразования . . . 999
36.3.6. Операции в стиле STL . . . 1000
36.3.7. Семейство функций find . . . 1003
36.3.8. Подстроки . . . 1004
36.4. Советы . . . 1005

37. Регулярные выражения . . . 1006
37.1. Регулярные выражения . . . 1006
37.1.1. Нотация регулярных выражений . . . 1007
37.2. regex . . . 1011
37.2.1. Результаты поиска соответствия . . . 1013
37.2.2. Форматирование . . . 1015
37.3. Функции регулярных выражений . . . 1016
37.3.1. regex_match() . . . 1016
37.3.2. regex_search() . . . 1018
37.3.3. regex_replace() . . . 1019
37.4. Итераторы регулярных выражений . . . 1020
37.4.1. regex_iterator . . . 1020
37.4.2. regex_token_iterator . . . 1021
37.5. regex_traits . . . 1023
37.6. Советы . . . 1024

38. Потоки ввода/вывода . . . 1025
38.1. Введение . . . 1025
38.2. Иерархия потоков ввода/вывода . . . 1027
38.2.1. Файловые потоки . . . 1028
38.2.2. Строковые потоки . . . 1029
38.3. Обработка ошибок . . . 1031
38.4. Операции ввода/вывода . . . 1032
38.4.1. Операции ввода . . . 1032

38.4.1.1. Форматированный ввод ... 1033
38.4.1.2. Неформатированный ввод ... 1034
38.4.2. Операции вывода ... 1035
38.4.2.1. Виртуальные функции вывода ... 1036
38.4.3. Манипуляторы ... 1037
38.4.4. Состояние потока ... 1038
38.4.5. Форматирование ... 1042
38.4.5.1. Состояние форматирования ... 1042
38.4.5.2. Стандартные манипуляторы ... 1044
38.4.5.3. Пользовательские манипуляторы ... 1046
38.5. Итераторы потоков ... 1048
38.6. Буферизация ... 1049
38.6.1. Потоки вывода и буферы ... 1052
38.6.2. Потоки ввода и буферы ... 1053
38.6.3. Итераторы буферов ... 1054
38.6.3.1. istreambuf_iterator ... 1054
38.6.3.2. ostreambuf_iterator ... 1055
38.7. Советы ... 1056

39. Локализация ... 1058
39.1. Культурные особенности ... 1058
39.2. Класс locale ... 1061
39.2.1. Именованные locale ... 1063
39.2.1.1. Конструирование новых locale ... 1065
39.2.2. Сравнение строк ... 1066
39.3. Класс facet ... 1067
39.3.1. Доступ к facet из locale ... 1068
39.3.2. Простой пользовательский facet ... 1068
39.3.3. Применение locale и facet ... 1071
39.4. Стандартные фасеты ... 1072
39.4.1. Операции сравнения для string ... 1074
39.4.1.1. Именованный collate ... 1076
39.4.2. Форматирование чисел ... 1077
39.4.2.1. Числовая пунктуация ... 1077
39.4.2.2. Числовой вывод ... 1078
39.4.2.3 Ввод чисел ... 1080
39.4.3. Форматы денег ... 1082
39.4.3.1. Денежная пунктуация ... 1082
39.4.3.2. Вывод денежных сумм ... 1085
39.4.3.3. Ввод денежных сумм ... 1085
39.4.4. Форматирование дат и времени ... 1086
39.4.4.1. time_put ... 1086
39.4.4.2. time_get ... 1087
39.4.5. Классификация символов ... 1089
39.4.6. Конверсия кодов символов ... 1092
39.4.7. Сообщения ... 1095
39.4.7.1. Использование сообщений из других facet ... 1098
39.5. Полезные интерфейсы ... 1099
39.5.1. Классификация символов ... 1099
39.5.2. Конверсия символов ... 1100

39.5.3. Конвертирование строк ... 1100
39.5.4. Конвертирование буферов ... 1101
39.6. Советы ... 1102

40. Численные расчеты ... 1103
40.1. Введение ... 1103
40.2. Числовые пределы ... 1104
40.2.1. Макросы числовых пределов ... 1106
40.3. Стандартные математические функции ... 1106
40.4. Комплексные числа ... 1108
40.5. Численные массивы: valarray ... 1109
40.5.1. Конструкторы и присваивания ... 1109
40.5.2. Индексирование ... 1111
40.5.3. Операции ... 1112
40.5.4. Срезы ... 1114
40.5.5. slice_array ... 1116
40.5.6. Обобщенные срезы ... 1117
40.6. Обобщенные численные алгоритмы ... 1118
40.6.1. accumulate() ... 1119
40.6.2. inner_product() ... 1120
40.6.3. partial_sum() и adjacent_difference() ... 1121
40.6.4. iota() ... 1122
40.7. Случайные числа ... 1122
40.7.1. «Движки» ... 1124
40.7.2. Аппаратные генераторы случайных чисел ... 1126
40.7.3. Распределения ... 1127
40.7.4. Случайные числа в языке C ... 1130
40.8. Советы ... 1130

41. Параллелизм ... 1132
41.1. Введение ... 1132
41.2. Модель памяти ... 1134
41.2.1. Память ... 1134
41.2.2. Реорганизация машинных команд ... 1135
41.2.3. Упорядочение памяти ... 1136
41.2.4. Гонка за данными ... 1137
41.3. Атомарность ... 1139
41.3.1. Атомарные типы ... 1141
41.3.2. Флаги и «ограждения» ... 1145
41.3.2.1. Флаги ... 1145
41.3.2.2. «Ограждения» ... 1146
41.4. volatile ... 1146
41.5. Советы ... 1147

42. Потоки и задачи ... 1148
42.1. Введение ... 1148
42.2. Потоки ... 1148
42.2.1. Идентификация ... 1150
42.2.2. Конструирование ... 1150
42.2.3. Деструктор ... 1152
42.2.4. join() ... 1152

42.2.5. detach() ... 1154
42.2.6. Пространство имен this_thread ... 1155
42.2.7. Уничтожение thread ... 1156
42.2.8. Переменные thread_local ... 1156
42.3. Устранение гонки за данными ... 1158
42.3.1. Мьютексы ... 1158
42.3.1.1. mutex и recursive_mutex ... 1160
42.3.1.2. Ошибки mutex ... 1162
42.3.1.3. timed_mutex и recursive_timed_mutex ... 1162
42.3.1.4. lock_guard и unique_lock ... 1163
42.3.2. Множественные блокировки ... 1166
42.3.3. call_once() ... 1167
42.3.4. Условные переменные ... 1168
42.3.4.1. condition_variable_any ... 1172
42.4. Параллелизм на базе задач ... 1172
42.4.1. future и promise ... 1173
42.4.2. promise ... 1174
42.4.3. packaged_task ... 1175
42.4.4. future ... 1178
42.4.5. shared_future ... 1180
42.4.6. async() ... 1181
42.4.7. Пример параллельной find() ... 1183
42.5. Советы ... 1186

43. Стандартная библиотека C ... 1188
43.1. Введение ... 1188
43.2. Файлы ... 1188
43.3. Семейство функций printf() ... 1189
43.4. Строки в C-стиле ... 1193
43.5. Память ... 1194
43.6. Даты и время ... 1195
43.7. Прочие функции ... 1199
43.8. Советы ... 1200

44. Совместимость ... 1201
44.1. Введение ... 1201
44.2. Расширения C++11 ... 1202
44.2.1. Средства языка ... 1202
44.2.2. Компоненты стандартной библиотеки ... 1203
44.2.3. Устаревшие средства ... 1204
44.2.4. Сравнение с ранними реализациями C++ ... 1205
44.3. Совместимость C/C++ ... 1205
44.3.1. C и C++ братья ... 1205
44.3.2. «Тихие» отличия ... 1207
44.3.3. C код, который не есть C++ код ... 1207
44.3.3.1. Проблемы «классического C» ... 1210
44.3.3.2. Средства C, которые не были приняты в C++ ... 1210
44.3.4. C++ код, который не есть C код ... 1211
44.4. Советы ... 1212

Предисловие

Все проблемы в информатике
можно решить еще одним уровнем косвенного обращения,
кроме проблемы слишком большого числа уровней
косвенного обращения.
— Дейвид Дж. Уилер

C++ кажется совсем новым языком. В том смысле, что я могу выражать свои идеи яснее, проще и прямолинейнее в C++11, чем я мог в C++98. Более того, получающиеся программы лучше проверяются компилятором и работают быстрее.

В этой книге, я стремлюсь к *детальности*. Я описываю каждое средство языка, каждый компонент стандартной библиотеки, которые могут потребоваться профессиональному программисту. Для каждого из них я предоставляю:

- *Обоснование*: какого вида проблемы это средство предназначено решать? На каких принципах оно спроектировано? Какие фундаментальные ограничения?
- *Спецификация*: Какое определение этого средства? Детальность описания рассчитана на программиста-эксперта; тот, кто хочет стать «адвокатом» в языке, может воспользоваться многочисленными ссылками на стандарт ISO.
- *Примеры*: Как это средство может использоваться само по себе и в сочетании с другими средствами? Каковы ключевые техники и идиомы? Какие последствия для обслуживаемости и производительности кода?

Использование C++ за годы существенно изменилось и также изменился язык. С точки зрения программиста, большинство изменений являются улучшениями. Текущий стандарт ISO C++ (ISO/IEC 14882-2011, обычно называемый C++11) попросту лучше для написания качественного программного обеспечения. Чем именно лучше? Какие стили и техники программирования поддерживает современный C++? Какие средства языка и стандартной библиотеки позволяют применить эти техники? Что является «кирпичиками» элегантного, правильного, легко обслуживаемого и эффективного кода на C++? Это ключевые вопросы, на которые отвечает эта книга. Многие ответы уже другие, нежели чем в версиях C++ 1985, 1995 или 2005 годов: прогресс, однако.

C++ — язык программирования общего назначения, с акцентом на проектирование и использование легковесных абстракций с обильным набором типов. Он особенно подходит для приложений с ограниченными ресурсами, таких, как инфраструктурные компоненты. C++ вознаграждает программиста, который потратит время, чтобы как следует изучить техники написания качественного кода. C++ — язык для тех, кто относится к программированию серьезно. Наша цивилизация крайне зависит от программного обеспечения; лучше, пусть это будет качественное программное обеспечение.

Написаны уже миллиарды строк кода на C++. Это делает стабильность очень важной, так что код, написанный на C++ 1985 и 1995 годов, по-прежнему работает и будет работать десятилетиями. Но, для всех задач, лучше использовать современный C++; если вы будете придерживаться старых стилей, у вас будет получаться менее качественный и производительный код. Упор на стабильность также означает, что соответствующий стандартам код, который вы пишете сегодня, по-прежнему будет работать и через пару десятилетий. Весь код в этой книге соответствует стандарту ISO C++ 2011 года.

Эта книга рассчитана на следующую аудиторию:

- Программисты C++, которые хотят знать, что предоставляет новейший стандарт ISO C++,
- Программисты C, которые интересуются, что нового есть в C++ по сравнению с C, и
- Люди с опытом работы с прикладными языками, такими как Java, C#, Python и Ruby, ищущие что-нибудь «ближе к аппаратуре» — что-нибудь более гибкое, предоставляющее более тщательную проверку при компиляции или что-нибудь дающее лучшую производительность.

Естественно, эти три группы могут пересекаться — профессиональный разработчик владеет более, чем одним языком.

Эта книга предполагает, что читатели — программисты. Если вы задаете вопрос «Что такое цикл for?» или «Что такое компилятор?», то эта книга (пока) не для вас; в таком случае, я рекомендую мою книгу «*Программирование: принципы и практика с использованием C++*» (*Programming: Principles and Practice Using C++*), чтобы разобраться в том, что есть программирование и C++. Более того, я предполагаю, что читатели имеют некоторый опыт разработки программ. Если вы спрашиваете «Зачем возиться с тестированием?» или говорите «Все языки в общем-то одинаковы; просто покажите синтаксис» или уверены, что есть язык, идеальный для любой задачи, то эта книга не для вас.

Какую функциональность предлагает C++11 по сравнению с C++98? Модель аппаратуры, подходящая к современным компьютерам с большим объемом параллельных вычислений. Средства языка и стандартной библиотеки для низкоуровнего параллельного программирования (напр., для многоядерных систем). Обработка регулярных выражений, указатели управления ресурсами, случайные числа, улучшенные контейнеры (включая хэш-таблицы) и многое другое. Обобщенная и однородная инициализация, упрощенные операторы for, перемещающая семантика, базовая поддержка Unicode, лямбды, обобщенные константные выражения, управление классовыми умолчаниями, шаблоны с переменным числом параметров пользовательские литералы и многое другое. Помните, что эти библиотеки и средства языка задуманы под соответствующие техники программирования. Они предназначены для использования в сочетании — как кирпичики в наборе, а не по-отдельности в относительной изоляции для решения конкретной проблемы. Компьютер — универсальная машина и C++ служит ему в этом качестве. В особенности, C++ спроектирован быть достаточно гибким и общим чтобы справляться с будущими проблемами, о которых и не догадывались его разработчики.

Благодарности

Помимо людей, упомянутых в разделах благодарности предыдущих изданий, я хочу поблагодарить Pete Becker, Hans-J. Boehm, Marshall Clow, Jonathan Coe, Lawrence Crowl, Walter Daugherty, J. Daniel Garcia, Robert Harle, Greg Hickman, Howard Hinnant, Brian Kernighan, Daniel Krügler, Nevin Liber, Michel Michaud, Gary Powell, Jan Christiaan van Winkel и Leor Zolman. Без их помощи, эта книга была бы намного беднее.

Спасибо Howard Hinnant за ответы на многочисленные вопросы о стандартной библиотеке.

Andrew Sutton — автор библиотеки Origin, которая была платформой для опытов, которые помогли написать главы о шаблонах, и матричной библиотеки, которая рассматривается в Главе 29. Библитека Origin имеет открытый исходный код и может быть найдена в Интернете с помошью запросов "Origin" и "Andrew Sutton."

Спасибо моим аспирантам за то, что нашли больше неточностей в «пробных главах», чем кто-либо другой.

Если бы я мог учесть все советы моих рецензентов, книга без сомнения была бы намного улучшена, но она стала бы и на несколько сот страниц длиннее. Каждый рецензирующий мою книгу специалист предлагал добавить технические детали, расширенные примеры и многие полезные методики проектирования; каждый рецензент-новичок (или преподаватель) предлагал добавить примеров; при этом большинство (правильно) замечали, что книга может быть слишком длинна.

Спасибо факультету информатики Принстаунского Университета и особенно профессору Брайану Кернигану (Brian Kernighan) у которого я проводил часть отпуска во время которого я писал эту книгу.

Спасибо моему редактору, Peter Gordon, и его комманде в Addison-Wesley за их помощь и терпение.

Колледж Стейшн, Техас

Бьерн Страуструп

Предисловие автора к третьему изданию на русском языке

В августе 1998 года был ратифицирован стандарт языка C++ (ISO/IEC 14882 «Standard for the C++ Programming Language», результат голосования национальных комитетов по стандартизации: 22–0). Это событие знаменует новую эру стабильности и процветания C++, а также связанных с языком инструментальных средств и приемов программирования.

Лично для меня главным является то, что стандартный C++ лучше чем любая его предыдущая версия соответствует моему замыслу C++. Стандартный C++ и его стандартная библиотека позволяют мне писать лучшие, более элегантные и более эффективные программы, чем те, что я мог писать в прошлом.

Чтобы добиться этого, я и сотни других людей долго и интенсивно работали в комитетах по стандартизации ANSI и ISO. Целью этих усилий было описать язык и библиотеку, которые будут исправно служить всем пользователям языка, не предоставляя каких-либо преимуществ одной группе пользователей, компании или стране перед остальными. Процесс стандартизации был открыт, справедлив, нацелен на качество и согласие.

Усилия по стандартизации были инициированы Дмитрием Ленковым (Dmitry Lenkov) из Hewlett-Packard, который работал в качестве первого председателя комитета. Во время завершения работ над стандартом председателем был Стив Кламаг (Steve Clamage) из Sun. Джонатан Шапиро (Jonathan Shopiro) и Эндрю Кениг (Andrew Koenig) (оба из AT&T) были редакторами, которые — с помощью многих членов комитета — подготовили текст стандарта, основанный на моем исходном справочном руководстве по C++.

Открытый и демократичный процесс стандартизации потенциально таит в себе одну опасность: результат может оказаться «комитетской разработкой». В случае с C++ этого по большей части удалось избежать. Во-первых, я состоял председателем рабочей группы по расширению языка. В этом качестве я оценивал все основные предложения и составлял окончательные варианты тех из них, которые я, рабочая группа и комитет считали стоящими и в тоже время реализуемыми. Таким образом, комитет в основном обсуждал направляемые ему относительно завершенные проекты, а не занимался разработкой таковых. Во-вторых, главная из новых составляющих стандартной библиотеки — стандартная библиотека шаблонов STL, предоставляющая общую, эффективную, типобезопасную и расширяемую среду контейнеров, итераторов и алгоритмов — была, в основном, результатом работы одного человека, Александра Степанова (Alexander Stepanov) из Hewlett-Packard.

Важно, что стандарт C++ — это не просто бумажка. Он уже встроен в наиболее свежие реализации C++. Большинство основных реализаций поддерживают стандарт (с очень немногочисленными исключениями), причем все обещают полное соответствие

в течение ближайших месяцев. Чтобы стимулировать честность производителей, две компании предлагают пакеты, проверяющие «стандартность» реализации C++.

Итак, код, который я пишу сейчас, использует, если это необходимо, большинство возможностей, предлагаемых стандартным C++ и описанных в данном третьем издании «Языка программирования C++».

Улучшения в языке C++ и расширение стандартной библиотеки заметно изменили то, как я пишу код. Мои теперешние программы короче, яснее и эффективнее прежних, что является прямым результатом лучшей, более ясной и систематичной поддержки абстракций в стандартном C++. Улучшенная поддержка таких средств, как шаблоны и исключения, сокращают потребность в более низкоуровневых и запутанных возможностях языка. Поэтому я считаю, что стандартный C++ проще в использовании, чем его предшествующие варианты, где мне приходилось имитировать средства, которые теперь поддерживаются непосредственно.

Основываясь на приведенных наблюдениях, я решил, что необходимо полностью переписать второе издание «Языка программирования C++». Никак иначе нельзя было должным образом отразить новые возможности, предлагаемые языком, и поддерживаемые ими приемы программирования и проектирования. В результате 80% материала третьего издания добавлено по сравнению со вторым. Изменилась и организация книги — теперь больший упор делается на технику и стиль программирования. Рассматриваемые изолированно, особенности языка скучны — они оживают лишь в контексте приемов программирования, ради поддержки которых и задумывались.

Соответственно, C++ может теперь преподаваться как язык более высокого уровне. То есть основное внимание можно уделять алгоритмам и контейнерам, а не жонглированию битами, объединениями, строками в стиле C, массивами и проч. Более низкоуровневые понятия (такие как массивы, нетривиальное использование указателей и приведения типов), естественно, также придется изучить. Но их представление теперь удается отложить до тех пор, пока новичок в программировании на C++, читатель или студент не обретут зрелость, необходимую, чтобы воспринимать эти средства в контексте более высокоуровневых концепций, к применению которых обучаемые уже привыкли.

В частности, невозможно переусердствовать в подчеркивании важности преимущества статически типобезопасных строк и контейнеров перед стилями программирования, предполагающими использование множества макросов и приведений типов. В третьем издании «Языка программирования C++» мне удалось избавиться от почти всех макросов и свести использование приведений типов к немногочисленным случаям, когда они действительно существенны. Я считаю серьезным недостатком принятую в C/C++ форму макросов, которую теперь можно считать устаревшей благодаря наличию более подходящих средств языка, таких как шаблоны, пространства имен, встроенные функции и константы. Точно также, широкое использование приведений типов в любом языке сигнализирует о плохом проектировании. Как макросы, так и приведения являются частыми источниками ошибок.

Тот факт, что без них можно обойтись, делает программирование на C++ куда более безопасным и элегантным.

Стандарт C++ предназначен для того, чтобы изменить наш способ программировать на C++, проектировать программы и обучать программированию на C++. Подобные изменения не происходят за один вечер. Я призываю вас не торопясь, внимательно присмотреться к стандарту C++, к приемам проектирования и программирования, используемым в третьем издании «Языка программирования C++», — и к вашему собственному стилю программирования. Предполагаю, что вполне возможны значительные улучшения.

Но пусть ваша голова остается холодной! Чудес не бывает — при написании кода опасно использовать языковые возможности и приемы, которые вы понимаете лишь частично. Настало время изучать и экспериментировать — только поняв новые концепции и приемы, можно воспользоваться по-настоящему существенными преимуществами стандартного C++.

Добро пожаловать! Бьерн Страуструп

Предисловие к третьему изданию

Программирование есть понимание.
— Кристин Нюгард

Применение языка C++ доставляет мне все большее удовольствие. За прошедшие годы средства поддержки проектирования и программирования на C++ кардинально улучшились, а также было предложено множество полезных и удобных методик их использования. Но речь идет не только об удовольствии. Практикующие программисты достигли значительного повышения производительности, уровня поддержки, гибкости и качества в проектах любого уровня сложности и масштаба. К настоящему времени C++ оправдал большую часть возлагавшихся на него надежд, а кроме того он преуспел в решении задач, о которых я даже и не мечтал.

Эта книга представляет стандарт C++[1] и ключевые методы программирования и проектирования, поддерживаемые этим языком. Стандарт C++ намного мощнее и точнее, чем то, что излагалось в первом издании данной книги. Новые языковые черты, такие как пространства имен, исключения, шаблоны и механизм динамического определения типов, позволяют реализовывать многие методики программирования более непосредственно, чем это было возможно ранее. Стандартная библиотека предоставляет программистам возможность стартовать с существенно более высокоуровневых конструкций чем те, что доступны в «голом» языке.

Всего лишь треть информации из первого издания перекочевало во второе издание данной книги. Третье издание переработано еще более интенсивно. В ней есть кое-что, что будет полезно самым опытным программистам на C++. В то же время, она более доступна новичкам, чем предыдущие издания этой книги. Это стало возможным благодаря взрывному росту практического использования C++ и опыту, вытекающему из этой практики.

Стандартная библиотека позволяет по-новому представить основные концепции языка C++. Как и предыдущие издания, настоящая книга описывает язык C++ независимо от конкретных реализаций. Как и в предыдущих изданиях, все конструкции и концепции излагаются «снизу-вверх», так что к тому моменту, когда они начинают использоваться, они уже представлены и подробно описаны. Отметим, что начать использовать хорошо спроектированную библиотеку можно прежде, чем читатели смогут изучить ее внутрен-

[1] ISO/IEC 14882:2003(E), Стандарт языка программирования C++.

нее устройство. Кроме того, стандартная библиотека сама по себе является ценнейшим источником примеров программирования и методов проектирования.

В настоящей книге рассматриваются все основные средства языка и стандартной библиотеки. Ее материал организован вокруг этих основных средств, однако фокус изложения направлен на применение средств языка как инструмента проектирования и программирования, а не на формальные конструкции как таковые. Книга демонстрирует ключевые методики, делающие C++ эффективным, и излагает эти методики так, чтобы их постижение повышало мастерство программиста. За исключением глав, иллюстрирующих технические детали, практические примеры кода взяты из области системного программирования. Дополнением к данной книге может служить исчерпывающий справочник *The Annotated C++ Language Standard.*

Основной целью данной книги является стремление помочь читателю понять, как средства языка C++ непосредственно поддерживают ключевые методики программирования. Задача ставится так, чтобы подвигнуть читателя отойти как можно дальше от простого копирования примеров кода и эмулирования методик, присущих другим языкам программирования. Только отличное понимание идей, стоящих за средствами языка, ведет к настоящему мастерству. Дополненная деталями реализации информация из настоящей книги является достаточной для выполнения реальных проектов заметного масштаба. Я надеюсь, что эта книга поможет читателю по-новому взглянуть на программирование на языке C++ и стать более квалифицированным программистом и проектировщиком.

Благодарности

Помимо людей, упомянутых в соответствующих разделах предыдущих изданий книги, я хотел бы поблагодарить еще и следующих: Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl, Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger и Chris Van Wyk — за комментирование отдельных глав третьего издания. Без их помощи текст книги был бы менее понятен, содержал бы больше ошибок, был бы менее полным, а книга была бы тоньше.

Я также хотел бы поблагодарить добровольцев из Комитета по стандартизации C++ за их титанический труд, сделавший C++ тем, чем он сейчас является. Несправедливо здесь выделять кого-либо, но еще более несправедливо было бы не упомянуть следующих: Mike Ball, Dag Brack, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, Jose Lajoie, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov и Mike Vilot — каждый из них работал со мной над отдельными частями C++ или стандартной библиотеки.

После выхода первого тиража этой книги десятки людей присылали мне исправления и предложения по ее улучшению. Многие из этих предложений я учел при подготовке более поздних тиражей, от чего они только выиграли. Переводчики этой книги на иностранные языки также высказывали свои соображения относительно пояснения отдельных моментов. Отвечая на просьбы читателей я добавил приложения D и E. Особо хотел бы при этом поблагодарить следующих людей: Dave Abrahams, Matt Austern, Jan Bielawski, Janina Mincer Daszkiewicz, Andrew Koenig, Dietmar Kuhl, Nicolai Josuttis, Nathan Myers, Paul E. Sevin, Andy Tenne-Sens, Shoichi Uchida, Ping-Fai (Mike) Yang и Dennis Yelle.

Мюррей Хилл, Нью Джерси

Бьерн Страуструп

Предисловие ко второму изданию

А дорога бежит и бежит.
— Бильбо Бэггинс

Как было предсказано в первом издании этой книги, C++ эволюционировал в соответствии с запросами его пользователей. Эта эволюция направлялась запросами пользователей самой разной квалификации, работающих в разных предметных областях. Сообщество программистов на C++ выросло в сотни раз за шесть лет, прошедшие с момента выхода первого издания этой книги. За это время были разработаны многочисленные методики, усвоены полезные практические уроки, приобретен бесценный опыт. Часть этого опыта отражена в настоящем издании.

Главной целью развития и расширения языка за эти шесть лет было стремление сделать C++ языком абстракции данных и языком объектно-ориентированного программирования в целом, языком, пригодным для написания высококачественных библиотек, в том числе библиотек пользовательских типов. Высококачественная библиотека — это библиотека, которая предоставляет пользователю концепцию в виде одного или нескольких классов, которые удобны в применении, безопасны и эффективны. Здесь безопасность означает предоставление пользователю интерфейса к содержимому библиотеки, безопасному в отношении типов. Эффективность означает, что классовая структура библиотеки не налагает дополнительной нагрузки на процессор и память по сравнению с ручным кодом, написанным на C.

В настоящей книге приводится полное описание языка C++. В главах 1–10 сосредоточен вводный учебный материал. Главы 11–13 посвящены обсуждению вопросов проектирования и реализации; наконец, далее приведено полное справочное руководство по C++. Естественно, в настоящей книге присутствуют новые черты и свойства языка, появившиеся после выхода первого издания книги: уточненный механизм разрешения перегрузки, средства управления памятью, безопасный по типам механизм компоновки, константные и статические члены классов, абстрактные классы, множественное наследование, шаблоны, обработка исключений.

C++ является языком общего назначения; его естественной сферой применения является системное программирование в самом общем смысле этого термина. Тем не менее, C++ с успехом используется и в других предметных областях. Реализации C++ имеются и для маломощных микрокомпьютеров, и для мощнейших суперкомпьютерных систем и разных операционных сред. Поэтому настоящая книга описывает собственно язык C++ без его привязки к конкретным операционным системам и их библиотекам.

В книге имеется большое число примеров классов, которые будучи полезными должны быть охарактеризованы как «игрушечные». Такой стиль подачи общих принципов и полезных методик позволяет лучше сфокусировать на них внимание, чем это возможно в контексте больших реальных программ, где они были бы похоронены в многочисленных деталях. Большую часть полезных классов, приведенных в книге, таких как строки, связанные списки, массивы, матрицы, графические классы, ассоциативные массивы и т.д. можно найти в более совершенном виде в разных коммерческих и некоммерческих источниках. Многие из этих промышленных классов являются прямыми или косвенными потомками приведенных здесь игрушечных классов.

По сравнению с предыдущим изданием настоящее больше сфокусировано на обучении, но форма подачи материала по-прежнему ориентирована на опытных программистов и разработчиков, дабы не оскорблять их опыт и квалификацию. Более широкое обсуждение вопросов проектирования отражает спрос на информацию, не замыкающуюся лишь на изучении средств языка и их непосредственного использования. Увеличено число приведенных технических деталей, а также качество их изложения. В частности, справочное руководство по языку отражает многолетний опыт работы в этом направлении. Мне хотелось написать книгу, которую к которой большинство программистов обращалось бы не один раз. Таким образом, данная книга представляет язык C++, его фундаментальные принципы и ключевые технологии, в рамках которых язык используется. Успехов!

Благодарности

Помимо людей, упомянутых в соответствующих разделах первого издания книги, я хотел бы поблагодарить еще и следующих: Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot, and Peter Weinberger — за комментирование отдельных глав второго издания.

Многие люди оказали влияние на развитие C++ с 1985 года по 1991 год. Я могу упомянуть лишь некоторых из них: Andrew Koenig, Brian Kernighan, Doug McIlroy и Jonathan Shopiro. Также большое спасибо всем участникам просмотра и рецензирования черновика справочного руководства и всем людям, настрадавшимся в первый год работы комитета X3J16.

Мюррей Хилл, Нью Джерси

Бьерн Страуструп

Предисловие к первому изданию

Язык формирует наше мышление
и определяет то, о чем мы можем думать.
— Б.Л. Ворф

C++ является языком общего назначения, цель которого — сделать работу серьезного программиста более приятной. За исключением некоторых деталей, C++ является надмножеством языка программирования C. Помимо возможностей языка C, C++ предоставляет гибкие и эффективные средства для определения новых типов. Программист может сегментировать приложение на фрагменты, определив для этого новые типы, отвечающие концепциям приложения. Такую технику программирования часто называют абстракцией данных. Объекты пользовательских типов содержат информацию, характерную для этих типов. Такие объекты можно с удобством и безопасностью использовать в контекстах, когда их тип неизвестен на этапе компиляции. Программы, использующие подобного рода объекты, часто называют объектными. При надлежащем применении такая техника ведет к более коротким программам, более понятным и более удобным в сопровождении.

Ключевой концепцией языка C++ является класс. Класс — это тип, определяемый пользователем. Классы обеспечивают сокрытие данных, гарантированную инициализацию данных, неявное преобразование между пользовательскими типами, динамическое определение типа, контролируемое управление памятью и механизмы перегрузки операций. C++ гораздо лучше контролирует типы, чем C, а также позволяет достичь более высокой модульности. Он также содержит улучшения, не имеющие прямого отношения к классам, такие как символические константы, встраиваемые функции, аргументы функций по умолчанию, перегрузка имен функций, операции по управлению выделением памяти, а также ссылки. C++ сохраняет возможности языка C по эффективной работе с низкоуровневыми аппаратнозависимыми базовыми типами (битами, байтами, словами, адресами и т.д.). Это позволяет реализовать пользовательские типы с высокой эффективностью.

C++ и его стандартные библиотеки построены с учетом переносимости. Текущая реализация будет работать на большинстве систем, поддерживающих язык C. Библиотеки языка C можно использовать в программах на C++, также как и большинство инструментальных средств языка C.

Данная книга предназначена в первую очередь для того, чтобы серьезные программисты могли изучить C++ и использовать его в нетривиальных разработках. Книга содержит полное описание C++, множество законченных примеров и еще большее количество фрагментов программ.

Благодарности

C++ никогда не достиг бы необходимой зрелости без интенсивного использования, советов и конструктивной критики со стороны друзей и коллег. В частности, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz и Jon Shopiro подали важные для развития языка идеи. Dave Presotto написал текущую реализацию библиотеки потокового ввода/вывода.

Кроме того, сотни людей способствовали развитию языка C++ и его компилятора, присылая мне свои предложения по улучшениям, описания проблем, с которыми они сталкивались, а также сообщения об ошибках компилятора. Я могу отметить здесь лишь некоторых: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, и Gary Walker.

Множество людей помогали с изданием данной книги. Среди них: Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro и участники семинара по C++ в Bell Labs, Columbus, штат Ohio (Огайо), 26-27 июня 1985 года.

Мюррей Хилл, Нью Джерси

Бьерн Страуструп

Часть 1

Введение

В этом введении предоставляется краткий обзор основных концепций и средств языка C++ и его стандартной библиотеки. Здесь также дается обзор этой книги и применяемых в ней методов описания средств языка и их использования. В добавок, вводные главы предлагают немного истории C++, описывают проект C++ и его применение.

Главы

1. Обращение к читателю
2. Обзор языка C++: Основы
3. Обзор языка C++: Механизмы абстракции
4. Обзор языка C++: Контейнеры и Алгоритмы
5. Обзор языка C++: Параллелизм и Утилиты

«... и ты, Маркус, ты дал мне многое. Теперь я дам тебе хороший совет. Будь многими. Брось играть в бытие одним лишь Маркусом Кокоза. Ты так сильно волновался за Маркуса Кокоза, что стал его рабом и пленником. Ты не делал ничего, не подумав сначала о его благополучии и престиже. Ты всегда боялся того, что Маркус совершит какую-либо глупость или ему станет скучно. А что в этом страшного? Во всем мире люди совершают глупости. Я хочу, чтобы ты жил легче. Будь больше, чем одним, будь многими людьми, столь многими, сколь можешь представить себе.»

— Карен Бликсен (Karen Blixen)
"*The Dreamers*" из книги "*Seven Gothic Tales*" (1934)

1

Обращение к читателю

Поспешай неспеша.
— Октавиан, Цезарь Август

- Структура книги

  Введение; Базовые средства; Механизмы абстракции; Стандартная библиотека; Примеры и ссылки
- Дизайн C++

  Стили программирования; Проверка типов; Совместимость с C; Язык, библиотеки и операционные системы
- Изучение C++

  Программирование на C++; Советы C++-программистам; Советы C-программистам; Советы Java-программистам)
- История C++

  Хронология; Ранний период; Стандарт 1998; Стандарт 2011; Где используется C++)
- Советы
- Библиография

1.1. Структура книги

В учебной литературе принято располагать материал таким образом, чтобы никакая концепция не использовалась прежде, чем она будет регулярным образом представлена; этот тип литературы предполагает последовательное чтение, начиная ровно со страницы номер один. По противоположному принципу строится справочная литература — в нее можно заглядывать в любом порядке и в любое место; в ней материал описывается посредством ссылок (прямых и обратных) на другой материал, так или иначе связанный с текущим. Чисто учебная литература позволяет пользоваться ей без предвариательных условий; она тщательно разжевывает все вопросы. А чисто справочную литературу может использовать лишь человек, знакомый с базовыми принципами предметной области (и, желательно, с некоторым практическим опытом). Настоящая книга не попадает ни в одну из этих категорий; она сочетает в себе оба подхода. Если вы уже знаккомы с C++, можете выбирать ту или иную главу или даже секцию главы. В противоположном случае вам придется все читать последовательно, стараясь все-таки не утонуть в деталях. Используйте индекс и перекрестные ссылки.

Попытка сделать отдельные части книги самосогласованными предполагает некоторую долю повторений, но повторения помогают в том числе и людям, читающим книгу последовательно. В книге много перекрестных ссылок как на материал самой этой книги, так и на стандарт ISO C++. Опытные программисты могут ограничиться обзором C++,

дающим общее представление, достаточное для того, чтобы далее использовать эту книгу как справочник.

Данная книга состоит из четырех частей:

Часть I *Введение*: Глава 1 служит введением в книгу и содержит немного основ C++. Главы 2–5 представляют быстрое введение в язык C++ и стандартную библиотеку.

Часть II *Базовые средства*: Главы 6–15 описывают встроенные типы языка C++ и основные средства построения программ на их основе.

Часть III *Механизмы абстракции*: Главы 16–29 описывают механизмы абстракции языка C++ и их применение для объектно-ориентированного и обобщенного программирования.

Часть IV Главы 30–44 посвящены обзору стандартной библиотеки и обсуждению вопросов совместимости.

1.1.1. Введение

В настоящей главе дается обзор всей книги, советы по ее использованию, а также самые основы языка C++ и его практического применения. Вы можете бегло просмотреть эту главу, останавливаясь лишь на том, что у вас вызвало интерес. Впоследствии, после прочтения других глав книги, вы можете вернуться и прочитать эту главу подробнее.

В следующих четырех главах представлен обзор основных концепций и средств языка C++ и его стандартной библиотеки:

Глава 2 *Обзор C++: Основы* — описывает модель памяти, вычисления и обработку ошибок.

Глава 3 *Обзор C++: Механизмы абстракции* — представляет средства языка, поддерживающие абстракцию данных, объектно-ориентированное программирование и обобщенное программирование.

Глава 4 *Обзор C++*: Контейнеры и алгоритмы — введение в строки, потоковый ввод/вывод, контейнеры и алгоритмы, как они представлены в стандартной библиотеке.

Глава 5 *Обзор C++*: Параллелизм и прочие компоненты библиотеки — рассматривает средства стандартной библиотеки, имеющие отношение к управлению ресурсами, параллелизму, математическим вычислениям, регулярным выражениям и прочие средства.

Столь быстрый обзор средств языка C++ нацелен на то, чтобы дать читателю общее представление о том, что предлагает язык C++. В том числе этот обзор поможет читателю осознать, какой долгий путь прошел язык C++ с тех пор, как вышли первое, второе и третье издание данной книги.

1.1.2. Базовые средства

Часть II книги фокусируется на подмножестве языка C++, которое поддерживает стиль программирования, традиционно свойственный языку C или подобным ему. Вводятся понятия типа, объекта, области видимости и хранения. В ней определяются самые фундаментальные понятия: выражения, операторы и функции. Модульность — как она поддерживается пространствами имен, исходными файлами и обработкой исключений — также не остается без рассмотрения:

Глава 6 Типы и объявления: фундаментальные типы, именование, области видимости, инициализация, простой вывод типа, время жизни объектов и синонимы типа.

Глава 7 *Указатели*, *Массивы* и *Ссылки*

Глава 8 *Структуры*, *Объединения* и *Перечисления*

Глава 9 *Операторы*: объявления как операторы, операторы выбора (**if** и **switch**), операторы цикла (**for**, **while** и **do**), **goto** и комментарии

Глава 10 *Выражения*: пример программы калькулятора, обзор операций, константные выражения и неявное приведение типа.

Глава 11 *Операции*: логические операции, условные выражения, инкремент и декремент, свободная память (**new** и **delete**), {}-списки, лямбда-выражения, явное приведение типов (**static_cast** и **const_cast**).

Глава 12 *Функции*: объявления и определения функций, **inline**-функции, **constexpr**-функции, передача аргументов, перегруженные функции, пред- и постусловия, указатели на функции, макросы.

Глава 13 *Обработка исключений*: разные стили обработки ошибок, гарантии исключений, управление ресурсами, инварианты, **throw** и **catch**, реализация **vector**

Глава 14 *Пространства имен*: **namespace**, модуляризация и интерфейс, композиция на основе пространств имен

Глава 15 *Исходные файлы и программы*: раздельная компиляция, компоновка, применение хедер-файлов, старт программы и ее завершение

Я полагаю, что вы так или иначе знакомы с большинством программных концепций из Части I. Я, например, объясняю, как с помощью C++ выразить итерацию и рекурсию, но не трачу время на то, чтобы подробно обосновать важность этих концепций.

Исключением из этого правила является тема обработки исключений. Многие программисты либо плохо знакомы с этой темой, либо знакомы с ней по опыту других языков (таких как Java), в которых управление ресурсами и обработка исключений не столь плотно интегрированы. Поэтому в главе, посвященной обработке исключений (Глава 13), я излагаю базовые основы философии подхода C++ к обработке исключений и управлению ресурсами.

1.1.3. Механизмы абстракции

Часть III описывает средства языка C++, поддерживающие различные формы абстракции, включая объектно-ориентированное и обобщенное программирование. В них рассматриваются классы, классовые иерархии и шаблоны.

Первые четыре главы концентрируются собственно на классах:

Глава 16 *Классы*: понятие пользовательского типа, класса, является основой всех механизмов абстракции C++

Глава 17 *Конструирование*, *очистка*, *копирование* и *перемещение* демонстрируют, как программист может определить процесс создания и инициализации объектов класса. Далее уточняются процессы копирования, перемещения и уничтожения объектов.

Глава 18 *Перегрузка операций* разъясняет правила, по которым можно определять операции для пользовательских типов (с упором на традиционные арифметические и логические операции, такие как **+**, ***** и **&**).

Глава 19 *Особые операции* — обсуждается применение для пользовательских типов таких операций, как операция **[]** для индексирования, **()** для функциональных объектов и **->** для интеллектуальных указателей.

Классы могут быть организованы в иерархии:

Глава 20 *Производные классы* — рассматриваются основы построения иерархий классов и их фундаментальное применение. Можно четко разграничить интерфейс (абстрактный класс) и его реализации (производные классы); связь между ними обеспечивается с помощью виртуальных функций. Рассматриваются методы контроля за доступом (**public**, **protected** и **private**).

Глава 21 *Иерархии классов* — рассматриваются эффективные приемы использования классовых иерархий. Рассматривается множественное наследование, когда класс имеет более одного прямого базового класса.

Глава 22 RTTI — способ навигации по классовым иерархиям, отталкиваясь от данных, хранящихся в объектах. Мы используем **dynamic_cast** для ответа на вопрос о том, определен ли объект базового класса как объект класса производного, а **typeid** позволяет получить небольшую информацию от объекта (например, каково имя его класса).

Многие из чрезвычайно гибких, эффективных и полезных абстракций опираются на параметризацию типов (классов) и алгоритмов (функций) другими типами и алгоритмами:

Глава 23 *Шаблоны* — в этой главе представлены самые основы шаблонов и техники их использования. Вводятся понятия шаблонов классов, шаблонов функций и синонимов шаблонов.

Глава 24 *Обобщенное программирование* — рассматриваются основы технологии обобщенного программирования. В центре рассмотрения находится технология «поднятия» абстрактного алгоритма из набора конкретного кода. Также рассматривается понятие концепций, специфицирующих требования обобщенного алгоритма к своим аргументам.

Глава 25 *Специализация* — рассматривается, как шаблоны используются для генерации классов и функций, специализаций, отталкиваясь от предоставленных им аргументов.

Глава 26 *Конкретизация* — рассмотрение фокусируется на правилах связывания имен.

Глава 27 *Шаблоны и иерархии* — объясняется, как шаблоны и классовые иерархии могут использоваться совместно.

Глава 28 *Метапрограммирование* — изучается вопрос о том, как шаблоны могут использоваться для генерации программ. Шаблоны полностью обеспечивают механизм Тьюринга для генерации кода.

Глава 29 *Проектирование матриц* — рассматривается большой проект, основанный на комбинировании различных механизмов языка для решения сложной задачи: как сформировать матрицы размерности N с практически произвольным типом элементов.

Средства языка, поддерживающие механизмы абстракции, рассматриваются в контексте рассмотрения этих механизмов. В Части III материал излагается не так, как в Части II: я больше не предполагаю, что читатель так или иначе знаком с этим материалом.

1.1.4. Стандартная библиотека

Главы, посвященные стандартной библиотеке, носят менее учебный характер, чем главы про средства языка. Я полагаю, что их можно читать в любом порядке, и что они носят справочный характер:

Гдава 30 *Обзор стандартной библиотеки* — перечисляются основные заголовочные файлы, а также представляются средства языка и средства, поддерживающие диагностику, такие как **exception** и **system_error**.

Глава 31 *STL контейнеры* — рассматриваются контейнеры на базе фундаментального набора итераторов, контейнеров и алгоритмов, традиционно именуемых как STL. Включают **vector**, **map** и **unordered_set**.

Глава 32 *STL алгоритмы* — рассматриваются алгоритмы из набора STL, включая **find()**, **sort()** и **merge()**.

Глава 33 *STL итераторы* — представлены итераторы и иные средства из STL, включая **reverse_iterator**, **move_iterator** и **function**.

Глава 34 *Память и ресурсы* — представлены компоненты библиотеки, имеющие отношение к работе с памятью и управлению ресурсами, такие как **array**, **bitset**, **pair**, **tuple**, **unique_ptr**, **shared_ptr**, аллокаторы, а также интерфейс к сборщику мусора.

Глава 35 *Прочие компоненты библиотеки* — представлены небольшие компоненты библиотеки, работающие со временем и типами.

Глава 36 *Строки* — изучается тип **string**, включая профили символов, являющиеся базисом для применения различных наборов символов.

Глава 37 *Регулярные выражения* — описывается синтаксис регулярных выражений, а также способы их использования для поиска строковых соответствий, включая **string_match()** для выявления полного совпадения, **regex_search()** для поиска образцов в тексте, **regex_replace()** для замены и **regex_iterator** для общего прохода в потоке символов.

Глава 38 *Потоки ввода/вывода* — документируют компоненты библиотеки, относящиеся ко вводу/выводу. Описывается форматированный и неформатированный ввод и вывод, обработку ошибочных ситуаций, а также буферизацию.

Глава 39 *Локализация* — описывает класс **locale** и различные **facet**, отражающие страновые различия в наборах символов, форматировании числовых величин, форматировании дат и времени, и т.д.

Глава 40 *Вычисления* — описываются средства для численных расчетов (такие как **complex**, **valarray**, случайные числа и обобщенные численные алгоритмы).

Глава 41 *Параллелизм* — представлены модель памяти C++ и средства для реализации параллельных вычислений без блокировок.

Глава 42 *Потоки и задачи* — представлены классы для реализации параллельных вычислений с потоками и блокировками (такие как **thread**, **timed_mutex**, **lock_guard**, **try_lock** и поддержка параллелизма на базе задач (**future** и **async()**).

Глава 43 *Стандартная библиотека языка С* — документируется стандартная библиотека языка С (включая **printf()** и **clock()**) как часть стандартной библиотеки C++.

Глава 44 *Вопросы совместимости* — рассматривается взаимоотношение языков С и C++, а также между Стандартом C++ (ISO C++) и предыдущими версиями C++.

1.1.5. Примеры и ссылки

Настоящая книга больше ориентирована на организацию программ, чем на разработку алгоритмов. Как следствие, я сознательно избегаю слишком умных и труднопонимаемых алгоритмов. Тривиальный алгоритм чаще всего помогает лучше уяснить какой-

либо аспект языка или структуру программы. Например, я применяю сортировку методом пузырька там, где на практике следует использовать что-либо вроде алгоритма быстрой сортировки. Часто я предлагаю переделать рассматриваемую программу под более совершенный алгоритм в качестве самостоятельного упражнения. Кроме того, нужно отметить, что на практике почти всегда лучше вызвать соответствующую библиотечную функцию, чем использовать код, рассматриваемый в данной книге в качестве иллюстрации.

Учебные примеры дают, конечно, искаженный взгляд на реальный процесс разработки программ. Из-за того, что в учебниках приходится все упрощать и разъяснять, в итоге не видны сложности, проистекающие из-за объемов реального практического (а не учебного) кода. Нет никакой иной возможности почувствовать, что есть на самом деле практическое программирование, кроме как начать это делать. Но данная книга вынуждена концентрироваться на средствах языка и его стандартной библиотеки. Это составляет базовые основы, на которых строится любая программа. Мы также рассмотрим правила и технологии, широко применяемые при построении программ.

Выбор примеров отражает мой собственный опыт в проектировании и программировании компиляторов, фундаментальных библиотек и моделирующих программ. Сквозь эти примеры проступает мой интерес к системному программированию; демонстрируемые программы являются упрощенными версиями реальных программ. Такое упрощение позволяет сфокусироваться на средствах языка и проектных идеях и не потонуть в многочисленных деталях. В идеале пример должен четко отразить проектную идею, технику программирования, языковую конструкцию или средство библиотеки. В случае примеров, иллюстрирующих исключительно технические подробности языка, я использую имена переменных **x** и **y**, имена типов **A** и **B**, а функции называю **f()** и **g()**.

Всюду, где это возможно, средства языка и стандартной библиотеки рассматриваются в контексте их использования, а не в сухой манере технического руководства. Средства языка, которые я рассматриваю, и степень детализации такого рассмотрения отражают мой взгляд на то, как они должны использоваться для эффективного программирования на C++; я хочу подсказать идею о том, как следует использовать эти средства (часто в комбинации с другими средствами). Знание абсолютно всех средств языка и библиотеки не является необходимым и достаточным для написания хороших программ. На самом деле, одержимость в достижении абсолютного знания буквально каждой детали может привести к переусложненным и «слишком умным» программам. Что действительно нужно, так это понимание роли проектирования и технологий программирования вкупе с хорошим знанием затрагиваемой предметной области.

Я предполагаю, что читатели данной книги имеют доступ к online-источникам. А конечным арбитром для всех правил языка и стандартной библиотеки является собственно стандарт ISO C++ [C++,2011].

Ссылки на различные части данной книги даются в форме §2.3.4 (Глава 2, секция 3, подсекция 4) и § iso.5.3.1 (Стандарт ISO C++, §5.3.1). Наклонный шрифт используется в основном для фокусирования, в случае первого появления важных концепций (например, *полиморфизм*), а также в комментариях в коде примеров.

Ради спасения некоторого количества деревьев сотни примеров для настоящей книги переведены на страницы сайта www.stroustrup.com.

То, что рассматривается в настоящей книге, ограничивается чистым стандартом C++ [C++,2011]. Так что примеры из книги должны пойти на любом современном компиляторе C++, следующим стандарту. Большая часть примеров из книги проверялась на разных компиляторах. Понятно, что самые последние (по времени введения) новации языка не могут успешно компилироваться на всех компиляторах. Однако, я не вижу необходимости

отвлекаться от основной линии изложения и то и дело перечислять, на каких именно компиляторах что идет, а что нет. Все равно очень скоро все современные компиляторы будут в состоянии переваривать все средства языка и библиотеки (разработчики ведь не спят). В Главе 44 рассматриваются вопросы, что делать, если используются старые компиляторы и что делать с кодом, написанным под компиляторы с языка С.

Я полагаюсь на средства С++11 всюду, где считаю это необходимым и уместным. Например, я предпочитаю инициализаторы в {}-стиле и применяю **using** для синонимов типов. Иногда это может озадачить тех, кто сильно прирос к старым версиям языка, но это будет стимулировать их для пересмотра знаний. С другой стороны, я не сторонник применения исключительно новых черт языка просто потому, что они новые; я хочу использовать новые средства, чтобы точнее и яснее отразить идеи, которые могли уже годами применяться в С++ и даже в С.

Очевидно, что если вы жестко поставлены перед необходимость использовать старые компиляторы (возможно, это требование заказчика), то вам придется отказаться от новаций в С++. Только не нужно думать, что применение исключительно старых конструкций обеспечит вам надежность просто потому, что эти средства старые и проверенные. Параграф §44.2 суммирует все различия между С++98 и С++11.

1.2. Проект С++

Цель языка программирования — помощь в реализации идеи в виде формального кода. В этом отношении язык программирования выполняет две связанные функции: он является средством, с помощью которого программист задает действия, предназначенные для выполнения процессором компьютера; но также язык формирует набор понятий, которым программист оперирует, осмысливая то, что должно быть сделано. Первую функцию идеально поддерживает язык, близкий к машинному, так что все аспекты, имеющие отношение к машинным деталям, отрабатываются программистом просто и эффективно. Первоначально, при создании языка С имелась в виду концепция подобного рода. Вторая функция нуждается в языке, близким к контексту предметной области, чтобы алгоритм решения задачи можно было бы выразить прямо в понятиях этой предметной области. Те средства, которые были добавлены к языку С при создании языка С++, такие как проверка аргументов функции, **const**, классы, конструкторы и деструкторы, исключения и шаблоны, предназначались в первую очередь для достижения указанной цели. Таким образом, С++ базируется на двух идеях:

- идея поддержки прямого соответствия встроенных операций и типов машинной архитектуре для эффективного использования памяти и эффективного использования низкоуровневых операций,
- идея поддержки не слишком затратного и гибкого механизма абстракции, обеспечивающего возможность определять пользовательские типы в синтаксисе, близким ко встроенным типам, и с сопоставимой ээфективностью их использования.

Первоначально для этого были заимствованы идеи из языка Simula. Со временем они эволюционировали в более изощренный и разнообразный набор средств, гибких и эффективных одновременно. Был достигнут синтез стилей программирования, эффективных и элегантных.

Проект языка С++ всегда фокусировался на программировании, имеющем дело с фундаментальными вещами вроде памяти, абстракций, управлении ресурсами, способами ясного отражения структуры алгоритмов, обработке ошибок, модульности. Все это является фундаментальными концепциями для системных программистов и для програм-

мистов, разрабатывающих системы в условиях ограничения ресурсов и требующих достижения бескомпромиссной производительности.

Опираясь на определения классов, классовых иерархий и шаблонов, вы можете писать на C++ программы на гораздо более высоком уровне, чем программы, рассматриваемые в данной книге. В частности, язык C++ широко применяется в таких областях, как финансовые системы, игровые программы, а также программирование вычислительных задач (§1.4.5). При написании высокоуровневых программ существенную помощь оказывают библиотеки. Без готовых библиотек современное программирование было бы весьма затруднительным, можно сказать болезненным. Но это справедливо и для любого другого языка программирования общего назначения.

В общем, для начала резюмируем:

- C++ — это *язык программирования общего назначения* с уклоном в сторону *системного программирования*. Со временем все большее внимание стало уделяться мощи и гибкости механизмов абстракции.
- C++ — это *язык программирования общего назначения*, обеспечивающий непосредственное отражение машинной архитектуры в комбинации с удобными механизмами абстракции.
- C++ — это язык для разработки и использования *элегантных и эффективных абстракций*.

Под языком программирования общего назначения я понимаю язык, который может использоваться в самых разных областях. На практике язык C++ используется в невероятно широком диапазоне — от микроконтроллеров до тяжелых распределенных коммерческих систем. Но что важно, так это то, что C++ никогда не подстраивался специально ни для одной из областей применения. Ясно, что не существует языка программирования, идеального для всех типов приложений и для всех программистов. Но идеалом для C++ служит стремление к охвату возможно более широкого набора предметных областей и типов применения.

Под системным программированием я понимаю программирование, непосредственно использующее машинные ресурсы, имеющее жесткие ограничения по использованию таких ресурсов, или тесно взаимодействующее с таким кодом. Например, это реализация инфраструктурных компонентов, таких как драйверы, сетевые стеки, виртуальные машины, операционные системы, среды программирования и фундаментальные библиотеки. Подчеркну, что такое смещение в сторону системного программирование является осознанным выбором не допускать упрощений в надежде сделать язык более простым и удобным для некоторых прикладных областей.

В то же время язык C++ предоставляет программисту широкий выбор: можно совершенно не беспокоиться об особенностях машинной архитектуры, можно использовать дорогие абстракции (все объекты только в свободной памяти, все функции виртуальные), можно использовать неэлегантные стили, обремененные «сверхабстракциями», или вообще не использовать абстракций. Это, однако, можно делать во многих других языках программирования, так что это не какая-то особая заслуга C++.

В книге *Design and Evolution of C++* [Stroustrup,1994] идеи и принципы, заложенные в проекте по разработке C++, рассматриваются с большей детальностью и полнотой. Но здесь мне хотелось бы отметить два момента:

- *Не оставлять зазора для более низкоуровневого программирования по сравнению с C++* (в редких случаях можно воспользоваться ассемблерными вставками) — иначе иной язык стал бы языком системного программирования.

- *Если не используются какие-либо средства C++, то за них и ничего не платится.* В противном случае нашлись бы отдельные программисты, которые писали бы более эффективный «ручной код», а более многочисленные программисты следовали бы за ними. Поэтому все средства C++ работают так, чтобы зря не пропадал ни один байт памяти и ни один машинный цикл. Это известно как принцип *ноль дополнительных затрат*.

Это драконовские принципы, но они крайне важны в некоторых (не во всех, конечно) контекстах. Они часто помогают создавать средства, более элегантные и более эффективные чем то, что можно было заранее предвидеть. Ярчайшим примером служит библиотека STL (§4.1.1, §4.4, §4.5, Глава 31, Глава 32, Глава 33). Эти принципы были весьма нужны в наших усилиях поднять уровень программирования как можно выше.

1.2.1. Стили программирования

Средства языка программирования осуществляют поддержку того или иного стиля программирования. Не следует рассматривать индивидуальные свойства языка, как средство для решения задачи. Нужно рассматривать их в качестве строительных кирпичей, в комбинации которых лежит ключ к успеху.

Вот идеал, к которому следует стремиться:

- Отражайте идеи непосредственно в коде.
- Независимые идеи отражайте в независимых частях кода.
- Отражайте зависимости между идеями в самом коде.
- Свободно комбинируйте идеи в коде — везде, где это имеет смысл.
- Выражайте простые идеи просто.

Многие люди могли бы подписаться под этими идеалами, а вот языки для поддержки таких идеалов отличаются друг от друга радикальным образом. Фундаментальной причиной такого отличия служит разный опыт людей, разрабатывавших языки, отражающий разные вкусы, нужды и историю. Ответ C++ на перечисленные идеалы кроется в его истории: он создавался как язык системного программирования (в ранние годы языков C и BCPL, [Richards,1980]), а также в стремлении справиться со сложностью посредством введения абстракций (восходит к языку Simula).

Язык C++ непосредственно поддерживает четыре стиля программирования:

- Процедурное программирование
- Абстракция данных
- Объектно-ориентированное программирование
- Обобщенное программирование

Но ключевой подход состоит в эффективном комбинировании этих стилей: это для большинства нетривиальных задач приводит к наиболее читаемым, наименьшего размера и наибыстрейшим программам.

Как это всегда происходит с популярными терминами в компьютерном мире, появилось множество определений для перечисленных стилей программирования, зависящее от разных подходов и ниш в индустрии программирования и в академической (университетской) среде. Например, вместо термина «стиль программирования» иногда говорят о «технике программирования» или о «парадигме». Я, все же, предпочитаю использовать «стиль программирования», как более конкретный и менее претенциозный.

Моим идеалом служит язык, средства которого в различных комбинациях позволяют поддерживать самые разные стили программирования и широкий набор технологий создания программ.

- *Процедурное программирование*: это программирование, фокусирующееся на обработке и структурах данных. Это то, что поддерживает язык С (а также Algol, Fortran и множество других языков). Язык С++ вносит здесь свой вклад — это и встроенные типы, операции, операторы, функции, **struct**, **union** и т.д. С минимальными исключениями язык С является подмножеством С++. По сравнению с С, язык С++ обеспечивает большую поддержку процедурного программирования, опираясь на дополнительные языковые конструкции и на более гибкую и контролируемую систему типов.
- *Абстракция данных*: это программирование, фокусирующееся на проектировании интерфейсов, упрятывании деталей реализации вообще и представлений в частности. С++ поддерживает конкретные и абстрактные классы. Такие средства определения классов, как закрытые части реализации, конструкторы и деструкторы, и связанные с ними операции напрямую поддерживают абстракцию данных. Абстрактные классы используются для полного сокрытия данных.
- *Объектно-ориентированное программирование*: это программирование, фокусирующееся на проектировании, реализации и использовании классовых иерархий. Помимо того, что С++ позволяет создавать иерархии классов, он имеет средства для навигации по классовыи иерархиям, а также позволяет удобным образом определять классы на основе уже существующих классов. Иерархии обеспечивают полиморфизм времени исполнения (§20.3.2, §21.2) и инкапсуляцию (§20.4, §20.5).
- *Обобщенное программирование*: это программирование, фокусирующееся на проектировании, реализации и использовании общих алгоритмов. Здесь слово общих означает, что алгоритмы могут работать с широким набором типов данных, не противоречащих требованиям алгоритмов к аргументам. Шаблоны языка С++ являются основным средством этого языка, предназначенным для поддержки обобщенного программирования. Шаблоны обеспечивают (на этапе компиляции) параметрический полиморфизм.

Все, что в языке помогает увеличить гибкость и эффективность классов, увеличивает и поддержку всех перечисленных стилей программирования. В общем, можно сказать, что С++ является языком, *ориентированным на классы*.

Все перечисленные стили языка учитывались в процессе эволюционировавшей разработки С++. Попытка искуственно ограничиваться на каком-либо одном стиле является ошибочной: кроме как, разве, в игрушечных примерах, такое искусственное ограничение в общем случае приводит к напрасно потраченным усилиям по проектированию программного обеспечение, которое при этом получается, скорее всего, негибким, с неважной производительностью или же с затрудненной поддержкой, и с другими подобными «достоинствами».

Меня коробит, когда кто-то характеризует С++ сквозь призму одного из этих стилей (например, что С++ это язык объектно-ориентированного программирования) или же говорит о гибридной/смешанной парадигме, предполагая, что язык с большими ограничениями мог бы оказаться более предпочтительным. Такие высказывания умаляют ту роль, которую каждый из стилей внес в окончательный синтез, или вообще отрицают преимущества синтеза. Каждый из стилей не является альтернативным: каждый вносит свою лепту в результирующий выразительный и эффективный стиль программирования; С++ обеспечивает прямую поддержку использования всех стилей в их комбинации.

С самого начала С++ был нацелен на синтез программных стилей и стилей проектирования. Даже самый первый опубликованный вариант [Stroustrup,1982] содержит приме-

ры, использующие различные стили в комбинации, и предоставляет средства для такого комбинирования:

- *Классы* поддерживают все перечисленные стили; все они опираются на выражение идей пользователем посредством пользовательских типов и объектов пользовательских типов.
- *Public/private* контроль доступа поддерживает абстракцию данных и объектно-ориентированное программирование, проводя четкую грань между интерфейсом и реализацией.
- *Функции-члены*, *конструкторы*, *деструкторы* и *определяемое пользователем присваивание* формируют четкий пользовательский доступ к объектам, что требуется для абстракции данных и объектно-ориентированного программирования. Они также обеспечивают унифицированный синтаксис, необходимый для обобщенного программирования. Более общих форм перегрузки пришлось ждать вплоть до 1984 года и более однородных форм инициализации — вплоть до 2010 года.
- *Объявления функций* реализуют статическую проверку интерфейсов к функциям-членам, как и к обычным функциям, что необходимо всем стилям программирования. Это необходимо для перегрузки.
- *Обобщенные функции* и *параметризованные типы* поддерживают обобщенное программирование. Шаблоны пришлось ждать до 1988 года.
- *Базовые и производные классы* формируют основание для объектно-ориентированного программирования и некоторых форм абстракции данных. Виртуальных функций пришлось ждать до 1983 года.
- *Встраивание* (*inlining*) позволяет использовать перечисленные средства достаточно эффективным образом ради системного программирования, встраиваемых систем и компактных библиотек.

Все ранние средства языка C++ скорее обеспечивали базовый вариант механизмов абстракции, чем разрозненные стили. Современный C++ предоставляет лучшую поддержку проектированию и программированию на базе легковесных абстракций, а помощь в написании элегантного и эффективного кода была в нем от самого начала. Все новации в C++ с 1981 года обеспечили лучшую поддержку синтеза стилей и их интеграции.

Любой фундаментальный объект в языке C++ однозначно идентифицируется тем, что он имеет точное положение в памяти и может быть разграничен от других объектов с (потенциально) одинаковыми значениями по их адресу. Выражения языка, обозначающие такие объекты, принято обобщенно называть термином *lvalue* (§6.4). Однако, в языке C++ с самых ранних этапов были объекты, не имеющие такой идентификации (объекты, для которых адрес нельзя было сохранить на определенное время). В C++11 соответствующее понятие *rvalue* развилось в понятие значения, перемещение которого можно выполнить весьма дешево (§3.3.2, §6.4.1, §7.7.2). Такие объекты лежат в основе технологии, напоминающей те, что имеются в функциональном программировании. Это чудесным образом дополняет потенциальные приемы программировании и языковые средства (например, лямбда-выражения), первоначально предназначенные для обобщенного программирования. Это также позволяет решить классическую задачу о «простых абстрактных типах данных», таких как задача легко и эффективно вернуть большую матрицу из операции (например, matrix +).

С самых ранних дней существования языка C++ значительное внимание уделялось управлению ресурсами. Здесь идеальным представлялось (и сейчас представляется), чтобы управление ресурсами было:

- простым (для реализаторов, но особенно для пользователей),
- общим (ресурс вообще — это то, что сначала захватывается, используется, а потом возвращается),
- эффективным (в соответствии с принципом нулевых накладных расходов; §1.2),
- совершенным (никаких утечек не допускается), и
- статически надежным по типам.

Многие важные классы стандартной библиотеки C++, такие как **vector**, **string**, **thread**, **mutex**, **unique_ptr**, **fstream** и **regex** являются дескрипторами ресурсов. В других библиотеках, как фундаментальных, так и прикладных, тоже встречаются подобного рода классы, например, Matrix или Widget (их названия говорят сами за себя). Первый шаг к поддержке понятия дескриптора ресурсов был сделан в самом первом черновике языка «С с классами» посредством введения конструкторов и деструкторов. Вскоре это было подкреплено механизмом контроля копирования, что достигалось определением пользовательских операций копирования, равно как и копирующими конструкторами. Наконец, в C++11 этот процесс завершился введением перемещающих конструкторов и перемещающего присваивания (§3.3), которые реализуют дешевое перемещение потенциально больших объектов из одной области видимости в другую (§3.3.2), а также путем управления временем жизни полиморфных и разделяемых объектов (§5.2.1).

Средства управления ресурсами могут опираться и на другие абстракции, не являющимися непосредственно дескрипторами ресурсов. Любой класс, определяющий и поддерживающий инварианты, полагается на подмножество общих средств языка C++.

1.2.2. Проверка типов

Связь между языком, на котором мы думаем/программируем, и теми решениями проблем, которые мы можем себе представить, весьма тесная. По этой причине ограничения языка ради понижения вероятности программистских ошибок опасны. Язык снабжает программиста набором концептуальных инструментов; если набор неадекватен поставленной задаче, он будет отвергнут. Хороший дизайн и отсутствие ошибок не могут гарантироваться простым исключением тех или иных средств языка. На самом деле, средства языка и система типов помогают программисту четко и кратко отражать проект решения (дизайн) непосредственно в коде программы.

Понятие статических типов и проверка типов на этапе компиляции являются ключом к эффективному использованию C++. Следуя языку Simula, в языке C++ проектирование пользовательских типов с интерфейсами, проверяемыми на этапе компиляции, приводит к высокой выразительности языка. Система типов языка C++ является нетривиально расширяемой (Глава 3, Глава 16, Глава 18, Глава 19, Глава 21, Глава 23, Глава 28, Глава 29) и одинаково нацеленной как на встроенные типы, так и на типы, определяемые пользователем.

В C++ проверка типов и упрятывание данных полагаются на анализ программы во время компиляции для предотвращения некорректного обращения с данными и их повреждения. Здесь нет никакого режима секретности или защиты от намеренного нарушения правил: C++ защищает от инцидентов, а не от сознательного взлома. Эти механизмы C++ не приносят никаких дополнительных накладных расходов ни в отношении производительности, ни в отношении излишнего расхода памяти.

Система статических типов языка C++ весьма гибкая, и применение простых пользовательских типов предполагает мельчайшую дополнительную нагрузку, а иногда и вообще обходится без таковой. Стиль программирования с обильным набором типов дела-

ет код более понятным, читаемым, а также помогает анализировать и поддерживать код. Тривиальная система типов допускает лишь тривиальный анализ, в то время как богатая система типов позволяет нетривиальным образом обнаруживать ошибки и выполнять эффективную оптимизацию кода. Компиляторы и другие инструменты разработки поддерживают анализ типов [Stroustrup,2012].

Поддержка большей части C в качестве подмножества и сохранение прямой поддержки аппаратуры ради наиболее требовательных к эффективности системных задач подразумевает необходимость в неполной реализации статической системы типов. Я-то сам всегда был сторонником всеобъемлющей типизации и типовой надежности. Я полностью согласен с высказыванием Денниса Ритчи о том, что «C — это язык со строгой типизацией, но с нестрогой ее проверкой». В то же время, язык Simula является одновременно и гибким, и надежным по типам. В то время, как я начинал работу над C++, он мне представлялся скорее как «Algol68 с классами», чем как «C с классами». Однако ж, список серьезных возражений против базирования работы на Algol68 [Woodward,1974] весьма длинный. В итоге, идеальная и предельно надежная система типов — это то, к чему C++ стремится по-возможности (но полностью не достигает). Программисты на C++, особенно разработчики библиотек, не должны опускать руки в этом отношении и всегда должны стремиться поддерживать недостижимый идеал: по мере развития язык C++ приобретал новые черты, а также компоненты стандартной библиотеки, которые сильно помогают такому стремлению. Если не касаться принципиально низкоуровневого кода, который следует четко изолировать посредством введения безопасных по типу интерфейсов, то остальной код, взаимодействующий с операционной системой в соответствии со стандартными интерфейсами, а также реализации фундаментальных абстракций (вроде типов **vector** или **string**), практически не нуждаются в кодировании со слабой проверкой типов.

1.2.3. Совместимость с C

C++ разрабатывался на базе языка C и, с небольшими исключениями, до сих пор включает C как подмножество. Основной причиной опоры на C было стремление воспользоваться успешно наработанным множеством низкоуровневых средств этого языка, а также желанием не вступать в противоречия с многочисленным техническим сообществом. Большое значение уделялось совместимости с языком C [Koenig,1989] [Stroustrup,1994] (Глава 44); пришлось, к сожалению, почистить синтаксис языка C. Далее, языки C и C++ продолжали развиваться параллельно, что все время требовало постоянного внимания [Stroustrup, 2002]. Ведь наличие двух комитетов, присматривающих за необходимостью соблюдения совместимости «по максимуму возможного», не является надежной гарантией, так как всегда имеют место разногласия по вопросам об уровне совместимости, по взглядам на то, что такое хороший стиль программирования и что требуется для его поддержки. Одни лишь усилия по организации взаимодействия между комитетами — уже большая работа.

Стопроцентная совместимость никогда не рассматривалась при разработке C++, ибо она повлекла бы за собой компрометацию надежной системы типов и препятствовала бы гладкой интеграции встроенных и пользовательских типов. Тем не менее, определение C++ все время пересматривалось, чтобы уровень несовместимости постепенно уменьшался; теперь C++ более совместим с C, чем это было изначально. В стандарт C++98 были включены многие детали из C89 (§44.3.1). По мере эволюции языка C от C89 [C,1990] к C99 [C,1999], язык C++ впитывал все новшества, кроме VLA (variable-length arrays — массивы переменной длины) и назначающих инициализаторов (designated initializers). Средства языка C для низкоуровневого системного программирования были сохранены

и усилены, например, смотри встраивание (§3.2.1.1, §12.1.5, §16.2.8) или **constexpr** (§2.2.3, §10.4, §12.1.6).

С другой стороны, современный вариант языка С многое заимствовал из С++ (например, **const**, прототипы функций и встраивание; [Stroustrup,2002]).

Определение С++ проверялось на предмет того, чтобы одинаково легальные конструкции в С и С++ имели также и одинаковый смысл в обоих языках (§44.3).

Первоначальным посылом при разработке языка С было стремление заменить язык ассемблера в самых требовательных системных задачах. При разработке С++ мы стремились не испортить это достижение. Различие в С и С++ лежит преимущественно в степени сфокусированности на типах. Язык С выразителен и многое позволяет. А язык С++, посредством системы типов, выразителен в не меньшей (если не в большей) степени, причем не за счет потери эффективности. Детальное знание С не является обязательным при изучении С++. Многие трюки и приемы программирования на С на языке С++ рассматриваются как необязательные. Например, явное приведение типов в программах на С++ применяется реже, чем в программах на С (§1.3.3). Но в то же время, хорошие программы на С имеют тенденцию оставаться корректными и для С++. К примеру, таковыми являются все программы на С из книги «The C Programming Language, Second Edition» [Kernighan,1988]. Опыт в любом статически типизируемом языке будет в помощь при изучении С++.

1.2.4. Язык, библиотеки и операционные системы

В языке С++ фундаментальные (встроенные) типы, операции и операторы — этот тот набор, с которым аппаратура компьютера работает напрямую: числа, символы и адреса. В языке С++ нет встроенных высокоуровневых типов, как нет и примитивных операций высокого уровня. Например, в языке С++ нет матричного типа с операцией инверсии или строкового типа с операцией конкатенации. Если пользователю нужны такие типы, он может сам определить их средствами языка С++. Фактически, разработка общеупотребительных или крайне узких специфических типов — вот наиболее фундаментальное поле деятельности программиста на С++. Хорошо спроектированные пользовательские типы отличаются от встроенных типов только способом их определения, а не тем, как они используются. Стандартная библиотека языка С++ (Глава 4, Глава 5, Глава 30, Глава 31 и т.д.) предоставляет великое множество примеров таких типов и вариантов их применения. С пользовательской точки зрения разница между встроенными типами и пользовательскими типами весьма мала. За исключением малозначительных отклонений (чаще всего по историческим причинам) сама библиотека написана на С++. Разработка стандартной библиотеки С++ на самом С++ — это жесткая проверка системы типов языка С++ и его механизмов абстракции: они должны быть достаточно мощными (таковыми и являются) и эффективными для практически всех критически важных системных задач. Все это позволяет использовать С++ для разработки крупномасштабных систем, в которых одни слои кода базируются на других слоях кода, реализуя разные уровни абстракции.

В языке С++ отсутствуют средства, которые нагружают память и процессор компьютера излишним образом. Например, конструкции, которые помимо целевой информации содержат еще и ряд вспомогательных данных (дополнительных, так сказать «для порядка»), в языке С++ не встречаются, так что если объявлена структура, состоящая из двух 16-битовых величин, то она и в самом деле может быть размещена в 32-битовом регистре. За исключением операций **new**, **delete**, **typeid**, **dynamic_cast** и **throw** и **try**-блока все индивидуальные выражения и операторы не требуют никакой поддержки на этапе выполнения. Это

крайне важно для встроенных приложений, а также приложений, сильно нагружающих аппаратуру компьютера. Механизмы абстракции C++ не препятствуют применению C++ в программировании подобного рода приложений. И программисты при этом избавлены от необходимости массированного применения низкоуровневых (и часто приводящих к ошибкам) средств языка.

Язык C++ спроектирован так, что его можно использовать в традиционном контексте — в той же самой среде программирования, что и в случае языка C на UNIX [UNIX,1985]. Но, по счастью, C++ никак не ограничивается средой разработки ОС UNIX — просто UNIX и C служат исходным ориентиром, наглядно и предметно показывающим связь между языком, библиотеками, компиляторами, компоновщиками (linkers), средой выполнения и т.д. Столь минимальная и четко очерченная модель позволяет применять C++ практически на любой платформе (операционной системе и аппаратуре). Конечно, лучше это делать на платформах с богатыми средствами поддержки в виде динамической загрузки, инкрементальной компиляции и другими средствами, не связанными непосредственно с языком как таковым.

Разумеется, невозможно абсолютно весь код сделать простым и легким для чтения. C++ содержит средства, позволяющие напрямую манипулировать аппаратурой с высокой эффективностью без всяких оглядок на простоту восприятия кода и легкость его написания. Но он также содержит и средства, позволяющие спрятать такой код за элегантными и безопасными интерфейсами.

Естественно, когда C++ используется для разработки больших систем, над проектом обычно работают разные группы программистов. C++ здесь помогает посредством модульности и сильно типизированных интерфейсов. По мере разрастания размера программы проблема перемещается из чисто языковой сферы в область удобства и мощности инструментов проектирования, программирования и поддержки.

Настоящая книга рассматривает приемы общего назначения, примеры разработки полезных типов общего назначения, библиотек и т.д. Это поможет всем программистам, в том числе и тем, кто разрабатывает большие программы, потому что рассмотренные нами примеры вполне могут служить составными частями больших систем.

Я рассматриваю реализацию и использование компонентов стандартной библиотеки, таких как **vector**, в качестве примеров. Это служит знакомством с компонентами библиотеки и концепциями, на которых они базируются. Эти примеры показывают, как программисты могли бы проектировать и реализовывать свои собственные библиотеки. Хочу сразу отметить, что в случае, если компонент стандартной библиотеки отвечает вашим намерениям, то лучше использовать именно его, а не строить собственные версии. Даже если ваш собственный компонент выйдет в чем-то лучше, он все равно будет менее практически оттестированным, менее применимым в иных контекстах и менее знакомым прочим программистам. В крупномасштабной временной перспективе компоненты стандартной библиотеки реально снизят ваши совокупные затраты на разработку, модификацию и поддержку программных систем.

Можно было бы предположить, что использование более детализированных структур данных приведет к увеличению размера исходного текста программы. Но это не так. В целом, C++-программа, объявляющая и использующая классы, объявляющая все функциональные интерфейсы и т.д., оказывается даже короче, чем эквивалентная C-программа, не использующая перечисленные средства. Стандартная библиотека языка C++ помогает укорачивать текст программы на C++ еще сильнее.

C++ поддерживает системное программирование. Это подразумевает, что код на C++ способен осуществлять взаимодействие с программами, написанными на иных языках

и присутствующих в системе. Невозможно все программное обеспечение написать на единственном языке. С самого начала язык C++ был способен взаимодействовать с C, ассемблером и Фортраном. Это означает, что функции, написанные на C++, C, ассемблере и Фортране, могут вызывать функции, написанные на отличном языке без каких-либо излишеств в плане передачи или преобразовании структур данных между ними.

C++ разрабатывался в расчете на единственное адресное пространство. Использование множественных процессоров и множества адресных пространств возможно посредством поддержки от операционной системы. В частности, системная поддержка осуществляется в том числе и посредством того или иного командного языка и соответствующего интерпретатора. На UNIX-системах речь идет о языке оболочки, но в иных системах может использоваться какой-либо иной командный (скриптовальный) язык. С самого начала предполагалось, что C++ будет использоваться в больших мультиязыковых системах.

1.3. Изучение C++

Никакой язык программирования не может быть во всех отношениях совершенством. Ему и не требуется быть таковым, чтобы оказаться хорошим инструментом для создания отличных программных систем. Более того, язык общего назначения не может быть идеальным для всех типов программ, для которых его будут использовать. Что идеально подходит для некоторого класса задач, может оказаться далеко не идеальным в иных случаях. Язык C++ разрабатывался как универсальный язык, применимый для широкого круга задач так, чтобы можно было с его помощью элегантно выражать широкий круг идей и концепций.

Однако, не все можно выразить одними лишь средствами языка. Это даже и не является самоцелью. Средства языка должны быть достаточными для поддержки множества стилей и методик программирования. Следовательно, изучение языка должно фокусироваться на овладении стилями, органичными и естественными для данного языка, а не на усилиях в попытке глубоко разобраться буквально в каждой детали каждого языкового средства. Самым важным является практическое создание программ; понимание языка программирования — это не чисто интеллектуальное упражнение. Нужно практическое воплощение идей.

В практическом программировании не так важно детально разбираться в самых заковыристых вопросах и мало пользы от того, что в процесс вовлекается как можно больше языковых средств. Каждое такое средство обретает истинный смысл лишь в контексте технологии, используемой для решения задачи. Поэтому, когда вы будете читать последующие главы книги, помните, что настоящей сверхзадачей является достижение понимания, как следует использовать средства языка и библиотеки в комплексе для достижения хорошего стиля программирования относительно надежно представленного проекта решения задачи.

Не существует реально полезных программных систем, построенных лишь в терминах собственно языка. Мы применяем библиотеки как для упрощения программирования, так и для того, чтобы повысить качество целевой системы. Библиотеки помогают повысить простоту поддержки, переносимости и производительности. Самые фундаментальные концепции задачи представляются адекватными абстракциями (классами, шаблонами и классовыми иерархиями), сосредоточенными в библиотеках. Многие из фундаментальных концепций представлены в стандартной библиотеке языка C++. Ее изучение является интегральной частью изучения C++. Сама она является квинтэссенцией знаний о том, как лучше всего применять язык C++.

C++ широко используется в целях обучения, а также для исследований. Это часто поражает тех людей, которые считают C++ не самым подходящим языком для вышеозначенных целей. Тем не менее, язык C++:

- Достаточно ясный для успешного обучения основам проектирования и типовым приемам программирования.
- Достаточно мощный для обучения продвинутым концепциям и технологиям.
- Достаточно эффективный и гибкий для применения в требовательных к эффективности проектам.
- Достаточно практичный, чтобы изученный материал мог с успехом быть перенесен из академической среды на идустриальную почву.
- Достаточно широко представленный на разных операционных и аппаратных платформах, на которых нужно обеспечить функционирование разрабатываемых программных систем.

C++ — это язык, с которым можно расти (а не стоять на месте).

В процессе изучения C++ следует четко фокусироваться на фундаментальных концепциях (таких как безопасность по типам, управление ресурсами или инварианты) и технологиях программирования (управление ресурсами на основе локальных объектов или применение итераторов в алгоритмах) и не потеряться в многочисленных мелких деталях языка. Конечной вашей целью должно быть овладение мастерством, позволяющим быть более эффективным в проектировании и реализации новых систем и в поддержании старых. Для этого достаточно разобраться с технологиями проектирования и программирования. Знание абсолютно всех технических деталей придет со временем.

C++ базируется на строгой проверке типов, а большинство приемов программирования сосредотачивается на достижении высокого уровня абстракции и прямого представления идей из предметной области решаемой задачи. Обычно этого можно достичь без компромисса с эффективностью по сравнению с применением низкоуровневых приемов программирования. Для этого программисты, переходящие на C++ с других языков или с более ранних версий C++, должны проникнуться присущими языку C++ идиомами и технологиями.

Бездумное копирование идиом и практик из иных языков на почву C++ обычно приводит к нелепым по форме и плохо функционирующим программам, которые вдобавок еще и трудно сопровождать. В процессе написания такой программы программист будет постоянно огорашиваться сообщениями от компилятора, напоминающими ему о том, что это вовсе не его привычный по старым практикам язык. Никто вам не может запретить программировать на C++ в стиле языков Fortran, C, Lisp, Java и т.д., но это вряд ли принесет удовольствие, да и не будет эффективным и рациональным занятием. На самом деле, любой ваш прежний язык может стать рациональной отправной точкой, только нужно идеи трансформировать так, чтобы они соответствовали общей структуре и системе типов языка C++. Тогда все пойдет гладко и приведет к эффективной программе. Над базовой системой типов языка можно одержать лишь пиррову победу.

Теперь обратимся к непрекращающимся дебатам о том, стоит ли сначала изучить C перед тем, как заняться языком C++? Мое твердое мнение состоит в том, что лучше сразу переходить к C++. Язык C++ более выразительный и позволяет создавать более надежный и безопасный код, свободный от всяких низкоуровневых трюков. С последними будет проще ознакомиться после того, как вы изучите общее подмножество C и C++. Глава 44 специально предназначена для тех, кто переходит от C++ к C для того, чтобы, скажем, поддерживать ранее написанный код на C. Подробнее о том, как, по-моему, лучше всего знакомить новичков с C++, изложено в [Stroustrup,2008].

В реальности существует несколько независимо разработанных реализаций C++. Они сопровождаются массой инструментов, библиотек и графических сред. Для ознакомления с ними вам придется найти соответствующие учебники, руководства и великое множество online-ресурсов. Я настоятельно рекомендую вам в случае возникновения вопросов обращаться к таким ресурсам, желательно к каким-нибудь двум разным источникам по любому вопросу. Ведь у всякого на любую тему имеются разные мнения.

1.3.1. Программирование на C++

Вопрос «как писать хорошие программы на C++?» примерно эквивалентен вопросу «как писать хорошую прозу на английском языке?». На столь серьезные вопросы имеются «простые» ответы: «Нужно знать, что хочешь сказать» и «Нужно практиковаться и изучать лучшие образцы», правда следовать этим простым правилам не так уж и легко.

Идеал для программиста на C++ — как и для программиста на любом другом языке высокого уровня — выражать идеи (понятия, концепции ...) из проекта решения задачи непосредственно в коде:

1. Отражаем проектные идеи в коде.
2. Представляем взаимосвязи между идеями также с помощью конструкций языка непосредственно (иерархии, параметризация и владение).
3. Независимые идеи отображаем в коде независимо.
4. Простые вещи реализуем как можно проще.
5. По-возможности используем решения, имеющие надежную статическую проверку.
6. Стараемся максимально локализовывать информацию (избегаем глобальных переменных, минимизируем использование указателей).
7. Стараемся держать уровень абстракции адекватным, а не избыточным (не нужно все переобобщать сверх очевидных пределов и практической необходимости).

Более частные советы приводятся в §1.3.2.

1.3.2. Советы C++ — программистам

К настоящему времени многие люди программируют на C++ уже в течение одного-двух десятилетий. Часть из них (возможно, большая часть) вынуждена была использовать единственную среду и окружение, налагаемое ранними версиями компиляторов и библиотек первых поколений. Часто опытные программисты на C++ отслеживают ситуацию с появлением в языке новых средств как таковых, но от них ускользает их взаимосвязь, лежащая в основе новых приемов и технологий программирования. Другими словами, то, что вам ранее было неизвестно или казалось непрактичным, теперь вполне может оказаться самым практичным и эффективным. Для этого следует перетряхнуть основы.

Читайте книгу главу за главой. Если вам знаком материал главы, то вы с ней справитесь за минуты. Если же материал вам незнаком, то с ним следует ознакомиться. Вряд ли любой программист знаком абсолютно со всеми деталями языка. Я сам, например, многому научился, пока писал эту книгу. Любому, кто хочет всесторонне изучить и применять C++, лучше сразу ознакомиться с широкоформатной перспективой, вносящей порядок и направление, позволяющей в дальнейшем использовать те или иные средства и возможности языка, отталкиваясь от деталей решаемой задачи. Настоящая книга посредством своей организации и примеров раскрывает такую перспективу.

Воспользуйтесь новыми возможностями, предлагаемыми языком в стандарте C++11 для того, чтобы модернизировать ваши проекты и программы:

1. Применяйте конструкторы для формирования инвариантов (§2.4.3.2, §13.4, §17.2.1).
2. Используйте пару конструктор/деструктор для управления ресурсами (§5.2, §13.3).
3. Избегайте «голых» **new** и **delete** (§3.2.1.2, §11.2.1).
4. Применяйте контейнеры и алгоритмы, а не встроенные массивы и самопальный код (§4.4, §4.5, §7.4, Глава 32).
5. Предпочитайте стандартную библиотеку другим, мнеее стандартным вариантам (§1.2.4).
6. Используйте исключения вместо индикации ошибок условными значениями, чтобы обрабатывать ошибки не локально, а именно в том месте, где это более всего уместно (§2.4.3, §13.1).
7. Применяйте семантику перемещения, чтобы избежать копирования больших объектов (§3.3.2, §17.5.2).
8. Для ссылок на полиморфные объекты применяйте **unique_ptr** (§5.2.1).
9. Для ссылок на разделяемые объекты (объекты со многими владельцами) применяйте **shared_ptr** (§5.2.1).
10. Используйте шаблоны для поддержания надежности статической системы типов и для того, чтобы избежать излишнего раздувания классовых иерархий (§27.2).

Неплохо также ознакомиться с советами программистам на C и Java (§1.3.3, §1.3.4).

1.3.3. Советы C-программистам

Кажется, что чем лучше программист знает C и чем более он преуспел в программировании на этом языке, тем сложнее ему отказаться от привычки писать на C++ в стиле языка C, теряя при этом многие преимущества и дополнительные возможности. Пожалуйста, ознакомьтесь с Главой 44, в которой сравниваются языки C и C++. В результате:

1. Не рассматривайте C++ как C с некоторыми добавками. Можно, конечно, использовать C++ в таком ключе, но это не оптимально. Для того, чтобы выжать максимум из C++, придется пересмотреть привычные стили и приемы проектирования и реализации.
2. Не используйте C в C++-программах. Это не только неоптимально, но и чревато дополнительными ошибками.
3. Применяйте стандартную библиотеку в том числе и как образец новых стилей программирования. Обратите внимание на отличия этой библиотеки от стандартной библиотеки языка C (например, = вместо **strcpy()** для копирования и == вместо **strcmp()** для сравнения строк).
4. Макроподстановки почти никогда не нужны в программах на C++. Применяйте **const** (§7.5), **constexpr** (§2.2.3, §10.4), **enum** или **enum class** (§8.4) для констант, **inline** (§12.1.5), чтобы избежать расходов на вызов функции, **template** (§3.4, Глава 23) для определения семейств функций и типов, **namespace** (§2.4.2, §14.3.1), чтобы избежать коллизии имен.
5. Не объявляйте переменную до того момента, как она вам понадобится, и тут же инициализирйте ее. Объявление допустимо всюду, где допустим оператор (§9.3), в инициализаторах **for**-операторов (§9.5), а также в условиях (§9.4.3).
6. Не применяйте **malloc()**. Операция **new** (§11.2) выполняет эту работу лучше, а вместо **realloc()** воспользуйтесь типом **vector** (§3.4.2). Не следует заменять **malloc()** и **free()** «голыми» **new** и **delete** (§3.2.1.2, §11.2.1).

7. Избегайте применять **void***, **union** и приведения типов во всех случаях, кроме как глубоко запрятанными в определениях некоторых функций и классов. Их применение ограничивает поддержку от системы типов языка, а также может внести проблемы в функционирование программы. В большинстве случаев использование приведения типов свидетельствует о проблемах с дизайном программы. Если уж без приведения типов не обойтись, используйте именованные варианты (например, **static_cast**; §11.5.2) для более четкой индикации ваших намерений.
8. Минимизируйте применение встроенных массивов и строк в С-стиле. Очень часто использование таких типов из стандартной библиотеки, как **string** (§4.2), **array** (§8.2.4) и **vector** (§4.4.1), приводит к программам, которые проще понять и проще поддерживать. В целом, нет никакого смысла самим воспроизводить то, что уже реализовано в стандартной библиотеке.
9. Избегайте арифметики указателей за исключением уж очень специализированного кода (например, менеджера памяти) и простого траверса массивов (например, **++p**).
10. Не стоит думать, что некий хитроумный код в С-стиле (не используются ни классы, ни шаблоны, ни исключения) является более эффективным чем более короткая альтернатива с применением средств стандартной библиотеки. Часто справедливо обратное.

Чтобы не противоречить правилам компоновки языка С, функция в С++ должна явно объявляться соответствующим образом (§15.2.5).

1.3.4. Советы Java-программистам

С++ и Java — это довольно разные языки со схожим синтаксисом. Их конечные цели сильно различаются, как и области применения. Java нельзя считать развитием языка С++ с целью делать все лучше и больше. Отличия существенные и не только в том, чтобы не забывать вызывать **delete** в случае динамического создания объектов из-за отсутствия сборщика мусора:

1. Не стоит пытаться имитировать Java-стиль на С++ — это неоптимально и для производительности, и для удобства поддержки кода.
2. Нужно активно использовать механизмы абстракции языка С++ (классы, шаблоны) и не откатываться к С-стилю из-за ложного чувства формальной близости синтаксиса.
3. Рассматривайте стандартную библиотеку и как средство обучения новым технологиям и стилям программирования.
4. Не пытайтесь на ходу изобретать уникальный базовый класс для всех своих классов (типа **Object** класс). В большинстве случаев в этом нет необходимости.
5. Минимизируйте использование переменных-указателей и переменных-ссылок; используйте локальные переменные и члены-данные классов (§3.2.1.2, §5.2, §16.3.4, §17.1).
6. Помните: переменные по умолчанию не являются ссылками.
7. Думайте об указателях как о С++-эквиваленте ссылкам языка Java (ссылки языка С++ более лимитированы, например, их нельзя реинициализировать).
8. Функция не является виртуальной по умолчанию. Не каждый класс предназначен для наследования.
9. Применяйте абстрактные классы как интерфейсы для классовых иерархий; избегайте базовых классов с членами-данными.

10. Применяйте управление ресурсами через локальные объекты всюду, где это возможно.
11. Используйте конструкторы для формирования классовых инвариантов (в противном случае генерируйте исключение).
12. Когда при уничтожении объекта нужна очистка ресурсов, делайте это в деструкторе. Не пытайтесь имитировать **finally** (в долгосрочной перспективе это потребует больше усилий, чем использование деструктора).
13. Избегайте «голых» **new** и **delete**; вместо этого применяйте контейнеры (**vector**, **string** или **map**) и классы-дескрипторы (**lock** или **unique_ptr**).
14. Для минимизации связей в программе применяйте глобальные функции (см., например, стандартные алгоритмы) и применяйте пространства имен (§2.4.2, Глава 14) для локализации области видимости имен глобальных функций.
15. Не используйте спецификаций исключений (кроме **noexcept**; §13.5.1.1).
16. Внутренние классы языка C++ не имеют доступа к объекту объемлющего класса.
17. В C++ имеется лишь самая минимальная поддержка для получения информации о типе на этапе исполнения: **dynamic_cast** и **typeid** (Глава 22). Сильнее полагайтесь на средства времени компиляции (полиморфизм времени компиляции; Глава 27, Глава 28).

Большая часть этих рекомендаций подойдет и для программистов на языке C#.

1.4. История C++

Я, можно сказать, изобрел C++, написал все ранние определения и выполнил самую первую реализацию. Я выбрал и сформулировал все основные критерии для разработки C++, спроектировал основные конструкции языка, разработал или помогал разрабатывать многие из ранних библиотек, а также был ответственным за рассмотрение и прием предложений по развитию C++ в комитете по стандартизации.

C++ был нацелен на то, чтобы обеспечить средства языка Simula для организации программ [Dahl,1970][Dahl,1972] вкупе с эффективностью языка C и его гибкостью, полезной в области системного программирования [Kernighan,1978][Kernighan,1988]. Simula является отправной точкой для механизмов абстракции в языке C++. Например, понятие класса (с наследованием и виртуальными функциями) было заимствовано из Simula. В то же время, шаблоны и исключения появились в C++ совсем из других источников.

Эволюция C++ шла параллельно практике его использования. Я тратил массу времени, выслушивая пожелания пользователей и разыскивая мнения опытных программистов. В частности, мои коллеги по AT&T Bell Laboratories внесли огромный вклад в процесс развития C++ в первое десятилетие.

В настоящем разделе дается очень краткий обзор этапов эволюции языка; в нем нет упоминания обо всех средствах языка и обо всех компонентах библиотеки. Также отсутствует какое-либо углубленное проникновение в детали. Если требуется больше информации, особенно по персоналиям, смотри [Stroustrup,1993][Stroustrup,2007] и [Stroustrup,1994]. Две мои статьи на конференции ACM History of Programming Languages и моя книга «Design and Evolution of C++» описывают процесс первоначальной разработки и дальнейшей эволюции C++ в деталях, а также там документировано влияние других языков программирования на C++.

Большинство документов, выпущеннх как часть стандарта ISO C++ доступны online [WG21]. В моих FAQ (online-ответы на вопросы) я стараюсь вести обсуждение средств

языка в связке с людьми, предлагающими уточнения и поправки к ним [Stroustrup,2010]. Язык C++ не является продуктом обезличенного анонимного комитета; это общее произведение множества преданных, опытных и много поработавших над данной темой людей.

1.4.1. Хронология

Работа, которая привела к языку C++, началась осенью 1979 года под названием «C с классами». Вот точная хронология:

1979 Работа над проектом «C с классами» стартовала. Первоначальный набор средств включал классы и наследование классов, контроль доступа посредством public/private, конструкторы и деструкторы, а также проверка объявлений функций с проверкой аргументов. Первая библиотека поддерживала невытесняющую многозадачность и генераторы случайных чисел.

1984 Проект «C с классами» получил новое имя C++. К этому моменту C++ приобрел виртуальные функции и перегрузку операций, ссылки и потоковый ввод/вывод, а также библиотеку комплексных чисел.

1985 Первый коммерческий выпуск C++ (14 октября). Библиотека содержала потоковый ввод/вывод, комплексные числа и невытесняющую многозадачность.

1985 Выход книги «The C++ Programming Language» (14 октября) [Stroustrup,1986].

1989 Книга «The Annotated C++ Reference Manual».

1991 Книга «The C++ Programming Language, Second Edition» [Stroustrup,1991], представляющая обобщенное программирование с применением шаблонов и обработку ошибок на базе исключений.

1997 Книга «The C++ Programming Language, Third Edition» [Stroustrup,1997], представляет ISO C++, включая пространства имен, dynamic_cast и множество усовершенствований шаблонов. К стандартной библиотеке присоединили STL-контейнеры и алгоритмы.

1998 Стандарт ISO C++.

2002 Начало работы над ревизией стандарта под условным названием C++0x.

2003 Выпуск пакета исправлений и дополнений для стандарта ISO C++ — новые компоненты стандартной библиотеки, такие как регулярные выражения, неупорядоченные контейнеры (хэш-таблицы).

2006 Технический документ ISO C++ посвящен вопросам детерминированности и технологий, относящихся ко встроенным системам.

2009 Проект C++0x наполнен окончательным набором новых средств: однородная инициализация, семантика перемещения, шаблоны с переменным числом параметров, лямбда-выражения, синонимы типов, модель памяти для параллельных вычислений и многое другое. В стандартную библиотеку добавлен ряд компонентов, включая потоки, блокировки и большинство компонентов из технического документа 2003 года.

2011 Стандарт ISO C++11 получил формальное одобрение.

2012 Появилась первая реализация стандарта C++11.

2012 Работа над будущими стандартами ISO C++ (под названиями C++14 и C++17) стартовала.

2013 Книга «The C++ Programming Language, Fourth Edition», описывающая C++11.

Во время разработки стандарта C++11 он был известен как C++0x. Да, мы были слишком оптимистичны относительно конечных сроков работы.

1.4.2. Ранний период

Первоначально я разрабатывал язык с целью использовать его для переноса сервисов ядра операционной системы UNIX на мультипроцессорные системы и локальные сети (сейчас это многоядерные системы и кластеры). Для этого мне потребовались эмуляции, управляемые событиями, и для чего идеально подходил язык Simula, если не брать в расчет проблемы с производительностью. Также мне нужен был прямой доступ к аппаратуре и нужно было обеспечить высокопроизводительные параллельные вычисления, для чего идеально подходил язык C, за исключением плохой проверки типов и слабой модульности. В результате мы добавили к C классы в стиле Simula и получился «C с классами», который мы интенсивно использовали во многих программных проектах, где требовалось минимальное время исполнения. На этом этапе в языке отсутствовали перегрузка операций, ссылки, виртуальные функции, шаблоны, исключения и огромная масса более мелких деталей [Stroustrup,1982]. Случай первого использования C++ за пределами исследовательской организации зафиксирован в июле 1983 года.

Имя C++ возникло у Рика Маскити (Rick Mascitti) и было мною использовано как замена названию «C с классами». Новое имя хорошо отражало эволюционную природу языка в его движении от языка C; операция ++ — это операция инкремента в языке C. Более короткое имя C+ символизирует синтаксическую ошибку. Имя D не было использовано потому, что оно не отражает факт расширения языка C, и к тому же оно уже было занято другими языками, претендующими на развитие языка C. Про иные интерпретации имени C++ смотри приложение к [Orwell,1949].

C++ разрабатывался так, чтобы я и мои друзья не испытывали потребность обращаться к ассемблеру, C, а также к иным популярным в то время языкам высокого уровня. Предполагалось, что на нем будет легче и приятнее писать хорошие программы. Не было никакого письменного проекта языка; проектирование, реализация и документирование шли параллельно. Не было и никакого комитета по разработке C++. Язык постепенно эволюционировал так, чтобы с его помощью можно было преодолевать разные практические проблемы, с которыми сталкивался я, мои друзья и коллеги.

1.4.2.1. Средства языка и средства библиотеки

Самый первый проект языка C++ (тогда называвшегося «C с классами») включал в себя объявление функций с проверкой аргументов и неявными приведениями типов, классы с public/private для разграничения интерфейса и реализации, наследование и конструкторы/деструкторы. Для примитивной параметризации я применял макросы. Это было в ходу в середине 1980-ых. Позже я представил набор языковых средств, способный поддерживать согласованный набор стилей программирования; см. §1.2.1. В ретроспективе я считаю червычайно важным появление конструкторов/деструкторов. Тут как раз кроется исток для современных технологий управления ресурсами (также требующих исключений) и других приемов, делающих код короче и понятнее. Если в те времена и были языки с множественными конструкторами, то мне они были решительно неизвестны. Деструкторы же были откровенной новацией языка C++.

Коммерческий выпуск C++ состоялся в октябре 1985 года. К тому времени я добавил встраивание функций (§12.1.5, §16.2.8), **const** (§2.2.3, §7.5, §16.2.9), перегрузку функций (§12.3), ссылки (§7.7), перегрузку операций (§3.2.1.1, Глава 18, Глава 19) и виртуальные функции (§3.2.3, §20.3.2). Из всех перечисленных средств полиморфизм времени исполнения на базе виртуальных функций вызывал наибольшие споры. Я никак не мог убедить людей из мира системного программирования в его пользе. Системные программисты с подозрением относились к косвенным вызовам функций, а люди с подготовкой в иных

языках программирования не могли поверить, что механизм виртуальных функций достаточно быстр для применения в системных задачах. Программисты из мира объектно-ориентированного программирования никак не могли привыкнуть к тому, что виртуальные функции используются для реализации выбора, выполняемого на этапе исполнения программы. Сопротивление виртуальным функциям основывалось также на том, что многие не верили, что можно получить более регулярный (и в то же время эффективный) код просто на базе средства языка. Программисты на C считали, что такого можно добиться лишь тщательным ручным программированием буквально каждой мелкой программной детали. Я же считал, что если можно получить от самого языка хоть какую-то автоматическую помощь, то это чрезвычайно важно при построении сложнейших программных систем (где и без того нагрузка на программиста запредельная).

Можно сказать, что большая часть развития проекта C++ выполнялась на настенных досках моих коллег. Обратная связь от Stu Feldman, Alexander Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy и Dennis Ritchie была просто бесценной.

Во второй половине 80-ых я продолжал добавлять новые средства к языку C++ под влиянием просьб со стороны пользователей. Наиболее существенные добавки включали шаблоны [Stroustrup,1988] и обработку исключений [Koenig,1990], которые на начало проекта по разработке C++ считались экспериментальными. Проектируя шаблоны, я разрывался между гибкостью, эффективностью и ранней проверкой типов. В то время никто не знал как совместимость все три эти требования, да еще и чтобы это конкурировало с эффективностью языка C в случае особо требовательных приложений. Я чувствовал, что нужно выбрать первые два из трех перечисленных свойств. Я и сейчас считаю, что выбор был сделан правильно, в то время как поиски проверки типов шаблонов продолжались [Gregor,2006], [Sutton,2011] и [Stroustrup,2012a]. Архитектура обработки ошибок нацелена на многоуровневое распространение исключений, на передачу аргументов обработчику и на интеграцию исключений с системой управления ресурсами на основе локальных объектов с деструкторами для автоматического освобождения ресурсов (§13.3).

Я обобщил механизм наследования классов до возможности иметь несколько базовых классов [Stroustrup,1987a]. Это получило название *множественного наследования* и рассматривалось как сложное и противоречивое. Этот механизм я считал менее важным, чем шаблоны или исключения. В то же время множественное наследование абстрактных классов (часто называемых *интерфейсами*) в настоящее время является обыденным в языках программирования со статической проверкой типов, а также в объектно-ориентированном программировании.

Язык C++ эволюционировал параллельно с некоторыми ключевыми средствами библиотеки, представленными в настоящей книге. Например, я разработал класс комплексных чисел [Stroustrup, 1984], классы **vector**, **stack** и классы потоков ввода/вывода [Stroustrup, 1985], а также механизм перегрузки операций. Совместно с Jonathan Shopiro я разрабатывал первые варианты строкового класса и класса **list**. Настоящий вариант строкового класса **string** из стандартной библиотеки восходит к этому раннему варианту. Библиотека для выполнения параллельных задач, описанная в работе [Stroustrup, 1987b], была частью проекта «C с классами». Я написал ее для поддержки эмуляций в стиле языка Simula. К сожалению, пришлось ждать долгие 30 лет вплоть до 2011 года, чтобы получить стандартизированную поддержку параллелизма (§1.4.4.2, §5.3, Глава 41). Разработка механизма шаблонов находилась под влиянием разновидности классов **vector**, **map**, **list** и **sort**, первоначально предложенных такими людьми, как Andrew Koenig, Alex Stepanov и др.

Язык C++ рос и мужал в окружении множества признаных и экспериментальных языков программирования (таких как, например, Ada [Ichbiah,1979], Algol 68 [Woodward,

1974] и ML [Paulson, 1996]). В отдельные моменты времени мою работу подпитывали порядка 25 языков, и их влияние документировано в работах [Stroustrup,1994] и [Stroustrup, 2007]. И все-таки, решающее влияние оказывали практические приложения, с которыми я реально сталкивался в жизни.

1.4.3. Стандарт 1998.

По мере того, как применение C++ росло взрывным образом, стало ясно, что его стандартизация просто неизбежна. Нам пришлось начать готовиться к организации этого процесса [Stroustrup,1994]. Была установлена тесная связь между реализаторами компиляторов и основными пользователями. Она поддерживалась как по электронной почте, так и в личных беседах на конференциях и в иных местах.

Корпорация AT&T Bell Labs внесла огромный вклад в развитие C++, позволив мне делиться предварительными версиями с разработчиками компиляторов и пользователями. Поскольку многие из этих людей работали на конкурирующие компании, значимость этого решения нельзя недооценивать. Менее ответственная компания могла внести серьезные проблемы с фрагментацией версий C++, просто ничего не делая. В итоге, сотни людей из десятков организаций смогли знакомиться с материалами по стандартизации прежде, чем они стали официально принятым базовым документом — результатом усилий по созданию ANSI C++ стандарта. Их имена можно найти в работе The Annotated C++ Reference Manual [Ellis,1989]. Комитет X3J16 под эгидой ANSI был созван в декабре 1989 года по инициативе компании Hewlett-Packard. В июне 1991 года процесс стандартизации ANSI (национальный американский комитет) C++ стал частью интернациональной ISO стандартизации под названием WG21. Начиная с 1990 года, эта совместная работа стала основным форумом для эволюции и совершенствования языка C++. Я работал в этих комитетах с самого начала. В частности, как глава рабочей группы по расширениям, в которой был ответственным за рассмотрение предложений по важным изменениям в C++ и по добавлению новых средств языка. Первоначальный план стандарта был опубликован в апреле 1995 года. А в 1998 году был проголосован (счет голосов 22–0) и принят стандарт ISO/IEC14882–1998 [C++,1998]. Исправление ошибок вышло затем в 2003 году, так что иногда люди говорят о C++03, но это в сущности тот же самый C++98.

1.4.3.1. Средства языка

К моменту начала процесса стандартизации по ANSI и ISO большинство средств языка уже присутствовало на месте и было документировано в работе [Ellis,1989]. Поэтому большая часть работы сводилась к уточнениям основных средств языка и их спецификации. Например, механизм шаблонов был значительно усовершенствован в результате серьезных дополнительных усилий. Для того, чтобы легче справляться с разрастающимися объемами программ и увеличением количества обращающихся библиотек, были введены пространства имен. По инициативе Дмитрия Ленкова из Hewlett-Packard были введены минимальные средства информации о типах на этапе исполнения (RTTI; Глава 22). Ранее я избегал таких средств из-за их избыточного использования в языке Simula. Я пытался протащить идею об опциональном консервативном сборщике мусора, но потерпел неудачу. Пришлось ожидать положительного решения вплоть до 2011 года.

В итоге, язык 1998 года был гораздо более богат на средства и лучше специфицирован, чем язык от 1989 года. Однако, не все новшества оказались улучшениями, которые в ретроспективе не показались целесообразными:

- Спецификации исключений призваны были усилить информацию о том, какие исключения генерируются функцией на этапе исполнения. Они были внесены по на-

стойчивым просьбам людей из Sun Microsystems. Впоследствии выяснилось, что спецификации исключений оказались более чем бесполезными для улучшения читаемости, надежности и производительности. В стандарте 2011 они промаркированы как устаревшие (подлежащие устранению в будущем). Стандарт 2011 вводит ключевое слово **noexcept** (§13.5.1.1) как более простое средство решения проблем, для которых ранее предполагалось использовать спецификации исключений.

- Всегда было понятно, что раздельная компиляции шаблонов и их использование было бы идеальным решением [Stroustrup,1994]. Проблема в том, что было не совсем понятно, как это сделать в мире реального программирования. После долгих обсуждений для стандарта 1998 было предложено компромиссное решение на базе ключевого слова **export** применительно к шаблонам. Это решение не было элегантным, всего один производитель решился его реализовать [Edison and Design Group], и в итоге из стандарта 2011 оно было исключено. Мы по-прежнему находимся в поиске более удачного решения. Мое мнение таково, что проблема кроется не в раздельной компиляции как таковой, а в том, что не достаточно четко разграничены спецификации интерфейса и реализации шаблонов, так что решение на базе ключевого слова **export** било не по той цели. В будущем, возможно, языковая поддержка «концепций» (§24.3) сможет помочь, обеспечивая более точную спецификацию требований к шаблонам. Сейчас это находится в активной разработке [Sutton, 2011] [Stroustrup,2012a].

1.4.3.2. Стандартная библиотека

Самой большой новацией в стандарте 1998 было включение STL (коллекция алгоритмов и контейнеров) в стандартную библиотеку языка C++ (§4.4, §4.5, Глава 31, Глава 32, Глава 33). Это был результат работ Алекса Степанова (а также Dave Musser, Meng Le и др.), проводившихся целое десятилетие и посвященных обобщенному программированию. Я, Andrew Koenig и Beman Dawes настаивали на включении STL в стандарт 1998 [Stroustrup,2007]. Библиотека STL встречала массовое приятие в среде программистов на C++, и даже за ее пределами.

Если не считать STL, то в остальном стандартная библиотека представляла собой просто некий набор компонентов, а не единый унифицированный проект. Мне не удалось настоять на включении унифицированной фундаментальной библиотеки в первоначальный вариант C++ [Stroustrup,1993], а не слишком-то помогавший нам менеджер из AT&T помешал мне и моим коллегам исправить этот недостаток в последующем. В итоге, почти все крупные организации (такие как Borland, IBM, Microsoft и Nexas Instruments) самостоятельно реализовали свои собственные фундаментальные библиотеки к моменту начала работы над стандартом. Таким образом, комитет был вынужден ограничиться тем лоскутным одеялом, которое присутствовало «от века» (например, библиотека комплексных чисел), и которое можно было включать в стандарт без риска конфронтации с библиотеками основных поставщиков.

Тип **string** из стандартной библиотеки (§4.4, Глава 36) восходит к моим ранним работам совместно с Jonathan Shopiro из Bell Labs, но в процессе работы над стандартом он был существенно модернизирован усилиями многих людей. А вот тип **valarray** для численных расчетов (§40.5) — это преимущественно работа одного лишь Kent Budge. Jerry Schwarz трансформировал мою библиотеку потокового вывода/вывода (§1.4.2.1) в **iostream** (§4.3, Глава 38), опираясь на работы Andrew Koenig в технике манипуляторов (§38.4.5.2) и другие идеи. Затем это подверглось уточнениям в процессе стандартизации, где основной вклад внесли Jerry Schwarz, Nathan Myers и Norihiro Kumagai.

По сравнению со многими коммерческими библиотеками стандартная библиотека языка C++ невелика. Она не содержит никаких стандартных GUI, баз данных или части, относящейся к Web-приложениям. Все это широко присутствует в многочисленных коммерческих (и свободных) библиотеках, но не является частью ISO стандарта. Причина такого положения дел вовсе не техническая; она уходит корнями в практические и коммерческие аспекты. В то же время, если сравнивать стандартную библиотеку C++ со стандартной библиотекой C, то ее уже не назовешь небольшой.

1.4.4. Стандарт 2011

Текущий стандарт языка C++, C++11, ранее известный как C++0x, является продуктом труда членов комитета WG21. Все вопросы тщательно рассматривались и согласовывались. Это вело к лучшим (и более строгим) спецификациям, но часто ограничивало инновации [Stroustrup,2007]. Самый первый черновик стандарта был опубликован для общественного обсуждения в 2009 году. Второй вариант стандарта ISO C++ (ISO/IEC14882–2011) [C++,2011] был ратифицирован со счетом 21–0 в августе 2011 года.

Столь большой временной разрыв между стандартами C++ объясняется преимущественно мнением членов комитета и моим, что после опубликования прошлого стандарта требовалось значительное время для того, чтобы все хорошенько улеглось, прошло длительную практическую проверку, прежде чем нужно было приступить к работе над новым стандартом. В итоге, работа над новым стандартом началась лишь в 2002 году. Другие причины заключались в том, что язык сильно разросся, а также возрос объем и требования к стандартной библиотеке. В цифрах, относящихся к страницам стандарта, объем стандарта языка вырос на 30%, а библиотеки — на все 100%. При этом большая часть увеличения объема объясняется более детальными спецификациями, а не возросшей функциональностью. Важное значение также имела необходимость в тщательной проверке вопроса о совместимости с предыдущим стандартом, так чтобы не скомпрометировать основную массу наработанного кода на C++: миллиарды строк кода на C++ имеют практическое хождение, и комитет не имел права поставить их под удар.

Самые общие цели для C++11 можно сформулировать очень просто:

- Сделать язык C++ еще более удобным для системного программирования и написания общеупотребительных библиотек.
- Сделать так, чтобы язык C++ было проще преподавать и изучать.

Эти цели документированы и детализированы в работе [Stroustrup,2007].

Большие усилия были направлены на то, чтобы сделать параллельное системное программирование более безопасным по типам и более переносимым. Для этого были разработаны модель памяти (§41.2) и набор средств программирования, не требующих применения блокировок (§41.3), что являлось результатом работ Hans Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams и других. Для иллюстрации того, что можно сделать поверх базовой библиотеки параллелизма, я предложил организовать обмен информацией между задачами без явного применения блокировок, что привело к **future** и **async()** (§5.3.5); большая часть работы в этом направлении была сделана Lawrence Crowl и Detlef Vollmann. Невозможно перечислить всех участников процесса разработки в области параллелизма, так что я даже и не буду пытаться сделать это.

1.4.4.1. Средства языка

Полный список средств языка и стандартной библиотеки, добавленных в стандарт C++11 по сравнению со стандартом C++98, приведен в §44.2. За исключением поддержки параллелизма все остальные изменения и дополнения можно было бы назвать «незначительны-

ми», но поступать так, значит игнорировать важное обстоятельство: средства языка предназначены для совместного использования с целью сделать программы более совершенными. То есть, это программы, которые легче читать, проще сопровождать, и которые выполняются более эффективно (выше скорость выполнения, меньше потребление ресурсов и т.д.).

Здесь я остановлюсь на тех средствах, которые я считаю очень важными кирпичиками, относящимися к стилю программирования на C++11, а также перечисляю людей, внесших наибольший вклад в эту тему:

- Умолчательный контроль: **=delete** и **=default**: §3.3.4, §17.6.1, §17.6.4; Lawrence Crowl и Bjarne Stroustrup.
- Вывод типа объекта из его инициализатора, **auto**: §2.2.2, §6.3.6.1; Bjarne Stroustrup. Впервые я ввел auto в 1993 году, но из-за проблем совместимости с C вынужден был обратно изъять это новшество.
- Обобщенное вычисление константных выражений, **constexpr**: §2.2.3, §10.4, §12.1.6; Gabriel Dos Reis и Bjarne Stroustrup [DosReis,2010].
- Внутриклассовая инициализация членов: §17.4.4; Bjarne Stroustrup, Michael Wong и Michel Michaud.
- Наследование конструкторов: §20.3.5.1; Bjarne Stroustrup, Michael Wong и Michel Michaud.
- Лямбда-выражения, способ неявного задания функциональных объектов в точке их использования в выражении: §3.4.3, §11.4; Jaakko Jarvi.
- Перемещающая семантика, способ перемещения информации без копирования: §3.3.2, §17.5.2; Howard Hinnant.
- Способ сообщения о том, что функция не может генерировать исключения **noexcept**: §13.5.1.1; David Abrahams, Rani Sharoni и Doug Gregor.
- Более подходящее имя для нулевого указателя, §7.2.2; Herb Sutter и Bjarne Stroustrup.
- Оператор цикла range-**for**: §2.2.5, §9.5.1; Thorsten Ottosen и Bjarne Stroustrup.
- Управление замещением: **final** и **override**: §20.3.4; Alisdair Meredith, Chris Uzdavinis и Ville Voutilainen.
- Синонимы типов, механизм предоставления синонима для типа или шаблона. В частности, способ определения шаблона путем связывания части аргументов другого шаблона: §3.4.5, §23.6; Bjarne Stroustrup и Gabriel Dos Reis.
- Тип для энумераторов: **enum class**: §8.4.1; David E. Miller, Herb Sutter и Bjarne Stroustrup.
- Универсальная и однородная инициализация (включая списки инициализации произвольной длины и защита от сужения типов): §2.2.2, §3.2.1.3, §6.3.5, §17.3.1, §17.3.4; Bjarne Stroustrup и Gabriel Dos Reis.
- Шаблоны с переменным числом аргументов, механизм передачи шаблону произвольного числа аргументов произвольных типов: §3.4.4, §28.6; Doug Gregor и Jaakko Jarvi.

Очень многих можно было бы перечислять в списке заслуживших признательность. Эти имена перечислены в техническом отчете [WG21] и в C++11 FAQ [Stroustrup,2010a]. Причина, по которой мое имя упоминается здесь столь часто, состоит не в моем тщеславии, а в том, что я считал необходимым принять личное участие в аспектах разработки, которые мне казались особенно важными. Это те черты и средства языка, которые должны присутствовать в любом хорошем коде. Их основная цель — комплексно и самосогласованно поддерживать основные стили программирования (§1.2.1). Они-то и составляют фундамент синтеза C++11.

Много усилий было потрачено на предложение, которое так и не вошло в стандарт. «Концепции» — это средство специфицирования и проверки требований к аргументам шаблонов [Gregor,2006], которое базировалось на предшествующих работах и которому было уделено серьезное внимание во время работы в комитете над стандартом. Оно было разработано, специфицировано, реализовано и протестировано, но большинство в комитете решило, что оно все еще не совсем готово. Будь оно принято, оно составило бы наиболее существенное отдельное новшество (наряду с параллелизмом). Однако, комитет не принял его, аргументировав сложностью и проблемами с производительностью на этапе компиляции [Stroustrup,2010b]. В настоящее время это средство продолжает активно исследоваться [Sutton,2011], [Stroustrup,2012a].

1.4.4.2. Стандартная библиотека

Работа над проектом, которая в итоге вылилась в стандартную библиотеку C++11, началась с технической бумаги комитета по стандартизации («TR1»). Первоначально, Matt Austern был главой группы по стандартной библиотеке, после чего к этой работе присоединился Howard Hinnant вплоть до финального проекта стандарта в 2011 году.

Я сейчас перечислю ряд компонентов стандартной библиотеки со ссылками на людей, внесших наибольший вклад. Более детальный список приведен в §44.2.2. Некоторые компоненты, такие как **unordered_map** (хэш-таблицы), были почти готовы к включению в стандарт C++98, но все-таки не вошли туда. Многие иные компоненты, такие как **unique_ptr** и **function**, были частью технического отчета (TR1), базировавшегося на библиотеке Boost (Boost — это открытая организация, призванная разрабатывать полезные библиотечные компоненты на базе STL).

- Хэшированные контейнеры, такие как **unordered_map**: §31.4.3; Matt Austern.
- Базовые компоненты библиотеки для параллелизма, такие как **thread**, **mutex** и **lock**: §5.3, §42.2; Pete Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams и др.
- Запуск асинхронных вычислений и возврат результата, **future**, **promise** и **async()**: §5.3.5, §42.4.6; Detlef Vollmann, Lawrence Crowl, Bjarne Stroustrup и Herb Sutter.
- Интерфейс к сборщику мусора: §34.5; Michael Spertus и Hans Boehm.
- Регулярные выражения, **regexp**: §5.5, Глава 37; John Maddock.
- Случайные числа: §5.6.3, §40.7; Jans Maürer и Walter Brown.

Некоторые вспомогательные компоненты были испытаны в составе библиотеки Boost:

- Указатель для простой и безопасной передачи ресурсов, **unique_ptr**: §5.2.1, §34.3.1; Howard E. Hinnant.
- Указатель на разделяемые ресурсы, **shared_ptr**: §5.2.1, §34.3.2; Peter Dimov.
- **tuple**: §5.4.3, §28.5, §34.2.4.2; Jakko Jarvi и Gary Powell.
- Общий **bind()**: §33.5.1; Peter Dimov.
- Тип **function** для владения вызываемыми объектами: §33.5.3; Doug Gregor.

1.4.5. Где используется C++?

В наше время C++ используется практически везде: в вашем компьютере, смартфоне, автомобиле, возможно что и в фотоаппарате. Вы, конечно, вряд ли разглядите его там, ведь C++ — это язык системного программирования и его наиболее значимые применения находятся глубоко в инфраструктуре, которую мы, как пользователя, никогда не видим.

Язык C++ используется миллионами программистов практически в каждой предметной области. В текущем доступе находятся миллиарды строк кода на C++. Инфраструк-

тура программирования на C++ включает десяток независимых реализаторов, несколько тысяч библиотек, сотни учебных пособий и десятки сайтов поддержки. Обучающие курсы разного уровня также широко распространены.

Ранние приложения откровенно тяготели к задачам системного программирования. Например, несколько операционных систем были написаны на C++: [Campbell, 1987] (academic), [Rozier,1988] (real time), [Berg,1995]. В большинстве современных операционных систем (Windows, операционные системы фирмы Apple, Linux и большинство мобильных ОС) многие важные подсистемы написаны на C++. В ваших смартфонах и Интернет-маршрутизаторах также много кода на C++. Я всегда считал крайне важной низкоуровневую бескомпромиссность языка C++. Это позволяет применять C++ для написания драйверов и другого кода, напрямую манипулирующего устройствами в условиях жестких ограничений по производительности. В такого рода программах предсказуемость поведения также важна не меньше чистой скорости. И, наконец, также важна и результирующая компактность кода. Каждое средство языка C++ проектировалось с учетом жестких требований ко времени выполнения и минимизации размера кода (§1.2.4) [Stroustrup, 1994, §4.5].

Многие из известных компьютеризированных коммерческих систем имеют в своем составе важные части, написанные на C++. Примерами являются системы Amadeus (заказ и бронирование авиабилетов), Amazon (Web-коммерция), Bloomberg (финансовая информация), Google (Web-поиск) и Facebook (социальная среда). Множество других языков программирования критически зависят от производительности и надежности кода на C++: это и виртуальные машины Java (к примеру, Oracle HotSpot), и интерпретаторы языка JavaScript (например, V8 от Google), а также браузеры (Microsoft Internet Explorer, Mozilla Firefox, AppleSafari, Google Chrome) и среды выполнения программ (вроде Microsoft. NET). Я всегда считал C++ исключительно важным для построения средств программной инфраструктуры [Stroustrup,2012a].

Конечно, многие приложения имеют в своем составе важные участки кода, критически зависимые от скорости выполнения. Однако, еще больший объем кода в этих приложениях нуждается в удобстве поддержки, расширяемости и простоте тестирования. То, что язык C++ имеет поддержку во всем перечисленном, привело к тому, что он широко применяется в областях, где крайне важны надежность и где требования к поведению программ могут существенно изменяться время от времени. Здесь в качестве примеров можно перечислить финансовые системы, системы телекоммуникаций, управление оборудованием и военные применения. Десятилетиями управление междугородними телефонными звонками в США и все звонки с префиксом 800 (звонки, за которые платит тот, кому звонят) маршрутизируются при помощи кода, написанного на C++ [Kamath,1993]. Все подобные приложения весьма большие и служат длительное время. Поэтому стабильность, совместимость и масштабируемость всегда стояли в центре внимания при разработке C++, ведь вполне типичным является размер программ на C++ во много миллионов строк кода.

При разработке компьютерных игр используется множества языков программирования и инструментов, совместимых с языком программирования с предельной эффективностью. В итоге, игры представляют из себя огромную область для применения C++.

То, что обычно называется системным программированием, находит широкое применение во встраиваемых системах, например в компьютерной томографии, системах управления полетом (например, Lockheed-Martin), в управлении ракетами, корабельными двигателями, в автомобильном ПО (BMW, к примеру) и в системах управления ветрогенераторами (например, Vesta).

При разработке C++ численные математические расчеты не входили в число приоритетных областей применения. Однако, в реальной практике большая часть математи-

ческих расчетов базируется на C++. Причина такого выбора заключается в том, что вычисления часто опираются на стандартные структуры данных, а их результаты требуется представлять в наглядной графической форме, что плохо сочетается с традиционным Fortran [Root,1995]. Я особо рад видеть, как C++ используется в таких масштабных научных проектах, как проект расшифровки генома, проекта CERN по поиску фундаментальных основ вселенной, и во многих других.

Возможность применять C++ в самых разных предметных областях (локальные и глобальные сети, графика, численные расчеты, базы данных и др.), для которых ранее традиционно использовались разные языки программирования, а также способность сосуществовать в рамках одного проекта с другими языками программирования, является очень сильной стороной C++. Опять-таки, в этом отношении очень важна стабильность этого языка на протяжении десятилетий. Идеи межъязыкового взаимодействия, заложенные в проект C++, оказались исключительно важными с практической точки зрения.

Большинство приложений не пишутся исключительно на «голом» языке программирования. Для программистов на C++ в распоряжении находится огромное число библиотек, таких как Boost, POCO (Web-разработка), QT (кроссплатформенная разработка приложений), wxWidgets (кроссплатформенная GUI-библиотека), WebKit (рендеринг для Web-браузеров), CGAL (вычислительная геометрия), QuickFix (обмен финансовой информацией), OpenCV (обработка изображений в реальном времени), Root (физика высоких энергий) и много-много других полезных библиотек.

1.5. Советы

Каждая глава настоящей книги завершается разделом «Советы», в котором даются конкретные рекомендации, имеющие отношение к содержанию этой главы. Эти советы можно применять там, где они уместны. Они нисколько не заменяют глубокие знания, интеллект, опыт, чувство меры и хороший вкус.

Я не считаю полезными советы в форме «не делайте так ...». Поэтому мои советы имеют форму предложений, как нужно делать. В разделе «Советы» вы не найдете детальных пояснений. Вместо этого к каждому совету прикреплена ссылка на соответствующую часть главы.

Вот некоторые советы для начинающих:

[1] Представляйте идеи (концепции) непосредственно в коде, например, в виде функции или класса; §1.2.

[2] Старайтесь сделать ваш код одновременно элегантным и эффективным; §1.2.

[3] Не стремитесь к излишней абстракции; §1.2.

[4] Фокусируйтесь на проектировании элегантных и эффективных абстракций, возможно в форме библиотек; §1.2.

[5] Отражайте зависимости между идеями непосредствыенно в коде, например с помощью параметризации или классовой иерархии; §1.2.1.

[6] Реализуйте несвязанные идеи изолированными частями кода, например, избегая зависимостей между классами; §1.2.1.

[7] C++ не просто объектно-ориентированный язык; §1.2.1.

[8] C++ не только предназначен для обобщенного программирования; §1.2.1.

[9] Предпочитайте решения, которые проверяются статически; §1.2.1.

[10] Явно представляйте ресурсы (как объекты классов); §1.2.1, §1.4.2.1.

[11] Простые идеи выражайте просто; §1.2.1.

[12] Применяйте библиотеки, особенно стандартную библиотеку, а не пытайтесь все сделать самостоятельно; §1.2.1.

[13] Используйте стиль программирования, богатый типами; §1.2.2.

[14] Низкоуровневый код не обязательно самый эффективный; не избегайте классов, шаблонов и компонентов стандартной библиотеки из-за опасений в потере производительности; §1.2.4, §1.3.3.

[15] Если данные содержат инвариант, инкапсулируйте его; §1.3.2.

[16] C++ это не C с расширениями; §1.3.3.

И вообще: написание хороших программ требует интеллекта, вкуса и терпения. У вас вряд ли все получится с самого начала. Экспериментируйте!

1.6. Библиография

[Austern,2003] Matt Austern et al.: *Untangling the Balancing and Searching of Balanced Binary Search Trees.* Software — Practice & Experience. Vol 33, Issue 13. November 2003.

[Barron,1963] D. W. Barron et al.: *The main features of CPL.* The Computer Journal. 6 (2): 134. (1963). comjnl.oxfordjournals.org/content/6/2/134.full.pdf+html.

[Barton,1994] J. J. Barton and L.R. Nackman: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley. Reading, Massachusetts. 1994. ISBN0–201–53393–6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project.* CACM. Vol. 38, No. 10. October 1995.

[Boehm,2008] Hans-J. Boehm and Sarita V. Adve: *Foundations of the C++ concurrency memory model.* ACM PLDI'08.

[Boost] The Boost library collection. www.boost.org.

[Budge,1992] Kent Budge, J.S. Perry, and A.C. Robinson: *High-Performance Scientific Computation Using C++.* Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990] X3 Secretariat: *Standard — The C Language.* X3J11/90–013. ISO Standard ISO/IEC9899–1990. Computer and Business Equipment Manufacturers Association. Washington, DC.

[C,1999] ISO/IEC9899. *Standard — The C Language.* X3J11/90–013–1999.

[C,2011] ISO/IEC9899. *Standard — The C Language.* X3J11/90–013–2011.

[C++,1998] ISO/IEC JTC1/SC22/WG21: *International Standard — The C++ Language.* ISO/IEC14882:1998.

[C++Math,2010] *International Standard — Extensions to the C++ Library to Support Mathematical Special Functions.* ISO/IEC29124:2010.

[C++,2011] ISO/IEC JTC1/SC22/WG21: *International Standard — The C++ Language.* ISO/IEC14882:2011.

[Campbell,1987] Roy Campbell et al.: *The Design of a Multiprocessor Operating System.* Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien,1995] James O. Coplien: *Curiously Recurring Template Patterns.* The C++ Report. February 1995.

[Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast.* January 2007. swtch.com/~rsc/regexp/regexp1.html.

[Czarnecki,2000] K. Czarnecki and U. Eisenecker: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. Reading, Massachusetts. 2000. ISBN0–201–30977–7.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972] O-J. Dahl and C. A.R. Hoare: *Hierarchical Program Construction* in *Structured Programming*. Academic Press. New York. 1972.

[Dean,2004] J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation. 2004.

[Dechev,2010] D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC2010 Symposium. May 2010.

[DosReis,2006] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.

[DosReis,2010] Gabriel Dos Reis and Bjarne Stroustrup: *General Constant Expressions for System Programming Languages*. SAC-2010. The 25th ACM Symposium On Applied Computing. March 2010.

[DosReis,2011] Gabriel Dos Reis and Bjarne Stroustrup: *A Principled, Complete, and Efficient Representation of C++*. Journal of Mathematics in Computer Science. Vol. 5, Issue 3. 2011.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN0–201–51459–1.

[Freeman,1992] Len Freeman and Chris Phillips: *Parallel Numerical Algorithms*. Prentice Hall. Englewood Cliffs, New Jersey. 1992. ISBN0–13–651597–5.

[Friedl,1997]: Jeffrey E.F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN978–1565922570.

[Gamma,1995] Erich Gamma et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN0–201–63361–2.

[Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.

[Hennessy,2011] John L. Hennessy and David A. Patterson: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann. San Francisco, California. 2011. ISBN978–0123838728.

[Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Pro gramming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72, No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN0–13–110362–8.

[Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible — but no closer*. The C++ Report. Vol. 1, No. 7. July 1989.

[Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised).* Proc USENIX C++ Conference. April 1990.

[Kolecki,2002] Joseph C. Kolecki: *An Introduction to Tensors for Students of Physics and Engineering.* NASA/TM-2002–211716.

[Langer,2000] Angelika Langer and Klaus Kreft: *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference.* Addison-Wesley. 2000. ISBN978–0201183955.

[McKenney] Paul E. McKenney: *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org. Corvallis, Oregon. 2012. http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[Maddock,2009] John Maddock: *Boost.Regex.* www.boost.org. 2009.

[Orwell,1949] George Orwell: *1984.* Secker and Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: *ML for the Working Programmer.* Cambridge University Press. Cambridge. 1996. ISBN0–521–56543-X.

[Pirkelbauer,2009] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup: *Design and Evaluation of C++ Open Multi-Methods.* Science of Computer Programming. Elsevier Journal. June 2009. doi:10.1016/j.scico.2009.06.002.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL — The Language and Its Compiler.* Cambridge University Press. Cambridge. 1980. ISBN0–521–21965–5.

[Root,1995] *ROOT: A Data Analysis Framework.* root.cern.ch. It seems appropriate to represent a tool from CERN, the birthplace of the World Wide Web, by a Web address.

[Rozier,1988] M. Rozier et al.: *CHORUS Distributed Operating Systems.* Computing Systems. Vol. 1, No. 4. Fall 1988.

[Siek,2000] Jeremy G. Siek and Andrew Lumsdaine: *Concept checking: Binding parametric polymorphism in C++.* Proc. First Workshop on C++ Template Programming. Erfurt, Germany. 2000.

[Solodkyy,2012] Y. Solodkyy, G. Dos Reis, and B. Stroustrup: *Open and Efficient Type Switch for C++.* Proc. OOPSLA'12.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library.* HP Labs Technical Report HPL-94–34 (R. 1). 1994.

[Stewart,1998] G. W. Stewart: *Matrix Algorithms, Volume I. Basic Decompositions.* SIAM. Philadelphia, Pennsylvania. 1998.

[Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language.* Sigplan Notices. January 1982. The first public description of "C with Classes."

[Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++.* Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.

[Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++.* Proc. Summer 1985 USENIX Conference.

[Stroustrup,1986] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley. Reading, Massachusetts. 1986. ISBN0–201–12078-X.

[Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++.* Proc. EUUG Spring Conference. May 1987.

[Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming.* Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. 1988.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN0–201–53992–6.

[Stroustrup,1993] B. Stroustrup: *A History of C++: 1979–1991*. Proc. ACM History of Programming Languages conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN0–201–54330–3.

[Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN0–201–88954–4. Hardcover ("Special") Edition. 2000. ISBN0–201–70073–5.

[Stroustrup,2002] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++1991–2006*. ACM HOPL–III. June 2007.

[Stroustrup,2008] B. Stroustrup: *Programming — Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN0–321–54372–6.

[Stroustrup,2010a] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2010b] B. Stroustrup: *The C++0x "Remove Concepts" Decision*. Dr. Dobb's Journal. July 2009.

[Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12–0041. January 2012.

[Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

[Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE2011 (International Conference on Software Language Engineering). July 2011.

[Tanenbaum,2007] Andrew S. Tanenbaum: *Modern Operating Systems, Third Edition*. Prentice Hall. Upper Saddle River, New Jersey. 2007. ISBN0–13–600663–9.

[Tsafrir,2009] Dan Tsafrir et al.: *Minimizing Dependencies within Generic Classes for Faster and Smaller Programs*. ACM OOPSLA'09. October 2009.

[Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley. Reading, Massachusetts. 1996. ISBN0–201–48345–9.

[UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Vandevoorde,2002] David Vandevoorde and Nicolai M. Josuttis: *C++ Templates: The Complete Guide*. Addison-Wesley. 2002. ISBN0–201–73484–2.

[Veldhuizen,1995] Todd Veldhuizen: *Expression Templates*. The C++ Report. June 1995.

[Veldhuizen,2003] Todd L. Veldhuizen: *C++ Templates are Turing Complete*. Indiana University Computer Science Technical Report. 2003.

[Vitter,1985] Jefferey Scott Vitter: *Random Sampling with a Reservoir*. ACM Transactions on Mathematical Software, Vol. 11, No. 1. 1985.

[WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: *C++ Concurrency in Action — Practical Multithreading*. Manning Publications Co. ISBN978–1933988771.

[Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge, Mass. 1996. ISBN0–262–73118–5.

[Wood,1999] Alistair Wood: *Introduction to Numerical Analysis*. Addison-Wesley. Reading, Massachusetts. 1999. ISBN0–201–34291-X.

[Woodward,1974] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. 1974.

2

Обзор языка C++: Основы

В первую очередь, разделаемся
с защитниками языка.
— Генрих VI, Часть II

- Введение
- Основы

  Типы; Переменные и арифметика; Константы; Операторы выбора и циклы; Указатели, массивы и циклы
- Пользовательские типы

  Структуры; Классы; Перечисления
- Модульность

  Раздельная компиляция; Пространства имен; Обработка ошибок
- Заключение
- Советы

2.1. Введение

Целью данного раздела, а также следующих трех разделов, является попытка обрисовать язык C++ в самых общих чертах, не углубляясь в детали. В настоящем разделе знакомство с основными понятиями C++ осуществляется неформально, рассматривается модель памяти C++ и модель выполнения программ, а также представляются основные способы организации кода. Все эти средства поддерживают стили программирования, встречающиеся в языке C, и традиционно называющиеся *процедурным программированием*. В Главе 3 представляются механизмы абстракции языка C++, а в Главе 4 и Главе 5 на примерах представлены средства стандартной библиотеки.

Предполагается, что читатель имеет опыт программирования. В противном случае, лучше сначала прочитать учебник «Программирование: Принципы и практика применения C++» [Страуструп, 2009], после чего вернуться к настоящей книге. Но даже при наличии опыта программирования может оказаться, что применяемые вами языки программирования или типы программ, которые вы создавали, разительно отличаются от стиля программирования на C++, который рассматривается в настоящей книге. Если вас смутит «молниеносный стиль» этой части книги, пропустите ее и перейдите к более систематическому изложению, начиная с Главы 6.

Такой старт, однако, поможет нам отойти от строгого стиля «снизу-вверх» при рассмотрении языка и стандартной библиотеки и сразу же в начальных главах позволит использовать более широкий набор средств и возможностей. Например, циклы не рассматриваются подробно вплоть до Главы 10, однако в очевидных рамках они будут применяться

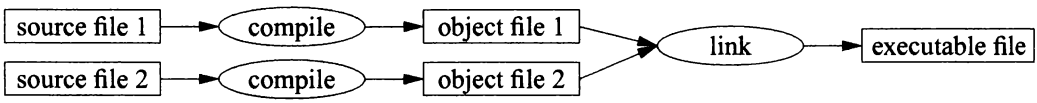
нами задолго до этого. Аналогично, детальное описание классов, шаблонов и динамического выделения свободной памяти, а также подробное рассмотрение средств стандартной библиотеки выполняется постепенно на протяжении многих глав, и такие типы стандартной библиотеки, как **vector**, **string**, **complex**, **map**, **unique_ptr** и **ostream**, широко применяются всюду, где это улучшает код примеров.

В качестве аналогии, рассмотрим краткий туристический визит в Копенгаген или Нью-Йорк. За несколько часов вы посещаете наиболее известные места, вам сообщают общую полезную информацию и дают рекомендации, что следует посмотреть и посетить в следующий раз. Естественно, нельзя сказать, что вы после этого хорошо знаете город. Вряд ли вы поняли все, что вам было рассказано и показано. Чтобы по-настоящему изучить город, в нем нужно пожить, возможно даже много лет. Тем не менее, в удачном варианте вы все же получите общее представление о городе, почувствуете, что в нем особенного, что конкретно вас может в нем заинтересовать. После такого беглого знакомства вы смело можете продолжить знакомство уже на более подробном уровне.

Настоящая глава представит язык C++ в целом, а не отдельными слоями. Как следствие, не будет подчеркиваться различие между средствами, относящимися к C, или к C++98, или к C++11. Такая историческая информация находится в §1.4 и в Главе 44.

2.2. Основы

C++ является компилируемым языком. Чтобы программа могла быть исполнена, ее исходный код должен быть откомпилирован в *объектные файлы* (*object files*), которые затем собираются компоновщиком (линкером — *linker*) в *исполняемый файл* (*executable file*). В типичном случае программа на C++ состоит из многих файлов с исходным кодом (вкратце, из *исходных файлов* — *source files*).



Исполняемая программа создается под конкретную аппаратно/операционную платформу; она не переносима, например, с платформы Mac на платформу Windows PC. Когда мы говорим о переносимости C++-программ, мы имеем ввиду переносимость исходного кода; исходный код может быть успешно скомпилирован и после этого запущен на выполнение на разных системах.

Стандарт ISO для C++ означает две вещи:

- *Базовые возможности языка*, такие как встроенные типы (например, **char** или **int**) или циклы (например, **for**-операторы или **while**-операторы)
- *Компоненты стандартной библиотеки*, такие как контейнеры (например, **vector** или **map**) или операции ввода/вывода (например, **<<** или **getline()**)

Компоненты стандартной библиотеки представляют из себя C++-код, поставляемый с каждой реализацией C++. То есть, стандартная библиотека сама реализуется на C++ (с минимальным использованием машинного кода для таких вещей, как например переключение контекста выполнения программных потоков). Это означает, что C++ достаточно выразителен и эффективен для решения наиболее требовательных системных задач.

C++ является языком со статической типизацией. Это означает, что тип любой сущности (например, объекта, значения, идентификатора или выражения) должен быть известен компилятору в точке использования. Тип объекта определяет множество операций, применимых к объекту.

2.2.1. Привет, Мир!

Вот минимальная программа на C++:

```
int main() {}    // минимальная C++ программа
```

В ней определяется функция с именем **main**, которая не принимает аргументов и ничего не делает (§15.4).

Фигурные скобки, {}, используются в C++ для группировки. Здесь они отмечают начало и конец определения тела функции. Двойная наклонная черта, //, начинает комментарий, который распространяется на оставшуюся часть строки кода. Комментарий предназначен для человека; компилятор его игнорирует.

В каждой C++-программе должна присутствовать одна глобальная функция **main()**. Исполнение программы начинается с этой функции. Значение типа **int**, возвращаемое этой функцией (если оно есть), передается системе в качестве возврата. При отсутствии возвращаемого значения системе передается значение, означающее успешное завершение. Ненулевое возвращаемое значение сигнализирует об ошибке. Не всякая операционная система и среда выполнения используют этот возврат: среды на базе операционных систем Linux/Unix делают это, в то время как среда в Windows редко когда обращается к этому возврату.

В типичном случае программа осуществляет какой-нибудь вывод информации. Вот пример программы, выводящей строку текста **Hello, World!**:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

Строка **#include <iostream>** инструктирует компилятор, чтобы он *включил* объявления стандартных средств потоков ввода/вывода из файла **iostream**. Без этих объявлений выражение

```
std::cout << "Hello, World!\n"
```

не имеет смысла. Операция **<<** («вставка в поток») пишет значение своего второго аргумента (правого), то есть текстовый литерал **"Hello, World!\n"** в стандартный поток вывода (**std::cout**), являющийся ее первым аргументом (левым). Строковый литерал — это последовательность символов, заключенная в двойные кавычки. В строковом литерале последовательность из обратной наклонной черты \ и следующего за ней символа означает «единый специальный символ». Например, **\n** — это символ перехода на новую строку.

Префикс **std::** означает, что имя **cout** следует искать в пространстве имен стандартной библиотеки (§2.4.2, Глава 14). Я обычно опускаю этот префикс при обсуждении средств стандартной библиотеки; в §2.4.2 показано, как обеспечить доступ к именам из пространства имен без явной квалификации.

Весь значимый исполняемый код размещается в функциях и вызывается прямо или косвенно из функции **main()**. Например:

```
#include <iostream>
using namespace std;          // видимость имен из std без std:: (§ 2.4.2)

double square(double x)       // возвести в квадрат число типа double
{
    return x*x;
}
```

```
void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}
int main()
{
    print_square(1.234); // печать: квадрат 1.234 есть 1.52276
}
```

Тип возврата **void** означает, что функция не выполняет возврата вообще.

2.2.2. Типы, переменные и арифметические операции

Каждое имя и каждое выражение имеет тип, определяющий операции, которые можно над ними выполнять. Например, объявление

```
int inch;
```

декларирует, что **inch** имеет тип **int**; то есть, **inch** — это целая переменная.

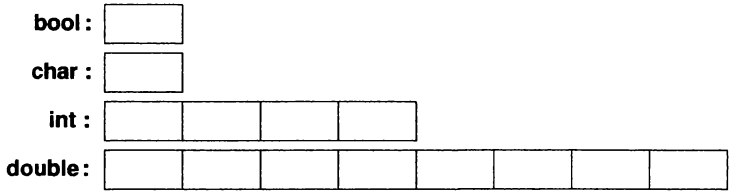
Объявление — это оператор, который вводит имя в программу. Он специфицирует тип для именованной сущности:

- *Тип* определяет набор допустимых значений и операций (над объектом).
- *Объект* есть блок памяти, содержащий значение для некоторого типа.
- *Значение* — это набор битов информации, интерпретируемый согласно типу.
- *Переменная* — это именованный объект.

Язык C++ располагает множеством фундаментальных типов. Например:

```
bool    // Логический, возможные значения true и false
char    // символьный, например, 'a', ' z', и '9'
int     // целый, например, 1, 42, и 1066
double  // с плавающей запятой двойной точности, пример, 3.14 и 299793.0
```

Каждый фундаментальный тип отображается в аппаратные особенности компьютера (процессора) и имеет фиксированный размер, от которого зависит диапазон хранимых значений:



Переменная типа **char** имеет естественный для конкретной машины размер для хранения символа (в типичном случае это 8 бит), а остальные типы имеют кратные размеры относительно типа **char**. Размер типа является машиннозависимым (может различаться для разных машин) и его можно запросить при помощи операции **sizeof**; например, **sizeof(char)** дает **1**, а **sizeof(int)** часто равен **4**.

Арифметические операции можно применять в надлежащих комбинациях:

```
x+y     // плюс
+x      // унарный плюс
x-y     // минус
-x      // унарный минус
x*y     // умножить
x/y     // поделить
x%y     // остаток (модуль) для целых
```

То же справедливо и для операций сравнения:

```
x==y    // равно
x!=y    // не равно
x<y     // меньше чем
x>y     // больше чем
x<=y    // меньше чем или равно
x>=y    // больше чем или равно
```

В языке C++ для присваиваний и арифметических операций выполняются все имеющие смысл преобразования (§10.5.3) между базовыми типами, так что их можно свободно смешивать:

```
void some_function()    // у функции нет возврата
{
    double d = 2.2;     // инициализация числа типа double
    int i = 7;          // инициализация целого
    d = d+i;            // присвоить d значение суммы
    i = d*i;            // присвоить i значение d*i с усечением до целого
}
```

Обратите внимание на то, что = это операция присваивания, а == это операция сравнения на равенство.

В языке C++ есть несколько форм записи для инициализации, таких как = в приведенном выше примере, а также есть универсальная запись с фигурными скобками и списком инициализации:

```
double d1 = 2.3;
double d2 {2.3};

complex<double> z = 1;      // комплексное число со скалярами двойной точности
complex<double> z2 {d1, d2};
complex<double> z3 = {1,2}; // = необязательно с {...}

vector<int> v {1,2,3,4,5,6};    // вектор целых
```

Применение знака = традиционно и восходит к языку C, но в сомнительных случаях используйте общую {}-списковую форму (§6.3.5.2). При прочих равных, это предохранит вас от преобразований, теряющих информацию (сужающих преобразований; §10.5).

```
int i1 = 7.2;       // i1 становится 7
int i2 {7.2};       // ошибка: конверсия в целое из числа с плавающей запятой
int i3 = {7.2};     // та же ошибка:  (знак = избыточен)
```

Константу (§2.2.3) нельзя не инициализировать, а переменные следует оставлять неинициализированными в исключительно редких случаях. Не вводите имя в программу до тех пор, пока для него не найдется подходящего значения. Пользовательские типы (такие как **string**, **vector**, **Matrix**, **Motor_controller** или **Orc_warrior**) можно определить так, что станет возможной неявная инициализация (§3.2.1.1).

При определении переменной нет необходимости явно специфицировать тип, если его можно вывести из инициализирующего значения:

```
auto b = true;      // bool
auto ch = 'x';      // char
auto i = 123;       // int
auto d = 1.2;       // double
auto z = sqrt(y);   // z имеет тип возврата sqrt(y)
```

Применяя **auto**, используем синтаксис со знаком =, поскольку при этом отсутствует преобразование типов, которое может породить проблемы (§6.3.6.2).

Применение **auto** возможно в случаях, когда нет специфических причин указывать тип явно. Вот эти «специфические причины»:

- Определение распространяется на большую область видимости, в которой мы хотим, чтобы тип был ясно различим для всех читателей нашего кода.
- Мы хотим явно задать точность вычислений для переменной (например, **double**, а не **float**).

Применение **auto** исключает избыточность и помогает избежать записи длинных имен типов. Это особенно актуально для обобщенного программирования, когда затруднительно знать истинный тип объекта, а типы объектов могут быть весьма длинными (§4.5.1).

Кроме стандартных форм арифметических и логических операций (§10.3) в C++ есть и более специфические варианты этих операций для изменения значений переменных:

```
x+=y    // x = x+y
++x     // инкремент: x = x+1
x-=y    // x = x-y
--x     // декремент: x = x-1
x*=y    // масштабирование: x = x*y
x/=y    // масштабирование: x = x/y
x%=y    // x = x%y
```

Эти операции компактны, удобны и часто используются на практике.

2.2.3. Константы

Язык C++ поддерживает две концепции *неизменяемости* (*immutability*) (§7.5):

- **const**: примерно означает «обещаю не изменять эту величину» (§7.5). Применяется преимущественно на уровне интерфейсов, так что можно передавать в функции данные, не опасаясь того, что там они будут модифицированы. Компилятор подкрепляет обещание, которое дается посредством **const**.
- **constexpr**: приблизительно означает «подлежит вычислению во время компиляции» (§10.4). Применяется преимущественно для констант, позволяет размещать данные в памяти, где маловероятна их порча, а также ради производительности.

Например:

```
const int dmv = 17;                          // dmv это именованная константа
int var = 17;                                // var не константа
constexpr double max1 = 1.4*square(dmv);     // OK если square(17) константа
constexpr double max2 = 1.4*square(var);     // ошибка: var не константа
const double max3 = 1.4*square(var);         // OK, вычисляется при выполнении
double sum(const vector<double>&);           // sum не изменяет аргумент (§ 2.2.5)
vector<double> v {1.2, 3.4, 4.5};            // v не константа
const double s1 = sum(v);                    // OK: вычисляется при выполнении
constexpr double s2 = sum(v);                // ошибка: sum(v) не констант. выраж-е
```

Чтобы функцию можно было использовать в *константном выражении*, то есть в выражении, вычисляемом компилятором, ее нужно определить с модификатором **constexpr**. Например:

```
constexpr double square(double x) {return x*x;}
```

В этом случае функция должна быть достаточно простой: всего лишь оператор **return**, вычисляющий и возвращающий значение. Такую функцию можно использовать с неконстантными аргументами, но тогда результат не является константным выражением. Это разрешается делать в случаях, не требующих константных выражений, так что не нужно определять практически идентичную функцию дважды: один раз для константных выражений и второй раз для переменных.

В небольшом количестве случаев константные выражения требуются в соответствии с правилами языка (границы массивов (§2.2.5, §7.3), case-метки (§2.2.4, §9.4.2), ряд аргументов шаблонов (§25.2) и константы, объявленные с **constexpr**). В остальных случаях вычисления во время компиляции актуальны для производительности. Кроме того, в независимости от вопросов эффективности концепция неизменяемости (для объекта с неизменным состоянием) является важной концепцией проектирования (§10.4).

2.2.4. Операторы выбора и циклы

Язык C++ предоставляет традиционный набор операторов выбора и циклов. Например, вот пример простой функции, которая запрашивает ввод у пользователя и возвращает булевское значение, отражающее введенное значение:

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";   // вывести вопрос

    char answer = 0;
    cin >> answer;                                  // прочитать ответ

    if (answer == 'y') return true;
    return false;
}
```

В пару к операции вывода << для ввода используется операция >>, где **cin** есть стандартный поток ввода. Тип правого операнда операции ввода определяет, какой ввод допустим, а сам этот операнд представляет собой конечную цель операции ввода. Символ **\n** в конце выводимой строки означает «перевод строки» (§2.2.1).

Приведенный пример можно улучшить, если принять во внимание и ответ **n** (то есть «нет»):

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";   // написать вопрос

    char answer = 0;
    cin >> answer;                                  // прочитать ответ

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

Оператор **switch** сравнивает значение с набором констант. Константы в **case**-ветвях должны быть уникальными, и если проверяемое значение не совпадает ни с одной из них, то выполняется ветвь **default**. Если **default**-ветвь отсутствует, то не выполняется ничего (переходят к следующему оператору).

Мало какая программа может обойтись без циклов. Например, мы могли бы предоставить пользователю несколько попыток произвести приемлемый ввод:

```
bool accept3()
{
    int tries = 1;
    while (tries<4) {
        cout << "Do you want to proceed (y or n)?\n";   // написать вопрос
        char answer = 0;
        cin >> answer;                                  // прочитать ответ

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "Sorry, I don't understand that.\n";
    ++tries;        // инкремент
    }
}
cout << "I'll take that for a no.\n";
return false;
}
```

Оператор **while** выполняется до тех пор, пока его проверяемое условие не станет ложным (**false**).

2.2.5. Указатели, массивы и циклы

Массив элементов типа **char** можно объявить следующим образом:

```
char v[6];      // массив из 6 символов
```

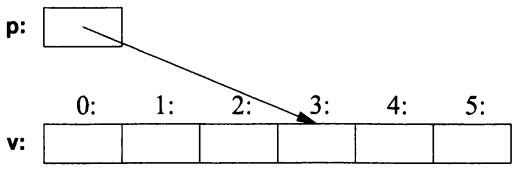
Аналогично, указатель объявляется так:

```
char*p;         // указатель на char
```

В объявлении [] означает массив, а * означает указатель. В массивах наименьшее значение индекса равно **0**, так что **v** имеет шесть элементов, с **v[0]** по **v[5]**. Размер массива должен быть константным выражением (§2.2.3). Переменная типа указатель может содержать адрес объекта соответствующего типа:

```
char* p = &v[3];    // p указывает на 4-ый элемент v
char x = *p;        // *p - объект, указуемый через p
```

В выражениях унарный префикс * означает «содержимое адреса памяти», а унарный префикс **&** означает «взять адрес». Приведенные определения с инициализацией можно графически отобразить следующим образом:



Рассмотрим копирование десяти элементов из одного массива в другой:

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];                         // будет копией v1
```

```
    for (auto i=0; i!=10; ++i)        // копируем элементы
        v2[i]=v1[i];
    // ...
}
```

Оператор **for** можно трактовать следующим образом: «присвоим **i** значение **0**; пока **i** не равно **10**, копируем i-ый элемент и инкрементируем **i**». Когда операция инкремента **++** применяется к целой величине, она эквивалентна прибавлению **1**. Язык C++ предлагает также и более простой вариант оператора **for**, называемый оператором range-**for**, который выполняет циклический проход последовательности простейшим способом:

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v)                  // для каждого x из v
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

Первый оператор range-**for** можно трактовать так: «для всех элементов **v**, от первого до последнего, помещать их копию в **x** и осуществлять вывод». Заметим, что нет необходимости уточнять границы массива при его инициализации списком значений. Оператор range-**for** можно применять для любой последовательности элементов (§3.4.1).

Если не нужно копировать элементы из **v** в переменную **x**, а нужно лишь, чтобы **x** ссылался на элемент, то можно написать следующее:

```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto& x : v)
        ++x;
    // ...
}
```

В объявлениях унарный суффикс **&** означает «ссылка на». Ссылка аналогична указателю, но в этом случае нет необходимости использовать префикс * для доступа к значению, адресуемому ссылкой. Кроме того, ссылку нельзя переназначить после ее инициализации. Знаки операций (такие как **&**, * и []) в составе операторов объявления называются *операциями объявления*:

```
T a[n];    // T[n]: массив из n эл-в типа T
T* p;      // T*: указ-ль на T (§ 7.2)
T& r;      // T&: ссылка на T (§ 7.7)
T f(A);    // T(A): ф-я с аргументом типа A и возвратом типа T (§ 2.2.1)
```

Нужно следить за тем, чтобы указатель всегда ссылался на объект, иначе доступ по указателю (dereferencing) будет недействительным. Когда у нас нет объекта, на который нужно ссылаться, или же когда нужно отразить понятие «отсутствие объекта» (например, достигнут конец списка), указателю нужно присвоить значение **nullptr** («нулевой указатель»). Значение **nullptr** применяется для всех типов указателей:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;  // указ-ль на Link<Record>
int x = nullptr;              // ошибка: nullptr указа-ль, а не целое
```

Во многих случаях полезно проверять, что аргумент типа указатель на самом деле указывает на нечто:

```
int count_x(char* p, char x)
    // сосчитать число вхождений x в p[]
    // p указывает на нуль-терминирован. массив символов
{
    if (p==nullptr) return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

Обратите внимание на то, как мы «передвигаем указатель» на следующий элемент массива при помощи операции **++**, и что мы можем обойтись без инициализации в операторе **for**, если в ней нет необходимости.

В определении функции **count_x()** предполагается, что **char*** указывает на *строку* в *С-стиле*, то есть что указатель «смотрит» на массив элементов типа **char** с терминальным нулем.

Раньше в C++ вместо **nullptr** использовались **NULL** или **0** (§7.2.2). Применение **nullptr** устраняет возможную путаницу между целыми (**0** или **NULL**) и указателями (**nullptr**).

2.3. Пользовательские типы

Мы называем типы, которые можно построить на фундаментальных типах (§2.2.2), модификаторе **const** (§2.2.3) и с помощью операций объявления (§2.2.5), *встроенными типами*. Набор встроенных типов и операций в C++ богат, но он намеренно низкоуровневый. Они прямолинейно и эффективно отражают возможности традиционного компьютерного «железа». Но они не снабжают программиста высокоуровневыми средствами для простого и удобного программирования продвинутых приложений. Язык C++ подкрепляет встроенные типы и операции изощренными *механизмами абстракции*, с помощью которых программист сам создает удобные высокоуровневые средства. Механизмы абстракции языка C++ в первую очередь предназначены для того, чтобы программист мог конструировать и реализовывать собственные типы, с подходящей структурой и удобными операциями, и чтобы другие программисты могли без труда пользоваться ими. Типы, которые создаются на базе встроенных типов с применением механизмов абстракции языка C++, называются *пользовательскими типами*. Оставшаяся часть настоящей главы представляет простейшие и наиболее фундаментальные средства для работы с пользовательскими типами. В Главе 3 механизмы абстракции и поддерживающие их стили программирования рассматриваются в деталях. В Главе 4 и Главе 5 дается обзор стандартной библиотеки, состоящей в основном из пользовательских типов, так что в этих главах демонстрируются примеры того, что можно создать, используя средства языка и технологии (приемы) программирования, представленные в Главе 2 и Главе 3.

2.3.1. Структуры

Первым шагом в построении нового типа часто является решение организовать необходимые для него элементы в соответствующую структуру (**struct**) данных:

```
struct Vector {
    int sz;          // число элементов
    double* elem;    // указатель на элементы
};
```

Наша первая версия для типа **Vector** состоит из элементов типа **int** и **double***.

Переменную типа **Vector** создают следующим образом:

```
Vector v;
```

Однако, это мало что дает, ибо при этом указатель **elem** ни на что не указывает. Ясно, что нужно сделать так, чтобы **v** мог указывать на какие-то элементы вектора. К примеру, можно было бы конструировать вектор типа **Vector** следующим образом:

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s];        // массив из s эл-ов типа double
    v.sz = s;
}
```

То есть мы присваиваем указателю **elem** адрес, производимый операцией **new**, а размер **size** приравнивается к числу элементов. Знак **&** в **Vector&** означает, что мы передаем **v** при помощи неконстантной (non-**const**) ссылки (§2.2.5, §7.7); таким образом, функция **vector_init()** может модифицировать (изменять) передаваемый ей вектор.

Операция **new** выделяет блок из *свободой памяти* (*free store*), также известной как *динамическая память* (*dynamic memory*) или *куча* (*heap*; (§11.2).

Приведем простой пример, использующий тип **Vector**:

```
double read_and_sum(int s)
    // читаем s целых из cin и возвращаем их сумму; s положительно
{
    Vector v;
    vector_init(v, s);          // аллоцируем s элементов для v
    for (int i=0; i!=s; ++i)
        cin>>v.elem[i];         // читаем значения элементов

    double sum = 0;
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i];         // сумма элементов
    return sum;
}
```

Предстоит еще много чего сделать, чтобы наш тип **Vector** стал таким же элегантным и гибким, как **vector** из стандартной библиотеки. В частности, начальный вариант типа **Vector** заставляет пользователя знать все детали его представления. Далее мы будем постепенно улучшать тип **Vector**, демонстрируя при этом возможности языка и техники программирования. В Главе 4 мы представим тип **vector** из стандартной библиотеки, содержащий множество усовершенствований, а Глава 31 представит этот тип в контексте других средств стандартной библиотеки.

Я применяю **vector** и другие компоненты стандартной библиотеки в качестве примеров для:

- иллюстрации средств языка и приемов проектирования,
- помощи в изучении и использовании стандартной библиотеки.

Не следует заново изобретать компоненты стандартной библиотеки, такие как **vector** или **string**; просто используйте их.

Для доступа к членам структуры по ее имени (или по ссылке) используется точка, а для доступа по указателю используется знак **->**. Вот соответствующий пример:

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;          // доступ по имени
    int i2 = rv.sz;         // доступ по ссылке
    int i4 = pv->sz;        // доступ по указателю
}
```

2.3.2. Классы

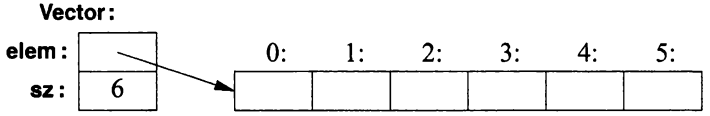
Определение данных в отрыве от производимых над данными операций имеет свои преимущества, позволяя использовать эти данные произвольным образом. Однако, требуется более тесная связь между данными и операциями для того, чтобы пользовательские данные походили на «настоящие типы». В частности, желательно, чтобы внутреннее представление данных было скрыто от пользователей, так как в этом случае упрощается их использование, гарантируется внутренняя согласованность данных, а также сохраняется возможность модифицировать структуру данных для ее постепенного совершенствования. Для этого нужно четко различать интерфейс к типу (используется всеми) и реализацию (в которой есть доступ к любым данным). Средством языка C++ для реализации изложенной концепции является *класс* (*class*). Класс состоит из *членов*, которые могут быть данными, функциями или типами. Интерфейс класса состоит из открытых членов (**public**), а закрытые члены (**private**) доступны только через интерфейс класса. Например:

```
class Vector {
public:
    Vector(int s) : elem{new double[s]}, sz{s} {}      // конструируем Vector
    double& operator[](int i) {return elem[i];}    // доступ к элементам: индексирование
    int size() {return sz;}
private:
    double* elem; // указатель на элементы
    int sz;       // число элементов
};
```

Имея этот класс, мы можем определить переменную нового типа **Vector**:

```
Vector v(6);        // Vector из 6 элементов
```

Объект типа **Vector** можно изобразить графически:



По сути, объект типа **Vector** является дескриптором, содержащим указатель на элементы (**elem**) и число элементов (**sz**). Количество элементов (6 в данном примере) может изменяться от одного объекта типа **Vector** к другому, и у объекта типа **Vector** может быть разное число элементов в разные моменты времени (§3.2.1.3). При этом размер самого объекта типа **Vector** все время один и тот же. Это традиционная для C++ схема работы с переменным количеством информации: дескриптор фиксированного размера, отсылающий к данным переменного размера, хранящимся где-то в другом месте (например, в свободной памяти, выделенной операцией **new**; §11.2). В Главе 3 подробно рассматриваются вопросы проектирования и использования таких объектов.

Возвращаясь к нашему примеру с типом **Vector**, заметим, что его представление (члены **elem** и **sz**) доступно только через интерфейс этого типа, формируемый открытыми членами: **Vector()**, **operator[]()** и **size()**. Пример функции **read_and_sum()** из §2.3.1 теперь упрощается:

```
double read_and_sum(int s)
{
    Vector v(s);    // вектор из s элементов
    for (int i=0; i!=v.size(); ++i)
        cin>>v[i]; // читаем элементы
```

```
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v[i];                    // сумма элементов
    return sum;
}
```

Функция с тем же именем, что и имя класса, называется *конструктором*: она используется для конструирования объектов класса. Таким образом, конструктор **Vector()** заменяет функцию **vector_init()** из §2.3.1. В отличие от обычной функции конструктор гарантированно применяется для инициализации объектов класса. Таким образом, определение конструктора устраняет проблему неинициализированных переменных.

Конструктор **Vector(int)** определяет, как именно конструируются объекты типа **Vector**. В частности, он требует целочисленного аргумента для своей работы. Этот аргумент задает число элементов. Конструктор инициализирует члены класса с помощью списка инициализации:

```
:elem{new double[s]}, sz{s}
```

Здесь мы сначала инициализируем **elem** указателем на **s** элементов типа **double**, получаемым из свободной памяти. А затем мы инициализируем **sz** числом **s**.

Доступ к элементам обеспечивается индексной функцией, именуемой **operator[]**. Она возвращает ссылку на соответствующий элемент (**double&**).

Функция **size()** сообщает пользователям о числе элементов.

Мы пока что опустили обработку ошибочных ситуаций, но мы вернемся к этому вопросу в §2.4.3. Аналогично, мы не рассмотрели механизм для возврата памяти, выделенной операцией **new** под массив элементов типа **double**; в §3.2.1.2 мы рассмотрим применение деструктора для элегантного решения этой проблемы.

2.3.3. Перечисления (enumerations)

В дополнение к классам язык C++ поддерживает простую форму пользовательских данных, для которых мы можем перечислить все значения:

```
enum class Color {red, blue, green};
enum class Traffic_light {green, yellow, red};

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Отметим, что элементы перечислений (например, **red**) находятся в области видимости своего класса (**enum class**), так что их можно повторно использовать в составе иных классов. Например, **Color::red** относится к классу **Color** и не имеет никакого отношения к **Traffic_light::red**.

Перечисления используются для представления небольших наборов целых значений. Они позволяют улучшить читаемость кода и позволяют избегать потенциальных ошибок, вероятность которых без перечислений была бы выше.

Ключевое слово **class**, стоящее после ключевого слова **enum**, утверждает, что перечисления являются типом, заключающим элементы (константы) в свою область видимости. Это помогает избегать путаницы с этими константами. В частности, мы не можем перепутать значения из **Traffic_light** и **Color**:

```
Color x = red;                // ошибка: чей red?
Color y = Traffic_light::red; // ошибка: этот red не из Color
Color z = Color::red;         // OK
```

Аналогично, мы не сможем осуществить неявное смешение значений из типа **Color** и целых значений:

```
int i = Color::red;   // ошибка: Color::red не int
Color c = 2;          // ошибка: 2 не есть Color
```

Если вы не хотите явно квалифицировать имена из перечислений и хотите, чтобы значения элементов перечислений легко согласовывались с типом **int** (без явного приведения типов), тогда уберите ключевое слово **class** и вы получите *простейшие перечисления* (plain **enum**) (§8.4.2).

По умолчанию, тип **enum class** поддерживает присваивания, инициализацию и сравнения (например, == и <) (§2.2.2). Так как это в полной мере пользовательский тип, мы можем определять для него операции:

```
Traffic_light& operator++(Traffic_light& t)
    // префиксный инкремент: ++
{
    switch (t) {
    case Traffic_light::green: return t=Traffic_light::yellow;
    case Traffic_light::yellow:    return t=Traffic_light::red;
    case Traffic_light::red:       return t=Traffic_light::green;
    }
}

Traffic_light next = ++light;   // следующ. будет Traffic_light::green
```

Имеется и менее строгий тип — *простейшее перечисление* (plain **enum**) (§8.4.2).

2.4. Модульность

Программа на языке C++ состоит из многих раздельно разрабатываемых частей, таких как функции (§2.2.1, Глава 12), пользовательские типы (§2.3, §3.2, Глава 16), иерархии классов (§3.2.4, Глава 20) и шаблоны (§3.4, Глава 23). Очень важно четко определить, как эти части взаимодействуют между собой. В первую очередь нужно отделить интерфейс к той или иной части программы от ее реализации. Язык C++ вводит интерфейсы через объявления. *Объявление* специфицирует все, что нужно для использования функции или типа. Вот пример:

```
double sqrt(double);      // ф-я принимает double и  возвращает double

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;         // elem указывает на массив из sz эл-ов double
    int sz;
};
```

Важно то, что *определения* функций (то есть код, составляющий тела функций), находятся в другом месте. В этом примере можно было бы и представление типа **Vector** вынести в иное место, но мы этим займемся позже (абстрактные типы; §3.2.2). Определение функции **sqrt()** может выглядить следующим образом:

```
double sqrt(double d)   // определение sqrt()
{
    // ... алгоритм из книги по математике ...
}
```

Для типа **Vector** мы должны определить все три метода (member functions):

```
Vector::Vector(int s)                    // определение конструктора
    : elem{new double[s]}, sz{s}         // инициализация членов
{
}

double& Vector::operator[](int i)        // определение индексации
{
    return elem[i];
}

int Vector::size()                       // определение size()
{
    return sz;
}
```

Мы должны определить функции для типа **Vector**, но не должны определять функцию **sqrt()**, так как она входит в стандартную библиотеку. Здесь, однако, нет никаких особенностей: библиотека это просто некий код, который нам можно использовать, и этот код написан при помощи тех же языковых средств, что и те средства, которые мы здесь изучаем.

2.4.1. Раздельная компиляция

C++ поддерживает раздельную компиляцию, где пользовательский код видит лишь объявления типов и используемых функций. Определения этих типов и функций находятся в отдельных исходных файлах и компилируются по-отдельности. Это позволяет организовать код программы в виде множества полунезависимых фрагментов кода. В результате удается минимизировать время компиляции и строго разделить между собой логически отличающиеся части программы (попутно минимизируя потенциальные ошибки). Библиотеки часто являются просто раздельно скомпилированными фрагментами кода (например, функций).

В типичном случае мы помещаем объявления, специфицирующие интерфейс к модулю, в файл с именем, отражающим его предполагаемое использование. Например:

```
// Vector.h:

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;       // elem указывает на массив из sz doubles
    int sz;
};
```

Это объявление следует поместить в файл **Vector.h**, а пользователи будут *включать* в свой код этот файл, называемый *заголовочным файлом* (*header file*), с тем, чтобы иметь доступ к этому интерфейсу. Например:

```
// user.cpp:

#include "Vector.h"      // получаем интерфейс для Vector
#include <cmath>         // интерфейс математической функции sqrt()
using namespace std;     // делаем члены std видимыми (§ 2.4.2)
double sqrt_sum(Vector& v)
{
```

```
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=sqrt(v[i]);          // сумма квадратных корней
    return sum;
}
```

Чтобы помочь компилятору гарантировать согласованность, **cpp**-файл с реализацией типа **Vector** также включает **h**-файл, объявляющий интерфейс:

```
// Vector.cpp:

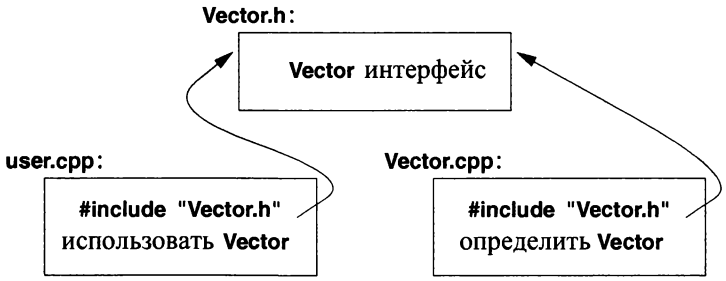
#include "Vector.h"       // получаем интерфейс

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}
```

Код в файле **user.cpp** и код в файле **Vector.cpp** используют информацию об интерфейсе, представленную в файле **Vector.h**. В остальном эти два файла независимы и могут компилироваться отдельно друг от друга. Графически это можно проиллюстрировать следующим образом:



Строго говоря, использование раздельной компиляции не является вопросом языка; это способ оптимального использования его конкретной реализации. И это вопрос огромной практической важности. Желательно по максимуму обеспечить модульность, выразить ее логически средствами языка, а затем использовать на уровне файлов для эффективной раздельной компиляции (Глава 14, Глава 15).

2.4.2. Пространства имен

Помимо функций (§2.2.1, Глава 12), классов (Глава 16) и перечислений (§2.3.3, §8.4) язык C++ предоставляет еще и *пространства имен* (*namespaces*, Глава 14) в качестве механизма группировки объявлений и в качестве механизма гарантирования, что их имена не вступят в противоречие с другими именами. Например, я мог бы поэкспериментировать с собственным типом для комплексных чисел (§3.2.1.1, §18.3, §40.4):

```
namespace My_code {
    class complex {/* ... */};
    complex sqrt(complex);
    // ...
    int main();
}

int My_code::main()
{
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
};

int main()
{
    return My_code::main();
}
```

Помещая код в пространство имен **My_code**, я избегаю конфликта с именами стандартной библиотеки из пространства имен **std** (§4.1.2). Это не напрасная предосторожность, ибо стандартная библиотека также обеспечивает поддержку арифметики комплексных чисел (§3.2.1.1, §40.4).

Обеспечить доступ к имени из некоторого пространства имен можно, квалифицировав его именем этого пространства имен (например, **std::cout** или **My_code::main**). Настоящая функция **main()** определяется в глобальном пространстве имен, то есть не в локальном по отношению к определяемому пространству имен, классу или функции. Для доступа к именам стандартной библиотеки можно использовать **using**-директиву (§14.2.3):

```
using namespace std;
```

Пространства имен используются преимущественно для организации больших программных компонентов, таких как библиотеки. Они упрощают сборку программы из частей, разработанных по-отдельности.

2.4.3. Обработка ошибок

Обработка ошибок — это большая и сложная тема, которая уходит своими корнями далеко за пределы языка и касается технологий программирования и используемых инструментов. И тем не менее, C++ предоставляет некоторые средства, которые могут помочь в этой теме. Наиболее радикальным средством является система типов сама по себе. Вместо того, чтобы мучительно и долго строить приложения исключительно на встроенных типах (таких как **char**, **int** или **double**) и операторах (например, **if**, **while** или **for**), мы разрабатываем типы, подходящие для нашего приложения (например, **string**, **map** или **regex**), а также алгоритмы (например, **sort()**, **find_if()** и **draw_all()**). Такие более высокоуровневые конструкции упрощают программирование, ограничивают возможности ошибаться (например, маловероятно, что вы примените обход дерева к диалоговому окну) и повышают шанс для компилятора выловить ошибки. Большинство конструкций языка C++ предназначены для проектирования и реализации высокоуровневых абстракций (то есть пользовательских типов и нацеленных на них алгоритмов). Следствием модульности и абстрагирования (в особенности из-за применения библиотек) является то обстоятельство, что точка обнаружения ошибки во время выполнения программы отделена от точки, где эту ошибочную ситуацию можно обработать. По мере роста программы и особенно в случае использования библиотек стандарты для обработки ошибок становятся все более важными.

2.4.3.1. Исключения (exceptions)

Рассмотрим снова пример с типом **Vector**. Что следует предпринять, если мы затребуем доступ к элементу за пределами актуального диапазона элементов вектора из §2.3.2?

- Разработчик типа **Vector** не может знать, что нужно пользователю в таком случае (он даже не знает, в какой программе **Vector** будет использоваться).
- Пользователь типа **Vector** не может обнаружить проблему (если бы мог, то проблемы не возникало бы вообще).

В итоге остается лишь одно: разработчик **Vector** должен детектировать попытки подобного рода доступа и сообщать об этом пользователю. А пользователь в таком случае должен сам решать, как ему поступать с этим далее. Например, **Vector::operator[]()** может обнаружить попытку доступа за пределы диапазона элементов и сгенерировать исключение типа **out_of_range**:

```
double& Vector::operator[](int i)
{
    if (i<0 || size()<=i) throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Оператор **throw** передает управление обработчику исключений типа **out_of_range** в некоторую функцию, откуда прямо или косвенно был выполнен вызов **Vector::operator[]()**. Для этого реализация должна раскрутить стек вызовов функций с тем, чтобы попасть назад в контекст вызова (§13.5.1). Например:

```
void f(Vector& v)
{
    // ...
    try {// с исключениями работает обработчик (см. ниже)

        v[v.size()] = 7; // попытка выйти за конец v
    }
    catch (out_of_range) {// oops: out_of_range ошибка
        // ... обрабатываем ошибку диапазона ...
    }
    // ...
}
```

Помещаем код, который мы подозреваем в необходимости обработки исключений, в так называемый **try**-блок. Показанная в примере попытка присваивания для **v[v.size()]** завершится неудачей. Поэтому управление будет передано в **catch**-блок, содержащий обработчик исключений (ошибок) типа **out_of_range**. Тип **out_of_range** определен в стандартной библиотеке и на самом деле используется при доступе к элементам некоторых контейнеров.

Применение механизмов обработки исключений упрощает обработку ошибок, делает ее более систематизированной и более читаемой. Смотри Главу 13 для дальнейшего рассмотрения темы, более подробных деталей и примеров.

2.4.3.2. Инварианты

Использование исключений для сигнализирования о попытке доступе за пределами актуального диапазона является примером проверки аргументов функции и отказа от работы из-за того, что базовое предположение (предусловие, *precondition*) не выполнено. Если бы мы формально определяли операцию индексирования для типа **Vector**, мы бы сказали что-нибудь типа «индекс должен быть внутри диапазона [**0:size()**)»; именно это мы

и тестировали в нашем **operator[]()**. Определяя функции, мы должны учитывать их предусловия и, если возможно, проверять их (см. §12.4, §13.4).

Итак, **operator[]()** оперирует над объектами типа **Vector**, но он не может сделать что-либо осмысленное, если члены класса **Vector** не имеют «разумных» зачений. В частности, мы говорим что «**elem** указывает на массив из **sz** чисел типа **double**», но мы это говорим лишь в комментарии. Утверждение о том, что именно должно иметь место для корректности класса, называется *инвариантом класса*, или просто *инвариантом*. Это задача конструктора устанавливать инвариант для класса (чтобы функции-члены могли на него положиться), а функциям-членам должно проверять выполнение инварианта при выходе из функций. К сожалению, конструктор типа **Vector** выполняет это лишь частично. Он надлежащим образом инициализирует поля класса, но он не может проверить, насколько передаваемые ему аргументы имеют смысл. Рассмотрим следующее:

```
Vector v(-27);
```

С большой вероятностью это приведет к хаосу.

Вот более подходящее определение конструктора:

```
Vector::Vector(int s)
{
    if (s<0) throw length_error{};
    elem = new double[s];
    sz = s;
}
```

Я использую исключение **length_error** из стандартной библиотеки для сообщения об отрицательном значении числа элементов потому, что некоторые операции стандартной библиотеки используют это исключение о проблемах подобного рода. Если операция **new** не может найти необходимую память, она генерирует исключение **std::bad_alloc**. Мы можем написать:

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error) {
        // случай отрицательного размера
    }
    catch (std::bad_alloc) {
        // исчерпание памяти
    }
}
```

Вы можете определить собственные классы, которые будут использоваться в качестве исключений, и использовать их, чтобы переносить информацию из точки обнаружения ошибки в точку, где эта ошибка может быть обработана (§13.5).

Часто функция не может далее выполнять свою работу после того, как сгенерировано исключение. В таком случае обработка исключения сводится к минимальной зачистке и повторной генерации исключения (§13.5.2.1).

Понятие инвариантов располагается в самом центре при проектировании классов, а предусловия выполняют ту же роль при проектировании функций. Инварианты

- помогают нам понять, чего именно мы хотим
- заставляют нас быть конкретными, что помогает получать корректный код (после отладки и тестирования)

Понятие инвариантов лежит в основе понятия управления ресурсами, поддерживаемое конструкторами (§2.3.2) и деструкторами (§3.2.1.2, §5.2). См. также §13.4, §16.3.1 и §17.2.

2.4.3.3. Статические утверждения (static assertions)

Исключения сообщают об ошибках во время работы программы. Но если ошибка может быть обнаружена во время компиляции, то это желательно сделать именно на этом этапе. Тут нам помогают система типов и средства определения интерфейсов к пользовательским типам. Мы можем проводить простые проверки любых свойств, известных во время компиляции, и сообщать о них как об ошибках компиляции. Например:

```
static_assert(4<=sizeof(int), "integers are too small");  // проверка размера целого
```

Сообщение **integers are too small** будет выведено, если не будет выполняться условие **4<=sizeof(int)**, то есть если **int** на вашей системе меньше 4 байтов. Мы называем такие предположения *статическими утверждениями*.

Механизм **static_assert** может применяться к чему-либо, что может быть выражено в терминах константных выражений (§2.2.3, §10.4). Например:

```
constexpr double C = 299792.458;        // km/s

void f(double speed)
{
    const double local_max = 160.0/(60*60);     // 160 km/h == 160.0/(60*60) km/s

    static_assert(speed<C,"can't go that fast"); // ошибка: speed должна быть constant
    static_assert(local_max<C,"can't go that fast");  // OK

    // ...
}
```

В общем случае, **static_assert(A, S)** выводит **S** в качестве сообщения об ошибке компиляции, если **A** не выполняется.

Наиболее важным случаем использования **static_assert** является применение утверждений о типах, используемых в качестве параметров в обобщенном программировании (§5.4.2, §24.3).

Про утверждения, проверяемые во время выполнения, см. §13.4.

2.5. Заключение

Темы, рассмотренные в данной главе, примерно соответствуют материалу Части II (Главы 6–15). Это те части C++, которые лежат в основании всех технологий программирования и стилей, поддерживаемых языком C++. Опытным программистам на языках C и C++ стоит принять во внимание, что эти основы не полностью коррелируют с подмножествами C или C++98 стандарта C++11.

2.6. Советы

[1] Не паникуйте! Все станет понятно в свое время; §2.1.

[2] Чтобы писать хорошие программы, не обязательно знать каждую деталь языка C++; §1.3.1.

[3] Фокусируйтесь на технологиях программирования, а не на свойствах языка; §2.1

3

Обзор языка C++: Механизмы абстракции

Не паниковать!
— Дуглас Адамс

- Введение
- Классы

  Конкретные типы; Абстрактные типы; Виртуальные функции; Классовые иерархии
- Копирование и перемещение

  Копирование контейнеров; Перемещение контейнеров; Управление ресурсами; Подавление операций
- Шаблоны

  Параметризуемые типы; Шаблоны функций; Функциональные объекты; Шаблоны с переменным числом параметров; Синонимы
- Советы

3.1. Введение

Настоящая глава посвящена поддержке языком C++ механизмов абстракции и управления ресурсами без углубления в многочисленные детали. В ней неформальным образом представлены методы, с помощью которых можно определять и использовать новые типы (типы, определяемые пользователем или пользовательские типы). В частности, в данной главе рассматриваются основополагающие свойства, методы реализации и средства языка, используемые для построения *конкретных классов*, *абстрактных классов* и *классовых иерархий*. Шаблоны представлены в качестве механизма параметризации типов и алгоритмов другими типами и алгоритмами. Вычисления с пользовательскими и встроенными типами оформляются в виде функций, которые иногда обобщаются до *шаблонных функций* и *функциональных объектов*. Все вместе это поддерживает стили программирования, известные как *объектно-ориентированное программирование* и *обобщенное программирование*. В последующих двух главах представлены примеры средств стандартной библиотеки и их практическое применение.

Предполагается, что вы уже программировали ранее. В противном случае, сначала прочитайте какой-либо иной учебник, например «Программирование: Принципы и практическое использование C++» [Страуструп, 2009], и после этого возвращайтесь к настоящей главе. Но даже если вы уже программировали, ваш язык программирования или типы программ, которые вы писали, могут очень сильно отличаться от стиля языка C++. Если наш «молниеносный стиль» смутит вас, переходите к более систематическому изложению, которое начинается с Главы 6.

Как и ранее в Главе 2, мы здесь представляем C++ в интегральном виде, а не в виде отдельных «слоев пирога». Так что мы не выделяем отдельно те средства, которые имеются в языке C, в стандарте C++98 и не противопоставляем их C++11. Историческую информацию можно найти в §1.4 и в Главе 44.

3.2. Классы

Центральной конструкцией языка C++ является *класс*. Класс это тип, определяемый пользователем для отражения в коде программы некоторой концепции. Во всех случаях, когда при разработке программы мы встречаемся с идеей, сущностью, полезной концепцией, мы пытаемся представить их в виде класса, который непосредственно присутствует в коде программе, а не только в голове, документации или в комментариях. Программу, которая строится на базе прозрачного набора классов, проще понять и привести в корректное состояние, нежели программу, которая напичкана кодом на базе встроенных типов. В частности, многие библиотеки определяют общеполезные классы.

По существу, в языке C++ все, кроме фундаментальных типов, операций и операторов, предназначено для того, чтобы можно было лучше определять классы и было бы удобнее их использовать. Когда я говорю «лучше», я имею в виду «более корректно», с возможностью поддержки, более эффективно и элегантно, с возможностью легче читать код, понимать его и рассуждать в связанных с классами терминах. Большинство технологий программирования полагается на проектирование и реализацию определенных типов классов. Вкусы и нужды программистов варьируются в больших пределах. Здесь мы рассмотрим базовые понятия для трех важных типов классов:

- Конкретные классы (§3.2.1)
- Абстрактные классы (§3.2.2)
- Классы, образующие иерархии (§3.2.4)

Огромное количество полезных классов часто оказываются одним из этих типов, или же представляют собой некоторую их комбинацию.

3.2.1. Конкретные типы

Основная идея *конкретных классов* состоит в том, что они ведут себя «прямо как встроенные типы». Например, тип комплексных чисел или тип чисел с бесконечной точностью вполне соответствуют встроенному числовому типу **int**, за исключением конечно того обстоятельства, что у них своя семантика и набор операций. Аналогично, типы **vector** или **string** вполне аналогичны встроенным массивам, но только с более удобным поведением (§4.2, §4.3.2, §4.4.1).

Определяющей чертой конкретных типов является то, что их представление является частью их определения. Во многих важных случаях, таких как **vector**, представление сводится к одному или нескольким указателям на данные, которые хранятся в отдельных местах, но представление присутствует в каждом объекте конкретного класса. Это позволяет реализовывать их эффективно по объему памяти и по времени исполнения. В частности, это позволяет:

- помещать объекты конкретных типов в стек, в статически выделяемую память, а также в другие объекты (§6.4.2);
- напрямую обращаться к объектам (не только через указатели или ссылки);
- немедленно и в полной мере инициализировать объекты (используя конструкторы; §2.3.2);
- копировать объекты (§3.3)

Представление может быть закрытым (как в случае **Vector** из §2.3.2) и доступным лишь с помощью предназначенных для этого методов (функций-членов), но оно реально присутствует, так что при существенном изменении пользователю придется перекомпилировать код. Это цена, которую приходится платить за то, чтобы иметь конкретные типы, которые ведут себя так же, как встроенные. Для типов, которые не подвергаются частым модификациям, а также в тех случаях, когда полезно применять локальные переменные ради ясности и эффективности, это приемлемо и часто оптимально. Для увеличения гибкости конкретные типы могут сохранять большую часть своего представления в свободной памяти (динамической памяти, в куче) с доступом из другой части, находящейся непосредственно в самом классовом объекте. Так реализованы типы **vector** и **string**; их можно рассматривать как дескрипторы ресурсов с тщательно проработанными интерфейсами.

3.2.1.1. Арифметический тип

Классическим примером арифметического пользовательского типа является тип **complex**:

```
class complex {
    double re, im;  // представление: два double
public:
    complex(double r, double i) : re{r}, im{i} {}    // complex из двух скаляров
    complex(double r) : re{r}, im{0} {}              // complex из одного скаляра
    complex() : re{0}, im{0} {}                      // по умолчанию complex: {0,0}

    double real() const {return re;}
    void real(double d) {re=d;}
    double imag() const {return im;}
    void imag(double d) {im=d;}

    complex& operator+=(complex z) {re+=z.re, im+=z.im; return *this;}  // добавляем
                                                                        // к re и im и вовзвращаем результат
    complex& operator-=(complex z) {re-=z.re, im-=z.im; return *this;}

    complex& operator*=(complex);     // определено вне класса
    complex& operator/=(complex);     // определено вне класса
};
```

Это слегка упрощенная версия типа **complex** из стандартной библиотеки (§40.4). Определение класса содержит лишь операции для доступа к представлению (внутренним данным). Представление же, само по себе, очень простое и традиционное. По практическим соображениям оно должно быть совместимым с вариантом, соответствующим версии языка Fortran 50-летней давности, и еще нужно обеспечить традиционные операции. Помимо высказанных логических требований **complex** должен быть эффективным, иначе его никто не будет использовать. Из этого следует, что простые операции должны быть встраиваемыми (**inline**). То есть, простые операции (вроде конструкторов, **imag()** и **+=**) должны реализовываться в машинном коде непосредственно без функциональных вызовов. По умолчанию, если классовый метод полностью определяется внутри определения класса, то он автоматически реализуется как встраиваемый. Образцовый **complex** из стандартной библиотеки тщательно продуман с точки зрения встраивания его методов.

Конструктор, который вызывается без аргументов, называется *конструктором по умолчанию*. То есть **complex()** есть конструктор по умолчанию для типа **complex**. Определяя конструктор по умолчанию, вы избегаете возможности создавать неинициализированные переменные этого типа.

Спецификаторы **const** у функций, возвращающих действительную и мнимую части, означают, что эти функции не подвергают изменениям объект, для которого они вызваны.

Существует много полезных операций, не требующих прямого доступа к представлению **complex**, так что их можно определять отдельно от определения класса:

```
complex operator+(complex a, complex b) {return a+=b;}
complex operator-(complex a, complex b) {return a-=b;}
complex operator-(complex a) {return {-a.real(), -a.imag()};} // унарный минус
complex operator*(complex a, complex b) {return a*=b;}
complex operator/(complex a, complex b) {return a/=b;}
```

Здесь я отталкиваюсь от факта, что аргумент, передаваемый по значению, копируется. Таикм образом, я могу модифицировать этот аргумент без какого-либо воздействия на экземпляр, имеющийся в распоряжении вызвавшего функцию, и использовать результат вычислений для возврата.

Определения операций == и!= абсолютно прозрачны:

```
bool operator==(complex a, complex b)       // равно
{
    return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)       // не равно
{
    return !(a==b);
}

complex sqrt(complex);

// ...
```

Использовать класс **complex** можно следующим образом:

```
void f(complex z)
{
    complex a {2.3};          // создаем {2.3,0.0} из 2.3
    complex b {1/a};
    complex c {a+z*complex{1,2.3}};
    // ...
    if (c != b)
        c = -(b/a)+2*b;
}
```

Компилятор конвертирует операции над числами типа **complex** в надлежащие функциональные вызовы. Например, **c!=b** означает вызов **operator!=(c, b)**, а **1/a** означает вызов **operator/(complex{1}, a)**.

Пользовательские операции (перегруженные операции) должны использоваться с осторожностью и стандартным образом. Синтаксис полностью определен языком, так что не получится определить унарную операцию **/**. Также отсутствует возможность переопределять операции для встроенных типов: вы не сможете определить свою операцию **+** для вычитания чисел типа **int**.

3.2.1.2. Контейнеры

Контейнер — это объект, содержащий коллекцию (набор) элементов, поэтому мы называем **Vector** контейнером, ибо объекты этого типа суть контейнеры. В соответствии

с определением из §2.3.2 **Vector** не просто бессмысленный контейнер элементов типа **double**: для него легко определить полезный инвариант (§2.4.3.2), обеспечить доступ к элементам с проверкой корректности диапазона (§2.4.3.1) и предоставить **size()**, чтобы мы могли спокойно осуществлять итерацию по его элементам. Однако, у него есть серьезный пробел: он выделяет память под элементы операцией **new**, но при этом он никогда не освобождает эту память. Это серьезный недостаток, ибо несмотря на то, что C++ определяет интерфейс к сборщику мусора (§34.5), совершенно не гарантируется, что он присутствует в системе и будет возвращать неиспользуемую память обратно. Даже если где-то и присутствует сборщик мусора, все равно может оказаться предпочтительным самим точно контролировать уничтожение объектов (§13.6.4) ради прозрачности кода или же в целях эффективности. Итак, есть нужда в механизме, который гарантировал бы, что память, выделенная конструктором, возвращается системе обратно; этот механизм называется *деструктором*:

```
class Vector {
private:
    double*elem;        // elem указывает на массив из sz эл-ов типа double
    int sz;
public:
    Vector(int s) : elem{new double[s]}, sz{s}    // конструктор: захват ресурсов
    {
        for (int i=0; i!=s; ++i) elem[i]=0;       // инициализация элементов
    }

    ~Vector() {delete[] elem;}                    // деструктор: освобождение ресурсов

    double& operator[](int i);
    int size() const;
};
```

Имя деструктора строится из имени класса и префикса ~; это дополнение к конструктору. Конструктор класса **Vector** выделяет память (в куче), используя операцию **new**. Деструктор выполняет очистку памяти, используя операцию **delete**. Все это выполняется без явного вмешательства пользователя класса **Vector**. Пользователь создает и использует объекты класса **Vector** практически так, как если бы они были переменными встроенных типов. Вот пример:

```
void fct(int n)
{
    Vector v(n);

    // ... используем v ...

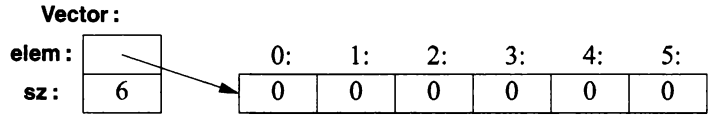
    {
        Vector v2(2*n);
        // ... используем v и v2 ...
    } // здесь v2 уничтожается

    // ... используем v ..

} // здесь v уничтожается
```

Тип **Vector** подчиняется тем же правилам именования, области видимости, времени жизни и т.д., что и встроенные типы вроде **int** или **char**. За подробностями о том, как контролировать время жизни объекта, обратитесь к §6.4. Рассмотренный нами тип **Vector** упрощен в том числе и за счет отбрасывания обработки ошибок; см. §2.4.3.

Комбинация конструктор/деструктор лежит в основе многих элегантных приемов программирования. В частности, она лежит в основе общих технологий управления ресурсами (§5.2, §13.3). Рассмотрим графическую иллюстрацию для **Vector**:



Конструктор надлежащим образом выделяет память под элементы **Vector** и инициализирует их. А деструктор освобождает ранее выделенную конструктором память. *Модель дескриптор-данные* часто применяется для управления данными, которые изменяются в размере в течение жизни объекта. Техника захвата ресурсов в конструкторе и освобождение их деструктором, называемая *Получение Ресурса Есть Инициализация* (*Resorce Acquisition Is Initialization* — RAII), позволяет избавиться от «голых **new**» в коде общего назначения, запрятав их глубоко в реализации хорошо запроектированных абстракций. Аналогичным образом нужно упрятывать и «голые **delete**». Это делает код более защищенным от разных ошибок и особенно от утечек памяти (§5.2).

3.2.1.3. Инициализация контейнеров

Контейнер предназначен для хранения элементов, так что совершенно очевидно, что мы нуждаемся в удобных способах помещения элементов в контейнер. Мы можем создать объект типа **Vector** с указанным числом элементов, а помещать их можем туда потом. Но это не самое элегантное решение. Упомянем здесь пару более приемлемых решений:

- *Конструктор* со *списком инициализации*: инициализирует списком элементов
- **push_back()**: добавляет новый элемент в конец последовательности

Вот возможное объявление:

```
class Vector {
public:
    Vector(std::initializer_list<double>); // инициализация списком
    // ...
    void push_back(double);      // добавляем элемент с конца увеличивая size на единицу
    // ...
};
```

Метод **push_back()** полезен для ввода произвольного числа элементов. Вот пример:

```
Vector read(istream& is)
{
    Vector v;
    for (double d; is>>d;)    // читаем вещественн. значения в d
        v.push_back(d);       // добавляем d к v
    return v;
}
```

Цикл ввода завершается достижением конца файла или же ошибкой формата. Пока этого не произошло, каждое прочитанное число добавляется к **Vector**, так что поле **size** для объекта **v** равно числу прочитанных элементов. Реализация **push_back()** обсуждается в §13.6.4.3. Создание для типа **Vector** перемещающего конструктора, при котором возможен недорогой возврат потенциально огромного объема данных из функции **read()**, обсуждается в §3.3.2.

Использованный в списке инициализации конструктора **std::initializer_list** является типом из стандартной библиотеки, известным компилятору: когда мы используем {}-список,

например {1,2,3,4}, компилятор создает объект типа initializer_list, который и будет использоваться в программе. Таким образом, мы можем написать:

```
Vector v1 = {1,2,3,4,5};          // v1 имеет 5 элементов
Vector v2 = {1.23, 3.45, 6.7, 8}; // v2 имеет 4 элемента
```

Конструктор для типа Vector, работающий со списком инициализации, может быть определен следующим образом:

```
Vector::Vector(std::initializer_list<double> lst)     // инициализируем списком
    : elem{new double[lst.size()]}, sz{lst.size()}
{
    copy(lst.begin(), lst.end(), elem);          // копируем из lst в elem
}
```

3.2.2. Абстрактные типы

Такие типы, как complex или Vector, называются *конкретными типами*, потому что их представление является частью их определения. В этом они напоминают встроенные типы. Напротив, *абстрактными типами* называются типы, которые полностью изолируют пользователя от деталей реализации. Для этого интерфейс отделяется от представления; в итоге мы расстаемся с локальными переменными. Так как мы ничего не знаем о представлении абстрактного типа (даже о размере), то мы должны создавать объекты в свободной памяти (§3.2.1.2, §11.2) и обращаться к ним по указателям или ссылкам (§2.2.5, §7.2, §7.7).

Сначала определим интерфейс к классу Container, который мы создадим в виде более абстрактной версии Vector:

```
class Container {
public:
    virtual double& operator[](int) = 0;     // чисто виртуальная функция
    virtual int size() const = 0;            // константная функция-член (§ 3.2.1.1)
    virtual ~Container() {}                  // деструктор (§ 3.2.1.2)
};
```

Этот класс служит чистым интерфейсом к более специфическим контейнерам, которые мы определим позже. Слово virtual означает «может быть переопределено позже в производном классе». Ничего удивительно в том, что функция, определенная с virtual, называется *виртуальной функцией*. Класс, который будет наследоваться от Container, обеспечит реализацию заявленного интерфейса. Необычный синтаксис =0 говорит о том, что функция является *чисто виртуальной*; производный от Container класс *обязан* определить эту функцию. Таким образом, невозможно создать объект типа Container; этот тип лишь задает интерфейс к классу, производному от Container, который и будет реализовывать функции operator[]() и size(). Класс, содержащий чисто виртуальную функцию, называется *абстрактным классом*.

Тип Container можно использовать следующим образом:

```
void use(Container& c)
{
    const int sz = c.size();

    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

Обратите внимание на то, как функция use() использует интерфейс Container, полностью игнорируя детали реализации. Она использует size() и [], не имея ни малейшей идеи

о том, какой именно тип обеспечивает их реализацию. Класс, который задает интерфейс ко множеству других классов, часто называется полиморфным типом (§20.3.2).

Класс **Container** не имеет конструктора, что типично для абстрактных классов. В конце концов, у него нет никаких данных для инициализации. С другой стороны, у **Container** есть деструктор, и этот деструктор **virtual**. Опять-таки, это типично для абстрактных классов, поскольку ими манипулируют через ссылки или указатели, и тот, кто уничтожает **Container** через указатель, не знает, какие ресурсы захвачены реализацией; см. §3.2.4.

Класс, который реализует интерфейс, заданный абстрактным классом **Container**, может использовать конкретный класс **Vector**:

```
class Vector_container: public Container {    // Vector_container реализует Container
    Vector v;
public:
    Vector_container(int s) : v(s) {}    // Вектор из s элементов
    ~Vector_container() {}

    double& operator[](int i) {return v[i];}
    int size() const {return v.size();}
};
```

Здесь конструкция **:public** может быть прочитана как «наследуется от» или как «является подтипом». Про класс **Vector_container** говорят, что он *унаследован* от класса **Container**, а класс **Container** называют *базовым классом* для **Vector_container**. По альтернативной терминологии классы **Vector_container** и **Container** называются *подклассом* и *суперклассом*, соответственно. Говорят, что производный класс наследует члены своего базового класса, так что использование базовых и производных классов часто обозначают термином *наследование*.

Про функции **operator[]()** и **size()** говорят, что они *замещают* (*override*) соответствующие члены базового класса **Container** (§20.3.2). Деструктор (**~Vector_container()**) замещает деструктор базового класса (**~Container()**). Заметьте, что деструктор для поля данных **v** (то есть **~Vector()**) вызывается неявно деструктором **~Vector_container()**.

Для того, чтобы функция вроде **use(Container&)** могла использовать **Container** при полном неведении о реализации, некоторая другая функция должна создать объект, над которым и оперирует функция **use(Container&)**. Например:

```
void g()
{
    Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    use(vc);
}
```

Так как функция **use()** ничего не знает о **Vector_container**, а знает лишь интерфейс **Container**, она будет работать точно так же и с другой реализацией **Container**. Вот пример:

```
class List_container: public Container {// List_container реализует Container
    std::list<double> ld; // (стандартный) list из double (§ 4.4.2)
public:
    List_container() {}  // пустой список
    List_container(initializer_list<double> il) : ld{il} {}
    ~List_container() {}
    double& operator[](int i);
    int size() const {return ld.size();}

};
```

```
double& List_container::operator[](int i)
{
    for (auto& x : ld) {
        if (i==0) return x;
        --i;
    }
    throw out_of_range("List container");
}
```

Здесь представление использует **list<double>** из стандартной библиотеки. В обычных случаях я не стал бы реализовывать контейнер с операцией индексирования на базе **list**, ибо эффективность индексирования на этом типе не сравнима с таковой для **vector**; я просто хотел продемонстрировать реализацию, радикально отличающуюся от обычной.

Создаем теперь объект типа **List_container** и отдаем его функции **use()**:

```
void h()
{
    List_container lc = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    use(lc);
}
```

Весь фокус в том, что функция **use(Container&)** не знает, имеет ли ее аргумент тип **Vector_container**, или **List_container**, или же еще какой тип; этой функции и не нужно это знать. Она может работать с любым **Container**, так как знает она именно этот интерфейс. Поэтому функцию **use(Container&)** не нужно перекомпилировать, если например реализация **List_container** изменится, или же будет применен совершенно новый класс, производный от **Container**.

Обратной стороной такой гибкости является необходимость манипулировать объектами через указатели или ссылки (§3.3, §20.4).

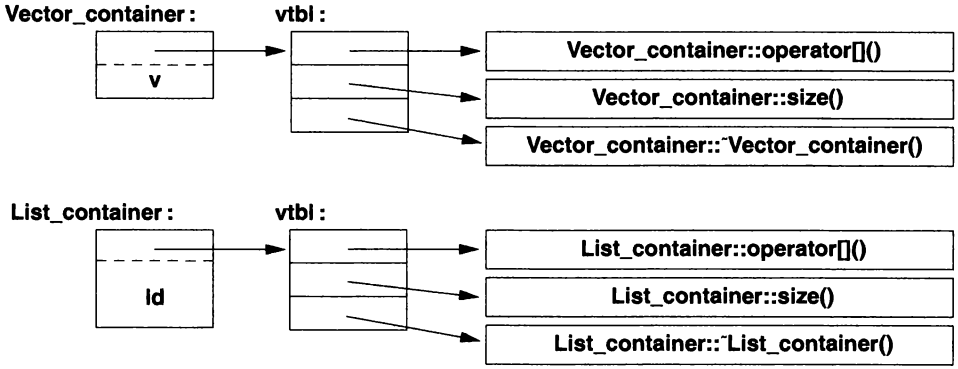
3.2.3. Виртуальные функции

Вернемся снова к коду, использующему **Container**:

```
void use(Container& c)
{
    const int sz = c.size();

    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

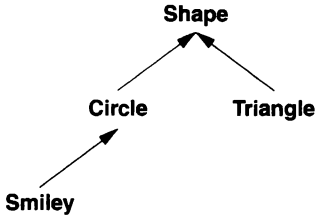
Каким образом вызов **c[i]** в теле функции **use()** соотносится с правильным **operator[]()**? Когда функция **h()** вызывает **use()**, **operator[]()** для типа **List_container** должен быть вызван. А когда **g()** вызывает **use()**, **operator[]()** должен быть вызван уже для типа **Vector_container**. Чтобы иметь возможность правильно разрешать вызовы, объекты **Container** должны содержать информацию, позволяющую во время исполнения программы вызвать надлежащую функцию. Обычным способом достижения этого эффекта является прием, когда компилятор конвертирует имя виртуальной функции в индекс в таблице указателей на функции. Эту таблицу обычно называют *таблицей виртуальных функций* или просто **vtbl**. Каждый класс с виртуальными функциями имеет свою собственную таблицу **vtbl**, идентифицирующую его виртуальные функции. Это можно проиллюстрировать графически следующим образом:



Функции в таблице **vtbl** обеспечивают корректное использование объекта, даже если размер объекта и расположение его данных неизвестны вызывающему коду. Последнему нужно лишь иметь доступ к указателю на **vtbl** для **Container** и еще нужно знать индексы виртуальных функций класса. Такой механизм вызова виртуальных функций можно сделать почти столь же эффективным, как и механизм обычного вызова функций (отличие в пределах 25%). Он требует дополнительно одного указателя на каждый объект класса с виртуальными функциями и одну таблицу **vtbl** для каждого такого класса.

3.2.4. Иерархии классов

Пример с **Container** демонстрирует простую иерархию классов. *Иерархия классов* — это набор классов, организованных в виде решетки наследования (при помощи конструкции : **public**). Мы используем иерархии классов для отражения концепций с иерархическими взаимосвязями, такие как «Пожарная машина есть вид грузовой машины, которая в свою очередь есть подвид транспортного средства», или «Улыбающееся лицо есть вид окружности, которая есть подвид фигуры». Вполне допустимы мощные иерархии, состоящие из сотен классов, взаимосвязанных вширь и вглубь. В качестве полуреалистичного примера рассмотрим фигуры на экране:



Стрелка изображает связь наследования. Например, класс **Circle** наследует от **Shape**. Чтобы отразить эту простую диаграмму в коде, мы должны сначала объявить класс, определяющий наиболее общие черты всех фигур:

```
class Shape {
public:
    virtual Point center() const =0;     // чисто виртуальная
    virtual void move(Point to) =0;

    virtual void draw() const = 0;       // рисуем на текущем "Canvas"
    virtual void rotate(int angle) = 0;

    virtual ~Shape() {}  // деструктор
    // ...
};
```

Естественно, что этот интерфейс является абстрактным классом: нет ничего общего в представлении данных (кроме расположения указателя на **vtbl**) для всех фигур. Исходя из этого определения, мы можем писать код для функции, манипулирующей вектором указателей на фигуры:

```
void rotate_all(vector<Shape*>& v, int angle) // вращаем элементы на angle градусов
{
    for (auto p : v)
        p->rotate(angle);
}
```

Определяя конкретную фигуру, мы должны сказать, что это **Shape** и определить ее конкретные особенности (включая ее виртуальные функции):

```
class Circle : public Shape {
public:
    Circle(Point p, int rr);    // конструктор

    Point center() const {return x;}
    void move(Point to) {x=to;}

    void draw() const;
    void rotate(int) {}         // простой алгоритм
private:
    Point x;  // центр
    int r;    // радиус
};
```

До сих пор пример с **Shape** и **Circle** не дал ничего нового по сравнению с **Container** и **Vector_container**, но мы можем идти дальше:

```
class Smiley : public Circle {// Circle как базов. класс для Smiley (лицо)
public:
    Smiley(Point p, int r) : Circle{p, r}, mouth{nullptr} {}

    ~Smiley()
    {
        delete mouth;
        for (auto p : eyes) delete p;
    }
    void move(Point to);

    void draw() const;
    void rotate(int);

    void add_eye(Shape* s) {eyes.push_back(s);}
    void set_mouth(Shape* s);
    virtual void wink(int i);  // моргнуть глазом (eye) с номером i

    // ...

private:
    vector<Shape*> eyes;    // обычно два глаза
    Shape* mouth;
};
```

Метод **push_back()** добавляет аргумент к **vector** (здесь это **eyes**), увеличивая его размер на единицу.

Теперь мы можем определить **Smiley::draw()**, используя соответствующий метод базового класса и применяя методы **draw()** для членов-данных (полей):

```
void Smiley::draw()
{
    Circle::draw();
    for (auto p : eyes)
        p->draw();
    mouth->draw();
}
```

Заметьте, что **Smiley** хранит свои «глаза» (eye — это глаз по-русски) в **vector** из стандартной библиотеки и удаляет их в своем деструкторе. Деструктор в **Shape** виртуальный и деструктор в **Smiley** замещает его. Виртуальный деструктор важен для абстрактного класса, так как работа с объектом производного класса обычно ведется через интерфейс, предоставляемый абстрактным базовым классом. В частности, он может удаляться с помощью указателя на базовый класс. В таком случае, механизм вызова виртуальных функций гарантирует вызов правильного деструктора, который неявно вызовет деструкторы базовых классов и деструкторы для членов-данных.

В приведенном нами упрощенном примере обязанностью программиста является задача корректного размещения глаз (eyes) и рта (mouth) внутри окружности, представляющей улыбающееся лицо (smiley).

Мы можем добавлять новые поля, методы или и то, и другое в рамках определения нового класса путем использования наследования. Это обеспечивает значительную гибкость вместе с потенциальными возможностями для недоразумений и плохого дизайна (см. Главу 21). Классовые иерархии предоставляют два механизма, из которых мы можем извлекать выгоду:

- *Наследование интерфейса*: Объект производного класса может использоваться везде, где затребован объект базового класса. См. пример с классами **Container** и **Shape**. Такие классы часто являются абстрактными.
- *Наследование реализации*: Базовый класс предоставляет данные и методы, которые облегчают реализацию производных классов. Пример: использование классом **Smiley** конструктора базового класса **Circle** и метода **Circle:: draw()**. Такого рода базовые классы часто имеют члены-данные и конструкторы.

Конкретные классы — особенно классы с минимальным представлением — похожи на встроенные типы: можно определять локальные переменные этих типов, осуществлять к ним доступ по именам, копировать их и т.д. Классы же из классовых иерархий иного типа: мы применяем их преимущественно для создания объектов в свободной памяти при помощи операции **new**, а доступ к ним осуществляем по указателям или ссылкам. К примеру, рассмотрим функцию, которая читает данные, описывающие фигуры, из потока и конструирует соответствующие объекты типа **Shape**:

```
enum class Kind {circle, triangle, smiley};

Shape* read_shape(istream& is)    // читаем описание формы из потока is
{
    // ... читаем описание формы из is и находим ее Kind k ...

    switch (k) {
    case Kind::circle:
        // читаем данные круга {Point, int} в p и r
        return new Circle{p, r};
    case Kind::triangle:
        // читаем данные треугольника {Point, Point, Point} в p1, p2, и p3
        return new Triangle{p1, p2, p3};
```

```
        case Kind::smiley:
            // читаем smiley-данные {Point, int, Shape, Shape, Shape} в p, r, e1, e2, и m
            Smiley* ps = new Smiley{p, r};
            ps->add_eye(e1);
            ps->add_eye(e2);
            ps->set_mouth(m);
        return ps;
        }
}
```

Можно использовать эту функцию следующим образом:

```
void user()
{
    std::vector<Shape*> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);                    // вызываем draw() для каждого элемента
    rotate_all(v,45);               // вызываем rotate(45) для каждого элемента
    for (auto p : v) delete p;      // не забываем удалить элементы
}
```

Очевидно, что пример упрощен — особенно в отношении обработки ошибок — но он в живой манере иллюстрирует тот факт, что **user()** не имеет ни малейшего представления о том, с какими именно фигурами идет работа. Код функции **user()** может быть скомпилирован один раз, после чего он может работать даже с фигурами, которые будут добавляться в программу позднее. Обратите внимание на то, что нет никаких указателей вне функции **user()**, так что ответственность за их освобождение лежит на **user()**. Это делается с помощью операции **delete** (что критически зависит от виртуального деструктора в базовом классе **Shape**). Поскольку деструктор в **Shape** виртуальный, операция **delete** вызывает деструктор класса, который лежит ниже в иерархии. Это чрезвычайно важно, ибо производный класс может захватить ресурсы всевозможных типов (файловые дескрипторы, блокировки, потоки вывода), которые должны быть освобождены. В нашем примере, **Smiley** удаляет объекты «глаза» (**eyes**) и «рот» (**mouth**).

Опытные программисты заметят, что я оставил две лазейки для ошибок:

- Пользователь может столкнуться с невозможностью удалить ресурс по указателю, возвращаемому функцией **read_shape()**.
- Владелец контейнера указателей на **Shape** может не удалить объекты, на которые смотрят указатели.

В этом отношении функции, возвращающие указатели на объекты, размещенные в свободной памяти, потенциально опасны.

Решением обоих проблем может быть возврат **unique_ptr** из стандартной библиотеки (§5.2.1), а не «голого указателя», и хранение этих **unique_ptr** в контейнере:

```
unique_ptr<Shape> read_shape(istream& is) // читаем описание формы из потока is
{
    // читаем описание формы из is и находим ее Kind k

    switch (k) {
    case Kind::circle:
        // читаем данные круга {Point, int} в p и r
        return unique_ptr<Shape>{new Circle{p, r}}; // §5.2.1
    // ...
}
```

```
void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);        // вызываем draw() для каждого элемента
    rotate_all(v,45);   // вызываем rotate(45) для каждого элемента
} // здесь все формы уничтожаются неявно
```

Теперь объект принадлежит **unique_ptr**, который и удалит объект, когда выйдет из области видимости.

Чтобы новая версия **user()** смогла работать, нужны новые версии **draw_all()** и **rotate_all()**, которые принимают **vector<unique_ptr<Shape>>**. Написание многочисленных **_all()**-функций может показаться утомительным; альтернатива рассматривается в §3.4.3.

3.3. Копирование и перемещение

По умолчанию, объекты можно копировать. Это верно как для пользовательских типов, так и для встроенных типов. Умолчательная семантика копирования объектов сводится к почленному копированию. Для примера воспользуемся типом **complex** из §3.2.1.1:

```
void test(complex z1)
{
    complex z2 {z1};    // инициализация копированием
    complex z3;
    z3 = z2;            // копирующее присваивание
    // ...
}
```

Теперь **z1**, **z2** и **z3** имеют одинаковые значение, так как при выполнении операции присваивания и при инициализации копируются все члены.

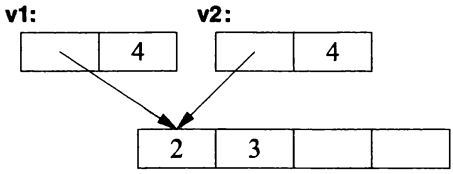
Когда мы проектируем класс, мы должны обязательно остановиться на вопросе, когда, где и как могут копироваться объекты класса. Для простых конкретных классов почленное копирование часто является корректной семантикой. Для некоторых более изощренных конкретных типов, вроде **Vector**, почленное копирование неправильно, а для абстрактных типов оно почти всегда не подходит.

3.3.1. Копирование контейнеров

Когда класс относится к категории *ресурсных дескрипторов* (*resource handles*), он отвечает за объект, адресуемый через указатель, и в таких случаях почленное копирование является катастрофой. Оно нарушает инвариант ресурсного дескриптора (§2.4.3.2). Например, почленное копирование приведет к тому, что копия **Vector** будет ссылаться на те же элементы, что и у оригинала:

```
void bad_copy(Vector v1)
{
    Vector v2 = v1;     // копируем представление v1 в v2
    v1[0] = 2;          // v2[0] теперь тоже 2!
    v2[1] = 3;          // v1[1] теперь тоже 3!
}
```

Предположим, что **v1** имеет четыре элемента, тогда результат иллюстрируется следующим образом:



По счастью, тот факт, что **Vector** имеет деструктор, является хорошим намеком на то, что умолчательное (почленное) копирование не подходит, и компилятору было бы правильным выдать предупреждение в таком случае (§17.6). Нам нужно самим определить правильный вариант для копирования.

Копирование объекта класса определяется двумя членами: *копирующим конструктором* (*copy constructor*) и операцией *копирующего присваивания* (*copy assignment*):

```
class Vector {
private:
    double* elem;                // elem указывает на массив из sz double
    int sz;
public:
    Vector(int s);                    // конструктор: создает инвариант, захват ресурсов
    ~Vector() {delete[] elem;}        // деструктор: освобождение ресурсов

    Vector(const Vector& a);              // копирующий конструктор
    Vector& operator=(const Vector& a);   // копирующее присваивание

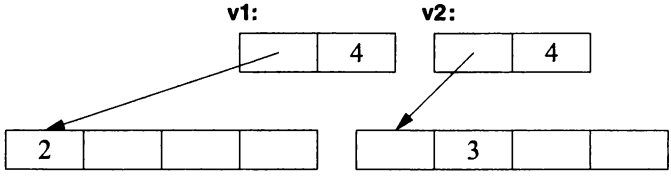
    double& operator[](int i);
    const double& operator[](int i) const;

    int size() const;
};
```

Подходящее определение копирующего конструктора для **Vector** выделяет память под требуемое число элементов, а затем копирует элементы в эту память, так что после копирования каждая копия **Vector** имеет свой собственный набор элементов:

```
Vector::Vector(const Vector& a)    // копирующий конструктор
    : elem{new double[sz]},        // выделить память под элементы
    sz{a.sz}
{
    for (int i=0; i!=sz; ++i)      // копируем элементы
        elem[i] = a.elem[i];
}
```

Результат для присваивания **v2=v1** можно графически представить следующим образом:



Естественно, что помимо копирующего конструктора нужна и операция копирующего присваивания:

```
Vector& Vector::operator=(const Vector& a)  // копирующее присваивание
{
    double* p = new double[a.sz];
    for (int i=0; i!=a.sz; ++i)
        p[i] = a.elem[i];
    delete[] elem;        // удаляем старые элементы
    elem = p;
    sz = a.sz;
    return *this;
}
```

Ключевое слово **this** имеет предопределенный смысл в теле классовых методов и означает указатель на объект, для которого метод вызван.

Копирующий конструктор и операция копирующего присваивания для класса **X** в типичном случае принимают аргумент типа **const X&**.

3.3.2. Перемещение контейнеров

Мы можем управлять копированием посредством копирующего конструктора и операции копирующего присваивания. Но копирование при этом может оказаться чрезвычайно дорогим для больших контейнеров. Рассмотрим пример:

```
Vector operator+(const Vector& a, const Vector& b)
{
    if (a.size()!=b.size())
        throw Vector_size_mismatch{};

    Vector res(a.size());
    for (int i=0; i!=a.size(); ++i)
        res[i]=a[i]+b[i];
    return res;
}
```

Возврат из операции + включает копирование результата из локальной переменной **res** в другое место в памяти, где вызывающий код может получить к нему доступ. Вот пример использования запрограммированной нами операции +:

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
    Vector r;
    // ...
    r = x+y+z;
    // ...
}
```

Копировать **Vector** приходится по крайней мере дважды (для двух операций +). Если объекты типа **Vector** велики, например они содержат по 10000 элементов типа **double**, то все это приведет к огромным вычислительным затратам. А обиднее всего то, что переменная **res** в **operator+()** после копирования больше вообще не используется. На самом деле, нам такое копирование и не нужно; мы всего лишь хотим переместить результат из тела функции наружу: то есть мы хотим *переместить* **Vector**, а не *копировать* его. И мы можем выразить это намерение следующим образом:

```
class Vector {
    // ...

    Vector(const Vector& a);              // копирующий конструктор
    Vector& operator=(const Vector& a);   // копирующее присваивание
```

```
    Vector(Vector&& a);                 // перемещающий конструктор
    Vector& operator=(Vector&& a);      // перемещающее присваивание
};
```

Получив такое определение, компилятор применит *перемещающий конструктор* для реализации перемещения результата из функции наружу. Это означает, что **r=x+y+z** не потребует копирования объектов **Vector**. Вместо этого они будут перемещены.

Перемещающий конструктор для **Vector** имеет тривиальное определение:

```
Vector::Vector(Vector&& a)
    : elem{a.elem},     // "хватаем элементы" из a
    sz{a.sz}
{
    a.elem = nullptr;   // теперь у a нет элементов
    a.sz = 0;
}
```

Запись **&&** означает «ссылка на rvalue» и является ссылкой, которую можно связать с rvalue (§6.4.1). Термин rvalue дополняет собой термин lvalue, который приблизительно означает «то, что может стоять в левой части операции присваивания». Таким образом, rvalue это (в первом приближении) величина, которой вы не можете что-либо присвоить, как например целочисленной величине, являющейся возвратом вызова функции, а ссылка на rvalue это ссылка на что-либо, чему никто не сможет ничего присвоить. Локальная переменная **res** в **operator+()** для **Vector** как раз служит примером.

У перемещающего конструктора аргумент не-**const**, ибо, в конце концов, он удаляет значение из своего аргумента. Операция *перемещающего присваивания* определяется аналогично.

Перемещающие операции имеют место тогда, когда ссылка на rvalue используется в качестве инициализатора или же как правая часть в операции присваивания.

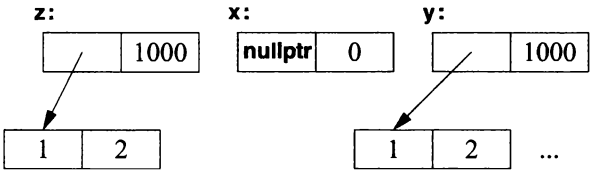
После перемещения объект, из которого переместили содержимое, должен находиться в состоянии, допускающем применение деструктора. В типичном случае, допускаются также присваивания для подобного объекта (§17.5, §17.6.2).

Если программист знает, что значение не будет далее использоваться, а компилятор не настолько интеллектуален, чтобы выявить эту ситуацию, то программист может явно отразить этот контекст:

```
Vector f()
{
    Vector x(1000);
    Vector y(1000);
    Vector z(1000);
    // ...
    z = x;              // получаем копию
    y = std::move(x);   // перемещаем
    // ...
    return z;           // перемещаем
};
```

Функция **move()** из стандартной библиотеки возвращает rvalue-ссылку на свой аргумент.

Непосредственно перед выполнением оператора **return** мы имеем следующее:



Когда **z** уже уничтожено, из него переместили содержимое (действием оператора **return**), оно, как и **x**, пустое, то есть не содержит элементов.

3.3.3. Управление ресурсами

Определяя конструкторы, копирующие и перемещающие операции, а также деструктор, программист получает полный контроль над жизненным циклом содержимого ресурса (вроде элементов контейнера). Более того, перемещающий конструктор позволяет легко и дешево перемещать объект из одной области видимости в другую. Таким образом, объекты, которые мы не можем или не хотим копировать из некоторой области видимости наружу, могут быть легко и дешево перемещены в указанном направлении вместо копирования. Рассмотрим **thread** из стандартной библиотеки для представления параллельной активности (§5.3.1) и **Vector** из миллиона элементов типа **double**. Мы не можем копировать первое из перечисленного и не хотим копировать второе.

```
std::vector<thread> my_threads;

Vector init(int n)
{
    thread t {heartbeat};                // запускаем heartbeat параллельно (в своем потоке)
    my_threads.push_back(move(t));       // перемещаем t в my_threads
    // ... дальнейшая инициализация ...
    Vector vec(n);
    for (int i=0; i<vec.size(); ++i) vec[i] = 777;
    return vec;                          // перемещаем результат из init()
}

auto v = init(); // стартуем heartbeat и инициализируем v
```

Это делает такие дескрипторы ресурсов, как **Vector** и **thread**, альтернативой использованию указателей во многих случаях. По сути, «интеллектуальные указатели» (smart pointers) из стандартной библиотеки, такие как **unique_ptr**, сами по себе являются дескрипторами ресурсов (§5.2.1).

Я применил **vector** из стандартной библиотеки для хранения объектов типа **thread**, поскольку вплоть до §3.4.1 мы не приступаем к параметризации **Vector** типом элементов.

Так же, как и исчезновение **new** и **delete** из кода приложения, мы можем организовать упрятывание указателей в дескрипторы ресурсов. В обоих случаях мы имеем более простой и читабельный код, не налагая при этом дополнительных нагрузок в смысле производительности. В особенности важно, что мы можем добиться *строгой ресурсной безопасности*, то есть мы можем в принципе устранить утечки ресурсов для самого общего понятия ресурса. Примерами служат **vector** по отношению к памяти, **thread** по отношению к системным потокам и **fstream** по отношению к файловым дескрипторам.

3.3.4. Подавление операций

Применение умолчательных копирования и перемещения для класса из иерархии в типичном случае является катастрофой: имея лишь указатель на базовый класс, мы просто не знаем, какие члены данных имеются у объекта производного класса (§3.2.2), и, соответственно, не можем их копировать. Так что лучше всего устранить эти умолчательные операции следующим образом:

```
class Shape {
public:
    Shape(const Shape&) =delete;          // "нет" операциям копирования
    Shape& operator=(const Shape&) =delete;

    Shape(Shape&&) =delete;               // "нет" операциям перемещения
    Shape& operator=(Shape&&) =delete;

    ~Shape();
    // ...
};
```

Теперь попытка скопировать **Shape** будет поймана компилятором. Если вам необходимо копировать объект из классовой иерархии, напишите какой-нибудь вариант функции клонирования (§22.2.4).

В рассмотренном конкретном случае, даже если бы мы забыли явно устранить копирующие и перемещающие операции, вреда бы не было. Дело в том, что операция перемещения автоматически (неявно) не генерируется для класса, в котором пользователь явно определил деструктор. Более того, подавляется также и автоматическая генерация копирующих операций (§44.2.3). Так что будет неплохо явно определять деструктор даже тогда, когда компилятор определил бы его неявно (§17.2.3).

Дескриптор ресурсов в общем случае не может копироваться почленным копированием его членов (§5.2, §17.2.2).

Механизм **=delete** является общим механизмом, то есть с его помощью можно подавить любую операцию (§17.6.4).

3.4. Шаблоны

Совершенно необязательно, что тот, кому нужен вектор, испытывает необходимость именно в векторе с элементами типа **double**. Вектор является общей концепцией, не зависящей от понятия числа с плавающей запятой. Следовательно, тип элементов вектора должен быть представлен независимо. *Шаблон* — это класс или функция, которые мы параметризуем набором типов или значений. Мы применяем шаблоны для отражения концепций, понимаемых в самом общем виде, и из которых можно порождать конкретные типы и функции, указывая аргументы вроде того, что тип элементов есть **double**.

3.4.1. Параметризуемые типы

Мы можем обобщить наш тип «вектор-элементов-double» до типа «вектор-элементов-некоторого типа» посредством определения шаблона, и заменив конкретный тип **double** произвольным параметром. Вот пример:

```
template<typename T>
class Vector {
private:
    T* elem;  // elem указывает на массив из sz элементов типа T
    int sz;
public:
    Vector(int s);                // конструктор: создает инвариант, захват ресурсов
    ~Vector() {delete[] elem;}    // деструктор: освобождение ресурсов

    // ... копирующие и перемещающие операции ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const {return sz;}
};
```

Префикс **template<typename T>** делает **T** параметром объявления, перед которым он стоит. Можно сказать, что это версия языка C++ для математического высказывания «для любого T» или «для всех типов T».

Классовые методы можно объявлять аналогично:

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0) throw Negative_size{};
    elem = new T[s];
    sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Отталкиваясь от этих определений, создаем конкретные векторы:

```
Vector<char> vc(200);    // вектор из 200 символов
Vector<string> vs(17);   // вектор из 17 строк
Vector<list<int>> vli(45); // вектор из 45 списков целых
```

Знаки >> в **Vector<list<int>>** завершают вложенные аргументы шаблонов; это не случайная описка с операцией потокового ввода. Не обязательно (как в C++98) помещать пробел между двумя идущими подряд знаками >.

Используем **Vector** следующим образом:

```
void write(const Vector<string>& vs)    // Vector строк
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Чтобы поддержать для нашего **Vector** цикл range-**for**, мы должны определить подходящие функции **begin()** и **end()**:

```
template<typename T>
T* begin(Vector<T>& x)
{
    return &x[0];   // указатель на первый элемент
}

template<typename T>
T* end(Vector<T>& x)
{
    return x.begin()+x.size(); // указатель на элемент "за последним"
}
```

Теперь мы можем написать:

```
void f2(const Vector<string>& vs) // Vector строк
{
    for (auto& s : vs)
        cout << s << '\n';
}
```

Аналогично, мы можем определять списки, вектора, ассоциативные массивы и др. как шаблоны (§4.4, §23.2, Глава 31).

Шаблоны являются механизмом времени компиляции, то есть они не налагают дополнительных обременений во время выполнения, в отличие от аналогичного по предназначению кода, написанного программистом вручную (§23.2.2).

3.4.2. Шаблоны функций

У шаблонов есть много вариантов применения, кроме параметризации контейнеров типом их элементов. В частности, они интенсивно используются в стандартной библиотеке для параметризации типов и алгоритмов (§4.4.5, §4.5.5). Например, мы можем написать функцию, которая вычисляет сумму значений элементов любого контейнера:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c)
        v+=x;
    return v;
}
```

Шаблонный аргумент **Value** и аргумент функции **v** позволяют коду, который вызывает эту функцию, задать тип и начальное значение аккумулятора (переменной, в которой аккумулируется сумма):

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
    int x = sum(vi,0);                          // сумма вектора целых (складываем целые)
    double d = sum(vi,0.0);                     // сумма вектора целых (складываем double)
    double dd = sum(ld,0.0);                    // сумма списка элементов типа double
    auto z = sum(vc, complex<double>{});        // сумма вектора complex<double>
                                                // начальное значение {0.0,0.0}
}
```

Причина, по которой целые слагаемые накапливаются в сумме типа **double**, состоит в том, чтобы элегантно превзойти ограничение на максимальное значение целых. Обратите внимание на то, как типы шаблонных аргументов для **sum<T, V>** выводятся из аргументов вызова функции. К счастью, нам не нужно явным образом специфицировать эти типы.

Рассмотренная функция **sum()** является упрощенной версией **accumulate()** из стандартной библиотеки (§40.6).

3.4.3. Функциональные объекты (объекты-функции)

Одним из наиболее полезных видов шаблонов является *функциональный объект*, называемый также *функтором*, который используется для определения объектов, обратиться к которым можно так, как выполняется вызов функции. Пример:

```
template<typename T>
class Less_than {
    const T val;   // значение для сравнения
public:
    Less_than(const T& v) : val(v) {}
    bool operator()(const T& x) const {return x<val;} // операция вызова
};
```

Метод **operator()** реализует «вызов функции».

Создадим именованные переменные типа **Less_than** для некоторого типа аргумента:

```
Less_than<int> lti {42};  // lti(i) сравнивает i с 42 используя < (i<42)
Less_than<string> lts {"Backus"}; // lts(s) сравнивает s с "Backus" используя < (s<"Backus")
```

Мы можем «вызвать» эти объекты так, как обычно вызываются функции:

```
void fct(int n, const string & s)
{
    bool b1 = lti(n);    // true если n<42
    bool b2 = lts(s);    // true если s<"Backus"
    // ...
}
```

Подобного рода функциональные объекты широко используются в качестве аргументов алгоритмов. Например, мы можем подсчитать количество случаев, для которых предикат (условие) истинен (возвращает **true**):

```
template<typename C, typename P>
int count(const C& c, P pred)
{
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

Предикат есть нечто, что мы можем вызвать и при этом будет возвращено значение **true** или **false**. Пример:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
        << ": " << count(vec, Less_than<int>{x})
        << '\n';
    cout << "number of values less than " << s
        << ": " << count(lst, Less_than<string>{s})
        << '\n';
}
```

Здесь **Less_than<int>{x}** конструирует объект, который операция вызова сравнивает с целым **x**; **Less_than<string>{s}** конструирует объект, который сравнивается со значением **s** типа **string**. Особая красота функциональных объектов состоит в том, что они все время носят с собой значение, с которым производится сравнение. Нам не нужно писать все новые функции для новых типов, и нам не нужно связываться с «ужасными» глобальными переменными, чтобы те содержали сравниваемые значения. Кроме того, для простых функциональных объектов, вроде **Less_than**, очень просто выполняется встраивание (inlining), и в итоге операция вызова для **Less_than** намного эффективнее непрямого вызова функции. Способность функциональных объектов обеспечивать эффективный вызов и носить с собой сравниваемое значение делает их особенно полезными в качестве аргументов алгоритмов.

Функциональные объекты, которые используются для уточнения смысла ключевых операций обобщенного алгоритма (как **Less_than** для **count()**), часто называют *объектами политики* (*policy objects*).

Мы определяем **Less_than** отдельно от его использования. Это бывает неудобно. Как следствие, имеется специальная нотация для неявной генерации функциональных объектов:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
        << ": " << count(vec,[&](int a){return a<x;})
        << '\n';
```

```
        cout << "number of values less than " << s
             << ": " << count(lst,[&](const string& a){return a<s;})
             << '\n';
}
```

Обозначение **[&](int a){return a<x;}** называется *лямбда выражением* (*lambda expression*) (§11.4) или просто лямбдой. Оно генерирует функциональный объект, в точности равный **Less_than<int>{x}**. Запись **[&]** — это *список захвата* (*capture list*), специфицирующий локальные имена (такие как **x**), передаваемые по ссылке. Если требуется «захватить» только **x**, то это можно выразить с помощью нотации **[&x]**. Если же нужно передавать генерируемой функции **x** по значению (то есть передавать копию **x**), то это намерение выражается так: **[=x]**. При пустом списке захвата пишут **[]**, захват всех локальных переменных для передачи по ссылке отражается в записи **[&]**, а их захват для передачи по значению — как **[=]**.

Применение лямбда выражений удобно, но оно может вызывать неясности. Для нетривиальных действий (более сложных, чем простое выражение) я предпочитаю именовать операцию так, чтобы это нагляднее отражало ее предназначение, а также чтобы ее можно было использовать в разных местах программы.

В §3.2.4 мы отмечали, что часто нас раздражает необходимость написания многочисленных операций над элементами вектора указателей, таких как **draw_all()** и **rotate_all()**. Функциональные объекты (в частности лямбды) могут помочь нам, отделяя траверс по контейнеру от спецификации действий над каждым элементом.

Сперва нам нужно определить функцию, которая применяет операцию к каждому элементу контейнера указателей:

```
template<class C, class Oper>
void for_all(C& c, Oper op)   // полагаем что C это контейнер указателей
{
    for (auto& x : c)
        op(*x);           // передаем op() ссылку на каждый указуемый элемент
}
```

Теперь можно написать версию **user()** из §3.2.4 без необходимости написания множества **_all**-функций:

```
void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    for_all(v,[](Shape& s){s.draw();});          // draw_all()
    for_all(v,[](Shape& s){s.rotate(45);});      // rotate_all(45)
}
```

Я передаю ссылку на **Shape** лямбде, так что ей нет нужды знать точно, как объекты хранятся в контейнере. В частности, все эти **for_all()** вызовы работали бы по-прежнему, если бы я заменил **v** на **vector<Shape*>**.

3.4.4. Шаблоны с переменным числом параметров

Можно определить шаблон с произвольным числом аргументов произвольного типа. Такой шаблон называется *шаблоном с переменным числом параметров* (*variadic template*).

Вот пример:

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
    g(head);  // делаем что-нибудь с head
    f(tail...);  // снова обращаемся к tail
}

void f() {}    // не делаем ничего
```

Ключом к реализации шаблона с переменным числом параметров является понимание того, что когда вы передаете ему список аргументов, вы имеете возможность отделить первый аргумент от остальных. Мы что-то делаем с первым аргументом (здесь это **head**), а затем рекурсивно вызываем **f()** для всех остальных аргументов (здесь это **tail**). Многоточие, ..., используется для обозначения понятия «оставшаяся часть списка». В конце концов **tail** станет пустым и нам нужно будет с этим что-то сделать.

Вызовем **f()** следующим образом:

```
int main()
{
    cout << "first: ";
    f(1,2.2,"hello");

    cout << "\nsecond: "
    f(0.2,'c',"yuck!",0,1,2);
    cout << "\n";
}
```

Здесь будет выполнен вызов **f(1,2.2,"hello")**, который породит вызов **f(2.2,"hello")**, который, в свою очередь, породит вызов **f("hello")**, а тот породит вызов **f()**. А что делает **g(head)**? В реальных программах он делает то, что мы хотим делать с каждым аргументом. К примеру, можно выводить аргумент (здесь это **head**) в поток вывода:

```
template<typename T>
void g(T x)
{
    cout << x << " ";
}
```

В результате получается такой вывод программы:

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

Похоже, что **f()** представляет собой простой вариант функции **printf()**, выводящей произвольный список значений, причем все это реализовано в трех строках кода плюс соответствующие объявления.

Сила шаблонов с произвольным числом параметров состоит в том, что они могут принять от вас любые аргументы, которые вам захочется дать им. А слабость заключается в том, что проверка типов в интерфейсе может сильно усложнить программу. Подробные детали см. в §28.6. Примеры рассмотрены в §34.2.4.2 и Главе 29.

3.4.5. Синонимы

Удивительно часто бывает полезным ввести синоним для типа шаблона (§6.5). Например, стандартный файл **<stddef>** содержит определение синонима **size_t**, например:

```
using size_t = unsigned int;
```

Истинный тип для синонима **size_t** зависит от конкретной реализации, так что он может оказаться и **unsigned long**. Применение синонима **size_t** позволяет программисту писать переносимый код.

Часто параметризуемые типы определяют синонимы для типов, связанных с их шаблонными аргументами. Вот пример:

```
template<typename T>
class Vector {
public:
    using value_type = T;
    // ...
};
```

Фактически каждый контейнер стандартной библиотеки предоставляет синоним **value_type** в качестве имени для его типа значений (§31.3.1). Это позволяет писать код, который будет работать для каждого контейнера, придерживающегося этого соглашения. Вот пример;

```
template<typename C>
using Element_type = typename C::value_type;

template<typename Container>
void algo(Container& c)
{
    Vector<Element_type<Container>> vec; // результаты храним здесь
    // ...
}
```

Механизм синонимии можно использовать, чтобы определить новый шаблон путем связывая части или всех шаблонных параметров. Например:

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string, Value>;

String_map<int> m;      // m это Map<string,int>
```

См. §23.6.

3.5. Советы

[1] Выражайте идеи непосредственно в коде; §3.2.

[2] Определяйте классы для отражения концепций приложения в коде; §3.2.

[3] Используйте конкретные классы для отражения простых концепций и для реализации критически важных с точки зрения производительности компонентов; §3.2.1.

[4] Избегайте «голых» **new** и **delete**; §3.2.1.2.

[5] Для управления ресурсами используйте дескрипторы ресурсов; §3.2.1.2.

[6] Используйте абстрактные классы в качестве интерфейсов, когда требуется четкое разделение интерфейса и реализации; §3.2.2.

[7] Используйте классовые иерархии для отражения концепций с внутренне присущей им иерархической структурой; §3.2.4.

[8] Проектируя классовую иерархию, понимайте разницу между наследованием реализации и наследованием интерфейса; §3.2.4.

[9] Управляйте конструированием, копированием, перемещением и уничтожением объектов; §3.3.

[10] Возвращайте контейнеры по значению (полагаясь на операцию перемещения ради эффективности); §3.3.2.

[11] Обеспечивайте строгую безопасность ресурсов; то есть не допускайте утечки чего-либо, понимаемого как ресурс §3.3.3.

[12] Используйте контейнеры, определяемые как шаблоны дескрипторов ресурсов, для хранения коллекций однотипных величин; §3.4.1.

[13] Используйте шаблоны функций для представления алгоритмов; §3.4.2.

[14] Используйте функциональные объекты, включая лямбды, для выражения намерений (политики) и действий; §3.4.3.

[15] Используйте синонимы типов и шаблонов для предоставления общей нотации для аналогичных типов и разных реализаций; §3.4.5.

4

Обзор языка C++: Контейнеры и Алгоритмы

Зачем тратить время на обучение,
если невежество достигается мгновенно.
— Хоббс

- Библиотеки
  (Обзор стандартной библиотеки; Заголовочные файлы стандартной библиотеки и ее пространство имен)
- Строки
- Потоковый ввод/вывод
  (Вывод; Ввод; Ввод/вывод пользовательских типов)
- Контейнеры
  (**vector**; **list**; **map**; **unordered_map**; Обзор контейнеров)
- Алгоритмы
  (Использование итераторов; Типы итераторов; Итераторы потоков; Предикаты; Обзор алгоритмов; Алгоритмы для контейнеров)
- Советы

4.1. Библиотеки

Ни одна значительная программа не может быть написана на базе лишь языка программирования. Сначала разрабатывается набор библиотек. Затем этот набор становится базисом для дальнейшей работы. Большинство программ бессмысленно писать, опираясь только на язык, так как многие программы можно сильно упростить за счет использования библиотек.

Как и предыдущие Главы 2 и 3, настоящая и следующая главы построены как быстрый обзор ключевых средств стандартной библиотеки. Я предполагаю, что вы уже программировали ранее. В противном случае, вам лучше сначала обратиться к учебнику, например к книге «Программирование: принципы и практика использования C++» [Страуструп, 2009]. Но даже если вы уже имеете опыт программирования, использованные вами библиотеки или типы программ, которые вы писали, могут сильно отличаться от стиля языка C++, излагаемого в данной книге. Если наш «молниеносный стиль» смутит вас, переходите к более систематическому изложению, который начинается с Главы 6. А более подробное изложение конкретно стандартной библиотеки начинается с Главы 30.

Я очень кратко рассматриваю полезные типы стандартной библиотеки, такие как **string**, **ostream**, **vector**, **map** (в данной главе), **unique_ptr**, **thread**, **regex** и **complex** (в Главе 5), наряду с наиболее типичными способами их использования. Это позволяет мне отложить

рассмотрение более содержательных примеров на последующие главы. Не беспокойтесь, что вам сразу не будет абсолютно все понятно. Цель настоящей главы — дать вам лишь базовое знание и понимание наиболее полезных средств стандартной библиотеки.

Точная спецификация стандартной библиотеки занимает две трети стандарта ISO C++. Изучайте именно ее, а не доморощенные спецификации. Очень многое вошло в этот проект, еще больше — в реализацию, а также огромные усилия направлены на поддержку и дальнейшее развитие.

Те средства стандартной библиотеки, которые рассматриваются в данной книге, входят во все полные реализации C++. Большинство реализаций помимо компонентов стандартной библиотеки предлагают также системы графического интерфейса пользователя (GUI), Web-интерфейсы, интерфейсы к базам данных и т.д. Аналогично, большинство специализированных сред разработки предоставляют еще и специализированные фундаментальные библиотеки для корпоративного (индустриального) стиля разработки и/или для специальных сред исполнения. Я не рассматриваю в настоящей книге ничего из перечисленного. Моя цель — предоставить согласованное изложение C++ так, как это определено в стандарте, и чтобы примеры были переносимыми, за исключением отдельных случаев, где это оговаривается особо. Естественно, программист может самостоятельно изучить дополнительные возможности, имеющиеся во многих системах.

4.1.1. Обзор стандартной библиотеки

Средства стандартной библиотеки можно классифицировать следующим образом:

- Средства поддержки времени исполнения (для динамического выделения памяти и информации о типах во время исполнения).
- Стандартная библиотека языка C (с минимальными модификациями для предотвращения нарушений системы типов); см. Главу 43.
- Строки и потоки ввода/вывода (с поддержкой международной системы символов и интернационализации); см. Главы 36, 38 и 39. Потоки ввода/вывода допускают расширение, так что пользователи могут добавлять свои собственные потоки, стратегии буферизации и наборы символов.
- Фундаментальный набор контейнеров (таких как **vector** и **map**) и алгоритмов (таких как **find()**, **sort()** и **merge()**); см. §4.4, §4.5, Главы 31–33. Этот набор, традиционно называемый STL [Stepanov,1994], допускает расширение за счет добавления собственных контейнеров и алгоритмов.
- Поддержка численных расчетов (стандартные математические функции, комплексные числа, векторы и генераторы случайных чисел); см. §3.2.1.1 и Главу 40.
- Поддержка регулярных выражений; см. §5.5 и Главу 37.
- Поддержка идиом параллельного исполнения, включая классы врода **thread** и **lock**; см. §5.3 и Главу 41. Идиомы представлены на столь фундаментальном уровне, что пользователи могут добавить новые модели параллельного исполнения в виде библиотек.
- Средства поддержки шаблонного метапрограммирования (см. §5.4.2, §28.2.4, §35.4), обобщенное программирование в стиле STL (например, **pair**; §5.4.3, §34.2.4.1), средства для программирования общих задач (например, **clock**; §5.4.1, §35.2).
- «Интеллектуальные указатели» для управления ресурсами (**unique_ptr** и **shared_ptr**; §5.2.1, §34.3) и интерфейс к сборщику мусора (§34.5).
- Специальные контейнеры, такие как **array** (§34.2.1), **bitset** (§34.2.2) и **tuple** (§34.2.4.2).

Для включения класса в стандартную библиотеку мы следовали следующим критериям:

- Он может быть полезным большинству программистов (как новичкам, так и экспертам),
- Он может быть представлен в общей форме, которая не подразумевает существенной потери в эффективности по сравнению с простой версией такого же класса,
- Простые случаи применения класса не слишком сложны в изучении (по сравнению с внутренней сложностью задачи, для решения которой он может использоваться).

По существу, стандартная библиотека языка C++ предоставляет наиболее фундаментальные структуры данных вместе с фундаментальными алгоритмами, работающими поверх этих структур данных.

4.1.2. Заголовочные файлы стандартной библиотеки и ее пространство имен

Каждое средство стандартной библиотеки представлено в своем заголовочном файле (хедер-файле). Например:

```
#include<string>
#include<list>
```

Это обеспечивает доступ к стандартным типам (классам) **string** и **list**.

Средства стандартной библиотеки определены в пространстве имен (§2.4.2, §14.3.1), названном **std**. Чтобы использовать эти средства, можно применить префикс **std::**:

```
std::string s {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is peace", "Freedom is Slavery",
                                "Ignorance is Strength"};
```

Ради простоты я редко использую префикс **std::** в примерах. Я также не всегда явно указываю директивой **#include** необходимые заголовочные файлы. Для компиляции фрагментов программ нужно самостоятельно включить в текст программы необходимые заголовочные файлы (§4.4.5, §4.5.5 и §30.2) и обеспечить видимость имен, объявляемых в этих файлах. Например:

```
#include<string>          // доступ к строкам стандартной библиотеки
using namespace std;      // доступ к std-именам без std-префикса

string s {"C++ is a general-purpose programming language"}; // OK: string это std::string
```

Не очень хорошо засорять глобальное пространство имен. Но в данной книге я использую почти исключительно стандартную библиотеку, и не добавляю скрупулезно префикс **std::** и не включаю в каждый пример директивой **#include** необходимые заголовочные файлы. Это предполагается по умолчанию.

Представим сейчас полезную выборку из заголовочных файлов стандартной библиотеки, каждая из которых объявляет свои средства в пространстве имен **std**:

Некоторые заголовочные файлы стандартной библиотеки			
<algorithm>	**copy(), find(), sort()**	§32.2	§ iso.25
<array>	**array**	§34.2.1	§ iso.23.3.2
<chrono>	**duration, time_point**	§35.2	§ iso.20.11.2
<cmath>	**sqrt(), pow()**	§40.3	§ iso.26.8
<complex>	**complex, sqrt(), pow()**	§40.4	§ iso.26.8
<fstream>	**fstream, ifstream, ofstream**	§38.2.1	§ iso.27.9.1

продолжение

Некоторые заголовочные файлы стандартной библиотеки			
<future>	**future, promise**	§5.3.5	§ iso.30.6
<iostream>	**istream, ostream, cin, cout**	§38.1	§ iso.27.4
<map>	**map, multimap**	§31.4.3	§ iso.23.4.4
<memory>	**unique_ptr, shared_ptr, allocator**	§5.2.1	§ iso.20.6
<random>	**default_random_engine, normal_distribution**	§40.7	§ iso.26.5
<regex>	**regex, smatch**	Chapter 37	§ iso.28.8
<string>	**string, basic_string**	Chapter 36	§ iso.21.3
<set>	**set, multiset**	§31.4.3	§ iso.23.4.6
<sstream>	**istrstream, ostrstream**	§38.2.2	§ iso.27.8
<thread>	**thread**	§5.3.1	§ iso.30.3
<unordered_map>	**unordered_map, unordered_multimap**	§31.4.3.2	§ iso.23.5.4
<utility>	**move(), swap(), pair**	§35.5	§ iso.20.1

Этот список далеко не полный; см. §30.2 ради более полной информации.

4.2. Строки

В дополнение к строковым литералам стандартная библиотека предоставляет тип **string**. Этот тип обеспечивает множество полезных строковых операций, таких как конкатенация. Вот пример:

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}

auto addr = compose("dmr","bell-labs.com");
```

Здесь **addr** инициализируется последовательностью символов dmr@bell-labs.com. Конкатенация — это «сложение» строк. Конкатенации подлежат объекты типа **string**, строковые литералы, С-строки и символы. У стандартного типа **string** есть перемещающий конструктор, так что возврат по значению даже больших объектов типа **string** эффективен (§3.3.2).

Во многих приложениях часто используется форма конкатенации, когда что-либо добавляется в конец объекта типа **string**. Это имеет прямую поддержку в виде операции **+=**. Например:

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n';      // с переходом на новую строку
    s2 += '\n';          // с переходом на новую строку
}
```

Два этих способа добавления в конец строки типа **string** семантически эквивалентны, но я предпочитаю второй из них, так как он более отчетливо проявляет характер операции, более лаконичен и потенциально более эффективен.

Объекты типа **string** *изменяемы* (*mutable*). Кроме операций **=** и **+=** допустимы также индексирование (используя **[]**) и операции с подстроками. Тип **string** стандартной библиотеки подробно описывается в Главе 36. У него есть множество операций с подстроками. Вот пример:

```
string name = "Niels Stroustrup";

void m3()
{
    string s = name.substr(6,10);      // s = "Stroustrup"
```

```
    name.replace(0,5,"nicholas");      // name становится "nicholas Stroustrup"
    name[0] = toupper(name[0]);        // name становится "Nicholas Stroustrup"
}
```

Операция **substr()** возвращает строку, которая является копией подстроки, указанной аргументами. Первый аргумент это индекс (позиция) в строке, а второй аргумент — длина подстроки. Так как значения индексов начинаются с нуля, то **s** принимает значение **Stroustrup**.

Операция **replace()** заменяет строку значением. Таким образом, подстрока, начинающаяся с нулевой позиции и имеющая длину **5**, есть **Niels**; она заменяется на **nicholas**. Наконец, я заменяю начальный символ на его эквивалент в верхнем регистре. Так что финальное значение для переменной **name** есть **Nicholas Stroustrup**. Отметим, что строка замены не обязана быть той же длины, что и заменяемая подстрока.

Естественно, объекты типа **string** можно сравнивать друг с другом и со строковыми литералами. Пример:

```
string incantation;

void respond(const string& answer)
{
    if (answer == incantation) {
        // ...нечто магическое
    }
    else if (answer == "yes") {
        // ...
    }
    // ...
}
```

В целом описание **string** дается в Главе 36. Общие технологии реализации стандартного класса **string** представлены в нашем примере для учебного класса **String** (§19.3).

4.3. Потоковый ввод/вывод

Стандартная библиотека обеспечивает форматированный ввод и вывод символов посредством **iostream**. Операции ввода типизированы и могут расширяться на типы, определяемые пользователем. В этом разделе **iostream** рассматривается очень кратко; в Главе 38 дается полное описание средств **iostream**.

Иные формы пользовательских интерфейсов, вроде графических интерфейсов, обеспечиваются библиотеками, которые не имеют ISO стандарта и не рассматриваются в настоящей книге.

4.3.1. Вывод

В библиотеке потокового ввода/вывода специфицирован вывод для каждого встроенного типа. Кроме того, нетрудно реализовать вывод и для пользовательских типов (§4.3.3). Знак **<<** используется в качестве обозначения для операции вывода в объекты типа **ostream**; **cout** — это стандартный поток вывода, а **cerr** — стандартный поток для сообщения об ошибках. По умолчанию, при записи значений в **cout** они конвертируются в последовательность символов. Например, чтобы вывести десятичное число **10**, мы можем написать:

```
void f()
{
    cout << 10;
}
```

В результате в стандартный поток вывода помещается цифра **1**, а за ней туда помещается цифра **0**.

```
void g()
{
    int i {10};
    cout << i;
}
```

С тем же результатом можно написать следующее:

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

Очевидным образом можно осуществлять вывод разных типов:

```
the value of i is 10
```

Для **h(10)** вывод будет таким:

При выводе нескольких величин может надоесть каждый раз повторять имя потока вывода. К счастью, результат операции вывода снова может быть использован для последующего вывода. Вот пример:

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

Вызов **h2()** дает тот же вывод, что и **h()**.

Символьная константа записывается как символ, заключенный в одиночные кавычки. Отметим, что символ выводится не как числовое значение, а именно как символ. Например:

```
void k()
{
    int b = 'b';    // char неявно конвертируется в int
    char c = 'c';
    cout << 'a' << b << c;
}
```

Числовой код для символа **'b'** равен **98** (в ASCII-кодировке на той реализации, что я использую), так что вывод программы будет **a98c**.

4.3.2. Ввод

Стандартная библиотека для ввода использует класс **istream**. Как и **ostream**, он имеет дело со строковыми представлениями для встроенных типов и легко может быть расширен для работы с пользовательскими типами.

Обозначение **>>** используется в качестве операции ввода; **cin** — это стандартный поток ввода. Тип правого операнда для **>>** определяет цель операции ввода и тип вводимого значения. Например:

```
void f()
{
    int i;
    cin >> i;       // читаем целое в i
```

```
    double d;
    cin >> d;       // читаем вещественное значение двойной точности в d
}
```

Здесь числовое значение, например **1234**, читается из стандартного потока ввода в целую переменную **i**, а в переменную **d**, имеющую тип вещественного числа с двойной точностью, из стандартного потока ввода читается соответствующее числовое значение, например **12.34e5**.

Часто нам требуется прочитать последовательность символов. В этом случае удобно читать в переменную типа **string**. Например:

```
void hello()
{
    cout << "Please enter your name\n";
    string str;
    cin >> str;
    cout << "Hello, " << str << "!\n";
}
```

Если ввести **Eric**, то вывод программы будет следующим:

```
Hello, Eric!
```

По умолчанию, пробельные символы (§7.3.2), такие как пробел, оканчивают процесс чтения, так что если вы будете вводить **Eric Bloodaxe**, намекая на злосчастного короля Йорка, вывод останется прежним:

```
Hello, Eric!
```

Прочесть полную строку (включая терминальный символ перевода строки) можно с помощью функции **getline()**. Вот пример:

```
void hello_line()
{
    cout << "Please enter your name\n";
    string str;
    getline(cin, str);
    cout << "Hello, " << str << "!\n";
}
```

Эта программа, получив на вход **Eric Bloodaxe**, обеспечит ожидаемый вывод:

```
Hello, Eric Bloodaxe!
```

Символ перехода на новую строку отбрасывается, и **cin** готов к приему следующей входной строки.

Стандартные строки обладают замечательной возможностью расширяться, чтобы вместить все символы, которые вы захотите поместить в них; таким образом, вы не должны выполнять предварительный расчет максимально необходимого размера. Если вдруг вы введете пару мегабайт точек с запятой, программа выдаст вам эту самую массу точек с запятой.

4.3.3. Ввод/вывод пользовательских типов

Дополнительно к возможности ввода/вывод встроенных типов и стандартных строк (тип **string**) стандартная библиотека потокового ввода/вывода позволяет программистам определять ввод/вывод для их собственных типов. Рассмотрим к примеру простой тип **Entry**, который можно использовать для представления записей в телефонной книге:

```
struct Entry {
    string name;
    int number;
};
```

Можно определить простую операцию вывода, которая пишет **Entry**, используя формат {«*name*», *number*}:

```
ostream& operator<<(ostream& os, const Entry& e)
{
    return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

Пользовательская операция вывода принимает (по ссылке) поток вывода в качестве первого аргумента и возвращает его в качестве результата. Подробности см. в §38.4.2.

Соответствующая операция ввода более сложна, так как она должна проверять надлежащее форматирование и отслеживать ошибки:

```
istream& operator>>(istream& is, Entry& e)
    // читаем пару {"name", number}. Форматирование: {" ", и }
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') {// стартуем с {"
        string name;                    // по умолчанию строка пустая: ""
        while (is.get(c) && c!='"')     // все что перед " это часть имени
            name+=c;

        if (is>>c && c==',') {
            int number = 0;
            if (is>>number>>c && c=='}') {// читаем число и }
                e = {name, number};     // очередной вход
                return is;
            }
        }
    }
    is.setf(ios_base::failbit);  // ошибка в потоке
    return is;
}
```

Операция ввода возвращает ссылку на поток ввода, и это можно использовать для проверки успешности операции. Таким образом, выражение **is>>c**, будучи использованным в качестве условия, означает «Удалось ли прочитать из потока **is** в переменную **c**?».

По умолчанию, в **is>>c** пробельные символы пропускаются, но в **is.get(c)** не пропускаются, так что представленная операция для ввода **Entry** игнорирует (пропускает) пробельные символы вне имен, а в именах не пропускает. Например:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin",987654}
```

Мы можем прочитать эту пару записей из потока ввода в переменную типа **Entry** следующим образом:

```
for (Entry ee; cin>>ee;)  // читаем из cin в ee
    cout << ee << '\n'; // пишем ee в cout
```

Вывод будет таков:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

См. §38.4.1 для более полного ознакомления с деталями написания операций ввода для типов, определяемых пользователем. См. §5.5 и Главу 37 для изучения более последовательной техники распознавания шаблонных конструкций в потоке символов (соответствие регулярным выражениям).

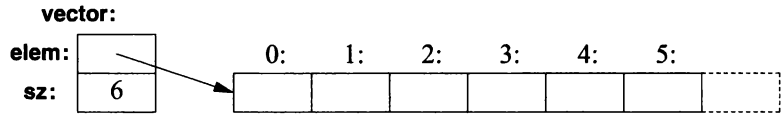
4.4. Контейнеры

В большинстве задач приходится создавать коллекции значений, а затем манипулировать этими коллекциями. Простейшим примером служит задача чтения символов в объект типа **string** с последующей распечаткой этой строки. Класс, предназначенный для хранения объектов, называют *контейнером*. В любой задаче очень важно предоставить подходящие контейнеры вместе с поддерживающими их фундаментальными операциями.

Для иллюстрации контейнеров стандартной библиотеки рассмотрим простую программу, хранящую имена и телефонные номера. Это такая учебная программа, в рамках которой рассмотрение разных подходов окажется одинаково простым и очевидным для людей с разной начальной подготовкой. Для хранения записей телефонной книги подойдет класс **Entry** из §4.3.3. Ради простоты мы будем игнорировать дополнительные сложности, возникающие в реальном мире, например то, что формат телефонных номеров не всегда имеет простое представление как 32-битовое целое.

4.4.1. Класс vector

Наверное, самым полезным контейнером стандартной библиотеки является тип **vector**. Он представляет последовательность элементов заданного типа. Элементы хранятся в памяти компьютера смежным образом:



Пример класса **Vector** из §3.2.2 и §3.4 иллюстрирует идею о том, как реализуется класс **vector**, а подробности рассматриваются в §13.6 и §31.4.

Мы можем проинициализировать **vector** набором значений его элементов:

```
vector<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

К элементам можно обратиться с помощью индексирования:

```
void print_book(const vector<Entry>& book)
{
    for (int i = 0; i!=book.size(); ++i)
        cout << book[i] << '\n';
}
```

Как обычно индексы начинаются с нуля, так что **book[0]** содержит запись для **David Hume**. Метод **size()** класса **vector** возвращает число элементов в этом контейнере.

Элементы контейнера **vector** формируют некий диапазон (range), так что можно использовать цикл range-**for** (§2.2.5):

```
void print_book(const vector<Entry>& book)
{
    for (const auto& x : book)    // про "auto" см. § 2.2.2
        cout << x << '\n';
}
```

Когда мы создаем объект типа **vector**, мы задаем его начальный размер:

```
vector<int> v1 = {1, 2, 3, 4};        // size есть 4
vector<string> v2;                    // size есть 0
vector<Shape*> v3(23);                // size есть 23; Началь. значение элемента: nullptr
vector<double> v4(32,9.9);            // size есть 32; Началь. значение элемента: 9.9
```

Явный размер размещают в круглых скобках, например **(23)**, и по умолчанию элементы принимают умолчательное значение, характерное для их типа (например **nullptr** для указателей, или **0** для чисел). Если же умолчательное значение не подходит, можно указать свое значение в качестве второго аргумента (к примеру, значение **9.9** для **32** элементов в **v4**).

Первоначальный размер можно изменить. Одной из самых полезных операций для типа **vector** служит **push_back()**, которая добавляет новый элемент в конец вектора, увеличивая его размер на единицу. Например:

```
void input()
{
    for (Entry e; cin>>e;)
        phone_book.push_back(e);
}
```

Здесь значения типа **Entry** читаются из стандартного потока ввода в **phone_book** до тех пор, пока не будет достигнут конец файла или же операция ввода не обнаружит ошибку формата. Операция **push_back()** для типа **vector** реализована в стандартной библиотеке очень эффективно.

Тип **vector** поддерживает копирование посредством операции присваивания и инициализации. Например:

```
vector<Entry> book2 = phone_book;
```

Копирование и перемещение объектов типа **vector** реализуется конструкторами и операциями присваивания, как это описано в §3.3. Присваивание для **vector** влечет за собой копирование элементов. Таким образом, после инициализации **book2**, **phone_book** и **book2** содержат отдельные копии каждой записи из телефонной книги. Когда элементов в векторе много, каждое невинно выглядящее присваивание или инициализация могут оказаться весьма дорогостоящими. Когда такая ситуация неприемлема, можно использовать ссылки или указатели (§7.2, §7.7), или же применять перемещающие операции (§3.3.2, §17.5.2).

4.4.1.1. Элементы

Как и все остальные контейнеры стандартной библиотеки, тип **vector** — это контейнер элементов типа **T**, то есть **vector<T>**. Практически все типы могут быть типами элементов: встроенные числовые типы (такие как **char**, **int** или **double**), пользовательские типы (например, **string**, **Entry**, **list<int>** или **Matrix<double,2>**), указатели (например **const char***, **Shape*** или **double***). Когда вы добавляете новый элемент, его значение копируется в контейнер. Например, когда вы помещаете в контейнер целое число со значением **7**, целевой элемент контейнера будет иметь значение **7**. Этот элемент не является ссылкой или указателем на некоторый объект, содержащий значение **7**. Это порождает компактные контейнеры с быстрым доступом.

4.4.1.2. Проверка диапазона

Стандартный **vector** не гарантирует проверку диапазона (§31.2.2). Например:

```
void silly(vector<Entry>& book)
{
    int i = book[ph.size()].number;      // book.size() вне диапазона
    // ...
}
```

Инициализация здесь, скорее всего, поместит в **i** какое-то случайное значение, а не породит ошибку. Это, конечно, нежелательно, и проверка диапазона является общей проблемой. Как следствие, я часто использую простую адаптацию **vector** для проверки диапазона:

```
template<typename T>
class Vec : public std::vector<T> {
public:
    using vector<T>::vector; // конструкторы из vector (под именем Vec);
                             // см. § 20.3.5.1

    T& operator[](int i)                 // проверка диапазона
        {return vector<T>::at(i);}

    const T& operator[](int i) const     // проверка диапазона константн. объектов; § 3.2.1.1
        {return vector<T>::at(i);}
};
```

Класс **Vec** наследует от **vector** все, кроме операции индексирования, которую он переопределяет для проверки диапазона. Операция **at()** — это операция индексирования для **vector**, которая генерирует исключение типа **out_of_range** в случае, когда ее аргумент выходит из корректного диапазона элементов (§2.4.3.1, §31.2.2).

Для типа **Vec** попытка обращения по индексу, выходящему за пределы корректного диапазона, вызывает исключение, которое пользователь может перехватывать. Например:

```
void checked(Vec<Entry>& book)
{
    try {
        book[book.size()] = {"Joe",999999};      // сгенерирует исключение
        // ...
    }
    catch (out_of_range) {
        cout << "range error\n";
    }
}
```

Исключение будет сгенерировано и перехвачено (§2.4.3.1, Глава 13). Если пользователь не будет перехватывать исключение, программа завершится предсказуемым образом, а не продолжит свое выполнение с неясными последствиями. Одним из способов минимизировать сюрпризы от неперехваченных исключений является помещение **try**-блока в тело функции **main()**:

```
int main()
try {
    // ...ваш код
}
catch (out_of_range) {
    cerr << "range error\n";
}
```

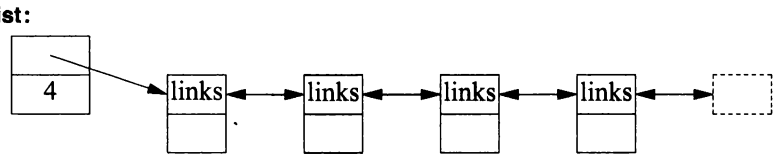
```
catch (...) {
    cerr << "unknown exception thrown\n";
}
```

Здесь имеются умолчательные обработчики исключений, так что если мы не сможем перехватить какое-либо исключение, сообщение об ошибке будет выведено в стандартный поток вывода диагностических сообщений **cerr** (§38.1).

Некоторые реализации самостоятельно предоставляют вариант **vector** с проверкой диапазона, избавляя вас от необходимости использования класса **Vec** (или чего-либо эквивалентного).

4.4.2. Контейнер list

Стандартная библиотека предлагает двусвязный список **list**:



Мы применяем **list** для хранения последовательностей, в которых мы хотим добавлять и удалять элементы без необходимости перемещения остальных элементов. Добавление и удаление записей в телефонной книге являются типичными операциями, и поэтому **list** подходит для представления простой телефонной книги. Например:

```
list<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Когда применяют связанный список, стараются не обращаться к элементам посредством индексирования (в отличие от векторов, для которых это типично). Вместо этого мы просматриваем список в поиске элемента с заданным значением. Тут мы опираемся на факт, что **list** является последовательностью, как это описано в §4.5:

```
int get_number(const string& s)
{
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
    return 0; // используем 0 для индикации "номер не найден"
}
```

Поиск **s** начинается с самого начала списка и продолжается до тех пор, пока **s** не будет обнаружен, или же не будет достигнут конец **phone_book**.

Иногда нужно идентифицировать некоторый элемент списка. Например, нам может потребоваться удалить его или вставить перед ним новый элемент. Для этого нам нужен *итератор*: итератор списка идентифицирует элемент списка и может использоваться для прохода (итерации — отсюда название) по всем элементам. Каждый контейнер стандартной библиотеки предоставляет функции **begin()** и **end()**, которые возвращают итераторы на первый элемент и на «следующий за последним» («one-past-the-last element»), соответственно (§4.5, §33.1.1). Используя явным образом итераторы, мы можем (хотя и менее элегантно) написать функцию **get_number()** следующим образом:

```
int get_number(const string& s)
{
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
        if (p->name==s)
            return p->number;
    return 0; // используем 0 для индикации "номер не найден"
}
```

На самом деле, компилятор примерно таким образом и реализует цикл range-**for**. Если **p** это итератор, то ***p** есть элемент, на который он указывает, **++p** заставляет **p** указывать на следующий элемент, и если **p** относится к классу с членом **m**, то **p->m** эквивалентно **(*p).m**.

В список типа **list** добавлять и удалять элементы очень легко:

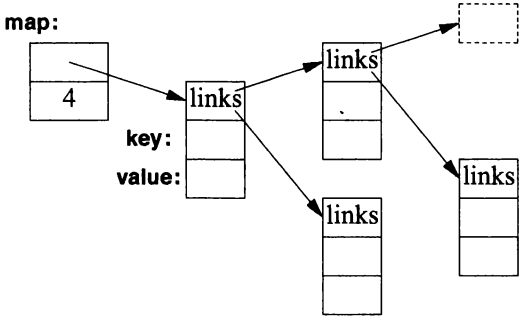
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
    phone_book.insert(p, ee);      // добавляем ее перед элементом со ссылкой p
    phone_book.erase(q);           // удаляем элемент со ссылкой q
}
```

Более полное описание методов **insert()** и **erase()** представлено в §31.3.7.

Показанные нами примеры с контейнером **list** можно применить точно так же и к контейнеру **vector**, причем на удивление второй вариант будет работать быстрее с векторами малых размеров, чем со списками малых размеров. Если все, что нам нужно, это просто последовательность элементов, то имеется выбор между **vector** и **list**. Если нет специальных причин против, следует предпочесть **vector**. Этот контейнер лучше работает для прохода по всем элементам, а также при сортировке и поиске (**sort()** и **binary_search()**).

4.4.3. Контейнер map

Достаточно утомительно писать код для поиска *имени* в списке элементов типа пары (*имя*, *число*). Кроме того, линейный поиск крайне неэффективен для всех списков, кроме очень коротких. Стандартная библиотека предлагает структуру поискового дерева (*красно-черное дерево — red-black tree*), названную **map**:



В других контекстах **map** известен как *ассоциативный массив* (*associative array*) или же как *словарь* (*dictionary*): реализуется как «сбалансированное бинарное дерево» («balanced binary tree»).

Контейнер **map** стандартной библиотеки (§31.4.3) — это контейнер пар значений, оптимизированный для поиска. Можно использовать тот же самый инициализатор, что и для **vector** или **list** (§4.4.1, §4.4.2):

```
map<string, int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

При индексации по значению первого типа (называемого *key*) **map** возвращает соответствующее значение второго типа (называемого *value* или *mapped value*). Например:

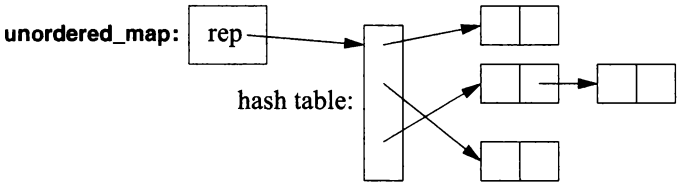
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

Другими словами, индексация для **map** — это по сути поиск, который мы просто назвали здесь **get_number()**. Если же ключ не отыскивается, он внедряется в контейнер **map** с умолчательным значением для **value**. Для целого типа умолчательным значением служит **0**; как раз то значение, которое я выбрал для индикации неправильного телефонного номера.

Если же мы хотим избежать внедрения неправильных телефонных номеров в телефонную книгу, мы можем использовать **find()** и **insert()** вместо операции индексации [] (§31.4.3.1).

4.4.4. Контейнер unordered_map

Стоимость поиска в **map** оценивается как **O(log(n))**, где **n** — это число элементов в **map**. Это хорошая оценка. Например, для **map** с 1000000 элементов, мы выполняем порядка 20 сравнений и косвенных обращений до нахождения элемента. Тем не менее, во многих случаях можно достичь лучшего результата, используя хэшированный поиск, а не сравнения на базе операций, таких как **<**. Хэшированные контейнеры стандартной библиотеки именуются как «unordered» («неупорядоченные»), потому что они не требуют упорядочивающей функции:



К примеру, мы можем применить **unordered_map** для нашей телефонной книги:

```
unordered_map<string, int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Как и для **map**, для **unordered_map** можно применять индексирование:

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

Контейнер **unordered_map** стандартной библиотеки предоставляет умолчательную хэш-функцию для типа **string**. При необходимости, вы можете предоставить собственную функцию (§31.4.3.4).

4.4.5. Обзор контейнеров

Стандартная библиотека предоставляет многие из наиболее общих и полезных типов контейнеров, и программист может выбрать тот из них, что лучше всего отвечает требованиям конкретного приложения:

Стандартные контейнеры	
vector<T>	Вектор переменной длины (§31.4)
list<T>	Двусвязанный список (§31.4.2)
forward_list<T>	Односвязанный список (§31.4.2)
deque<T>	Двусторонняя очередь (§31.2)
set<T>	Множество (§31.4.3)
multiset<T>	Множество со многими вхождениями значений (§31.4.3)
map<K, V>	Ассоциативный массив (§31.4.3)
multimap<K, V>	Ассоциатив. массив со многими вхождениями (§31.4.3)
unordered_map<K, V>	Контейнер map с хэшированным поиском (§31.4.3.2)
unordered_multimap<K, V>	Контейнер multimap с хэшированным поиском (§31.4.3.2)
unordered_set<T>	Множество с хэшированным поиском (§31.4.3.2)
unordered_multiset<T>	Контейнер multiset с хэшированным поиском (§31.4.3.2)

Неупорядоченные контейнеры оптимизированы для поиска по ключу (часто, строковому типу **string**); другими словами, они реализуются посредством хэш-таблиц.

Стандартные контейнеры рассматриваются в §31.4. Контейнеры определяются в пространстве имен **std** и их определения сосредоточены в файлах **<vector>**, **<list>**, **<map>** и т.д. (§4.1.2, §30.2). Дополнительно, стандартная библиотека предоставляет контейнерные адаптеры **queue<T>** (§31.5.2), **stack<T>** (§31.5.1), **deque<T>** (§31.4) и **priority_queue<T>** (§31.5.3). Стандартная библиотека предоставляет также и более специализированные контейнероподобные типы, такие как массив фиксированного размера **array<T, N>** (§34.2.1) и **bitset<N>** (§34.2.2).

Стандартные контейнеры и их базовые операции спроектированы так, чтобы быть сходными с точки зрения синтаксиса. Более того, смысл операций эквивалентен для разных контейнеров. Базовые операции применимы к любым контейнерам, где они имеют смысл и могут быть эффективно реализованы. Например:

- Функции **begin()** и **end()** возвращают итераторы на первый элемент и на элемент, следующий за последним, соответственно.
- Операция **push_back()** может эффективно добавлять элементы в конец последовательности для типов **vector**, **forward_list**, **list** и других.
- Метод **size()** возвращает число элементов.

Синтаксическая и семантическая однородности позволяют программисту создавать новые типы контейнеров, которые можно использовать практически так же, как и стандартные контейнеры. Вектор с проверкой диапазона, тип **Vector** (§2.3.2, §2.4.3.1) служат наглядными примерами. Однородность интерфейсов контейнеров позволяют нам определять алгоритмы независимо от индивидуальных типов контейнеров. Тем не менее, у всего есть сильные и слабые стороны. Например, индексирование и проход по всем элементам для типа **vector** являются дешевыми операциями; а для **list** все наоборот. Также отметим, что **vector** часто более эффективен для коротких последовательностей небольших элементов, чем **list** (даже для **insert()** и **erase()**). Я бы рекомендовал тип **vector** в качестве умолчательного типа для последовательностей элементов: нужно иметь конкретные причины для отказа от этого выбора.

4.5. Алгоритмы

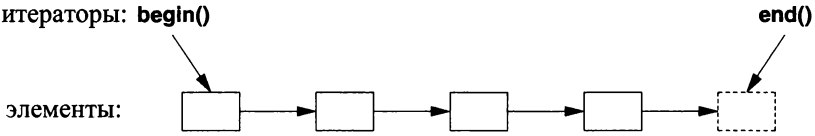
Любые структуры данных, такие как вектор или список, сами по себе дают мало пользы. Чтобы их использовать, нужны базовые операции, такие как добавление и уда-

ление элементов (как это имеет место для **list** и **vector**). Кроме того, мы редко когда ограничиваемся лишь хранением элементов в контейнерах. Нам нужно их сортировать, выводить их на печать, извлекать подпоследовательности, выполнять поиск объектов и т.д. Как следствие, стандартная библиотека предоставляет не только контейнерные типы, но и наиболее общие алгоритмы для них. В следующем примере мы выполняем сортировку для **vector** и помещаем копии всех уникальных элементов в **list**:

```
bool operator<(const Entry& x, const Entry& y)  // меньше чем
{
    return x.name<y.name;   // сортируем Entry по их именам
}

void f(vector<Entry>& vec, list<Entry>& lst)
{
    sort(vec.begin(), vec.end());                    // используем < для упорядочения
    unique_copy(vec.begin(), vec.end(), lst.begin()); // не копируем смежные равные эл-ты
}
```

Стандартные алгоритмы рассматриваются в Главе 32. Они формулируются в терминах последовательностей элементов. *Последовательность* представляется парой итераторов, специфицирующих первый элемент и элемент, следующий за последним:



В нашем примере **sort()** сортирует последовательность, зафиксированную парой итераторов **vec.begin()** и **vec.end()** — которые в данном случае являются всеми элементами **vector**. Для записи нужно специфицировать лишь первый элемент. Если записывается более одного элемента, все элементы, следующие за указуемым, будут переписаны. Таким образом, чтобы избежать ошибок, **lst** должен иметь по крайней мере столько же элементов, сколько имеется уникальных элементов в **vec**.

Если бы нам нужно было поместить уникальные элементы в новый контейнер, мы бы тогда написали следующий код:

```
list<Entry> f(vector<Entry>& vec)
{
    list<Entry> res;
    sort(vec.begin(), vec.end());
    unique_copy(vec.begin(), vec.end(), back_inserter(res));      // добавляем к res
    return res;
}
```

Метод **back_inserter()** добавляет элементы в конец контейнера, расширяя контейнер так, чтобы всем хватило места (§33.2.2). В результате, стандартные контейнеры плюс **back_inserter()** устраняют необходимость в чреватом ошибками применении функции **realloc()** (§31.5.1) для управления памятью в С-стиле. Контейнер **list** стандартной библиотеки располагает перемещающим конструктором (§3.3.2, §17.5.2), который возвращает **res** по значению весьма эффективно (даже в случае многих тысяч элементов).

Если вы находите стиль кода с парой итераторов неуклюжим, например **sort(vec.begin(), vec.end())**), то можете опеределить отдельную версию алгоритма для контейнера и писать **sort(vec)** (§4.5.6).

4.5.1. Использование итераторов

Для любого контейнера вы можете легко получить несколько итераторов на полезные элементы; **begin()** и **end()** служат примерами. Кроме того, многие алгоритмы возвращают итераторы. Например, стандартный алгоритм **find()** ищет значение в последовательности и возвращает итератор на найденный элемент:

```
bool has_c(const string& s, char c)          // содержит ли s символ c?
{
    auto p = find(s.begin(), s.end(), c);
    if (p!=s.end())
        return true;
    else
        return false;
}
```

Как и многие поисковые алгоритмы стандартной библиотеки, **find()** возвращает **end()** для индикации результата «не найдено». Эквивалентная, но более короткая версия **has_c()** имеет следующий вид:

```
bool has_c(const string& s, char c)          // содержит ли s символ c?
{
    return find(s.begin(), s.end(), c)!=s.end();
}
```

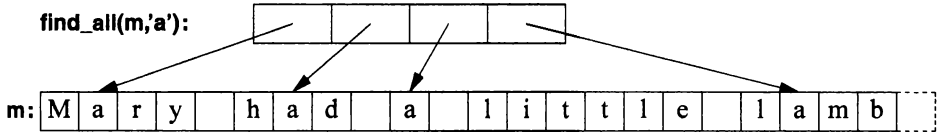
В качестве более интересного примера можно рассмотреть задачу о нахождении всех позиций заданного символа в строке. Вернуть при этом можно будет **vector** из итераторов для типа **string**. Выбор **vector** здесь будет эффективен, ибо этот тип обеспечивает семантику перемещения (§3.3.1). В предположении, что нам может понадобиться модифицировать найденные местоположения символа, передаем не-**const string**:

```
vector<string::iterator> find_all(string& s, char c) // найти все вхождения c в s
{
    vector<string::iterator> res;
    for (auto p = s.begin(); p!=s.end(); ++p)
        if (*p==c)
            res.push_back(p);
    return res;
}
```

Здесь последовательный просмотр строки выполняется в стандартном цикле, когда итератор **p** передвигается при каждой итерации от элемента к следующему элементу при помощи операции **++**, а доступ к элементам выполняется операцией разыменования ***.** Протестировать **find_all()** можно следующим образом:

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))
        if (*p!='a')
            cerr << "a bug!\n";
}
```

Вызов **find_all()** можно проиллюстрировать графически:



Итераторы и стандартные алгоритмы работают эквивалентным образом на всех стандартных контейнерах, для которых их использование имеет смысл. Как следствие, мы можем обобщить **find_all()**:

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)  // найти все вхождения v в c
{
    vector<typename C::iterator> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

Ключевое слово **typename** требуется для того, чтобы сообщить компилятору, что **iterator** у **C** это тип, а не значение некоторого типа, вроде целого числа **7**. Можно спрятать эту деталь реализации, введя синоним для типа (§3.4.5) **iterator**:

```
template<typename T>
using Iterator<T> = typename T::iterator;

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v) // найти все вхождения v в c
{
    vector<Iterator<C>> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

Теперь мы можем написать:

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))    // p это string::iterator
        if (*p!='a')
            cerr << "string bug!\n";

    list<double> ld {1.1, 2.2, 3.3, 1.1};
    for (auto p : find_all(ld,1.1))
        if (*p!=1.1)
            cerr << "list bug!\n";

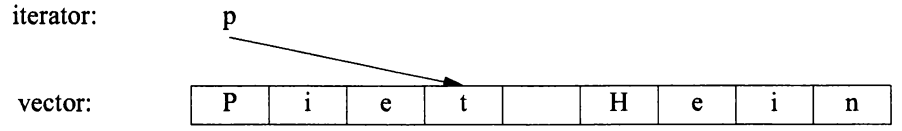
    vector<string> vs {"red", "blue", "green", "green", "orange", "green"};
    for (auto p : find_all(vs,"green"))
        if (*p!="green")
            cerr << "vector bug!\n";

    for (auto p : find_all(vs,"green"))
        *p = "vert";
}
```

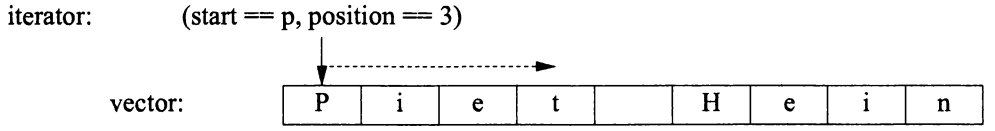
Итераторы позволяют отделить алгоритмы от контейнеров. Алгоритмы работают над данными через итераторы и ничего не знают о контейнерах, в которых располагаются элементы. В свою очередь, контейнеры ничего не знают об алгоритмах, которые оперируют над их элементами; они лишь предоставляют итераторы по запросу (то есть, через **begin()** и **end()**). Такая модель позволяет писать обобщенные и гибкие программы.

4.5.2. Типы итераторов

Что же в реальности представляют собой итераторы? Каждый конкретный итератор есть объект некоторого типа. Существует много разных типов итераторов, поскольку итераторы должны содержать информацию, необходимую для выполнения работы для конкретного типа контейнеров. Например, итератор для **vector** может быть простым указателем, поскольку это вполне разумный способ обращения к его элементам:

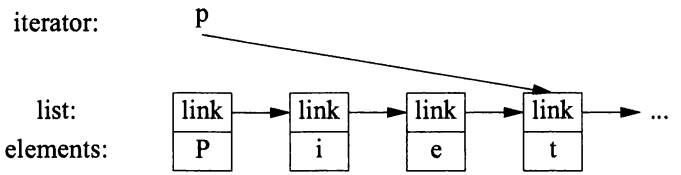


Можно итератор для типа **vector** организовать и по-другому — как указатель на его начало плюс индекс (смещение):



Применение такого итератора позволяет выполнять проверку диапазона.

Итератор для **list** устроен сложнее, чем просто указатель на элементы, потому что элемент списка в общем случае не знает, где находится следующий элемент списка. Например, итератор для **list** мог бы быть указателем на связь (link):



Общим для всех итераторов является их семантика и названия операций. Например, применение операции ++ к любому итератору дает итератор, указывающий на следующий элемент. Аналогично, операция * возвращает элемент, на который настроен итератор. Фактически, любой объект, подчиняющийся простым правилам вроде тех, что мы сейчас рассмотрели, является итератором (§33.1.4). Более того, пользователям редко когда может понадобиться знать тип конкретного итератора; каждый контейнер знает тип итераторов и предоставляет их под стандартными именами **iterator** и **const_iterator**. К примеру, итераторным типом для **list<Entry>** является тип **list<Entry>::iterator**. Редко когда мы должны беспокоиться о том, как именно этот тип определен.

4.5.3. Итераторы потоков

Итераторы являются полезной концепцией для работы с последовательностями элементов в контейнерах. Однако, контейнерами не исчерпываются все случаи, где мы имеем дело с последовательностями элементов. К примеру, входной поток производит последовательность значений, и последовательности значений мы пишем в выходной поток. Следовательно, понятие итераторов может быть легко применено к входным и выходным потокам.

Для создания **ostream_iterator** мы должны специфицировать поток, который будем использовать, а также тип объектов, которые мы будем писать в него. Например:

```
ostream_iterator<string> oo {cout}; // пишем строки в cout
```

Если присвоить что-либо выражению ***oo**, то присваиваемое значение будет записано в **cout**. Например:

```
int main()
{
    *oo = "Hello, ";     // cout<<"Hello, "
    ++oo;
    *oo = "world!\n";    // cout<<"world!\n"
}
```

Здесь показан еще один способ вывести каноническое приветствие (Hello, world!) в стандартный выходной поток.

Аналогично, **input_stream** есть нечто, что позволяет нам трактовать входной поток как контейнер «только для чтения». И снова мы должны специфицировать поток, который будем использовать, а также тип ожидаемых значений:

```
istream_iterator<string> ii {cin};
```

Входные итераторы используются парой, представляющей последовательность, так что мы должны предоставить и **istream_iterator** для индикации конца ввода. Это умолчательный **istream_iterator**:

```
istream_iterator<string> eos {};
```

В типичном случае, итераторы **istream_iterator** и **ostream_iterator** не используются напрямую. Вместо этого, они передаются алгоритмам в качестве аргументов. Например, мы можем написать простую программу для чтения файла, сортировки прочитанных слов, устранения дубликатов, а также для вывода результата в другой файл:

```
int main()
{
    string from, to;
    cin >> from >> to;                    // получить исходное и конечное имена файлов

    ifstream is {from};                   // поток ввода для файла "from"
    istream_iterator<string> ii {is};     // входной итератор для потока
    istream_iterator<string> eos {};      // input sentinel

    ofstream os{to};                      // выходной поток для файла "to"
    ostream_iterator<string> oo {os,"\n"}; // выходной iterator для потока

    vector<string> b {ii, eos};           // b-вектор, созданный из  потока [ii: eos)
    sort(b.begin(), b.end());             // сортируем буфер

    unique_copy(b.begin(), b.end(), oo);  // копируем буфер в выход, устраняем дубликаты

    return!is.eof() ||!os; // возвращаем состояние ошибки (§2.2.1, §38.3)
}
```

Тип **ifstream** — это **istream**, который может быть прикреплен к файлу, а **ofstream** — это **ostream**, который тоже прикрепляется к файлу. Второй аргумент при создании **ostream_iterator** используется для разделения выходных значений.

На самом деле, эта программа несколько длиннее, чем могла бы быть. Мы читаем строки в **vector**, затем сортируем их при помощи **sort()**, а потом выводим их, устраняя ду-

бликаты. Более элегантное решение — не хранить дубликаты вообще. Это можно сделать, если хранить строки в контейнере типа **set**, который не позволяет хранить дубликаты и сам сохраняет порядок в следовании элементов (§31.4.3). Таким образом мы можем заменить две строки кода с **vector** одной строкой кода с **set**, и заменить **unique_copy()** на **copy()**:

```
set<string> b {ii, eos};          // собираем входные строки
copy(b.begin(), b.end(), oo);     // копируем буфер в выход
```

Мы использовали имена **ii**, **eos** и **oo** лишь однажды, так что есть возможность еще подсократить код:

```
int main()
{
    string from, to;
    cin >> from >> to;          // получаем имена входного и выходного файлов

    ifstream is {from};         // поток ввода для файла "from"
    ofstream os {to};           // поток ввывода для файла "to"

set<string> b {istream_iterator<string>{is}, istream_iterator<string>{}}; // читаем вход
copy(b.begin(), b.end(), ostream_iterator<string>{os,"\n"});              // копируем в выход

return!is.eof() ||os;           // возвращаем состояние ошибки (§2.2.1, §38.3)
}
```

Вопрос о том, улучшила ли последняя версия читаемость кода, является дискуссионным.

4.5.4. Предикаты

В предыдущих примерах алгоритмы встраивали некоторое действие, которое должно было выполняться над каждым элементом последовательности. Часто бывает нужным сделать такое действие аргументом алгоритма. К примеру, алгоритм **find()** (§32.4) дает возможность искать заданное значение удобным способом. Более общим способом поиска является тот, что позволяет искать элемент, удовлетворяющий заданному условию — *предикату* (§3.4.2). Например, мы можем искать в контейнере **map** первый элемент, величина которого больше **42**. Контейнеры типа **map** открывают доступ к своим элементам как к последовательности пар (*key*, *value*), так что мы можем искать в последовательности **map<string, int>** пару **pair<const string, int>**, в которой значение типа **int** больше, чем **42**:

```
void f(map<string, int>& m)
{
    auto p = find_if(m.begin(), m.end(), Greater_than{42});
    // ...
}
```

Здесь **Greater_than** является функциональным объектом (§3.4.3), содержащим значение **42**, с которым нужно проводить сравнение:

```
struct Greater_than {
    int val;
    Greater_than(int v) : val{v} {}
    bool operator()(const pair<string, int>& r) {return r.second>val;}
};
```

Можно также применить и лямбда-выражение (§3.4.3):

```
int cxx = count_if(m.begin(), m.end(), [](const pair<string, int>& r) {return r.second>42;});
```

4.5.5. Обзор алгоритмов

Самым общим определением алгоритма служит фраза «конечный набор правил, порождающих последовательность операций для решения конкретного набора задач и имеющих пять существенных черт: Конечность ... Определенность ... Вход ... Выход ... Эффективность» [Кнут, 1968, §1.1]. В контексте стандартной библиотеки языка C++ алгоритмом является шаблонная функция, оперирующая на последовательности элементов.

Стандартная библиотека предоставляет десятки алгоритмов. Алгоритмы объявлены в пространстве имен **std** и определены в заголовочном файле **<algorithm>**. Все эти алгоритмы принимают последовательности в качестве входа (§4.5). Полуоткрытая последовательность от **b** до **e** обозначается как **[b:e)**. Ниже я перечисляю некоторые из алгоритмов, которые кажутся мне особо полезными:

Избранные стандартные алгоритмы	
p=find(b, e, x)	**p** первый **p** в **[b:e)** так что ***p==x**
p=find_if(b, e, f)	**p первый p в [b:e) так что f(*p)==true**
n=count(b, e, x)	**n** число элементов ***q** в **[b:e)** так что ***q==x**
n=count_if(b, e, f)	**n число элементов *q** в **[b:e)** так что **f(*q, x)**
replace(b, e, v, v2)	Заменить элементы ***q** в **[b:e)** так что ***q==v by v2**
replace_if(b, e, f, v2)	Заменить элементы ***q** в **[b:e)** так что **f(*q) by v2**
p=copy(b, e, out)	Копировать **[b:e)** в **[out:p)**
p=copy_if(b, e, out, f)	Копировать элементы ***q** из **[b:e)** так что **f(*q)** в **[out:p)**
p=unique_copy(b, e, out)	Копировать **[b:e)** в **[out:p)**; не копировать смежные дубликаты
sort(b, e)	Сортировать элементы из **[b:e)** используя **<** в качестве критерия сортировки
sort(b, e, f)	Сортировать элементы из **[b:e)** используя **f**
(p1, p2)=equal_range(b, e, v)	**[p1:p2)** подмножество сортированной последовательности **[b:e)** со значением **v** (ищется бинарным поиском)
p=merge(b, e, b2, e2, out)	Слияние сортированных последовательностей **[b:e)** и **[b2:e2)** в **[out:p)**

Эти алгоритмы, а также множество других (см. Главу 32), могут применяться к элементам контейнеров, к строкам типа **string** и к встроенным массивам.

4.5.6. Алгоритмы для контейнеров

Последовательность определяется парой итераторов **[begin:end)**. Это общее и гибкое определение, но чаще всего мы применяем алгоритмы к последовательности, являющейся содержимым контейнера. Например:

```
sort(v.begin(), v.end());
```

Почему бы не иметь вариант типа **sort(v)**? Это можно легко организовать следующим образом:

```
namespace Estd {
    using namespace std;

    template<class C>
    void sort(C& c)
```

```
    {
        sort(c.begin(), c.end());
    }

    template<class C, class Pred>
    void sort(C& c, Pred p)
    {
        sort(c.begin(), c.end(), p);
    }

    // ...
}
```

Я поместил контейнерную версию **sort()** в свое собственное пространство имен **Estd** (*extended* **std**) во избежание противоречий с иными вариантами, использующими пространство имен **std**.

4.6. Советы

[1] Не изобретайте колесо заново; используйте библиотеки; §4.1.

[2] Когда есть выбор, предпочитайте остальным вариантам именно стандартную библиотеку; §4.1.

[3] Не думайте, что стандартная библиотека идеальна для любых случаев; §4.1.

[4] Не забывайте про директивы **#include** с соответствующими хедер-файлами стандартной библиотеки; §4.1.2.

[5] Не забывайте, что все средства стандартной библиотеки определяются в пространстве имен **std**; §4.1.2.

[6] Предпочитайте строки типа **string** строкам в стиле языка C (**char***; §2.2.5); §4.2, §4.3.2.

[7] Потоки **iostream** безопасны по типам и расширяемы; §4.3.

[8] Вместо **T[]** используйте **vector<T>**, **map<K, T>** или **unordered_map<K, T>**; §4.4.

[9] Изучите варианты стандартных контейнеров, их достоинства и недостатки; §4.4.

[10] По умолчанию используйте контейнер **vector**; §4.4.1.

[11] Предпочитайте компактные структуры данных; §4.4.1.1.

[12] В случае сомнений используйте вектор с проверкой диапазона (вроде **Vec**); §4.4.1.2.

[13] Используйте **push_back()** или **back_inserter()** для добавления элементов в контейнер; §4.4.1, §4.5.

[14] Используйте **push_back()** для **vector**, а не **realloc()** для массивов; §4.5.

[15] Перехватывайте общие исключения в функции **main()**; §4.4.1.2.

[16] Изучайте стандартные алгоритмы и применяйте их вместо доморощенного кода; §4.5.5.

[17] Если надоело работать с итераторами, определите контейнерные алгоритмы; §4.5.6.

5

Обзор языка C++: Параллелизм и Утилиты

Обучая, обучай кратко.
— Цицерон

- Введение
- Управление ресурсами
  unique_ptr и **shared_ptr**
- Параллелизм
  Задачи и потоки; Передача аргументов; Возврат результатов; Разделяемые данные; Взаимодействие задач
- Утилиты — небольшие полезные компоненты библиотеки
  Время; Функции, для работы с типами; **pair** и **tuple**
- Регулярные выражения
- Математика
  Математические функции и алгоритмы; Комплексные числа; Случайные числа; Векторная арифметика; Числовые пределы)
- Советы

5.1. Введение

С точки зрения конечного пользователя идеальная стандартная библиотека должна предоставлять компоненты, непосредственно удовлетворяющие потребностям практически любой конкретной задачи. Для фиксированной предметной области какая-нибудь мощная коммерческая библиотека в состоянии плотную приблизиться к этому идеалу. Но это совсем не то, что старается сделать стандартная библиотека языка C++. Общедоступная и универсальная библиотека не может дать все и всем. Вместо этого стандартная библиотека C++ предоставляет компоненты, которые полезны большинству программистов в большинстве приложений для разных предметных областей. То есть, она старается удовлетворить пересечению множества потребностей, а не их объединению. А кроме этого, она еще дает некоторую поддержку крайне важным областям, таким как математические вычисления и обработка текстовой информации.

5.2. Управление ресурсами

Практически в любой нетривиальной программе ключевой задачей является управление ресурсами. Ресурс — это нечто, что нужно сначала заполучить, а позднее (явно или неявно) освободить. Примерами ресурсов служат память, блокировки, сокеты, дескрип-

торы потоков и файловые дескрипторы. В любой длительно исполняющейся программе серьезная задержка в освобождении ресурса приводит к постепенной деградации в эффективности, что в конце концов может превратиться в окончательный крах программы. Даже в коротких программах утечки ресурсов могут привести к их дефициту, увеличивая время исполнения программ многократно.

Стандартная библиотека разработана в манере, помогающей избегать утечки ресурсов. С этой целью она опирается на такие базовые средства языка C++, как конструкторы/деструкторы, гарантирующие, что захваченные ресурсы не переживут отвечающих за них объектов. Примером может служить пара конструктор/деструктор класса **Vector** для управления временем жизни элементов (§3.2.1.2), и все контейнеры стандартной библиотеки реализованы в этом плане аналогично. Очень важно, что такой подход отлично взаимодействует с обработкой ошибок посредством исключений. К примеру, эта техника используется в стандартной библиотеке в классах блокировок:

```
mutex m; // контролирует доступ к разделяемым данным
// ...
void f()
{
    unique_lock<mutex> lck {m}; // захват мьютекса m
    // ... манипулируем разделяемыми данными ...
}
```

Поток (**thread**) не будет исполняться, пока конструктор для **lck** не захватит свой семафор **m** (**mutex** — *mutual exclusion*, взаимное исключение; иногда называют исключающими семафорами) (§5.3.4). Соответствующий деструктор будет освобождать этот ресурс. В этом примере деструктор класса **unique_lock** освобождает мьютекс (типа **mutex**) тогда, когда поток исполнения покинет функцию **f()** (посредством корректного возврата или же, скажем, за счет генерации исключения).

Это является реализацией фундаментальной техники «Захват ресурса есть Инициализация» (§3.2.1.2, §13.3). Контейнеры (такие как **vector** и **map**), строки **string** и потоки **iostream** управляют ресурсами (вроде файловых дескрипторов и буферов) аналогичным образом.

5.2.1. Интеллектуальные указатели unique_ptr и shared_ptr

В примерах, рассмотренных до сих пор, объекты, захватившие ресурсы в некоторой области видимости, освобождали их по выходу из этой области видимости. А что происходит с ресурсами, выделяемыми для объектов в свободной памяти? Для управления объектами в свободной памяти стандартная библиотека в файле **<memory>** определяет два вида интеллектуальных указателей:

1. **unique_ptr** для отражения конценции уникального владения (§34.3.1)
2. **shared_ptr** для отражения конценции разделяемого владения (§34.3.2)

Самое важное применение этих интеллектуальных указателей нацелено на предотвращение утечек памяти, являющихся следствием недостаточно внимательного программирования. Например:

```
void f(int i, int j)                    // X* сравниваем с unique_ptr<X>
{
    X* p = new X;                       // создаем новый X
    unique_ptr<X> sp {new X};           // создаем новый X и указатель на него отдаем unique_ptr
    // ...
    if (i<99) throw Z{};                // может генерироваться исключение
    if (j<77) return;                   // ранний возврат
    p->do_something();                  // может генерироваться исключение
    sp->do_something();                 // может генерироваться исключение
```

```
    // ...
    delete p; // уничтожаем *p
}
```

Здесь мы «забыли» применить операцию **delete** к **p** в случае **i<99** и в случае **j<77**. В то же самое время, **unique_ptr** гарантирует, что владеемый им объект будет надлежащим образом уничтожен независимо от того, как был осуществлен выход из функции **f()** (посредством исключительной ситуации, через обычный оператор **return** или же по исчерпании кода тела функции). Так что мы решаем проблему, просто не применяя указателей и операций **new**:

```
void f(int i, int j)    // используем локальную переменную
{
    X x;
    // ...
}
```

К сожалению, избыточное использование **new** (и указателей, и ссылок) кажется все возрастающей проблемой.

А ведь использование **unique_ptr** в случаях, когда требуется семантика указателей, является простой и необременяющей заменой, не влекущей избыточного потребления памяти и времени работы по сравнению с применением встроенных указателей. Его иное применение заключается в передаче объектов, размещаемых в свободной памяти, в функцию и их возврат из нее:

```
unique_ptr<X> make_X(int i)
    // создаем X и отдаем его unique_ptr
{
    // ... проверяем i, и т. д....
    return unique_ptr<X>{new X{i}};
}
```

Интеллектуальный указатель **unique_ptr** является дескриптором для индивидуального объекта (или массива) аналогично тому, как **vector** служит дескриптором для последовательности объектов. Оба управляют временем жизни объектов и оба полагаются на семантику перемещения, чтобы выполнять возврат просто и эффективно.

Интеллектуальный указатель **shared_ptr** аналогичен **unique_ptr** за исключением того, что он скорее копирует, чем перемещает. Многочисленные **shared_ptr** на объект разделяют владение объектом, и объект уничтожается лишь тогда, когда уничтожается последний из этих **shared_ptr**. Вот пример:

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
    shared_ptr<fstream> fp {new fstream(name, mode)};
    if (!*fp) throw No_file{};    // проверка корректности открытия файла

    f(fp);
    g(fp);
    // ...
}
```

Теперь файл, открытый конструктором **fp**, будет закрываться последней функцией, которая (явно или неявно) уничтожает копию **fp**. Заметьте, что **f()** или **g()** могут запустить задачу с отдельной копией **fp**, или же иным каким-то способом хранить копию, которая переживет **user()**. Таким образом, **shared_ptr** реализует некоторую форму «сборки мусора»,

опирающуюся на деструкторы в управлении ресурсами при работе с объектами в памяти. Это не является бесплатным, и не является слишком дорогим методом, но делает время жизни объекта менее предсказуемым. Так что используйте **shared_ptr** исключительно в случаях, когда на самом деле требуется совместное владение объектами.

Применяя **unique_ptr** и **shared_ptr**, мы можем реализовать подход «никаких голых **new**» (§3.2.1.2) применительно ко многим программам. Тем не менее, эти «интеллектуальные указатели» концептуально по-прежнему указатели и поэтому я ставлю их на второе место в задачах управления ресурсами — после контейнеров и иных типов, которые управляют ресурсами на более высоком концептуальном уровне. В частности, **shared_ptr** сами по себе не формируют никаких правил, по которым их владельцы могут читать/писать в разделяемые объекты. «Гонки за данными» (§41.2.4) и иные формы неопределенностей сами по себе не устраняются отказом от концепций управления ресурсами.

Когда же следует использовать интеллектуальные указатели (такие как **unique_ptr**) вместо применения ресурсных дескрипторов с их операциями, специально разработанными под ресурсы (например, **vector** или **thread**)? Как это не удивительно, но ответ очень простой — тогда, когда нам нужна семантика указателей.

- Когда выполняется совместный (разделяемый) доступ к объекту, нам нужны указатели (или ссылки) на разделяемый объект, так что в этом случае **shared_ptr** является очевидным выбором.
- Когда мы ссылаемся на полиморфный объект, нам требуются указатели (или ссылки), потому что мы не знаем точного типа объекта и даже его размера, так что **unique_ptr** становится очевидным выбором.
- Разделяемые полиморфные объекты в типичном случае требуют **shared_ptr**.

В то же время, нет нужды в указателях при возврате коллекций объектов из функции; контейнер, являющийся ресурсным дескриптором, делает это просто и эффективно (§3.3.2).

5.3. Параллелизм

Параллелизм — это исполнение нескольких задач одновременно — широко используется для повышения производительности (применяется несколько процессоров для решения единственной задачи) или же для улучшения отклика (выполнение одной части задачи продвигается, в то время как другая часть задачи висит в ожидании ответа). Все современные языки программирования поддерживают в той или иной степени параллелизм. Поддержка параллелизма в стандартной библиотеке языка C++ реализует переносимый и безопасный по типам вариант того, что уже более 20 лет используется в C++ и практически без исключения поддерживается аппаратурой современных компьютеров. Подход стандартной библиотеки ориентирован преимущественно на параллелизм системного уровня, а не на изощренные высокоуровневые модели; последние можно реализовать в библиотеках, использующих средства стандартной библиотеки.

Стандартная библиотека напрямую поддерживает параллельное исполнение многих потоков в едином адресном пространстве. Для этого C++ предоставляет подходящую модель памяти (§41.2) и набор атомарных операций (§41.3). В то же время, большинство пользователей будут сталкиваться с параллелизмом в терминах стандартной библиотеки и библиотек, построенных поверх нее. В настоящем разделе кратко приводятся примеры основных средств параллелизма стандартной библиотеки: **thread**, **mutex**, **lock()**, **packaged_task** и **future**. Эти средства построены непосредственно на том, что предоставляют в этом отношении операционные системы, и не налагают никаких избыточных проблем с производительностью и эффективностью.

5.3.1. Задачи и потоки

Мы называем *задачей* (*task*) некую вычислительную активность, которая может потенциально выполняться параллельно с иными вычислениями. *Поток* (*thread*) является представлением задачи на системном уровне. Задача, которая будет выполняться параллельно другим задачам, инициируется путем создания объекта типа **std::thread** (определяется в файле **<thread>**), которому при конструировании эта задача будет передана в качестве аргумента. Задача это функция или функциональный объект:

```
void f();                   // функция

struct F {                  // функциональный объект
    void operator()();      // операция вызова для F (§3.4.3)
};

void user()
{
    thread t1 {f};          // f() выполняется в отдельном потоке
    thread t2 {F()};        // F()() выполняется в отдельном потоке

    t1.join();              // ждем t1
    t2.join();              // ждем t2
}
```

Вызовы **join()** гарантируют, что мы не выйдем из **user()** до того, как потоки будут завершены. Термин *join* здесь означает «ждать, пока не завершится поток».

Программные потоки разделяют одно и то же адресное пространство. В этом отношении потоки отличаются от процессов, которые в общем случае не разделяют данные между собой. А вот потоки, из-за их совместного адресного пространства, взаимодействуют через разделяемые (доступные отовсюду) объекты (§5.3.4). Такие взаимодействия в типичном случае контролируются посредством блокировок или иных механизмов для предотвращения «гонки за данными» (неконтролируемый параллельный доступ к переменной).

Программирование параллельных задач таит в себе много неожиданностей. Рассмотрим некоторую реализацию задачи **f** (функция) и задачи **F** (функциональный объект):

```
void f() {cout << "Hello ";}

struct F {
    void operator()() {cout << "Parallel World!\n";}
};
```

Это характерный ошибочный пример: **f** и **F()** — оба используют объект **cout** без какой-либо синхронизации. Результирующий вывод непредсказуем и может различаться для разных сеансов работы программы, потому что порядок выполнения индивидуальных операций в двух задачах не определен. Программа может выдать странный результат типа

```
PaHerallllel o World!
```

Определяя задачи для параллельной программы, нужно стараться формировать их независимое исполнение за исключением отдельных актов коммуникаций в простых и очевидных формах. Одной из простейших форм представления параллельной задачи является представление о функции, которая выполняется параллельно вызвавшему ее коду. Чтобы это работало, нужно передать аргументы, получить назад результат и гарантировать отсутствие разделяемых данных в процессе работы (отсутствие «гонки за данными»)

5.3.2. Передача аргументов

В типичном случае задача нуждается в данных, над которыми будет производиться работа. Рассмотрим следующее:

```
void f(vector<double>& v);          // функция что-то делает с v

struct F {                          // объект-функция: что-то сделать с v
    vector<double>& v;
    F(vector<double>& vv): v{vv} {}
    void operator()();              // операция вызова; § 3.4.3
};

int main()
{
    vector<double> some_vec {1,2,3,4,5,6,7,8,9};
    vector<double> vec2 {10,11,12,13,14};

    thread t1 {f, some_vec};        // f(some_vec) выполняется в отдельном потоке
    thread t2 {F{vec2}};            // F(vec2)() выполняется в отдельном потоке

    t1.join();
    t2.join();
}
```

Очевидно, что **F{vec2}** сохраняет ссылку на аргумент (вектор) в **F**. **F** может после этого использовать этот вектор и надеяться, что никакая другая задача не имеет доступа к **vec2**, пока исполняется **F**. Передача **vec2** по значению устраняет этот риск.

Инициализация значением **{f, some_vec}** использует шаблонный конструктор класса **thread** с переменным числом параметров (§28.6). Компилятор проверяет, что первый параметр может быть вызван с аргументами, указанными с помощью следующих параметров, и строит нужный функциональный объект, который и передается потоку. Таким образом, если **F::operator()()** и **f()** реализуют один и тот же алгоритм, выполнение двух задач будет приблизительно эквивалентным: в обоих случаях конструируется функциональный объект для исполнения в потоке.

5.3.3. Возврат результатов

В примере из §5.3.2 я передавал аргументы с помощью неконстантной ссылки. Я так поступаю в тех случаях, когда ожидаю, что задача будет модифицировать значение данных, адресуемых ссылкой (§7.7). Это несколько коварный, но не такой уж и редкий способ возврата результатов. Менее туманная техника заключается в том, чтобы передавать данные по константной ссылке, а также отдельным аргументом передавать место, в которое будет положен результат:

```
void f(const vector<double>& v, double* res);     // ввод из v; результат в *res

class F {
public:
    F(const vector<double>& vv, double* p): v{vv}, res{p} {}
    void operator()();    // помещаем результат в *res
private:
    const vector<double>& v;     // источник ввода
    double* res;                 // цель вывода
};
```

```
int main()
{
    vector<double> some_vec;
    vector<double> vec2;
    // ...

    double res1;
    double res2;
    thread t1 {f, some_vec,&res1};      // f(some_vec,&res1) в отдельном потоке
    thread t2 {F{vec2,&res2}};          // F{vec2,&res2}() в отдельном потоке

    t1.join();
    t2.join();

    cout << res1 << ' ' << res2 << '\n';
}
```

Я не считаю возврат результатов через параметры особо элегантным, и поэтому еще вернусь к этому вопросу в §5.3.5.1.

5.3.4. Разделяемые данные

Иногда задачам нужен совместный доступ к данным (разделяемые данные). В этом случае, доступ к данным должен быть синхронизирован таким образом, чтобы лишь одна задача обращалась к данным в определенный момент времени. Опытные программисты могут заметить, что нет ничего плохого в том, что многие задачи одновременно читают неизменяемые данные, но как вообще можно гарантировать, что лишь одна задача в конкретный момент времени имеет доступ к заданному набору объектов.

Фундаментальным решением проблемы служит **mutex**. Поток (**thread**) захватывает *mutex* (*семафор*), используя операцию **lock()**:

```
mutex m;        // управляющий mutex
int sh;         // разделяемые данные

void f()
{
    unique_lock<mutex> lck {m}; // захват мьютекса
    sh += 7;                    // манипулируем разделяемыми данными
} // неявно освобождаем мьютекс
```

Конструктор у **unique_lock** захватывает семафор (mutex) посредством вызова **m.lock()**. Если другой поток уже захватил семафор, поток ждет (блокируется) до тех пор, пока у другого потока не отпадет нужда в доступе. В случае, когда отпадает нужда в доступе к разделяемым данным, **unique_lock** освобождает семафор посредством вызова **m.unlock()**. Средства взаимного исключения доступа и блокировок находятся в файле **<mutex>**.

Соответствие между разделяемыми данными и семафорами очевидное: программист должен знать, какой семафор предназначен для каких данных. Это, конечно, может порождать ошибки, и мы пытаемся прояснять соответствие через различные средства языка. Например:

```
class Record {
public:
    mutex rm;
    // ...
};
```

Не надо быть гением, чтобы угадать, что для объекта **rec** типа **Record**, **rec.rm** является семафором, который вы должны захватывать перед тем, как обратиться к иным данным

rec (посредством комментария или более подходящего имени это можно сделать более понятным для читателя кода).

Нет ничего необычного в том, что иногда нужен одновременный доступ к нескольким ресурсам с целью выполнения определенного действия. Это может приводить к клинчу. Например, поток **thread1** захватил семафор **mutex1**, а затем пытается захватить еще и семафор **mutex2**, в то время как поток **thread2** уже захватил семафор **mutex2** и пытается захватить **mutex1**, а в результате ни одна из задач не может продвинуться в своей работе. Стандартная библиотека предлагает тут помощь в форме операции, позволяющей выполнить несколько одновременных блокировок:

```
void f()
{
    // ...
    unique_lock<mutex> lck1 {m1, defer_lock}; // defer_lock: нет попытке захватить мьютекс
    unique_lock<mutex> lck2 {m2, defer_lock};
    unique_lock<mutex> lck3 {m3, defer_lock};
    // ...
    lock(lck1, lck2, lck3);                    // захватываем три блокировки
    // ... манипулируем разделяемыми данными ...
} // неявно освобождаются все мьютексы
```

Здесь **lock()** сработает только тогда, когда будут захвачены все его аргументы и никогда не приведет к блокировке, пока он удерживает семафор. Деструкторы индивидуальных объектов типа **unique_lock** гарантируют, что семафоры будут освобождены, когда поток выйдет из области видимости.

Взаимодействие посредством разделяемых данных является весьма низкоуровневой задачей. В частности, программисту нужно каким-то образом суметь узнать, какие работы выполнены и какие не выполнены различными задачами. В этом отношении применение разделяемых данных является худшей альтернативой концепциям вызова и возврата. С другой стороны, некоторые люди считают, что разделяемые данные обеспечивают большую эффективность, чем копирование аргументов и выполнение возврата. Это на самом деле так в ситуациях, когда приходится иметь дело с огромными объемами данных, но сами по себе операции блокировки и снятия блокировок являются относительно дорогими операциями. А с другой стороны, современные машины очень даже хороши в копировании данных, особенно компактно расположенных данных, таких как элементы типа **vector**. В итоге, не следует кидаться применять разделяемые данные в целях эффективности без тщательных раздумий и без измерений.

5.3.4.1. Ожидание наступления событий

Иногда поток должен ждать наступления какого-либо внешнего события, например завершения работы другого потока или же истечения некоторого интервала времени. Рассмотрим следующее:

```
using namespace std::chrono; // см. § 35.2

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();
cout << duration_cast<nanoseconds>(t1-t0).count() << " nanoseconds passed\n";
```

Заметьте, что мне даже не пришлось запускать поток; по умолчанию, **this_thread** ссылается на один и только один поток (§42.2.6).

Я использовал **duration_cast** для настройки единиц часов в наносекунды. См. §5.4.1 и §35.2 перед тем, как попробовать со временем что-либо более сложное, чем то, что здесь представлено. Средства работы со временем располагаются в **<chrono>**.

Базовая поддержка коммуникаций на основе внешних событий обеспечивается посредством **condition_variable** и находится в файле **<condition_variable>** (§42.3.4). Тип **condition_variable** является механизмом, помогающим одному потоку ждать другого. В частности, он позволяет потоку ожидать выполнения некоторого *условия* (часто называемого *событием*), являющегося следствием результата работы других потоков.

Рассмотрим классический пример двух потоков, взаимодействующих посредством передачи сообщений через очередь (**queue**). Ради простоты я объявляю очередь (**queue**) и механизм устранения условий гонки на этой очереди глобальным образом:

```
class Message {    // объект коммуникации
    // ...
};

queue<Message> mqueue;    // очередь сообщений
condition_variable mcond; // ожидание событий коммуникации
mutex mmutex;             // механизм блокировки
```

Типы **queue**, **condition_variable** и **mutex** обеспечиваются стандартной библиотекой.

Потребитель (**consumer**) читает и обрабатывает сообщения (объекты типа **Message**):

```
void consumer()
{
    while(true) {
        unique_lock<mutex> lck{mmutex};              // захватываем mmutex
        while (mcond.wait(lck)) /* ничего не делаем */; // освобождаем lck и ждем;
                                                     // снова захватываем lck по пробуждению
        auto m = mqueue.front();                     // получаем сообщение
        mqueue.pop();
        lck.unlock();                                // освобождаем lck
        // ... обрабатываем m ...
    }
}
```

Здесь я явным образом ввожу защиту операций с очередью (**queue**) и условной переменной (**condition_variable**) с помощью **unique_lock** над **mutex**. В цикле ожидания **condition_variable** блокировка находится в освобожденном состоянии до тех пор, пока **condition_variable** не просигнализирует о событии (в очередь поступит сообщение), после чего блокировка снова захватывается.

Соответствующий производитель (**producer**) выглядит следующим образом:

```
void producer()
{
    while(true) {
        Message m;
        // ... заполняем сообщение ...
        unique_lock<mutex> lck {mmutex};     // защищаем операции
        mqueue.push(m);
        mcond.notify_one();                  // уведомляем
    }                                        // освобождаем блокировку (по области видимости)
}
```

Применение условных переменных (**conditional_variable**) поддерживает много разных форм элегантного и эффективного совместного доступа к данным (sharing), но эти приемы программирования не совсем очевидны и не так уж и просты (§42.3.4).

5.3.5. Взаимодействие задач

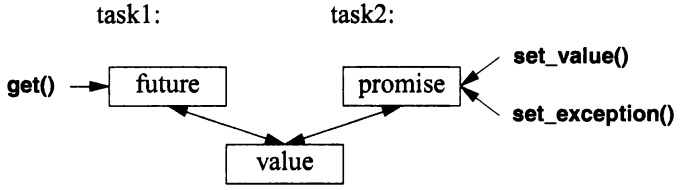
Стандартная библиотека предоставляет ряд средств, позволяющих программисту работать на концептуальном уровне задач, а не на низкоуровневом слое непосредственно с потоками и блокировками:

1. **future** и **promise** для возврата значения из задачи, запущенной на отдельном потоке
2. **packaged_task** для помощи в запуске задач и прикреплении механизмов возврата результатов
3. **async()** для запуска задачи в манере, близкой к вызову функции.

Эти средства можно найти в файле **<future>**.

5.3.5.1. future и promise

Самое важное в **future** и **promise** — это то, что они позволяют передать значение между задачами без явного применения блокировок; при этом система реализует эту передачу эффективно. Основная идея проста: когда задаче нужно передать значение другой задаче, она помещает это значение в **promise**. Неким образом реализация этого механизма обеспечивает появление этого значения в соответствующем **future**, из которого его и можно будет прочесть (в типичном случае тем, кто запустил задачу). Мы может представить все это графически следующим образом:



Если у нас есть объект **fx** типа **future<X>**, то мы можем вызовом метода **get()** получить значение типа **X** из этого объекта:

```
X v = fx.get();      // по необходимости ждем вычисляемое значение
```

Если значения там еще нет, поток блокируется до тех пор, пока оно там не появится. Если же значение не может быть вычислено, **get()** может сгенерировать исключение (получив его от системы или же от задачи, от которой мы пытаемся получить значение).

Главной задачей **promise** является предоставление простых «put»-операций (**set_value()** и **set_exception()**), парных к «get»-операциям **future**. Имейте в виду, что названия «promise» и «future» сложились исторически — не нужно придавать им слишком большого (буквального) значения.

Если вы располагаете **promise** и нуждаетесь в том, чтобы послать **future** результат типа **X**, вы можете сделать это двумя способами: передать значение или передать исключение. Вот пример:

```
void f(promise<X>& px) // задача: поместить результат в px
{
    // ...
    try {
        X res;
        // ... вычислить значение для res ...
        px.set_value(res);
    }
    catch (...) {        // oops: не получается вычислить res
        // передать исключение потоку future:
        px.set_exception(current_exception());
    }
}
```

Вызов **current_exception()** возвращает перехваченное исключение (§30.4.1.2).

Для работы с исключением, переданным через **future**, нужно при вызове **get()** перехватить его каким-либо образом. Например:

```
void g(future<X>& fx)    // задача: получить результат из fx
{
    // ...
    try {
        X v = fx.get();        // по необходимости ждем вычисляемое значение
        // ... use v ...
    }
    catch (...) {    // oops: кто-то не может вычислить v
        // ... обрабатываем ошибку ...
    }
}
```

5.3.5.2. packaged_task

Как мы должны внедрять **future** в задачу, нуждающуюся в результате, и как мы должны внедрять **promise** в задачу, которая этот результат производит? Тип **packaged_task** упрощает подготовку задач, взаимодействующих через **future** и **promise**, для запуска на потоках. Этот тип предоставляет код-обертку, который помещает возвращаемое значение или исключение из задачи в **promise** (как это делает код из §5.3.5.1). Если вы запросите его посредством вызова **get_future**, тогда **packaged_task** предоставит вам **future** для соответствующего **promise**. Например, мы можем организовать две задачи, каждая из которых вычисляет сумму своей половину элементов из **vector<double>**, используя **accumulate()** из стандартной библиотеки (§3.4.2, §40.6):

```
double accum(double* beg, double * end, double init)
// вычислить сумму [beg:end) начиная с начального значения init
{
    return accumulate(beg, end, init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*, double*, double);  // тип задачи

    packaged_task<Task_type> pt0 {accum};                // упаковывем задачу (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};                // удерживаем future у pt0
    future<double> f1 {pt1.get_future()};                // удерживаем future у pt1

    double* first = &v[0];
    thread t1 {move(pt0), first, first+v.size()/2,0};    // стартуем поток для pt0
    thread t2 {move(pt1), first+v.size()/2, first+v.size(),0};  // стартуем поток для pt1

    // ...

    return f0.get()+f1.get();                            // получаем результаты
}
```

Шаблон **packaged_task** принимает тип задачи в качестве своего шаблонного параметра (здесь, **Task_type**, синоним для **double(double*, double*, double)**) и задачу в качестве аргумента своего конструктора (здесь, **accum**). Операции **move()** нужны потому, что **packaged_task** не подлежит копированию.

5.3.5.3. async()

В данной главе я придерживаюсь простого, но в то же время мощного представления о задаче как о функции, которая возможно будет выполняться параллельно иным задачам. Это не единственная модель задачи, которая поддерживается в стандартной библиотеке, но она хорошо применима ко многим практическим ситуациям. Более изощренные модели, основанные на разделяемой памяти, также можно использовать при необходимости.

Для запуска задач, которые могут исполняться асинхронно, используется **async()**:

```
double comp4(vector<double>& v)
    // запустить много задач, если v достаточно велико
{
    if (v.size()<10000) return accum(v.begin(), v.end(),0.0);

    auto v0 = &v[0];
    auto sz = v.size();
    auto f0 = async(accum, v0, v0+sz/4,0.0);          // первая четверть
    auto f1 = async(accum, v0+sz/4, v0+sz/2,0.0);     // вторая четверть
    auto f2 = async(accum, v0+sz/2, v0+sz*3/4,0.0);   // третья четверть
    auto f3 = async(accum, v0+sz*3/4, v0+sz,0.0);     // четвертая четверть

    return f0.get()+f1.get()+f2.get()+f3.get();  // объединяем результаты
}
```

В основе своей, **async()** отделяет «вызывающую часть» функции от «части, возвращающей результат», а также отделяет их обеих от истинного выполнения задачи. Используя **async()**, вам не нужно заботиться о потоках и блокировках. Вместо этого, вы мыслите в категориях задач, которые выполняют свою работу асинхронно. Есть естественное ограничение: даже и не думайте применять **async()** для задач, которые разделяют ресурсы с необходимостью их блокировки — в этом случае вы даже не знаете, сколько потоков будет задействовано, ибо это определяет сам **async()**, основываясь на том, что он знает о доступных системных ресурсах в момент вызова. Например, **async()** может проверить, имеются ли в данный момент времени простаивающие процессорные ядра, и после этого принять решение о том, сколько потоков использовать.

Не думайте об **async()** исключительно как о специализированном механизме параллельных вычислений с повышенной производительностью. Он может также использоваться, например, для запуска задачи, предназначенной для получения информации от пользователя, в то время как «основная программа» может заниматься чем-то другим (§42.4.6).

5.4. Утилиты — небольшие полезные компоненты библиотеки

Не все компоненты стандартной библиотеки относятся к ее частям с говорящими названиями, такими как «контейнеры» или «ввод/вывод». В настоящем разделе мы рассмотрим несколько примеров небольших, но широко применяемых компонентов (утилит):

- **clock** и **duration** для измерения времени.
- Функции для работы с типами, такие как **iterator_traits** и **is_arithmetic**, предоставляют информацию о типах.
- **pair** и **tuple** для представления небольших, потенциально гетерогенных наборов значений.

Эти функции или типы не слишком сложные и не связаны теснейшим образом с массой других функций или типов. Эти небольшие компоненты стандартной библиотеки скорее являются строительными блоками для более мощных библиотечных средств.

5.4.1. Время

Стандартная библиотека предоставляет ряд средств для работы со временем. Например, вот основной способ для замера времени:

```
using namespace std::chrono; // см. §35.2

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1-t0).count() << "msec\n";
```

Часы возвращают **time_point** (момент времени). Разность между двумя моментами времени есть **duration** (промежуток времени). Разные часы выдают результаты в разных единицах времени (часы, которые использую я, измеряют время в наносекундах), так что целесообразно конвертировать **duration** в нужные вам единицы времени. Что и делает **duration_cast**.

Средства стандартной библиотеки, имеющие дело со временем, объявлены в подпространстве имен **std::chrono** и находятся в файле **<chrono>** (§35.2).

Не делайте априорных утверждений об эффективности кода, предварительно не проведя измерения. Предположения о производительности одни из самых ненадежных.

5.4.2. Функции для работы с типами

Функция для работы с типами (*type function*) — это функция, вычисляемая на этапе компиляции, и получающая тип в качестве аргумента или же возвращающая тип. Стандартная библиотека предоставляет некоторое количество функций для работы с типами, чтобы помочь тем, кто реализует библиотеку, а также вообще всем программистам для написания кода, использующим все аспекты языка C++ и стандартной библиотеки.

Для числовых типов, **numeric_limits** из файла **<limits>** предоставляет разнообразную полезную информацию (§5.6.5). Например:

```
constexpr float min = numeric_limits<float>::min();     // наимень. положит. float (§ 40.2)
```

Размеры объектов можно определить при помощи встроенной операции **sizeof** (§2.2.2). Например:

```
constexpr int szi = sizeof(int);     // число байт в int
```

Функции этого типа являются частью механизмов языка C++ для использования на этапе компиляции, которые помогают выполнять строгую проверку типов и помогают достигать более высокой производительности. Применение этих средств часто называют *метапрограммированием*, или в контексте шаблонов — *шаблонным метапрограммированием* (Глава 28). Здесь я рассматриваю пару таких средств из стандартной библиотеки: **iterator_traits** (§5.4.2.1) и предикаты типов (§5.4.2.2).

5.4.2.1. iterator_traits

В стандартной библиотеке **sort()** принимает пару итераторов, которые, как предполагается, задают последовательность (§4.5). Более того, эти итераторы должны предоставлять произвольный доступ к последовательности, то есть, они должны быть *итераторами произвольного доступа*. Некоторые контейнеры, такие как **forward_list**, не предоставляют таких итераторов. Конкретно, **forward_list** является односвязанным списком, так что индексирование для него очень дорогое, и нет разумных способов адресоваться к предыдущему элементу. Однако, как и большинство иных контейнеров, **forward_list** предоставляет *пря-*

мые итераторы, которые можно использовать для траверса последовательностей (прохода по всем элементам) в алгоритмах и **for**-операторах цикла (§33.1.1).

Стандартная библиотека предоставляет механизм **iterator_traits**, который помогает проверить, какой тип итераторов поддерживается. Теперь мы можем улучшить функцию **sort()** из §4.5.6, чтобы она принимала или **vector**, или **forward_list**. Вот соответствующий пример:

```
void test(vector<string>& v, forward_list<int>& lst)
{
    sort(v);   // сортируем вектор
    sort(lst); // сортируем односвязанный список
}
```

Приемы, которые заставят это работать, весьма интересны и полезны. Сначала я напишу две вспомогательные функции, которые принимают лишний аргумент, указывающий на то, какие им следует использовать итераторы — произвольного доступа, или же прямые итераторы. Версия для итераторов произвольного доступа тривиальная:

```
template<typename Ran>                                  // итераторы произвольного доступа
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)
                                                        // можем индексировать в [beg:end)
{
    sort(beg, end);          // просто сортируем
}
```

Версия для прямых итераторов почти столь же проста; копируем список в **vector**, сортируем и копируем в обратную сторону:

```
template<typename For>                                  // прямые итераторы
void sort_helper(For beg, For end, forward_iterator_tag)  // выполняем траверс [beg:end)
{
    vector<decltype(*beg)> v {beg, end};          // инициализируем вектор из [beg:end)
    sort(v.begin(), v.end());
    copy(v.begin(), v.end(), beg);                // копируем элементы обратно
}
```

Встроенная функция **decltype()** возвращает декларируемый тип ее аргумента (§6.3.6.3). Таким образом, **v** есть **vector<X>**, где **X** это тип элементов входной последовательности.

Настоящая «магия типов» состоит в выборе одной из этих вспомогательных функций:

```
template<typname C>
void sort(C& c)
{
    using Iter = Iterator_type<C>;
    sort_helper(c.begin(), c.end(), Iterator_category<Iter>{});
}
```

Здесь я использую две функции для работы с типами: **Iterator_type<C>** возвращает тип итераторов для **C** (то есть, **C::iterator**), после чего **Iterator_category<Iter>{}** конструирует «ярлык», указывающий на тип итераторов:

- **std::random_access_iterator_tag**, если итератор для типа **C** поддерживает произвольный доступ.
- **std::forward_iterator_tag**, если итератор для типа **C** поддерживает итерацию в прямом направлении.

Отталкиваясь от этого, мы можем осуществить выбор между двумя сортирующими алгоритмами прямо на этапе компиляции. Такую технику программирования называют

диспетчеризацией на базе тегов (*tag dispatch*), и она применяется в стандартной библиотеке, а также в стороннем коде для увеличения гибкости и производительности.

Форма, в которой стандартная библиотека поддерживает технику диспетчеризации на основе тегов, базируется на простом классовом шаблоне **iterator_traits** из файла **<iterator>** (§33.1.3). Это позволяет очень просто определить функции для типов, использующиеся в функции **sort()**:

```
template<typename C>
    using Iterator_type = typename C::iterator;              // тип итератора у C

template<typename Iter>
    using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;
                                                             // категория итератора
```

Если вам не хочется разбираться с «магией типов на этапе компиляции», которая используется для реализации возможностей стандартной библиотеки, вы можете проигнорировать такие средства, как **iterator_traits**. Но вместе с этим вы уже не сможете воспользоваться техниками, основанными на этих средствах, для улучшения вашего кода.

5.4.2.2. Предикаты типов

Предикат типа из стандартной библиотеки — это простая функция для работы с типами, которая отвечает на фундаментальный вопрос о типе. Вот пример:

```
bool b1 = Is_arithmetic<int>();          // да, int это арифметический тип
bool b2 = Is_arithmetic<string>();       // нет, std::string не арифметический тип
```

Эти предикаты находятся в файле **<type_traits>** и описываются в §35.4.1. Другие примеры это **is_class**, **is_pod**, **is_literal_type**, **has_virtual_destructor** и **is_base_of**. Наиболее полезны они при написании шаблонов. Например:

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
    // ...
};
```

Для улучшения читаемости по сравнению с прямолинейным использованием стандартной библиотеки я определяю следующую функцию для работы с типами:

```
template<typename T>
constexpr bool Is_arithmetic()
{
    return std::is_arithmetic<T>::value;
}
```

В старых программах **::value** используется вместо функционального вызова, но я считаю это несколько неизящным и демонстрирующим детали реализации.

5.4.3. pair и tuple

Иногда нам нужны данные как таковые, то есть просто коллекция значений, а не объект класса с четко определенной семантикой и инвариантом использования (§2.4.3.2, §13.4). В таких случаях можно определить **struct** с надлежащим набором надлежаще названных членов. Но можно также и попросить стандартную библиотеку сделать это за нас.

Например, алгоритм **equal_range** стандартной библиотеки (§32.6.1) возвращает пару (**pair**) итераторов, специфицирующих подпоследовательность, удовлетворяющую предикату:

```
template<typename Forward_iterator, typename T, typename Compare>
    pair<Forward_iterator, Forward_iterator>
    equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

Для отсортированной последовательности [**first:last**), **equal_range()** вернет **pair**, представляющую подпоследовательность, удовлетворяющую предикату **cmp**. Мы можем это использовать для поиска в отсортированной последовательности элементов типа **Record**:

```
auto rec_eq = [](const Record& r1, const Record& r2) {return r1.name<r2.name;};
                                                                    // сравнить имена

void f(const vector<Record>& v)   // полагаем что v отсортирован по полю "name"
{
    auto er = equal_range(v.begin(), v.end(), Record{"Reg"}, rec_eq);
    for (auto p = er.first; p!=er.second; ++p)      // выведем все совпадения
        cout << *p;     // полагаем что << определена для Record
}
```

Первый член пары называется **first**, а второй — **second**. Этот стиль именования сначала кажется издишне примитивным. Но он очень важен для обобщенного программирования.

Тип **pair** из стандартной библиотеки (из файла **<utility>**) часто используется как внутри стандартной библиотеки, так и вне ее. Тип **pair** предоставляет операции, такие как **=**, **==** и **<**, если это позволяют ее элементы. Функция **make_pair()** упрощает создание **pair** без явного упоминания типа (§34.2.4.1). Например:

```
void f(vector<string>& v)
{
    auto pp = make_pair(v.begin(),2);  // pp это pair<vector<string>::iterator, int>
    // ...
}
```

Если вам требуется более чем два элемента (или менее чем), можете воспользоваться типом **tuple** (из файла **<tuple>**; §34.2.4.2). Тип **tuple** представляет гетерогенную последовательность элементов, например:

```
tuple<string, int, double> t2("Sild",123, 3.14);      // тип явно специфицируется

auto t = make_tuple(string("Herring"),10, 1.23);      // тип выводится
                                                      // t это tuple<string, int, double>

string s = get<0>(t);      // получить первый элемент у tuple
int x = get<1>(t);
double d = get<2>(t);
```

Элементы у **tuple** пронумерованы (начиная с нуля), а не поименованы как у **pair** (**first** и **second**). Для выбора элементов во время компиляции приходится, к сожалению, использовать безобразные на вид конструкции вида **get<1>(t)**, а не что-нибудь более приличное вроде **get(t,1)** или **t[1]** (§28.5.2).

Для **tuple** (как и для **pair**) возможно сравнение, если это допускают их элементы.

В интерфейсах (в широком смысле) **pair** применяется широко, поскольку часто нам нужно вернуть более одного значения, например некую величину и индикатор качества вычислений. Реже приходится в результате наблюдать три и более частей, так что **tuple** чаще всего наблюдается в реализациях обобщенных алгоритмов.

5.5. Регулярные выражения

Регулярные выражения являются мощным инструментом обработки текстов. Они предоставляют простой и лаконичный способ описания схемы искомой последовательности символов (*patterns* — текстовых *паттернов*) в исходной строке (почтовый код вроде TX 77845, или дата в ISO-стиле, такая как 2009–06–07), а также выполняют по заданному паттерну эффективный поиск в тексте. Поддержка регулярных выражений в стандартной библиотеке предоставляется в файле **<regex>** классом **std::regex**, а также вспомогательными функциями. Для примера давайте определим паттерн и выведем его на печать:

```
regex pat (R"(\w{2}\s*\d{5}(-\d{4})?)");    // варианты ZIP кода: XXddddd-dddd
cout << "pattern: " << pat << '\n';
```

Все, кто хоть когда-нибудь использовал регулярные выражения в каком-либо языке, найдут знакомым выражение **\w{2}\s*\d{5}(-\d{4})?**. Оно определяет текстовый паттерн, начинающийся с двух букв **\w{2}**, за которыми опционально следует некоторый пробельный символ **\s*** с пятью цифрами после него (**\d{5}**), и опционально заканчивающийся на тире и четыре цифры (**-\d{4}**). А если вы еще не знакомы с регулярными выражениями, то сейчас самое время начать знакомство ([Stroustrup,2009]), [Maddock,2009], [Friedl,1997]. Информация по регулярным выражениям суммируется в §37.1.1.

Для записи текстовых паттернов я использую «*сырой строковый литерал*» (*raw string literal*; §7.3.2.1), начинающийся на **R"(** и заканчивающийся на **)"**. Это позволяет использовать обратные наклонные черты и кавычки непосредственно внутри строки.

Самое простое применение текстовых паттернов — поиск в потоке символов:

```
int lineno = 0;
for (string line; getline(cin, line);) {              // читаем в буфер line
    ++lineno;
    smatch matches;                                   // здесь сопоставленные строки
    if (regex_search(line, matches, pat))             // ищем pat в строке
        cout << lineno << ": " << matches[0] << '\n';
}
```

Вызов **regex_search(line, matches, pat)** ищет в **line** что-либо, что соответствует регулярному выражению, хранящимуся в **pat**, и если что-то находится, то результат сохраняется в **matches**. Если же ничего не находится, то возвращается **false**. Тип **matches** есть **smatch**. Буква «s» здесь соответствует приставке «sub», и **smatch** означает **vector** из sub-matches. Первый элемент, то есть **matches[0]**, соответствует полному совпадению.

Более детальное описание приведено в Главе 37.

5.6. Математика

При первоначальной разработке языка C++ математика не стояла в качестве основных приоритетов. Тем не менее, сейчас C++ интенсивно используется в численных расчетах, и стандартная библиотека отвечает на этот вызов.

5.6.1. Математические функции и алгоритмы

В файле **<cmath>** находятся «обычные математические функции», такие как **sqrt()**, **log()** и **sin()** с аргументами типа **float**, **double** и **long double** (§40.3). Комплексные варианты этих функций находятся в файле **<complex>** (§40.4).

В файле **<numeric>** находится небольшой набор общих численных алгоритмов, таких как, например, **accumulate()**. Пример:

```
void f()
{
    list<double> lst {1, 2, 3, 4, 5, 9999.99999};
    auto s = accumulate(lst.begin(), lst.end(),0.0);      // вычисляем сумму
    cout << s << '\n';                                    // печатаем 10014.9999
}
```

Эти алгоритмы работают для любых последовательностей стандартной библиотеки и могут иметь аргументы-операции (§40.6).

5.6.2. Комплексные числа

Стандартная библиотека поддерживает семейство типов комплексных чисел и класс **complex**, описанный в §2.3. Для поддержки комплексных чисел со скалярами типа **float**, **double** и т.д., **complex** в стандартной библиотеке определяется как шаблон:

```
template<typename Scalar>
class complex {
public:
    complex(const Scalar& re ={}, const Scalar& im ={});
    // ...
};
```

Обычные математические операции и большинство распространенных математических функций поддерживаются для комплексных чисел. Вот пример:

```
void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld {fl+sqrt(db)};
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

Функции **sqrt()** и **pow()** (возведение в степень) находятся среди обычных математических функций, определенных в **<complex>**. Дополнительные сведения представлены в §40.4.

5.6.3. Случайные числа

Случайные числа бывают полезны во многих случаях, например в тестировании, играх, эмуляции и в вопросах, связанных с безопасностью. Такое разнообразие предметных областей отражается во множестве генераторов случайных чисел, представленных в файле **<random>** стандартной библиотеки. Генератор случайных чисел состоит из двух частей:

1. *ядро*, которое генерирует последовательность случайных или псевдослучайных значений.
2. *отображение*, которое переводит эту последовательность в распределение величин в заданном диапазоне.

Примерами отображений служат **uniform_int_distribution** (где все генерируемые целые числа равновероятны), **normal_distribution** и **exponential_distribution** (экспоненциальный рост); каждое распределение для своего диапазона значений. Пример:

```
using my_engine = default_random_engine;          // тип движка
using my_distribution = uniform_int_distribution<>; // тип распределения

my_engine re {};                                  // движок по умолчанию
my_distribution one_to_six {1,6};                 // распределение под int 1..6
auto die = bind(one_to_six, re);                  // создаем генератор

int x = die();                                    // кидаем кость: x значение из [1:6]
```

Функция стандартной библиотеки **bind()** производит функциональный объект, который активирует свой первый аргумент (здесь, **one_to_six**), предоставляя ему свой второй аргумент (здесь, **re**) в качестве аргумента (§33.5.1). Таким образом, вызов **die()** эквивалентен вызову **one_to_six(re)**.

Представленный код вряд ли легок для понимания новичками. Оператор **using** делает его немного более читаемым. Но я бы предпочел просто написать

```
auto die = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```

В конце концов, какой вариант является более читаемым, зависит от читателя и контекста.

И все-таки, для новичков общий интерфейс к библиотеке случайных чисел может оказаться излишне сложным. Простой генератор равномерно распределенных случайных чисел может оказаться достаточным для начала. Например:

```
Rand_int rnd {1,10};      // генератор случай. чисел в  [1:10]
int x = rnd();            // x число из [1:10]
```

Итак, как нам получить его? Нам нужно будет применить что-то вроде **die()** в классе **Rand_int**:

```
class Rand_int {
public:
    Rand_int(int low, int high): dist{low, high} {}
    int operator()() {return dist(re);}              // вытянуть целое
private:
    default_random_engine re;
    uniform_int_distribution<> dist;
};
```

Несмотря на то, что это определение по-прежнему соответствует экспертному уровню знаний в данной области, использование класса **Rand_int** вполне доступно начинающим изучать эту тему в первую же неделю. Например:

```
int main()
{
    Rand_int rnd {0,4};          // однородный генератор случай. чисел

    vector<int> histogram(5);        // вектор размера 5
    for (int i=0; i!=200; ++i)
        ++histogram[rnd()];          // гистограмма частот чисел из [0:4]

    for (int i = 0; i!=mn.size(); ++i) {  // выводим столбцовую диаграмму
        cout << i << '\t';
        for (int j=0; j!=mn[i]; ++j) cout << '*';
        cout << endl;
    }
}
```

Вывод этой программы демонстрирует равномерное распределение (с разумной статистической вариацией):

```
0 ********************************************
1 ******************************************
2 ****************************
3 ******************************************
4 ******************************************
```

Так как не существует стандартной графической библиотеки для C++, я использую «ASCII-графику». Существует масса как открытых, так и коммерческих графических библиотек для C++, но в настоящей книге я ограничиваюсь исключительно ISO стандартом.

За дополнительной информацией по случайным числам обратитесь к §40.7.

5.6.4. Векторная арифметика

Тип **vector**, описанный в §4.4.1, разрабатывался с тем, чтобы он был общим механизмом для хранения значений, чтобы он был гибким, а также мог быть встроен в архитектуру контейнеров, итераторов и алгоритмов. Однако, он не поддерживает математические операции над векторами (векторную арифметику). Конечно, нетрудно было бы добавить такие операции к **vector**, но общность и гибкость этого типа потребовала бы серьезной работы для достижения производительности, необходимой в серьезной вычислительной работе. Поэтому в стандартной библиотеке присутствует тип **valarray** (в файле **<valarray>**), менее общий и гибкий, но более приспособленный для оптимизаций в сторону численных расчетов:

```
template<typename T>
class valarray {
    // ...
};
```

Для **valarray** поддерживаются арифметические операции и наиболее общие математические функции. Пример:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1; // арифметические операции с массивом *, +, /, и =
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

Детали смотрите в §40.5. В частности, **valarray** помогает реализовывать многомерные вычисления.

5.6.5. Числовые пределы

В файле **<limits>** стандартная библиотека предоставляет классы, которые описывают свойства встроенных типов — вроде максимальной степени для **float**, или же число байт в типе **int**; см. §40.2. Например, можно статически проверить, что **char** является знаковым типом:

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

Заметим, что здесь второй оператор работает потому, что **numeric_limits<int>::max()** является функцией с атрибутом **constexpr** (§2.2.3, §10.4).

5.7. Советы

[1] Применяйте ресурсные дескрипторы для управления ресурсами; §5.2.
[2] Используйте **unique_ptr** для ссылки на объекты полиморфных типов; §5.2.1.
[3] Используйте **shared_ptr** для ссылки на разделяемые объекты; §5.2.1.
[4] Применяйте механизмы параллелизма, безопасные по типам; §5.3.
[5] Минимизируйте использование разделяемых данных; §5.3.4.
[6] Не применяйте разделяемые данные ради «эффективности» без тщательных размышлений и без экспериментальных измерений; §5.3.4.
[7] Мыслите в категориях параллельных задач, а не потоков; §5.3.5.
[8] Чтобы быть полезной, библиотеке нет необходимости быть очень большой или очень сложной; §5.4.
[9] Проведите временные измерения перед утверждением об эффективности вашей программы; §5.4.1.
[10] Вы можете писать код, явно зависящий от свойств типов; §5.4.2.
[11] Применяйте регулярные выражения для поиска совпадений с паттернами; §5.5.
[12] Не опирайтесь в серьезных численных расчетах исключительно на свойства языка; используйте библиотеки; §5.6.
[13] Свойства числовых типов доступны посредством **numeric_limits**; §5.6.5.

Часть 2

Базовые средства

В этой части описаны встроенные типы C++ и базовые методы их использования при конструировании программ. С-подмножество C++ представлено вместе с дополнительной поддержкой C++ традиционных стилей программирования. Здесь также приведены основные средства компоновки программ C++ из логических и физических частей.

Главы

6. Типы и объявления
7. Указатели, массивы и ссылки
8. Структуры, объединения и перечисления
9. Операторы
10. Выражения
11. Операции
12. Функции
13. Обработка исключений
14. Пространства имен
15. Исходные файлы и программы

«... Я уже давно питаю сомнения относительно суждений философов по всем вопросам и чувствую в себе большую склонность спорить с их выводами, нежели соглашаться с ними. Существует одна ошибка, в которой повинны почти все философы без исключения: они слишком узки и ограниченны в своих принципах и не учитывают то обширное многообразие, которое столь свойственно действиям природы. Выдвинув какой-либо излюбленный принцип, который, быть может, соответствует многим действиям природы, философ расширяет применимость этого принципа на весь мир творения, сводя к нему путем насильственного и нелепого рассуждения любое явление. ...»

— Дэвид Юм
"*Essays, Moral, Political, and Literary.* PART I"

6

Типы и объявления

Совершенство достигается
только к моменту краха.
— К.Н. Паркинсон

- Стандарт ISO C++

  Реализации; Базовый набор символов
- Типы

  Фундаментальные типы; Логический тип; Типы символов; Целые типы; Типы чисел с плавающей запятой; Префиксы и суффиксы; **void**; Размеры типов; Выравнивание
- Объявления

  Структура объявлений; Объявление множественных имен; Имена; Область видимости; Инициализация; Вывод типа; **auto**; **decltype()**
- Объекты и величины

  lvalue и rvalue; Время жизни объектов
- Синонимы типов
- Советы

6.1. Стандарт ISO C++

Язык C++ и стандартная библиотека определяются ISO стандартами: ISO/IEC14882:2011. В нашей книге ссылки на стандарт идут в формате § iso.23.3.6.1. В тех случаях, когда текст данной книги может быть сочтен не достаточно точным, или неполным, обращайтесь к стандарту. Однако, стандарт — это не учебник, и его не слишком-то легко читать неспециалистам.

Строгое следование стандартам языка C++ и стандартной библиотеки еще не гарантирует, что ваш код будет качественным и переносимым. Стандарт не касается оценок качества кода; он просто сообщает, на что программист может твердо рассчитывать, сталкиваясь с разными реализациями. В принципе, нетрудно написать ужасную программу, не противоречащую стандарту, а полезные программы часто опираются на такие свойства и средства, для которых стандарт не гарантирует переносимости. Это касается доступа программ к системным интерфейсам и аппаратуре компьютера, которые не могут быть напрямую отражены в языке C++, и которые опираются на специфические особенности реализации.

Многие важные вещи отмечаются стандартом как *зависящие от реализации*. Это означает, что конкретная реализация должна обеспечить для такой конструкции четкое и однозначное поведение, и что оно должно быть документировано. Например:

```
unsigned char c1 = 64;      // ok: char содержит не менее 8 бит
unsigned char c2 = 1256;    // зависит от реализации: урезание, если char имеет 8 bit
```

Инициализация **c1** однозначно определена, так как **char** должен иметь размер не менее 8 бит. Инициализация **c2** не является однозначной и зависит от реализации из-за того, что точное число бит в типе **char** стандартом не фиксируется. Если в **char** всего лишь 8 бит, то значение **1256** будет урезано до **232** (§10.5.2.1). Большинство зависящих от реализации черт языка C++ имеет отношение к различию в аппаратуре компьютера.

Помимо поведения частей программы, зависящего от реализации, имеет место и *неопределенное поведение*, когда набор вариантов известен, но реализатор не обязан сообщать, какой именно вариант им выбран. Обычно, для подобных вещей имеются фундаментальные причины. Например, точное значение, которое возвращается операцией **new**, не определено. Точно так же не определено значение переменной, к которой имеют доступ два потока, если в коде этих потоков не предусмотрены какие-либо механизмы для предотвращения «гонки за данными» (§41.2).

При написании реальных программ часто приходится полагаться на поведение, зависящее от реализации. Это та цена, которую мы вынуждены платить за эффективность выполнения программы на большом числе операционных систем и компьютерных архитектур. Все было бы проще, если бы символы всюду эффективно кодировались 8 битами, а все указатели были бы 32-разрядными числами. Но в реальности встречаются и 16-битные, и 32-битные кодировки символов. Также на разных машинных архитектурах и указатели разные (от 16 бит до 64 бит).

Чтобы максимально обеспечить переносимость, нужно четко выделить части программ, в которых используются средства языка, зависящие от реализации, и по-возможности изолировать их и снабдить ясными комментариями на этот счет. Например, можно сосредоточить все размеры, зависящие от аппаратуры, а также определения типов в некотором заголовочном файле (хедер-файле). Для поддержки таких приемов стандартная библиотека предоставляет **numeric_limits** (§40.2). Многие предположения о величинах, зависящих от реализации, можно проверять статически на этапе компиляции (§2.4.3.3). Например:

```
static_assert(4<=sizeof(int),"sizeof(int) too small");
```

Неопределенное поведение гораздо коварнее. В типичных случаях, программы, прямодушно использующие неопределенные свойства, демонстрируют очень плохое поведение. Вот пример:

```
const int size = 4*1024;
char page[size];

void f()
{
    page[size+size] = 7;      // не определено
}
```

Здесь возможны как несанкционированная перезапись данных, так и генерирование аппаратных исключений. Применение мощных оптимизаторов машинного кода может привести к еще более неадекватному поведению программы. В тех случаях, когда имеется некий очерченный круг альтернативных реализаций, средство языка помечается не как неопределенное, а как зависящее от реализации, или же как недостаточно специфицированное.

Очень важно не полениться и гарантировать, что в программе не применяются на удачу средства языка, помеченные как неопределенные. Существуют автоматические инструменты, помогающие решить такую задачу.

6.1.1. Реализации

Реализации могут быть полными или частичными (§ iso.17.6.1.3). Полные реализации обязаны поддерживать все средства стандартной библиотеки, описанные в стандарте

(§30.2) и в настоящей книге. Частичные реализации могут предоставлять меньшее количество средств стандартной библиотеки при условии того, что следущий набор гарантируется:

Отдельные заголовочные файлы		
Типы	**<cstddef>**	§10.3.1
Свойства реализации	**<cfloat> <limits> <climits>**	§40.2
Целые типы	**<cstdint>**	§43.7
Старт и терминирование	**<cstdlib>**	§43.7
Динамическая память	**<new>**	§11.2.3
Идентификация типов	**<typeinfo>**	§22.5
Обработка исключений	**<exception>**	§30.4.1.1
Initializer lists	**<initializer_list>**	§30.3.1
Иная поддержка	**<cstdalign> <cstdarg> <cstdbool>**	§12.2.4, §44.3.4
Traits для типов	**<type_traits>**	§35.4.1
Атомарность	**<atomic>**	§41.3

Существуют реализации, предназначенные для минимальных, очень близких к аппаратуре программ, которые не поддерживают исключений.

6.1.2. Базовый набор символов исходного кода программ

Стандарт C++ и тексты программ в настоящей книге используют лишь базовый набор символов, состоящий из букв, цифр, графических и пробельных символов из американского варианта семибитной кодировки ISO 646–1983, известной как ASCII (ANSI3.4–1968). Это может приводить к проблемам, если работа ведется в окружениях с иными наборами символов:

- ASCII содержит символы пунктуации и символы операций (вроде], {и!), отсутствующие в некоторых символьных наборах.
- Нужно как-то обозначать символы, не имеющие традиционного символьного начертания (например, символ с кодом 17).
- ASCII не содержит символов (вроде μ или ñ), применяющихся в алфавитах, отличных от английского.

Чтобы иметь возможность применять в исходном коде нестандартные символы, реализации должны каким-то образом сопоставлять их с базовым символьным набором, например используя универсальные имена символов (§6.2.3.2).

6.2. Типы

Рассмотрим следующее:

```
x = y+f(2);
```

Чтобы эта запись имела смысл, имена **x**, **y** и **f** должны быть надлежащим образом объявлены. Это означает, что сущности, поименованные как **x**, **y** и **f**, существуют и имеют типы, для которых операции, вроде **=** (присваивание), **+** (сложение) и **()** (вызов функции), имеют смысл.

Каждое имя (идентификатор) в программе на языке C++ имеет связанный с ним тип. Этот тип задает набор операций, которые можно применять к этому имени (точнее, к сущности, скрывающейся за этим именем), и как эти операции интерпретируются. Например:

```
float x;      // x вещественная переменная
int y = 7;    // y целая переменная со значением 7
float f(int); // f – функция: аргумент типа int и возврат вещественного типа
```

Эти объявления делают приведенный выше пример осмысленным. Поскольку объявлено, что **y** имеет тип **int**, его можно присваивать, использовать в качестве операнда операции **+**, и так далее. Имя **f** объявлено как функция, которая принимает один аргумент типа **int**, и поэтому можно выполнить вызов этой функции, передав ей для примера в качестве аргумента число **2**.

В данной главе мы представляем фундаментальные типы (§6.2.1) и объявления (§6.3). Примеры в данной главе призваны лишь иллюстрировать характерные черты языка C++; они не предназначены ни для чего другого. Более сложные и более реалистичные примеры отложены до более поздних глав. А здесь нам нужно рассмотреть формальные конструкции, из которых состоят программы на языке C++. Эти конструкции, а также сопутствующую терминологию, вам нужно знать, чтобы писать собственные программы, а также понимать код, написанный другими. Исчерпывающее знание каждой рассматриваемой в настоящей главе детали не является абсолютно необходимым для понимания последующих глав. Как следствие, вы можете быстро пройтись по данной главе, схватить суть основных концепций, а позже вернуться к ней за более прочным овладением материала.

6.2.1. Фундаментальные типы

Язык C++ располагает набором *фундаментальных типов*, соответствующих наиболее общим единицам компьютера для хранения данных, а также наиболее общим способам использования таких данных:

§6.2.2 Логический тип (**bool**)

§6.2.3 Символьные типы (такие как **char** или **wchar_t**)

§6.2.4 Целые типы (вроде **int** или **long long**)

§6.2.5 Типы с плавающей запятой (**double** и **long double**)

§6.2.7 Тип **void**, означающий отсутствие информации

Из этих типов можно произвести новые типы, применяя декларатор (операции объявления):

§7.2 Типы указателей (такие как **int***)

§7.3 Типы массивов (вроде **char[]**)

§7.7 Типы ссылок (например, **double&** или **vector<int>&&**)

Кроме того, можно объявить еще ряд типов:

§8.2 Структуры данных и классы (Глава 16)

§8.4 Типы перечислений для представления специфических наборов значений (**enum** и **enum class**)

Логический тип, символьные и целые типы в совокупности называются *интегральными типами*. Интегральные типы и типы с плавающей запятой в совокупности называются *арифметическими типами*. Перечисления и классы (Глава 16) называются *типами, определяемыми пользователем*, или просто *пользовательскими типами*, поскольку эти типы пользователь языка (программист) должен сам определить до того, как их можно будет использовать (в отличие от фундаментальных типов). Соответственно, фундаментальные типы, ссылки, и указатели принято называть *встроенными типами*. В стандартной библиотеке определяется множество пользовательских типов (Глава 4, Глава 5).

Интегральные типы и типы с плавающей запятой представлены в нескольких вариантах их размеров, предоставляя программисту возможность выбрать размер потребляемой памяти, точность и диапазон значений, вовлекаемых в вычисления (§6.2.8). Предполага-

ется, что аппаратура компьютера предоставляет байты для хранения символов, слова для хранения и обработки целочисленных значений, некоторые блоки для реализации чисел с плавающей запятой, а также адреса для ссылок на все эти сущности. Фундаментальные типы языка C++ вместе с указателями и массивами представляют программисту эти понятия машинного уровня в манере, достаточно независимой от реализации.

В большинстве приложений мы можем использовать **bool** для логических значений, **char** для символов, **int** для целых значений и **double** для чисел с плавающей запятой. Остальные фундаментальные типы представляют собой вариации для оптимизации, специальных нужд и переносимости, которые можно игнорировать до тех пор, пока эти нужды не возникнут.

6.2.2. Логический тип

Логический тип **bool** имеет всего лишь два значения: **true** и **false**. Этот тип используется для отражения результатов логических операций. Например:

```
void f(int a, int b)
{
    bool b1 {a==b};
    // ...
}
```

Если **a** и **b** имеют одинаковые значения, **b1** примет значение **true**; иначе **b1** примет значение **false**.

Часто **bool** используется в качестве типа возврата функции, проверяющей выполнение некоторого условия (предиката). Например:

```
bool is_open(File*);

bool greater(int a, int b) {return a>b;}
```

По определению, **true** имеет значение **1** в случае конвертации в целое, а **false** имеет значение **0**. В обратном направлении целые неявно конвертируются в логические значения: ненулевые целые конвертируются в **true**, а нулевые значения — в **false**. Например:

```
bool b1 = 7;   // 7!=0, b становится true
bool b2 {7};   // ошибка: сужение (§ 2.2.2, § 10.5)

int i1 = true;  // i1 становится 1
int i2 {true};  // i2 становится 1
```

Если вы предпочитаете использовать {}-инициализаторы для предотвращения сужения значений, но по-прежнему хотите конвертировать **int** в **bool**, то можно выразить это явно:

```
void f(int i)
{
    bool b {i!=0};
    // ...
};
```

В арифметических и логических выражениях **bool** конвертируется в **int**; целочисленная арифметика и логические операции выполняются над конвертированными значениями. Если же результат нужно конвертировать обратно в **bool**, тогда **0** конвертируется в **false**, а ненулевое значение конвертируется в **true**. Например:

```
bool a = true;
bool b = true;

bool x = a+b;      // a+b есть 2, x становится true
bool y = a||b;     // a||b есть 1, y становится true ("||" значит "or")
bool z = a-b;      // a-b есть 0, z становится false
```

Указатель может неявно конвертироваться в **bool** (§10.5.2.5). Ненулевой указатель конвертируется в **true**; указатели со значением **nullptr** ковертируются в **false**. Пример:

```
void g(int* p)
{
    bool b = p;              // сужение до true или false
    bool b2 {p!=nullptr};    // явная проверка на nullptr

    if (p) {    // эквивалентно p!=nullptr
        // ...
    }
}
```

Я предпочитаю писать **if(p)** вместо **if(p!=nullptr)**, поскольку так лучше отражается понятие «если **p** имеет корректное значение», а также потому что оно короче. Чем короче, тем меньше шансов на описки.

6.2.3. Символьные типы

Имеется множество символьных наборов и используемых кодировок. Язык C++ располагает многими символьными типами, отражающими это многообразие:

- **char**: Символьный тип по умолчанию. Тип **char** используется для текстов программ в разных реализациях и обычно соответствует 8 битам.
- **signed char**: Гарантирован быть знаковым типом, то есть может содержать как положительные, так и отрицательные значения.
- **unsigned char**: Гарантирован быть беззнаковым типом.
- **wchar_t**: Предназначен для хранения символов из более широких символьных наборов, таких как Unicode (см. §7.3.2.2). Размер этого типа зависит от реализации, но он всегда достаточен для хранения наиболее широкого символьного набора из тех, что поддерживаются реализацией (Глава 39).
- **char16_t**: Предназначен для хранения символов из 16-битных символьных наборов, таких как UTF-16.
- **char32_t**: Предназначен для хранения символов из 32-битных символьных наборов, таких как UTF-32.

Итак, имеется шесть отдельных типов (несмотря на тот факт, что суффикс _t часто используется для обозначения синонимов; §6.5). В рамках любой реализации тип **char** эквивалентен либо **signed char**, либо **unsigned char**, но эти три типа все равно считаются отдельными типами.

Переменная типа **char** может содержать символ из символьного набора, предоставляемого реализацией. Например:

```
char ch = 'a';
```

Почти всегда тип **char** занимает 8 бит, так что он может содержать одно из 256 различных значений. В типичном случае, символьный набор является вариантом стандарта ISO-646, например ASCII, поддерживая, тем самым, все символы на вашей клавиатуре. Некоторые проблемы возникают из-за того, что набор символов на клавиатуре не полностью стандартизован.

Серьезные вариации имеют место между символьными наборами для разных национальных языков, а также между разными символьными наборами для одного и того же национального языка. Мы будем затрагивать эти вопросы лишь в связи с их влиянием на правила языка C++. Более обширная тема о программировании в мультиязыковых средах в целом выходит за рамки нашей книги, хотя она и немного затрагивается в отдельных местах (§6.2.3, §36.2.1, Глава 39).

Можно твердо положиться на то, что символьный набор в реализации поддерживает десятичные цифры, 26 алфавитных символов английского языка и основные символы пунктуации. Нельзя утверждать, что:

- В 8-битном наборе нет ничего, кроме 127 символов (например, некоторые наборы поддерживают 255 символов).
- Нет алфавитных символов иных, чем символы английского алфавита (есть иные европейские символы, такие как æ, þ или ß).
- Алфавитные символы образуют смежный набор (EBCDIC оставляет зазор между **i** и **j**).
- Все символы, необходимые для программирования на C++, содержатся на клавиатуре в виде отдельных клавиш (в случае использования некоторых национальных языков на клавиатуре отсутствуют знаки {,}, [,], | и \).
- Тип **char** умещается в 1 байт. Существуют процессоры для встроенных систем, которые не адресуют отдельные байты, и для которых **char** занимает 4 байта. Кроме того, возможно использование 16-битного UNICODE для базового типа **char**.

Мы всегда, по-возможности, будем стараться избегать предположений о реальном представлении объектов. Это общее правило применяется даже к символам.

Каждый символ имеет свое числовое значение в символьном наборе, используемом реализацией. Например, в ASCII числовое значение для символа **b** равно **98**. А вот пример цикла, который выводит целочисленные значения для каждого символа, которым вы интересуетесь:

```
void intval()
{
    for (char c; cin >> c;)
        cout << "the value of '" << c << "' is " << int{c} << '\n';
}
```

Запись **int{c}** дает целочисленное значение для символа **c** (конструируем **int** из символа **c**). Возможность конвертирования **char** в целое порождает вопрос: тип **char** знаковый или беззнаковый? 256 значений в 8-битовом представлении могут интерпретироваться как величины от **0** до **255**, или же как величины от **-127** до **127**. Именно так, а не как величины от **-128** до **127**, как можно было бы ожидать: стандарт C++ оставляет лазейку для аппаратуры с дополнением до единицы, что и удаляет одно значение; отсюда использование значения **-128** становится непереносимым. К сожалению, выбор для **char** быть знаковым или беззнаковым, зависит от реализации. В C++ для однозначного ответа на этот вопрос предусмотрены два типа: **signed char**, для которого диапазон значений от **-127** до **127**, и **unsigned char** с его диапазоном от **0** до **255**. По счастью, разница имеет значение лишь для величин за пределами диапазона от **0** до **127**, в который умещаются большинство общеупотребимых символов.

Значения вне этого диапазона, хранящиеся в «простом» **char**, могут приводить к неприятным проблемам с переносимостью. Смотрите §6.2.3.1, если вы намереваетесь использовать несколько символьных типов, или же если вы собираетесь хранить в символьных переменных целые числа.

Заметьте, что символьные типы — это интегральные типы (§6.2.1), так что к ним можно применять арифметические операции и побитовые логические операции (§10.3). Например:

```
void digits()
{
    for (int i=0; i!=10; ++i)
        cout << static_cast<char>('0'+i);
}
```

Это такой способ записать десять цифр в **cout**. Символьный литерал **'0'** конвертируется в целое и к нему прибавляется **i**. Результирующий **int** затем конвертируется в **char** и выводится (пишется) в **cout**. Выражение **'0'+i** имеет тип **int**, так что если бы я опустил **static_cast<char>**, то вывод был бы чем-то вроде **48**, **49** и так далее, а не **0**, **1** и т.д.

6.2.3.1. Знаковые и беззнаковые символьные типы

В зависимости от реализации тип **char** является либо знаковым, либо беззнаковым. Это открывает лазейку для неприятных сюрпризов. Например:

```
char c = 255; // 255 есть "все единицы", шестнадцатеричное 0xFF
int i = c;
```

Каково будет значение переменной **i**? К сожалению, ответ не определен. В 8-битовых представлениях все зависит от того, как интерпретируется значение **char**, состоящее из одних битовых единиц, при переводе его в **int**. На системах, где **char** является беззнаковым, ответ будет **255**. Там же, где **char** знаковый, там ответ будет **-1**. В последнем случае компилятор может предупредить о конвертации литерала **255** в **char** со значением, равным **-1**. В языке C++, однако, нет общего механизма преодоления проблем такого рода. Во избежание проблем можно отказаться от типа **char** и использовать более конкретные типы (в отношении их знаковости). К сожалению, некоторые средства стандартной библиотеки, например функция **strcmp()**, принимают лишь тип **char** (§43.4).

Тип **char** должен действовать идентично либо **signed char**, либо **unsigned char**. В то же время, все три типа считаются отдельными типами, так что нельзя смешивать указатели на эти типы. Пример:

```
void f(char c, signed char sc, unsigned char uc)
{
    char* pc = &uc;             // ошибка: нет конверсии в указатель
    signed char* psc = pc;      // ошибка: нет конверсии в указатель
    unsigned char* puc = pc;    // ошибка: нет конверсии в указатель
    psc = puc;                  // ошибка: нет конверсии в указатель
}
```

Переменные всех трех символьных типов могут присваиваться друг другу. Однако, присваивание слишком большого значения переменной типа **signed char** (§10.5.2.1) дает неопределенный результат. Например:

```
void g(char c, signed char sc, unsigned char uc)
{
    c = 255; // зависит от реализации если char знаковый и имеет 8 bit
    c = sc;  // OK
    c = uc;  // зависит от реализации если char знаковый и если величина uc велика
    sc = uc; // зависит от реализации если величина uc велика
    uc = sc; // OK: конверсия в unsigned
    sc = c;  // зависит от реализации если char беззнаковый и если величина c велика
    uc = c;  // OK: конверсия в unsigned
}
```

Для конкретности, предположим, что тип **char** занимает 8 бит:

```
signed char sc = -160;
unsigned char uc = sc;      // uc == 116 (потому что 256–160==116)
cout << uc;                 // печать 't'

char count[256];            // полагаем 8-bit char
++count[sc];                // вероятен крах: out-of-range доступ
++count[uc];                // OK
```

Эти проблемы не возникнут, если использовать только тип **char** и избегать отрицательных значений символов.

6.2.3.2. Символьные литералы

Символьный литерал — это одиночный символ, заключенный в одиночные кавычки, например, **'a'** или **'0'**. Типом символьного литерала является **char**. Символьный литерал может быть неявно конвертирован в целое значение в рамках символьного набора системы, на которой исполняется C++ программа. Например, если используется набор символов ASCII, то числовым значением для **'0'** будет **48**. Использование символьных литералов вместо десятичных чисел делает программы более переносимыми.

Некоторое количество символов имеет стандартные имена, использующие знак \ в качестве escape-символа:

Имя	ASCII имя	C++ имя
Новая строка	NL (LF)	**\n**
Горизонтальная табуляция	HT	**\t**
Вертикальная табуляция	VT	**\v**
Backspace (забой)	BS	**\b**
Возврат каретки	CR	**\r**
Подача листа	FF	**\f**
Тревога	BEL	**\a**
Обратная наклонная черта	\	****
Знак вопроса	?	**\?**
Одиночная кавычка		**\'**
Двойная кавычка		**\"**
Восьмеричное число	*ooo*	*****ooo*
Шестнадцатеричное число	*hhh*	**\x***hhh ...*

Несмотря на их внешний вид, это именно одиночные символы.

Мы можем представить символы из набора реализации с помощью восьмеричного числа, состоящего из одной, двух или трех цифр (символ \, за которым следуют восьмеричные цифры), или же как шестнадцатеричное число (символ **\x**, за которым следуют шестнадцатеричные цифры). Число шестнадцатеричных цифр не лимитируется. Последовательность восьмеричных или шестнадцатеричных цифр обрывается на первом же символе, который не является восьмеричной или шестнадцатеричной цифрой, соответственно. Например:

Восьмеричное	Шестнадцатеричное	Десятичное	ASCII
'\6'	**'\x6'**	**6**	ACK
'\60'	**'\x30'**	**48**	**'0'**
'\137'	**'\x05f'**	**95**	**'_'**

Так можно представить любой символ из символьного набора реализации, и в частности погрузить символы внутрь символьных строк (см. §7.3.2). Применение же числовых форм записи делает программу непереносимой между машинами с разными символьными наборами.

В принципе, допускается заключить несколько символов внутрь символьного литерала, например **'ab'**. Но такая конструкция архаична, зависит от реализации, и ее лучше не использовать. Типом такого многосимвольного литерала является **int**.

При погружении восьмеричной числовой константы внутрь символьной строки целесообразно использовать ровно три цифры. В этом случае не надо беспокоиться о том, какой символ следует далее: цифра или нет? В шестнадцатеричных константах используйте две цифры. Рассмотрим следующие примеры:

```
char v1[] = "a\xah\129";      // 6 chars: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";      // 5 chars: 'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";      // 4 chars: 'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";     // 5 chars: 'a' '\xad' '\012' '7' '\0'
```

Широкие символьные литералы имеют форму **L'ab'** и имеют тип **wchar_t**. Количество символов между одиночными кавычками и их смысл зависят от реализации.

В программах на C++ можно использовать символьные наборы, гораздо более богатые, нежели ASCII-набор из 127 символов, например набор UNICODE. Литералы таких расширенных наборов представляются последовательностью из четырех или восьми шестнадцатеричных цифр, которым предшествует **U** или **u**. Например:

```
U'\UFADEBEEF'
u'\uDEAD'
u'\xDEAD'
```

Укороченная форма **u'\uXXXX'** эквивалентна **U'U0000XXXX'** для любой шестнадцатеричной цифры **X**. Число шестнадцатеричных цифр, отличное от четырех или восьми, является ошибкой. Смысл шестнадцатеричной цифры определяется стандартом ISO/IEC10646 и эти значения называются *универсальными именами символов*. В стандарте C++ универсальные имена символов описываются в § iso.2.2, § iso.2.3, § iso.2.14.3, § iso.2.14.5 и § iso.E.

6.2.4. Целые типы

Как и в случае типа **char**, каждый целый тип представлен в трех вариантах: «простой», **signed** и **unsigned**. Кроме того, целые типы имеют четыре варианта размеров: **short int**, «просто» **int**, **long int** и **long long int**. Тип **long int** можно записать короче, просто как **long**, а тип **long long int** — как **long long**. Аналогично, **short** является эквивалентным **short int**, **unsigned** — эквивалентен **unsigned int**, а **signed** — **signed int**. Но нет никакого **long short int**, эквивалентного **int**.

Беззнаковые целые типы идеальны в случаях, когда переменные трактуются как хранилище битов. Применение **unsigned** вместо **int** лишь для того, чтобы получить в распоряжение лишний бит для представления положительных целых, не является хорошей идеей. А попытка гарантировать положительность некоторых переменных объявлением их **unsigned** в типичном случае нивелируется правилами неявных преобразований типов (§10.5.1, §10.5.2.1).

В отличие от **char**, «простой» **int** всегда знаковый. Знаковые варианты целых типов являются просто синонимами для соответствующих типов без явной спецификации ключевым словом **signed**, а вовсе не иными типами.

Если требуется более явный контроль за размерами целых типов, следует воспользоваться синонимами из **<cstdint>** (§43.7), такими как **int64_t** (знаковый целый с точным раз-

мером в 64 бита), **uint_fast16_t** (беззнаковый целый с размером в 8 бит, предположительно быстрейший из подобных целых) и **int_least32_t** (знаковый целый с размером минимум 32 бита, вроде простого **int**). «Простые» целые типы имеют точно определенные минимальные размеры (§6.2.8), так что **<cstdint>** часто избыточен.

Дополнительно к стандартным целым конкретные реализации могут предоставлять *расширенные целые типы* (знаковые и беззнаковые). Они обязаны вести себя как иные целые в контексте приведения типов и относительно литеральных значений, но они имеют больший диапазон значений (потребляя при этом больше памяти).

6.2.4.1. Целые литералы

Целые литералы могут быть представлены в трех обличьях: десятичном, восьмеричном и шестнадцатеричном. Десятичные литералы применяются чаще всего и выглядят вполне традиционно:

```
7    1234  976      12345678901234567890
```

Компилятор выдает предупреждение в случаях, когда представленное значение выходит за границы диапазона типа, но сообщение об ошибке возникнет лишь в случае {}-инициализаторов (§6.3.5).

Литерал, начинающийся с нуля, за которым следуют символы **x** или **X** (**0x** или **0X**), означает число в шестнадцатеричной форме записи. А если за нулем не следуют **x** или **X**, то это восьмеричное число. Пример:

Десятичное	Восьмеричное	Шестнадцатеричное
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x63

Буквы **a**, **b**, **c**, **d**, **e** и **f** (или их эквиваленты в верхнем регистре) используются для представления чисел **10**, **11**, **12**, **13**, **14** и **15**, соответственно. Восьмеричные и шестнадцатеричные формы записи целых чисел используются преимущественно для представления битовых комбинаций. Здесь вас, правда, могут поджидать некоторые неожиданности. Например, на машине, для которой **int** представим в виде 16-битов с дополнением до единицы, **0xffff** является отрицательным числом **-1**. Но если с указанной целью используется большее число бит, то **0xffff** означает положительное целое число **65535**.

Суффикс **U** применяется для явного указания беззнаковых литералов. А суффикс **L** явно указывает на **long**-литералы. Например, **3** это **int**, **3U** это **unsigned int**, а **3L** это **long int**.

Допускаются и комбинации суффиксов:

```
cout << 0xF0UL << ' ' << 0LU << '\n';
```

В отсутствие суффиксов компилятор приписывает целому литералу тип на основе его значения, сообразуясь с размерами целых в данной реализации (§6.2.4.2).

Будет правильным ограничивать использование явных (но неочевидных) значений некоторым количеством хорошо откомментированных констант (**const**; §7.5), выражений с **constexpr** (§10.4) и инициализаторами перечислений (§8.4).

6.2.4.2. Типы целых литералов

В общем случае, тип целого литерала зависит от формы его представления, величины и суффиксов:

- Если форма представления литерала десятичная и суффиксов нет, то ему приписывается первый тип, в котором умещается его значение: **int**, **long int**, **long long int**.
- Если форма литерала восьмеричная или шестнадцатеричная и суффиксов нет, то ему приписывается первый тип, в котором умещается его значение: **int**, **unsigned int**, **long int**, **unsigned long int**, **long long int**, **unsigned long long int**.
- Если есть суффиксы **u** или **U**, то ему приписывается первый тип, в котором умещается его значение: **unsigned int**, **unsigned long int**, **unsigned long long int**.
- Если форма литерала десятичная и есть суффиксы **l** или **L**, то ему приписывается первый тип, в котором умещается его значение: **long int**, **long long int**.
- Если форма восьмеричная или шестнадцатеричная и есть суффиксы **l** или **L**, то ему приписывается первый тип, в котором умещается его значение: **long int**, **unsigned long int**, **long long int**, **unsigned long long int**.
- Если есть суффиксы **ul**, **lu**, **uL**, **Lu**, **Ul**, **lU**, **UL** или **LU**, то литералу приписывается первый тип, в котором умещается его значение: **unsigned long int**, **unsigned long long int**.
- Если литерал десятичный и есть суффиксы **ll** или **LL**, то ему приписывается тип **long long int**.
- Если форма литерала восьмеричная или шестнадцатеричная и есть суффиксы **ll** или **LL**, то ему приписывается первый тип, в котором умещается его значение: **long long int**, **unsigned long long int**.
- Если есть суффиксы **llu**, **llU**, **ull**, **Ull**, **Llu**, **LLU**, **uLL** или **ULL**, то его тип **unsigned long long int**.

К примеру, **100000** имеет тип **int** на машине с 32-битными **int**, но имеет тип **long int** на машине с 16-битными **int** и 32-битными **long**. Аналогично, **0XA000** имеет тип **int** на машине с 32-битными **int**, но тип **unsigned int** на машине с 16-битными **int**. Такие зависимости от архитектуры устраняются суффиксами: **100000L** имеет тип **long int** на всех машинах, и **0XA000** имеет тип **unsigned int** для всех архитектур.

6.2.5. Типы чисел с плавающей запятой

Естественно, что типы с плавающей запятой представляют дробные числа с плавающей запятой. Последние являются аппроксимациями действительных чисел на ограниченном объеме памяти, который отводится под эти числа. Типов чисел с плавающей запятой три: **float** (одинарная точность представления), **double** (двойная точность) и **long double** (увеличенная точность).

Смысл этих понятий точности представления дробных чисел зависит от реализации. Здесь для полного понимания нужны хорошие знания теории вычислений. Если их у вас нет, то остается: получить сторонний совет, потратить время на изучение указанного предмета, или же выбрать **double** на удачу.

6.2.5.1. Литералы с плавающей запятой

По умолчанию литерал с плавающей запятой имеет тип **double**. Опять-таки, компилятор выдаст предупреждение в случае, если значение такого литерала не соответствует диапазону допустимых значений. Вот примеры некоторых литералов с плавающей запятой:

```
1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15
```

Никаких пробельных символов не допускается внутри литералов с плавающей запятой. Например, 65.43 e-21 это не литерал с плавающей запятой, а просто набор из четырех несвязанных лексем (вызывающих ошибку компиляции):

```
65.43 e - 21
```

Если вам нужны литералы с плавающей запятой в формате **float**, то нужно использовать суффикс **f** или **F**:

```
3.14159265f    2.0f  2.997925F      2.9e-3f
```

А если нужен тип **long double**, то следует использовать суффикс **l** или **L**:

```
3.14159265L    2.0L      2.997925L      2.9e-3L
```

6.2.6. Префиксы и суффиксы

Ниже представлен небольшой набор суффиксов и несколько префиксов, касающихся типов литералов:

Арифметические явные префиксы и суффиксы						
Нотация		**fix**	**Смысл**	**Пример**	**Ссылка**	**ISO**
0		prefix	восьмеричное	**0776**	§6.2.4.1	§ iso.2.14.2
0x	**0X**	prefix	шестнадцатеричное	**0xff**	§6.2.4.1	§ iso.2.14.2
u	**U**	suffix	**unsigned**	**10U**	§6.2.4.1	§ iso.2.14.2
l	**L**	suffix	**long**	**20000L**	§6.2.4.1	§ iso.2.14.2
ll	**LL**	suffix	**long long**	**20000LL**	§6.2.4.1	§ iso.2.14.2
f	**F**	suffix	**float**	**10f**	§6.2.5.1	§ iso.2.14.4
e	**E**	infix	вещественное	**10e-4**	§6.2.5.1	§ iso.2.14.4
		infix	вещественное	**12.3**	§6.2.5.1	§ iso.2.14.4
l		prefix	**char**	**'c'**	§6.2.3.2	§ iso.2.14.3
u'		prefix	**char16_t**	**u'c'**	§6.2.3.2	§ iso.2.14.3
U'		prefix	**char32_t**	**U'c'**	§6.2.3.2	§ iso.2.14.3
L'		prefix	**wchar_t**	**L'c'**	§6.2.3.2	§ iso.2.14.3
l!		prefix	строка	**"mess"**	§7.3.2	§ iso.2.14.5
R"		prefix	«сырая» строка	**R"(\b)"**	§7.3.2.1	§ iso.2.14.5
u8"	**u8R"**	prefix	UTF-8 строка	**u8"foo"**	§7.3.2.2	§ iso.2.14.5
u"	**uR"**	prefix	UTF-16 строка	**u"foo"**	§7.3.2.2	§ iso.2.14.5
U"	**UR"**	prefix	UTF-32 строка	**U"foo"**	§7.3.2.2	§ iso.2.14.5
L"	**LR"**	prefix	**wchar_t** строка	**L"foo"**	§7.3.2.2	§ iso.2.14.5

Здесь надпись «string» означает строковый литерал (§7.3.2), а не тип **std::string**.

Очевидно, что мы могли бы рассматривать . и **e** как инфиксы, а **R"** и **u8"** как начало набора разделителей. Однако, я считаю представление точной номенклатуры менее важным, чем обзор и без того большого разнообразия литералов.

Суффиксы **l** и **L** комбинируются с суффиксами **u** и **U** для отображения **unsigned long** типов. Пример:

```
1LU     // unsigned long
2UL     // unsigned long
3ULL    // unsigned long long
4LLU    // unsigned long long
5LUL    // ошибка
```

Для литералов с плавающей запятой суффиксы **l** и **L** обозначают тип **long double**. Пример:

```
1L      // long int
1.0L    // long double
```

Допускается комбинация префиксов **R**, **L** и **u**, например, **uR"**(foo\(bar))**"**. Обратите внимание на серьезное отличие в смысле префикса **U** для символа (**unsigned**) и для строк в кодировке UTF-32 (§7.3.2.2).

И, наконец, можно определять новые суффиксы для пользовательских типов. Вот пример использования определяемых пользователем операций задания типа литерала (§19.2.6):

```
"foo bar"s      // литерал типа std::string
123_km          // литерал типа Distance
```

Имейте в виду, что собственные суффиксы лучше начинать со знака подчеркивания, иначе может возникнуть противоречие с суффиксами, применяемыми в стандартной библиотеке.

6.2.7. void

С синтаксической точки зрения тип **void** — это фундаментальный тип. Но применять его можно лишь в сочетании с другими типами; не существует объектов типа **void**. Этот тип используется для того, чтобы просигнализировать о том, что функция не возвращает значение, а также в качестве базового типа для указателей на объекты неизвестного типа. Например:

```
void x;   // ошибка: не бывает void объектов
void& r;  // ошибка: не бывает ссылок на void
void f();  // функция f не имеет возврата (§12.1.4)
void* pv;  // указатель на объект неизвестного типа (§7.2.1)
```

Объявляя функцию, вы должны указать тип возвращаемого значения. С логической точки зрения, если функция ничего не возвращает, то вы можете просто опустить указание на тип возврата. Однако ж, это вносит серьезные путаницу в грамматику (§ iso.A). Поэтому мы используем **void** в качестве псевдотипа возврата функции, означающего «возврата нет».

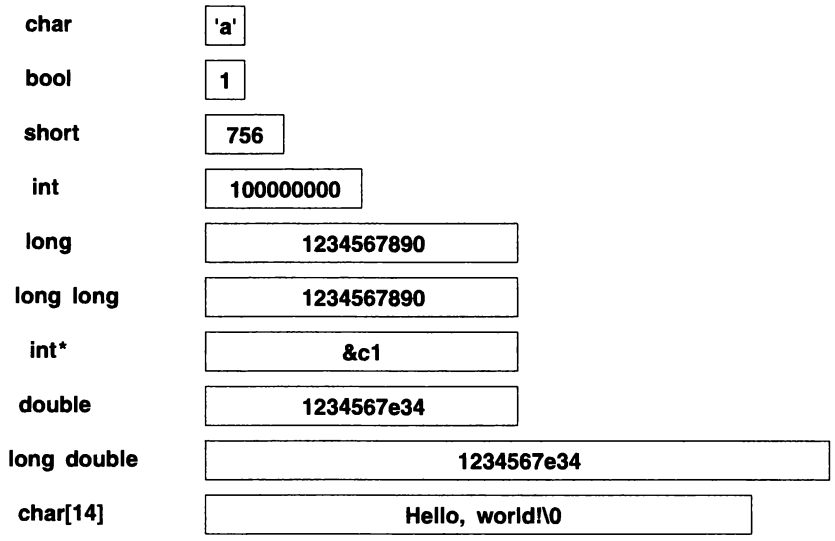
6.2.8. Размеры

Некоторые аспекты фундаментальных типов языка C++ являются зависящими от реализации (§6.1), например размер типа **int**. Я всегда обращаю на это внимание и советую избегать таких аспектов языка, или в случае невозможности, тщательно минимизировать и изолировать их влияние. Почему об этом стоит так беспокоиться. Ясно, что люди, программирующие для разных платформ, просто обязаны так поступать, иначе они захлебнутся в отслеживании и устранении тонких ошибок, возникающих во время исполнения программ. А люди, которые программируют строго для одной платформы на некотором фиксированном компиляторе могут расслабиться? Нет и еще раз нет. Это было бы слишком недальновидно. Ведь если вашу программу будет поджидать успех, то найдутся люди, которые обязательно будут портировать ее на другие платформы, и им достанутся подложенные вами «мины». Кроме того, может так случиться, что новые версии вашей программы придется компилировать на ином компиляторе (например, на новой версии вашего любимого компилятора) с несколько иными особенностями по отношению к размеру типов. В общем, надежнее и легче сразу осознать и минимизировать влияние частей программы, поведение которых может зависеть от реализации.

Но лимитировать это влияние в собстенном коде относительно несложно, а вот как быть с ним в случае системно зависимых библиотек? Одним из возможных решений проблемы может стать опора на стандартную библиотеку.

Причина, по которой предоставляется множество целых типов, множество unsigned типов и множество типов с плавающей запятой, заключается в желании дать программисту возможность максимально использовать аппаратуру. Для разных машин имеет место существенная разница в требованиях к потреблению памяти, в скорости доступа к памяти и в скорости выполнения вычислений относительно различных вариаций фундаментальных типов. Так что, если вы знакомы с машинной архитектурой, то будет нетрудно остановиться на подходящем целом типе для конкретной переменной. Однако, написать по-настоящему переносимый низкоуровневый код весьма непросто.

Ниже мы приводим графическое отображение фундаментальных типов на фоне взятых для примера строковых литералов (§7.3.2):



В заданном на рисунке масштабе мегабайт памяти занял бы расстояние в 3 мили (5 километров).

Размеры C++ объектов выражаются кратно по отношению к размеру типа **char**, так что по-определению размер **char** есть **1**. Размер объекта или типа можно получить с помощью операции **sizeof** (§10.3). Вот что гарантируется относительно размеров фундаментальных типов:

- **1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)**
- **1 ≤ sizeof(bool) ≤ sizeof(long)**
- **sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)**
- **sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)**
- **sizeof(N) ≡ sizeof(signed N) ≡ sizeof(unsigned N)**

В последней строке **N** может быть **char**, **short**, **int**, **long** или **long long**. Дополнительно гарантируется, что размер **char** не менее 8 бит, размер **short** — не менее 16 бит, а **long** — не менее 32 бит. Предполагается, что тип **char** определяется реализацией таким образом, чтобы было удобно представлять символы на конкретной машине; в типичном случае это 8-битовый байт. Аналогично, тип **int** определяется реализацией таким образом, чтобы было удобно хранить и манипулировать целыми числами на конкретной машине; в типичном случае это 4-байтовое (32-битовое) слово. Неразумно предполагать больший размер. Неразумно также считать, что тип **int** имеет тот же самый размер, что и указатель; на многих машинах (с 64-разрядной архитектурой) размер указателей больше размера целого.

Отметим также, что не гарантируется выполнение неравенств **sizeof(long)<sizeof(long long)** и **sizeof(double)<sizeof(long double)**.

Многие, зависящие от реализации аспекты фундаментальных типов, могут вскрываться с помощью **sizeof**, а другие можно найти в файле **<limits>**. Пример:

```
#include <limits> // §40.2
#include <iostream>

int main()
{
    cout << "size of long " << sizeof(1L) << '\n';
    cout << "size of long long " << sizeof(1LL) << '\n';

    cout << "largest float == " << std::numeric_limits<float>::max() << '\n';
    cout << "char is signed == " << std::numeric_limits<char>::is_signed << '\n';
}
```

Функции из **<limits>** (§40.2) все как один **constexpr** (§10.4), так что их применение не вносит накладных расходов во время выполнения и их можно применять в контекстах, требующих константных выражений.

Фундаментальные типы могут свободно комбинироваться в присваиваниях и выражениях. По мере возможности величины конвертируются так, чтобы избегать потерь информации (§10.5).

Если величина **v** может быть точно представлена типом **T**, то приведение **v** к типу **T** сохраняет величину неизменной. Иных приведений типов лучше избегать во избежание потерь информации (§2.2.2, §10.5.2.6).

Если вам потребуется конкретный размер для целой переменной, например 16-битовое целое, вы можете воспользоваться включаемым файлом **<cstdint>**, в котором определяется немалое множество типов (или же синонимов типов; §6.5). Например:

```
int16_t x {0xaabb};                 // 2 байта
int64_t xxxx {0xaaaabbbbccccdddd};  // 8 байтов
int_least16_t y;                    // по крайней мере 2 байта (как int)
int_least32_t yy                    // по крайней мере 4 байта (как long)
int_fast32_t z;                     // быстрейший int с 4 байтами по крайней мере
```

В стандартном включаемом файле **<cstddef>** определяется синоним, который широчайше используется как в стандартной библиотеке, так в пользовательском коде: **size_t** — это зависящий от реализации тип беззнакового целого, способного вместить размер в байтах для любого объекта. Естественно, что его применяют там, где нужно хранить размер объекта. Например:

```
void* allocate(size_t n); // получить n байт
```

Аналогично, **<cstddef>** определяет знаковый целый тип **ptrdiff_t** для хранения разности значений двух указателей, равной числу элементов.

6.2.9. Выравнивание

Для хранения объекта требуется не только достаточный объем памяти. На некоторых машинных архитектурах также требуется, чтобы байты, отводимые под хранение объекта, были определенным образом выравнены с тем, чтобы аппаратура могла осуществить эффективный доступ к объекту (в экстремальных случаях вообще могла бы выполнить доступ). Например, 4-байтовые **int** часто нужно выравнивать по границам слова (равного 4 байт), а 8-байтовый **double** нужно выравнивать по 8-байтовым границам. Конечно же,

все это очень специфично для конкретных машинных архитектур, и для большинства программистов абсолютно прозрачно. Вы можете с успехом программировать десятилетия без нужды явным образом беспокоиться относительно выравнивания. Один из примеров, где вопрос с выравниванием довольно заметен, это так называемые «дырки» в структурах (**struct**), существующие как раз для выравнивания (§8.2.1):

Операция **alignof()** возвращает величину выравнивания для своего аргумента-выражения. Пример:

```
auto ac = alignof('c');     // выравнивание char
auto ai = alignof(1);       // выравнивание int
auto ad = alignof(2.0);     // выравнивание double

int a[20];
auto aa = alignof(a);       // выравнивание int
```

Иногда мы должны специфицировать выравнивание в объявлении, для которого нельзя применить выражение вроде **align(x+y)**. Вместо этого можно использовать спецификатор **alignas**: **alignas(T)** означает «выровнять как для **T**». К примеру, мы можем отложить (про запас) неинициализированный блок памяти для некоторого типа **X** следующим образом:

```
void user(const vector<X>& vx)
{
    constexpr int bufmax = 1024;
    alignas(X) buffer[bufmax];          // не инициализировано

    const int max = min(vx.size(), bufmax/sizeof(X));
    uninitialized_copy(vx.begin(), vx.begin()+max, buffer);
    // ...
}
```

6.3. Объявления

До того, как имя (идентификатор) можно будет использовать в программе на C++, его нужно объявить. То есть, нужно информировать компилятор о типе, чтобы компилятор понял, с какой сущностью ему придется иметь дело. Например:

```
char ch;
string s;
auto count = 1;
const double pi {3.1415926535897};
extern int error_number;

const char* name = "Njal";
const char* season[] = {"spring", "summer", "fall", "winter"};
vector<string> people {name, "Skarphedin", "Gunnar"};

struct Date {int d, m, y;};
int day(Date* p) {return p->d;}
double sqrt(double);
template<class T> T abs(T a) {return a<0? -a: a;}

constexpr int fac(int n) {return (n<2)?1: n*fac(n-1);} // оценка во время компиляции (§2.2.3)
constexpr double zz {ii*fac(7)};                       // инициализация во время компиляции

using Cmplx = std::complex<double>;                    // синоним типа (§ 3.4.5, § 6.5)
struct User;                                           // имя типа
enum class Beer {Carlsberg, Tuborg, Thor};
namespace NS {int a;}
```

Как видно из показанных примеров, объявление может делать больше, чем просто ассоциировать имя с его типом. Большинство из представленных *объявлений* являются также и *определениями*. Определение — это такое объявление, в котором сообщается все, что нужно для начала использования определяемой сущности в программе. В частности, если сущность требует выделения некоторой памяти, эта память в определении выделяется (откладывается для последующего использования). В рамках иной терминологии говорится, что объявления составляют часть интерфейса, в то время как определения являются частью реализации. С этой точки зрения мы стараемся составлять интерфейсы из объявлений таким образом, чтобы группировать их в отдельные файлы (§15.2.2); с последующим включением туда, где это будет требоваться.

Предполагая, что эти определения находятся в глобальной области видимости (§6.3.4), имеем:

```
char ch;                        // выделить память под char и инициализировать нулем
auto count = 1;                 // выделить память под int и инициализировать единицей
const char* name = "Njal";      // выделить память для указателя на char
                                // выделить память для строкового литерала "Njal"
                                // инициализировать указатель адресом строкового литерала

struct Date {int d, m, y;};     // Date это структура с тремя полями
int day(Date* p) {return p->d;} // day это функция, выполняющая указанный код

using Point = std::complex<short>; // Point это имя для std::complex<short>
```

Из представленных объявлений только три не являются определениями:

```
double sqrt(double);            // объявление функции
extern int error_number;        // объявление переменной
struct User;                    // объявление имени типа
```

То есть, чтобы их использовать, их нужно где-то определить. Например:

```
double sqrt(double d) {/* ... */}
int error_number = 1;
struct User {/* ... */};
```

Нужно, чтобы для любого имени в программе на C++ присутствовало ровно одно определение (см. §15.2.3). А вот объявлений может быть произвольное количество.

Все объявления одной и той же сущности должны согласовываться по типу. Следующий фрагмент содержит две ошибки:

```
int count;
int count;                      // ошибка: повторное определение
extern int error_number;
extern short error_number;      // ошибка: несовпадение типа
```

А в следующем фрагменте ошибок нет (про использование **extern** см. §15.2):

```
extern int error_number;
extern int error_number;        // OK: повторное объявление
```

Определения могут содержать явное значение для определяемой сущности. Пример:

```
struct Date {int d, m, y;};
using Point = std::complex<short>;      // Point это имя для std::complex<short>
int day(Date* p) {return p->d;}
const double pi {3.1415926535897};
```

Для типов, синонимов, шаблонов, функций и констант «значение» неизменно. Для не-const типов данных начальное значение может быть впоследствии изменено. Например:

```
void f()
{
    int count {1};                    // инициализируем count в 1
    const char* name {"Bjarne"}; // name это переменная, указывающ. на константу(§ 7.5)
    count = 2;                        // присваиваем 2 переменной count
    name = "Marian";
}
```

Из представленных выше определений только два не специфицируют значение:

```
char ch;
string s;
```

В §6.3.5 и §17.3.3 можно прочитать о том, как и где переменная получает умолчательное значение. Любое объявление, специфицирующее значение, является определением.

6.3.1. Структура объявлений

Структура объявлений определяется грамматикой C++ (§ iso.A). Эта грамматика эволюционировала в течение десятилетий, начиная с ранней грамматики языка C, и является довольно-таки сложной. Однако, не опускаясь до слишком уж больших упрощений, можно считать, что объявление состоит из пяти частей:

- Опциональные префиксные спецификаторы (вроде **static** или **virtual**)
- Базовый тип (вроде **vector<double>** или **const int**)
- Декларатор с опциональным именем (например, **p[7]**, **n** или ***(*)[]**)
- Опционально суффиксные спецификаторы функций (например, **const** или **noexcept**)
- Опционально инициализатор или тело функции (например, **={7,5,3}** или **{return x;}**)

За исключением определений функций и пространств имен объявление завершается точкой с запятой. Рассмотрим, например, определение массива строк в C-стиле:

```
const char* kings[] = {"Antigonus", "Seleucus", "Ptolemy"};
```

Здесь базовый тип **const char**, декларатор ***kings[]**, а инициализатор — это знак **=**, за которым следует {}-список.

Спецификатором служит ключевое слово, такое как **virtual** (§3.2.3, §20.3.2), **extern** (§15.2) или **constexpr** (§2.2.3), которое специфицирует нетиповой атрибут того, что собственно объявляется.

Декларатор составляется из имени и опционально из ряда операций объявления. Вот список наиболее распространенных из них:

Операции в декларациях		
prefix	*	указатель
prefix	*const	constant указатель
prefix	*volatile	volatile указатель
prefix	&	lvalue ссылка (§ 7.7.1)
prefix	&&	rvalue ссылка (§ 7.7.2)
prefix	auto	функция (с суффиксным типом возврата)
postfix	[]	массив
postfix	()	функция
postfix	->	возврат из функции

Их применение было бы более простым, будь все они либо префиксными, либо постфиксными. Однако, *, [] и () спроектированы так, чтобы соответствовать их же применению

в выражениях (§10.3). В итоге, операция * префиксная, а [] и () постфиксные. Постфиксные операции объявления имеют приоритет связывания перед префиксными операциями. Следовательно, **char* kings[]** это массив указателей на **char**, в то время как **char(*kings)[]** это указатель на массив элементов типа **char**. Приходится использовать круглые скобки, чтобы объявлять типы вроде типа «указатель на массив» и «указатель на функцию»; смотри примеры из §7.2.

Отметим, что тип нельзя выкинуть из объявления. Например:

```
const c = 7;              // ошибка: нет типа

gt(int a, int b)          // ошибка: нет типа возврата
{
    return (a>b)? a: b;
}

unsigned ui;              // OK: "unsigned" означает "unsigned int"
long li;                  // OK: "long" означает "long int"
```

В этом отношении стандарт C++ отличается от ранних версий C и C++, в которых первые из двух примеров были разрешены и где по умолчанию предполагался тип **int** (§44.3). В результате такого умолчания на практике часто возникали тонкие ошибки и досадные недоразумения.

Ряд типов имеют имена, составленные из нескольких ключевых слов, такие как **long long** и **volatile int**. И есть типы, имена которых вообще не выглядят как имена, например **decltype(f(x))** (тип возврата вызова **f(x)**; §6.3.6.3).

Спецификатор **volatile** описывается в §41.4.

Спецификатор **alignas()** описывается в §6.2.9.

6.3.2. Объявление нескольких имен

В единственном объявлении можно объявить несколько имен сразу. Такое объявление просто содержит разделенный запятыми список деклараторов. Например, можно объявить сразу две целых переменных:

```
int x, y; // int x; int y;
```

Операции в объявлениях относятся непосредственно к отдельному имени, а не ко всем именам из данного объявления. Пример:

```
int* p, y; // int* p; int y; НЕ int* y;
int x, *q; // int x; int* q;
int v[10], *pv; // int v[10]; int* pv;
```

Вообще-то, объявления с несколькими именами и нетривиальными операциями делают программу менее читабельной, так что лучше ими не слишком сильно увлекаться.

6.3.3. Имена

Имя (идентификатор) состоит из последовательности букв и цифр. Первым символом имени должна быть буква. Символ подчеркивания, _, рассматривается как буква. Язык C++ не налагает никакого ограничения на количество символов в имени. Однако, следует иметь в виду, что некоторые части реализации находятся вне зоны ответственности разработчика компилятора. Например, это компоновщик (линкер). И иногда вот такие части реализации могут накладывать некотрые ограничения на имена. Ряд сред выполнения могут требовать расширения или же сужения набора символов, допустимых в идентификаторах.

Расширения (например, допускающие в именах символ **$**) приводят к непереносимым программам. В пользовательских именах запрещено использовать ключевые слова языка C++ (§6.3.3.1), такие как, например, **new** или **int**. Приведем примеры имен:

```
hello     this_is_a_most_unusually_long_identifier_that_is_better_avoided
DEFINED   foO      bAr      u_name      HorseSense
var0      var1     CLASS    _class ___
```

А вот примеры последовательностей символов, которые нельзя использовать в качестве имен:

```
012       a fool   $sys     class    3var
pay.due   foo˜bar  .name    if
```

Глобальные имена, начинающиеся со знака подчеркивания, традиционно стараются резервировать для внутренних нужд конкретных реализаций и сред выполнения, так что их не стоит использовать для пользовательских имен. Аналогично, резервируются имена, начинающиеся с двух символов подчеркивания или с символа подчеркивания, за которым следует буква в верхнем регистре (вроде **_Foo**) (§ iso.17.6.4.3).

Читая программу, компилятор ищет наиболее длинную последовательность символов, которую можно трактовать как имя. В результате **var10** является единственным именем, а не именем **var**, за которым следует число **10**. Также **elseif** это одно имя, а не ключевое слово **else** за которым следует ключевое слово **if**.

Буквы в нижнем и вернем регистрах трактуются как разные символы, так что **Count** и **count** это разные имена, что впрочем вряд ли является разумным выбором имен, ибо при быстром чтении текста программы это может сбить с толку читающего. Вообще лучше избегать имен, визуально слабо различающихся. В частности в ряде шрифтов буква **O** (в верхнем регистре) и знак нуля (**0**) слабо различимы, что также характерно и для буквы **l** (нижний регистр), буквы **I** (верхний регистр) и единицы (**1**). Таким образом, выбор идентификаторов **l0**, **lO**, **l1** **ll** и **I1l** вряд ли следует признать разумным. Не во всех шрифтах имеются такие проблемы, но в большинстве дело обстоит именно так.

Имена, работающие в широких областях видимости, лучше делать достаточно длинными и самоочевидными, вроде **vector**, **Window_with_border** и **Department_number**. В то же время ради краткости и простоты записи локальные имена лучше делать короткими и традиционными, вроде **x**, **i** и **p**. Функции (Глава 12), классы (Глава 16) и пространства имен (§14.3.1) могут применяться для сужения областей видимости имен. Часто применяемые имена стоит делать короткими, а редко применяемые — более длинными.

Выбирайте имена так, чтобы они отражали смысл именуемой сущности, а не внутренних деталей реализации. Например, **phone_book** лучше, чем **number_vector**, даже в том случае, когда телефонные имена на самом деле хранятся в контейнере **vector** (§4.4). Не стоит пытаться отражать в именах информацию о типе, как это часто делается в языках с динамической или же слишком слабой типизацией:

- Отражение типа в именах снижает уровень абстракции, в частности противоречит обобщенному программированию (базируется на именах, которые могут ссылаться на сущности разных типов)
- Компилятор лучше программиста отслеживает типы
- Если вы захотите сменить тип для какого-то имени, то вам придется всюду корректировать его имя, отражающее тип
- Система отражения типа в именах при возрастании количества применяемых типов неизбежно превратится в шифровальную систему (мало кому понятную)

Вообще, конечно, выбор оптимальных имен является большим искуством.

Старайтесь придерживаться согласованной системы именования. Например, начинайте с заглавной буквы имена типов, а имена нетиповых сущностей — с букв в нижнем регистре (например, **Shape** и **current_token**). Общепринятой практикой является запись имен макросов одними заглавными буквами (если уж вам нужны макросы (§12.6); например **HACK**), и только для них. Отделяйте смысловые слова в идентификаторах знаками подчеркивания; **number_of_elements** более читабельно, чем **numberOfElements**. Однако очевидно, что полной согласованности в именовании добиться вряд ли возможно, ибо в типичном случае разные части программ разрабатываются разными группами программистов, предпочитающими свои любимые стили. Отметим также, что в языке C++ и в стандартной библиотеке имена типов пишутся буквами нижнего регистра; это можно рассматривать как намек на то, что они относятся к стандарту.

6.3.3.1. Ключевые слова

Вот ключевые слова языка C++:

C++ ключевые слова					
alignas	**alignof**	**and**	**and_eq**	**asm**	**auto**
bitand	**bitor**	**bool**	**break**	**case**	**catch**
char	**char16_t**	**char32_t**	**class**	**compl**	**const**
constexpr	**const_cast**	**continue**	**decltype**	**default**	**delete**
do	**double**	**dynamic_cast**	**else**	**enum**	**explicit**
extern	**false**	**float**	**for**	**friend**	**goto**
if	**inline**	**int**	**long**	**mutable**	**namespace**
new	**noexcept**	**not**	**not_eq**	**nullptr**	**operator**
or	**or_eq**	**private**	**protected**	**public**	**register**
reinterpret_cast	**return**	**short**	**signed**	**sizeof**	**static**
static_assert	**static_cast**	**struct**	**switch**	**template**	**this**
thread_local	**throw**	**true**	**try**	**typedef**	**typeid**
typename	**union**	**unsigned**	**using**	**virtual**	**void**
volatile	**wchar_t**	**while**	**xor**	**xor_eq**	

Кроме того, слово **exports** зарезервировано на будущее.

6.3.4. Область видимости

Объявление вводит имя в некоторую область видимости; то есть имя можно использовать только в определенных частях программы.

- *Локальная область видимости*: Имя, объявляемое в функции (Глава 12) или в лямбде (§11.4), называется *локальным именем*. Его область видимости начинается в точке объявления и простирается до конца блока, содержащего это объявление. *Блок* — это секция кода, содержащаяся внутри фигурных скобок. Имена параметров функций или лямбд считаются локальными в теле их определения (за исключением внутренних блоков).
- *Область видимости класса*: имя называется *именем члена класса*, если оно определяется в классе вне функций, иных классов (Глава 16), enum class (§8.4.1) и пространств имен. Его область видимости начинается с открывающей фигурной скобки объявления класса и простирается до конца объявления класса.
- *Область видимости пространства имен*: имя называется *именем из пространства имен*, если оно определяется в пространстве имен (§14.3.1) вне функций, лямбд

(§11.4), классов (Глава 16), enum class (§8.4.1) и иных пространств имен. Его область видимости начинается в точке объявления и действительна всюду далее в пространстве имен. Имя из пространства имен достижимо из иных единиц трансляции (§15.2).

- *Глобальная область видимости*: имя называется *глобальным*, если оно определяется вне функций, классов (Глава 16), enum class (§8.4.1) и пространств имен (§14.3.1). Область видимости глобального имени начинается в точке его объявления и простирается до конца файла, в котором это объявление содержится. Глобальное имя достижимо из иных единиц трансляции (§15.2). С технической точки зрения глобальное пространство имен — это тоже пространство имен.
- *Область видимости оператора*: имя входит в область видимости оператора, если оно определяется внутри круглых скобок, относящихся к операторам **for**, **while**, **if** и **switch**. Эта область начинается в точке объявления и простирается до конца оператора.
- *Область видимости функции*: метка (§9.6) становится видимой с точки ее объявления и остается таковой до конца определения функции.

Объявление имени в блоке скрывает ее объявление в объемлющем блоке и глобальное имя. Таким образом, имя можно переобъявить в блоке для того, чтобы использовать его внутри указанного блока для ссылки на другую сущность. После выхода из этого блока восстанавливается действие предыдущего объявления. Пример:

```
int x;                  // глобальное x

void f()
{
    int x;              // локальное x скрывает глобальное x
    x = 1;              // присвоение локальному x
    {
        int x;          // скрывает первое локальное x
        x = 2;          // присвоение второму локальному x
    }
    x = 3;              // присвоение первому локальному x
}

int* p = &x;            // взятие адреса глобального x
```

Сокрытие имен неизбежно при разработке больших программ. В то же время, программисту-человеку заметить такое сокрытие (часто называемое затенением) не так-то просто. Поэтому сокрытия имен следует избегать. Применять имена вроде **i** или **x** в качестве глобальных или же в качестве локальных в больших функциях — это означает нарываться на неприятности.

На скрытое глобальное имя можно ссылаться при помощи операции::. Пример:

```
int x;

void f2()
{
    int x = 1;          // скрываем глобальное x
    ::x = 2;            // присвоение глобальному x
    x = 2;              // присвоение локальному x
    // ...
}
```

Не существует способа «открыть» скрытое локальное имя.

Область видимости имени, не являющегося именем члена класса, начинается с точки его объявления, то есть сразу же за полным декларатором и перед инициализатором. Это

означает, что имя может использоваться даже для отражения своего собственного начального значения. Пример:

```
int x = 97;

void f3()
{
    int x = x;  // извращение: инициализация x собственным (неинициализир.) значением
}
```

Хороший компилятор всегда предупредит о попытке использовать неинициализированную переменную.

Внутри блока возможно использование единственного имени для ссылки на два разных объекта (без применения операции::):

```
int x = 11;

void f4()           // извращение: использование двух разных x в одной области видимости
{
    int y = x;      // используем глобальное x: y = 11
    int x = 22;
    y = x;          // используем локальное x: y = 22
}
```

Но, конечно же, не стоит увлекаться такими «изысками».

Считается, что имена аргументов функции объявлены в теле функции в самом объемлющем блоке (то есть в самых внешних фигурных скобках):

```
void f5(int x)
{
    int x;          // ошибка
}
```

Здесь присутствует ошибка, ибо имя x определено дважды в одной и той же области видимости.

Имена, определяемые в операторе **for** локальны по отношению к этому оператору (область видимости оператора). Пример:

```
void f(vector<string>& v, list<int>& lst)
{
    for (const auto& x: v) cout << x << '\n';
    for (auto x: lst) cout << x << '\n';
    for (int i = 0, i!=v.size(), ++i) cout << v[i] << '\n';
    for (auto i: {1, 2, 3, 4, 5, 6, 7}) cout << i << '\n';
}
```

Здесь отсутствуют конфликты имен.

Объявление в качестве единственного оператора в ветви оператора **if** () запрещено.

6.3.5. Инициализация

Если у объекта присутствует инициализатор, то он задает для этого объекта начальное значение. Для инициализатора допустимо использовать четыре формы представления:

```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

Из представленных четырех вариантов только первый можно применять в любых контекстах и я рекомендую именно его. В то же время, эта самая первая форма (использо-

ванная для **a1**) является новацией C++11, так что другие три варианта всегда присутствуют в старом коде. Две формы со знаком = присутствуют еще в коде на языке C. Старые привычки отмирают медленно, так что я часто использую = в случаях инициализации простых переменных простыми значениями. Например:

```
int x1 = 0;
char c1 = 'z';
```

Все, что сложнее показанного примера, следует инициализировать при помощи {}. Эта форма инициализации называется инициализацией списком, и она не допускает сужения значений (iso.§8.5.4). Таким образом:

- Целое не может конвертироваться в иное целое, которое не в состоянии вместить заданное значение. Например, конвертация **char** в **int** допускается, а **int** в **char** — нет.
- Величина с плавающей запятой не может конвертироваться в иной тип с плавающей запятой при невозможности отразить в нем заданное значение. Например, конвертация **float** в **double** допустима, а **double** во **float** — нет.
- Величина с плавающей запятой не может конвертироваться в целое.
- Целая величина не может конвертироваться в тип с плавающей запятой.

Пример:

```
void f(double val, int val2)
{
    int x2 = val;       // если val==7.9, x2 становится 7
    char c2 = val2;     // если val2==1025, c2 становится 1

    int x3 {val};       // ошибка: возможно усечение
    char c3 {val2};     // ошибка: возможно сужение

    char c4 {24};       // OK: 24 можно представить как char
    char c5 {264};      // ошибка (если 8-bit char): 264 не представимо в виде char

    int x4 {2.0};       // ошибка: нет конверсии double в int

    // ...
}
```

По поводу правил конвертации встроенных типов смотри §10.5.

Но преимущества {}-инициализации исчезают в случае применения **auto** с целью вывода типа, ибо во многих случаях при {}-инициализации мы не нуждаемся в выводе типа. Например:

```
auto z1 {99};       // z1 есть initializer_list<int>
auto z2 = 99;       // z2 есть int
```

Так что применяйте = в случае использования **auto**.

Можно определять объект класса так, что он будет инициализироваться списком значений, а можно альтернативным образом указать каких-нибудь пару значений, которые вовсе не относятся к величинам, подлежащим хранению в объекте. Вот классический пример для объектов типа **vector** с целыми элементами:

```
vector<int> v1 {99};    // v1 вектор из одного элемента 1 со значением 99
vector<int> v2(99);     // v2 вектор из 99 элементов со значениями 0
```

Большинство типов не допускают таких альтернатив, даже для **vector** это не всегда допускается:

```
vector<string> v1{"hello!"};    // v1 вектор из одного элемента со значением "hello!"
vector<string> v2("hello!");    // ошибка: нет конструктора из строкового литерала
```

В сухом остатке, предпочитайте {}-инициализацию во всех случаях, если только у вас нет ясных причин не делать этого.

Пустой список инициализации означает, что требуется использовать умолчательное значение. Пример:

```
int x4 {};          // x4 становится 0
double d4 {};       // d4 становится 0.0
char* p {};         // p становится nullptr
vector<int> v4{};   // v4 становится пустым вектором
string s4 {};       // s4 становится ""
```

Большинство типов имеют умолчательные значения. Для интегральных типов это подходящее представление нуля. Для указателей умочательное значение есть **nullptr** (§7.2.2). Для пользовательских типов умолчательное значение (если таковое имеется) задается конструкторами этого типа (§17.3.3).

Для пользовательских типов имеется четкое различие между прямой инициализацией (в которой допускается неявное приведение типа) и инициализацией копированием (где это не допускается); см. §16.2.6.

Инициализация конкретных типов объектов обсуждается в соответствующих местах книги:

- Указатели: §7.2.2, §7.3.2, §7.4.
- Ссылки: §7.7.1 (lvalues), §7.2.2 (rvalues).
- Массивы: §7.3.1, §7.3.2.
- Константы: §10.4.
- Классы: §17.3.1 (не используя конструкторы), §17.3.2 (посредством конструкторов), §17.3.3 (умолчательными значениями), §17.4 (инициализация членов и инициализация части, достающейся от базового класса), §17.5 (копирование и перемещение).
- Пользовательские контейнеры: §17.3.4.

6.3.5.1. Отсутствие инициализаторов

Для многих типов, включая все встроенные типы, допускается отсутствие инициализаторов. Если вы так делаете, а на практике такое встречается нередко, вы только усложняете себе жизнь. Единственный разумный случай — это большой буфер для входных данных. Пример:

```
constexpr int max = 1024*1024;
char buf[max];
some_stream.get(buf, max);   // читать максимум max символов в buf
```

Но мы могли бы с легкостью проинициализировать этот буфер:

```
char buf[max] {};   // инициализируем каждый char в 0
```

Такая избыточная инициализация, однако, может существенно понизить эффективность выполнения программы. В принципе, лучше вообще избегать таких низкоуровневых буферов, или же все-таки не оставлять их непроинициализированными, кроме случаев, когда очевидна потеря в производительности.

В отсутствие инициализатора глобальный объект (§6.3.4), пространство имен (§14.3.1), локальная переменная с ключевым словом **static** (§12.1.8) или член класса с модификатором **static** (§16.2.12) (коллективно известные как *статические объекты*) инициализируются посредством {} для соответствующего типа. Например:

```
int a;          // значит "int a{};" a становится 0
double d;       // значит "double d{};" d становится 0.0
```

Локальные переменные и объекты, создаваемые в свободной памяти (иногда называемые *динамическими объектами* или *объектами кучи*; §11.2) не инициализируются умолчательными значениями кроме случаев пользовательских типов с умолчательными конструкторами (§17.3.3). Например:

```
void f()
{
    int x;                          // x не имеет точно определенного значение
    char buf[1024];                 // buf[i] не имеет точно определенного значения

    int* p {new int};               // *p не имеет точно определенного значения
    char* q {new char[1024]};       // q[i] не имеет точно определенного значения

    string s;                       // s=="" из-за умолчательного конструктора
    vector<char> v;                 // v=={} из-за умолчательного конструктора

    string* ps {new string};        // *ps is "" из-за умолчательного конструктора
    // ...
}
```

Если вам требуется инициализация локальных переменных встроенных типов или объектов встроенных типов, создаваемых операцией **new**, используйте {}. Пример:

```
void ff()
{
    int x {};                       // x становится 0
    char buf[1024]{};               // buf[i] становится 0 для всех i

    int* p {new int{10}};           // *p становится 10
    char* q {new char[1024]{}};     // q[i] становится 0 для всех i

    // ...
}
```

Элементы массивов и члены классов инициализируются по умолчанию при инициализации массивов и структур.

6.3.5.2 Списки инициализации

До сих пор мы рассматривали случаи, когда либо совсем не было инициализации, или же она выполнялась посредством единственного значения. Более сложные объекты могут требовать для инициализации большего количества значений. Это как правило выполняется с помощью списков инициализации, заключенных в фигурные скобки. Вот примеры:

```
int a[] = {1, 2};                           // инициализатор массива
struct S {int x, string s};
S s = {1, "Helios"};                        // инициализатор структуры
complex<double> z = {0, pi};                // применяется constructor
vector<double> v = {0.0, 1.1, 2.2, 3.3};    // используется конструктор списком
```

По поводу инициализации массивов в С-стиле смотри §7.3.1; структуры в С-стиле рассматриваются в §8.2; пользовательские типы с конструкторами — в §2.3.2 или §16.2.5; конструкторы со списками инициализации — §17.3.4.

В рассмотренных выше примерах использование = является излишним. Однако, многие любят применять = для акцентирования того факта, что набор значений используется для инициализации набора членов структуры.

В ряде случаев можно применять список аргументов в функциональном стиле (§2.3, §16.2.5). Например:

```
complex<double> z(0, pi);      // используется конструктор
vector<double> v(10,3.3);      // используется конструктор: v получает 10 элементов по 3.3
```

В объявлениях пара пустых круглых скобок всегда означает «функция» (§12.1). Таким образом, если вы хотите явным образом указать на необходимость «умолчательной инициализации», используйте {}. Пример:

```
complex<double> z1(1,2);       // инициализатор в функц. стиле (инициализация конструктором)
complex<double> f1();          // объявление функции

complex<double> z2 {1,2};      // инициализируем конструктором в {1,2}
complex<double> f2 {};         // инициализация конструктором в умолчатель. значение {0,0}
```

Не забывайте, что {}-инициализация не допускает сужения значений (§6.3.5).

Если используется **auto**, то тип для {}-списка выводится в формате **std::initializer_list<T>**. Например:

```
auto x1 {1,2,3,4};             // x1 есть initializer_list<int>
auto x2 {1.0, 2.25, 3.5};      // x2 есть an initializer_list of<double>
auto x3 {1.0,2};               // ошибка: нельзя вывести тип для {1.0,2} (§ 6.3.6.2)
```

6.3.6. Выведение типа: auto и decltype()

В языке имеются два механизма выведения типа для выражения:

- **auto** — для выведения типа объекта из инициализатора; тип может быть типом переменной, константы (**const**) или константного выражения (**constexpr**).
- **decltype(expr)** — для выведения типа из того, что не может быть простым инициализатором, вроде типа возврата функции или типа члена класса.

Выведение выполняется в общем-то очень просто: **auto** или **decltype()** просто объявляют тип выражения, уже известный компилятору.

6.3.6.1 Спецификатор типа auto

Когда в объявлении переменной присутствует инициализатор, нам нет необходимости явным образом специфицировать ее тип. Можно просто положить, что тип переменной совпадает с типом инициализатора. Рассмотрим следующее:

```
int a1 = 123;
char a2 = 123;
auto a3 = 123;          // тип a3 есть "int"
```

Тип целого литерала **123** есть **int**, так что **a3** также получает тип **int**. Ключевое слово **auto** просто стоит на месте типа переменной.

Применение **auto** вместо точной спецификации типа **int** мало что дает в простом случае инициализатора вроде **123**. Этот способ более полезен тем более, чем сложнее запись типа или сложнее выяснить этот тип. Например:

```
template<class T> void f1(vector<T>& arg)
{
    for (vector<T>::iterator p = arg.begin(); p!=arg.end(); ++p)
        *p = 7;

    for (auto p = arg.begin(); p!=arg.end(); ++p)
        *p = 7;
}
```

Применять **auto** в операторе цикла удобно и такую запись читать легче. К тому же, в таком случае код менее восприимчив к изменениям. Например, если я вдруг изменю **arg** так, чтобы он стал **list**, то цикл с **auto** останется корректным, в то время как первый оператор цикла потребует внесения исправлений.

В широких областях видимости (в отличие от узких областей видимости оператора) явное указание типа позволяет быстрее локализовать ошибки. Если же применять **auto**, то это может затянуть поиск ошибок. Например:

```
void f(double d)
{
    constexpr auto max = d+7;
    int a[max];              // ошибка: граница массива не целое
    // ...
}
```

Если применение **auto** приводит к сюрпризам, то здесь может помочь укорочение кода функций, что вообще является неплохой идеей (§12.1).

Можно «декорировать» выводимый тип спецификаторами и модификаторами (§6.3.1), такими как **const** и **&** (ссылка; §7.7). Например:

```
void f(vector<int>& v)
{
    for (const auto& x: v) {// x есть const int&
        // ...
    }
}
```

Здесь **auto** определяется типом элементов контейнера **v**, то есть типом **int**.

Заметьте, что тип выражения никогда не является ссылкой, ибо ссылки неявно разыменовываются в выражениях (§7.7). Пример:

```
void g(int& v)
{
    auto x = v;    // x есть int (а не int&)
    auto& y = v;   // y есть int&
}
```

6.3.6.2 auto и {}-списки

Когда мы явным образом указываем тип инициализируемого объекта, мы рассматриваем два типа: тип объекта и тип инициализатора. Например:

```
char v1 = 12345;    // 12345 есть int
int v2 = 'c';       // 'c' есть char
T v3 = f();
```

Применение {}-инициализаторов минимизирует риск нечаянных приведений типа:

```
char v1 {12345};    // ошибка: сужение
int v2 {'c'};       // отлично: неявная конверсия char->int
T v3 {f()};         // работает если и только если f() может неявно конвертироваться в T
```

Когда же мы используем **auto**, то у нас фигурирует единственный тип, тип инициализатора, так что можно не опасаясь использовать синтаксис со знаком **=**:

```
auto v1 = 12345;    // v1 есть int
auto v2 = 'c';      // v2 есть char
auto v3 = f();      // v3 некоторого подходящего типа
```

Фактически, использование = с **auto** обладает и преимуществом, ибо синтаксис с {}-списком способен удивить кого-нибудь:

```
auto v1 {12345};    // v1 есть список из int
auto v2 {'c'};      // v2 есть список из char
auto v3 {f()};      // v3 есть список элементов подходящего типа
```

Это логично. Рассмотрим следующее:

```
auto x0 {};         // ошибка: невозможно вывести тип
auto x1 {1};        // список из int с одним элементом
auto x2 {1,2};      // список из int с двумя элементами
auto x3 {1,2,3};    // список из int с тремя элементами
```

Тип однородного списка элементов типа **T** полагается типом **initializer_list<T>** (§3.2.1.3, §11.3.3). В частности, тип переменной **x1** выводится вовсе *не* как тип **int**. Если бы это было так, то какой тип имели бы **x2** и **x3**?

Как следствие, я рекомендую использовать = вместо {} для объектов с **auto**, если только мы не имеем в виду именно список.

6.3.6.3 Спецификатор decltype()

Можно использовать **auto**, если есть подходящий инициализатор. Но иногда мы можем захотеть получить выводимый тип без инициализации переменной. Тогда следует применить спецификатор типа в объявлении: **decltype(expr)** объявляет тип выражения **expr**. Такой способ чаще всего используется в обобщенном программировании. Рассмотрим код функции, в которой складываются две матрицы с потенциально разными типами элементов. Каков будет тип результата сложения? Конечно матрица, но каков тип ее элементов? Очевидный ответ таков: тип элементов суммарной матриц есть тип суммы элементов. Это объявляется так:

```
template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) -> Matrix<decltype(T{}+U{})>;
```

Я использую суффиксный тип результата возврата (§12.1), чтобы иметь возможность выразить тип возврата в терминах аргументов: **Matrix<decltype(T{}+U{})>**. То есть, результат есть **Matrix** с типом элементов, который совпадает с типом суммы элементов аргументов-матриц: **T{}+U{}**.

В определении я снова должен использовать **decltype()**, чтобы выразить тип элементов у **Matrix**:

```
template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) -> Matrix<decltype(T{}+U{})>
{
    Matrix<decltype(T{}+U{})> res;
    for (int i=0; i!=a.rows(); ++i)
        for (int j=0; j!=a.cols(); ++j)
            res(i, j) += a(i, j) + b(i, j);
    return res;
}
```

6.4. Объекты и значения

Мы можем создавать и использовать объекты, которые не имеют имени (например, при помощи **new**), и еще мы можем присваивать значения странно выглядящим выражениям (например, ***p[a+10]=7**). Поэтому нам нужно как-то называть это «что-то в памяти». По

сути это и есть *объект* как непрерывная область памяти; терминологически выражение, которое ссылается на объект, называется *lvalue* (нечто, располагающееся в левой части операции присваивания). Однако, не каждое lvalue может использоваться в левой части операции присваивания, так как некоторые lvalue могут ссылаться на константы (§7.7). В точной терминологии lvalue, которые могут использоваться в левой части операции присваивания, называются *модифицируемыми lvalue*. Данное нами только что столь низкоуровневое понятие объекта не следует путать с объектами классов и объектами полиморфных типов (§3.2.2, §20.3.2).

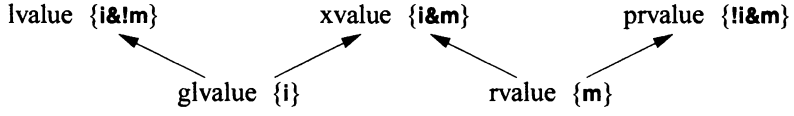
6.4. lvalue и rvalue

В дополнение к понятию lvalue имеется еще и понятие *rvalue*. Грубо говоря, rvalue это то, что не является lvalue (вроде возврата функции).

С технической точки зрения, требуется более детально рассмотреть понятия lvalue и rvalue. Два свойства объектов имеют важное значение при попытке адресовать объекты, копировать их или перемещать:

- *Идентифицируемость*: у программы есть имя объекта, указатель или ссылка на него, так что можно ответить на вопрос, не совпадают ли между собой два рассматриваемых объекта, изменялось ли значение объекта, и так далее.
- *Перемещаемость*: объект можно переместить (можно переместить значение объекта в другое место в памяти, оставив его в штатном состоянии с неопределенным значением — вместо копирования; §17.5).

Так выходит, что три из возможных четырех комбинаций перечисленных свойств объекта нужны для точного описания правил языка C++. Используя букву **m** для «перемещаемости — moveable» и букву **i** для «идентифицируемости — identity», мы можем графически представить классификацию выражений:



Таким образом, классическое lvalue это то, что имеет идентифицируемость и не имеет перемещаемости (ибо мы всегда имеем его значение), а классическое rvalue — это то, что может быть перемещено. Остальные альтернативы — *prvalue* («pure rvalue» — «чистое rvalue»), *glvalue* («generalized lvalue» — «обобщенное lvalue») и *xvalue* (здесь x для экстраординарного или экспертного). Например:

```
void f(vector<string>& vs)
{
    vector<string>& v2 = std::move(vs);     // перемещаем vs в v2
    // ...
}
```

Здесь **std::move(vs)** — это xvalue: имеет четкую идентификацию (мы всегда можем сослаться с помощью **vs**) и мы имеем явное разрешение на его перемещение при помощи вызова **std::move()** (§3.3.2, §35.5.1).

В практическом программировании (не в теории языков программирования) вполне достаточно мыслить в категориях lvalue и rvalue (любое выражение либо то, либо другое).

6.4.2. Время жизни объектов

Время жизни объекта начинается в момент, когда отработал его конструктор, и заканчивается тогда, когда за дело берется его деструктор. Объекты без объявляемых конструкторов, такие как **int**, могут представляться как имеющие умолчательные конструкторы и деструкторы, которые ничего не делают.

Можно классифицировать объекты в контексте их времен жизни:

- *Автоматические:* Если только программист не продекларирует иного (§12.1.8, §16.2.12), объявленные в функции объекты создаются в момент их определения и уничтожаются по выходу из области видимости. Часто такие объекты называют *автоматическими объектами*. В типичной реализации эти объекты создаются в стеке; каждый отдельный вызов функции получает свой фрейм (кадр) в стеке, содержащий автоматические объекты.
- *Статические*: Объекты, объявленные в глобальной области видимости или в пространстве имен (§6.3.4), а также объявленные с ключевым словом **static** в функции (§12.1.8) или классе (§16.2.12) создаются и инициализируются один раз и «живут» до конца работы программы (§15.4.3). Такие объекты называют *статическими объектами*. Все время, пока программа работает, их адреса остаются неизменными. В многопоточных программах статические объекты могут породить серьезные проблемы из-за того, что они доступны всем потокам и в типичном случае нуждаются в блокировках во избежание «гонки за данными» (§5.3.1, §42.3).
- *В свободной памяти*: Применяя **new** и **delete**, мы создаем объекты, время жизни которых мы сами контролируем напрямую (§11.2).
- *Временные:* предназначены для хранения промежуточных результатов вычислений или для хранения ссылки на константу. Время их жизни определяется способом их использования. В типичном случае временные объекты являются автоматическими.
- *Локальные для потока*: это объекты, объявленные как **thread_local** (§42.2.8): такие объекты создаются при создании их потока и вместе с потоком уничтожаются.

Термины *локальные* и *статические* также традиционно квалифицируются и как *классы хранения*.

Время жизни элементов массивов и нестатических членов классов определяется временем жизни объектов, частями которых они являются.

6.5. Синонимы типов

Иногда для имени типа требуется синоним. Причинами могут послужить следующие соображения:

- Оригинальное имя слишком длинное или малопривлекательное (в глазах какого-нибудь программиста).
- Технологии программирования могут потребовать, чтобы разные типы в выделенном контексте имели одно и то же имя.
- Требуется заданное имя для упрощения поддержки программы.

Например:

```
using Pchar = char*;          // указатель на символ
using PF = int(*)(double);    // указатель на функцию с аргументом double и возвратом int
```

Аналогичные по сути типы могут определять одно и то же имя в качестве синонима в своих членах:

```
template<class T>
class vector {
    using value_type = T;    // у всех контейнеров есть value_type
    // ...
};

template<class T>
class list {
    using value_type = T;    // у всех контейнеров есть value_type
    // ...
};
```

Хорошо это или плохо, но синонимы типов есть просто синонимы для существующих типов, а не новые типы. Пример:

```
Pchar p1 = nullptr;      // p1 есть char*
char* p3 = p1;           // отлично
```

Если требуются отдельные типы с идентичной по сути семантикой или идентичным представлением, то следует обратиться к перечислениям (§8.4) или классам (Глава 16).

Более старый вариант синтаксиса с ключевым словом typedef может по-прежнему использоваться во многих контекстах:

```
typedef int int32_t;          // эквивалентно "using int32_t = int;"
typedef short int16_t;        // эквивалентно "using int16_t = short;"
typedef void(*PtoF)(int);     // эквивалентно "using PtoF = void(*)(int);"
```

Синонимы часто используются тогда, когда важно изолировать код от деталей конкретной машины. Например, имя **int32_t** четко указывает на наше намерение иметь именно 32-битное целое. Применив **int32_t** вместо простого **int**, мы можем портировать наш код на машину, где **sizeof(int)==2**, переопределив **int32_t** для использования более длинных целых:

```
using int32_t = long;
```

Суффикс _t традиционно применяется для синонимов типов. Синонимы **int16_t**, **int32_t** и другие определяются в файле **<stdint>** (§43.7). Отметим, что именование типа согласно его представлению, а не согласно цели его использование, не является хорошей идеей (§6.3.3).

Ключевое слово **using** может также применяться для синонима шаблона (§23.6). Пример:

```
template<typename T>
    using Vector = std::vector<T, My_allocator<T>>;
```

К синонимам типа нельзя применять спецификаторы типа, вроде **unsigned**. Примеры:

```
using Char = char;
using Uchar = unsigned Char;     // ошибка
using Uchar = unsigned char;     // OK
```

6.6. Советы

[1] Последним арбитром в вопросах определений, связанных с языком, является стандарт ISO C++; §6.1.

[2] Избегайте неопределенного поведения кода; §6.1.

[3] Максимально изолируйте код, зависящий от реализации; §6.1.

[4] Не опирайтесь на предположения о числовых величинах, соответствующих символам; §6.2.3.2, §10.5.2.1.

[5] Запомните, что числовое значение, запись которого начинается с **0**, является восьмеричным; §6.2.4.1.
[6] Избегайте «магических констант»; §6.2.4.1.
[7] Не опирайтесь на предположения о размерах целых; §6.2.8.
[8] Не опирайтесь на предположения о диапазоне и точности типов с плавающей запятой; §6.2.8.
[9] Предпочитайте простой тип **char** типам **signed char** и **unsigned char**; §6.2.3.1.
[10] Имейте представление о преобразовании знаковых типов в беззнаковые и наоборот; §6.2.3.1.
[11] Объявляйте одно имя в рамках одного оператора объявления; §6.3.2.
[12] Старайтесь обычные и локальные имена делать короткими, а необычные и глобальные — длинными; §6.3.3.
[13] Избегайте похожих (трудноразличимых) имен; §6.3.3.
[14] Именуйте объекты так, чтобы имя отражало смысл объектов, а не их тип; §6.3.3.
[15] Поддерживайте согласованный стиль именования; §6.3.3.
[16] Избегайте имен, составленных из символов в верхнем регистре; §6.3.3.
[17] Старайтесь формировать небольшие области видимости; §6.3.4.
[18] Не используйте одно и то же имя в некоторой области видимости и в объемлющей области видимости; §6.3.4.
[19] Предпочитайте {}-инициализаторы в объявлениях с поименованным типом; §6.3.5.
[20] Предпочитайте инициализацию со знаком равно в объявлениях, в которых используется **auto**; §6.3.5.
[21] Избегайте неинициализированных переменных; §6.3.5.1.
[22] Применяйте синонимы со значащими именами для встроенных типов в случаях, когда встроенный тип, применяющийся для представления значения, может изменяться; §6.5.
[23] Синонимы типов просто вводят новые имена для типов; для новых типов применяйте перечисления и классы; §6.5.

7

Указатели, массивы и ссылки

От тонкости до нелепости столь близко,
что трудно классифицировать их порознь.
— Томас Пэн

- Введение
- Указатели
  void*; **nullptr**
- Массивы
  Инициализаторы массивов; Строковые литералы; «Сырые» символьные строки; Широкие символы
- Указатели на массивы
  Навигация по массивам; Многомерные массивы; Передача массивов
- Указатели и модификатор **const**
- Указатели и владение объектами
- Ссылки
  lvalue-ссылки; rvalue-ссылки; Ссылки на ссылки; Указатели и ссылки
- Советы

7.1. Введение

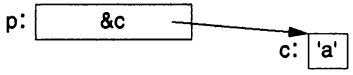
Настоящий раздел книги посвящен механизмам языка C++, адресующим блоки памяти. Ясно, что мы можем ссылаться на объекты по их имени, но в C++ любой объект, расположенный в памяти, характеризуется и вполне определенным адресом, так что им можно манипулировать и посредством адреса, если к этому присовокупить еще и знание о типе объекта. Средствами языка C++, обеспечивающими возможность хранения адресов и их дальнейшего использования, являются указатели и сылки.

7.2. Указатели

Для типа **T** запись **T*** означает тип «указатель на **T**»; переменная типа **T*** может содержать адрес объекта типа **T**. Например:

```
char c = 'a';
char* p = &c;  // p содержит адрес c; & это операция взятия адреса
```

или в графической форме:



Самой фундаментальной операцией с указателем является *операция разыменования* (*dereferencing*), то есть обращение к объекту, на который указывает указатель. Об этой операции также говорят как о косвенном обращении к объекту (*indirection*). Операция разыменования (префиксная) обозначается с помощью *. Пример:

```
char c = 'a';
char* p = &c;  // p содержит адрес c; & это операция взятия адреса
char c2 = *p;  // c2 == 'a'; * это операция разыменования
```

Объект, на который указывает **p**, есть **c**, который содержит символ **'a'**, так что после присваивания переменой **c2** значения ***p**, эта переменная получает значение **'a'**.

Допускаются некоторые арифметические операции над указателями на элементы массивов (§7.4).

Реализация указателей направлена на прямое использование механизмов адресации применительно к той машине, на которой работает программа. Большинство машин могут адресовать один байт. А те, которые не могут, опираются на аппаратные особенности, позволяющие извлекать байты из слов. С другой стороны, адресовать объекты размером в один бит могут лишь очень редкие машины. В итоге, мельчайшим объектом, который можно независимо располагать в памяти и адресовать с помощью встроенных указателей, является объект типа **char**. Заметьте, что **bool** занимает по крайней мере столько же памяти, что и **char** (§6.2.8). Чтобы компактно хранить в памяти более мелкие величины, используйте побитовые логические операции (§11.1.1), битовые поля в структурах (§8.2.7) или **bitset** (§34.2.2).

Символ * в качестве суффикса используется в имени типа, символизируя тип «указатель на». К сожалению, указатели на массивы или функции требуют несколько более сложного синтаксиса:

```
int* pi;            // указатель на int
char** ppc;         // указатель на указатель на char
int* ap[15];        // массив из 15 указателей на int
int (*fp)(char*);   // указатель на функцию с аргументом char* и возвращающую int
int* f(char*);      // функция с аргументом char*, возвращающая указатель на int
```

Смотри §6.3.3 по поводу разъяснений синтаксиса объевлений, а также § iso.A насчет полного описания грамматики.

Указатели на функции весьма полезны; они рассматриваются в §12.5. Указатели на члены классов рассматриваются в §20.6.

7.2.1. void*

В низкоуровневом коде часто возникают ситуации, когда нам нужно сохранить и передать куда-то адрес объекта в памяти, тип которого неизвестен. Для этого используется указатели типа **void***. Этот тип можно вербально трактовать как «указатель на объект неизвестного типа».

Переменной типа **void*** можно присваивать значения указателей на любой тип объекта, кроме указателей на функции (§12.5) или указателей на члены классов (§20.6). Кроме того, значение одной переменной типа **void*** можно присваивать другой переменной того же типа; эти переменные можно сравнивать друг с другом на равенство/неравенство и их можно явно конвертировать в иные типы. Все иные операции были бы опасными, ибо компилятор не знает тип объекта, адресуемого таким указателем. В результате попытка использовать эти операции приводит к ошибке компиляции. Чтобы использовать **void***

для доступа к объекту, его нужно сначала явно конвертировать в указатель иного типа. Например:

```
void f(int* pi)
{
    void* pv = pi; // ok: неявная конверсия из int* в void*
    *pv;           // ошибка: невозможно разыменовать void*
    ++pv;          // ошибка: нельзя инкрементировать void* (неизвестен размер указуемого объекта)

    int* pi2 = static_cast<int*>(pv);          // явная конверсия в int*

    double* pd1 = pv;                          // ошибка
    double* pd2 = pi;                          // ошибка
    double* pd3 = static_cast<double*>(pv);    // ненадежно (§11.5.2)
}
```

В общем случае, достаточно опасно использовать указатели, приведенные к типу, отличному от типа объекта, на который указывает указатель. Например, на машине может быть выравнивание объектов типа **double** по 8-байтовым границам. В этом случае может возникнуть странное поведение кода, если **pi** указывает на **int**, который не отвечает этому типу выравнивания. Поскольку явное приведение типов указателей может оказаться опасным, операция **static_cast** (§11.5.2) специально сконструирована так, чтобы выглядеть «безобразной» и ее в итоге можно было бы легко обнаруживать в коде.

Основной областью применения указателей типа **void*** является их передача в функции, которым запрещено делать предположения о типе адресуемых ими объектов, а также для возврата безтиповых объектов из функций. Использование же объектов допускается лишь после явного приведения типа указателя.

В типичном случае, функции, использующие указатели типа **void***, должны сосредотачиваться в самых низкоуровневых частях кода программы, где часто приходится манипулировать аппаратными ресурсами. Пример:

```
void* my_alloc(size_t n);      // аллоцировать n байт из моей специальной кучи
```

Наличие указателей типа **void*** в более высокоуровневых частях кода должно вызывать подозрение, и оно скорее всего является следствием ошибок в дизайне программы. В случае использования **void*** в целях оптимизации это следует скрывать за безопасными интерфейсами (§27.3.1).

Значения указателей на функции (§12.5) и указателей на члены классов (§20.6) нельзя присвоить указателям типа **void***.

7.2.2. nullptr

Литерал **nullptr** символизирует «нулевой указатель», то есть указатель, не указывающий ни на какой объект. Его можно присваивать любому указателю, но не другим встроенным типам:

```
int* pi = nullptr;
double* pd = nullptr;
int i = nullptr;          // ошибка: i не указатель
```

Существует всего один **nullptr**, а не разные «нулевые указатели» для разных типов указателей.

До того, как в язык был введен **nullptr**, вместо него использовался просто нуль (**0**). Например:

```
int* x = 0;      // x получает значение nullptr
```

Не существует объектов, располагаемых в памяти по адресу 0, так что нуль (все биты нулевые) является наиболее общим представлением для **nullptr**. Нуль (0) имеет тип **int**. Стандартные преобразования (§10.5.2.3) допускают использование **0** в качестве константы для указателей и указателей на члены классов.

На практике популярно применение макроконстанты **NULL** для представления нулевых указателей. Например:

```
int* p = NULL;      // используем макро NULL
```

Однако, в разных реализациях имеет место разное определение для **NULL**; к примеру, это может быть **0** или **0L**. В языке C макроконстанта **NULL** в типичном случае есть **(void*)0**, что неприемлемо в языке C++ (§7.2.1):

```
int* p = NULL;      // ошибка: нельзя присвоить void* указателю типа int*
```

Применение **nullptr** делает код более читаемым и помогает избегать недоразумений, когда функция перегружается так, чтобы принимать либо целое, либо указатель (§12.3.1).

7.3. Массивы

Для типа **T** запись **T[size]** означает тип «массив из **size** элементов типа **T**». Элементы индексируются значениями индекса от **0** до **size-1**. Например:

```
float v[3];         // массив трех вещественных значений: v[0], v[1], v[2]
char* a[32];        // массив из 32 указателей на char: a[0] .. a[31]
```

Доступ к массиву возможен с помощью операции индексации [], или же посредством указателя (используя операцию * или операцию []; §7.4). Например:

```
void f()
{
    int aa[10];
    aa[6] = 9;          // присвоение седьмому элементу массива aa
    int x = aa[99];     // неопределенное поведение
}
```

Доступ за пределы массива имеет неопределенные последствия, чаще всего катастрофические. Никакой проверки выхода индекса за допустимые границы во время выполнения программы не гарантируется и чаще всего не имеет места.

Количество элементов в массиве должно задаваться константным выражением (§10.4). Если вам требуется переменное число элементов, применяйте тогда контейнер **vector** (§4.4.1, §31.4). Пример:

```
void f(int n)
{
    int v1[n];          // ошибка: размер массива не является константным выражением
    vector<int> v2(n);  // OK: вектор из n элементов типа int
}
```

Многомерные массивы реализуются в виде массива массивов (§7.4.2).

Массив является фундаментальным способом отражения последовательности объектов в памяти. Если все, что вам нужно, это простая последовательность объектов известного типа в памяти фиксированного размера, то массив является идеальным решением. В остальных случаях массив порождает проблемы.

Массив можно располагать статически, в стеке и в свободной памяти (§6.4.2). Например:

```
int a1[10];      // 10 int в статическом хранилище

void f()
{
    int a2 [20];              // 20 int в стеке
    int*p = new int[40];      // 40 int в свободной памяти
    // ...
}
```

Встроенный массив языка C++ является низкоуровневым средством, которое следует использовать внутри реализации более высокоуровневых структур данных с более предсказуемым и менее опасным поведением, вроде **vector** или **array** из стандартной библиотеки. Для массивов не допускается присваивание, а имя массива неявно конвертируется в указатель на первый элемент этого массива (§7.4). Из-за этого следует избегать применения массивов в интерфейсах (например в аргументах функций; §7.4.3, §12.2.2), поскольку неявное приведение к указателю служит источником традиционных ошибок в С-коде и в C++-коде, имитирующем стиль языка С. Если вы создаете массив в свободной памяти, не забудьте вызвать **delete[]** ровно один раз сразу после того, как исчезнет необходимость в массиве (§11.2.2). Легче и удобнее всего контролировать время жизни массива в свободной памяти посредством дескриптора ресурсов (вроде **string** (§19.3, §36.3), **vector** (§13.6, §34.2) или **unique_ptr** (§34.3.1)). Если вы располагаете массив статически или в стеке, никогда не вызывайте **delete[]**.

Одним из самых широко применяемых массивов является массив элеметов типа **char** с терминальным нулем. Таким способом в языке С хранятся символьные строки, так что указанные массивы часто называют *строками в С-стиле* (*C-style strings*). Строковые литералы в языке C++ соответствуют такому же соглашению и многие библиотечные функции (**strcpy()** или **strcmp()**; §43.4) принимают такие строки. Часто полагают, что переменные типа **char*** или **const char*** указывают на последовательности символов с терминальным нулем.

7.3.1. Инициализаторы массивов

Массив можно инициализировать списком значений. Пример:

```
int v1[] = {1, 2, 3, 4};
char v2[] = {'a', 'b', 'c', 0};
```

Когда массив объявляется без указания размера, но со списком инициализации, размер массива вычисляется посредством подсчета количества элементов списка. Следовательно, **v1** и **v2** имеют тип **int[4]** и **char[4]**, соответственно. Если размер явно указан, то в случае превышения количества элементов в списке инициализации над указанным размером возникает ошибка компиляции. Пример:

```
char v3[2] = {'a', 'b', 0}; // ошибка: слишком много инициализаторов
char v4[3] = {'a', 'b', 0}; // OK
```

Если же в списке инициализации недостает полагающихся элементов, то используется **0** для элементов, оставшихся без явно указанного значения. Пример:

```
int v5[8] = {1, 2, 3, 4};
```

Это эквивалентно

```
int v5[] = {1, 2, 3, 4, 0, 0, 0, 0};
```

Не существует встроенных операций для копирования массивов. Вы не можете проинициализировать массив другим массивом (даже того же самого типа):

```
int v6[8] = v5;      // ошибка: нет копирования массивов (попытка присовения int* массиву)
v6 = v5;             // ошибка: нет присваивания массивов
```

Естественно, что вы не можете и передавать массивы по значению. Смотри также §7.4.

Если вы нуждаетесь в присваивании для коллекции объектов, используйте **vector** (§4.4.1, §13.6, §34.2), **array** (§8.2.4) или **valarray** (§40.5).

Массив символов можно удобным образом проинициализировать строковым литералом (§7.3.2).

7.3.2. Строковые литералы

Строковый литерал это последовательность символов, заключенная в двойные кавычки:

```
"this is a string"
```

В строковом литерале содержится еще один символ, неочевидный из внешнего вида литерала, который на самом деле всегда оканчивается нулевым символом **'\0'**, числовой код которого есть **0**. Пример:

```
sizeof("Bohr")==5
```

Тип символьного литерала есть «массив соответствующего числа элементов типа **const char**». Так, **"Bohr"** имеет тип **const char[5]**.

В языке C и в ранних версиях C++ вы могли строковым литералом инициализировать неконстантную переменную типа **char***:

```
void f()
{
    char* p = "Plato";   // ошибка, приемлемо в коде до C++11
    p[4] = 'e';          // ошибка: присвоение константе
}
```

Мы полагаем, что было бы крайне неосмотрительно сохранять допустимость такого присваивания. Оно порождает тонкие ошибки, так что вряд ли стоит сожалеть о такой утрате. То, что теперь мы требуем неизменности строковых литералов не только согласуется с элементарной логикой, но и позволяет компиляторам выполнить весьма существенную оптимизацию целевого кода.

В случае, если нам требуется изменять строковое значение, мы должны поместить символы в неконстантный массив:

```
void f()
{
    char p[] = "Zeno"; // p массив из 5 char
    p[0] = 'R';        // OK
}
```

Строковые литералы располагают в статической области памяти, и поэтому их возврат из функции безопасен. Например:

```
const char* error_message(int i)
{
    // ...
    return "range error";
}
```

Память, содержащая **"range error"** никуда не денется после вызова **error_message()**.

Вопрос о том, как именно хранятся два идентичных по значению строковых литерала (как два массива или как один), отдается на откуп реализациям (§6.1). Например:

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";

void g()
{
    if (p == q) cout << "one!\n"; // результат зависит от реализации
    // ...
}
```

Обратите внимание на то, что операция == сравнивает адреса, хранящиеся в указателях, а не значения, на которые они указывают.

Пустая строка записывается как "" и имеет тип **const char[1]**. Единственным символом пустой строки является терминальный нуль.

В строках можно применять и обозначения символов с помощью обратной наклонной черты для отражения непечатных символов (§6.2.3.2). Это позволяет включить в содержимое строки двойные кавычки и обратную наклонную черту. Самым популярным примером здесь служит включение в содержимое строки символа перехода на новую физическую строку (символ перевода строки) — **'\n'**. Пример:

```
cout<<"beep at end of message\a\n";
```

Специальный символ **'\a'** в ASCII-кодировке имеет стандартное имя **BEL**; он предназначен для генерации короткого звукового сигнала.

Не допускается внедрение реальных (физических) разрывов строк внутри строковых литералов:

```
"this is not a string
but a syntax error"
```

Длинные строковые литералы можно разрывать по пробельным символам следующим образом:

```
char alpha[] =  "abcdefghijklmnopqrstuvwxyz"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Компилятор автоматически сконкатенирует смежные строки, так что **alpha** примет такое значение:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Можно внедрить в строку и нулевой символ, но большинство программ и не догадаются о существовании символов после такого нуля. Например, строка **"Jens\000Munk"** будет трактоваться как **"Jens"** функциями стандартной библиотеки, такими как **strcpy()** или **strlen()**; см. §43.4.

7.3.2.1. «Сырые» символьные строки

Чтобы внедрять символы \ и " в содержимое строковых литералов, мы должны предварять их обратной наклонной чертой. Во многих случаях в этом нет ничего сложного. Однако, в тех случаях, когда в строковых литералах должно присутствовать большое количество двойных кавычек и символов \, представленное решение становится обременительным. Кроме того, в регулярных выражениях обратная наклонная черта используется еще и для обозначения класса символов (§37.1.1). Причем это соглашение имеет место

для многих языков программирования, так что мы не можем отказаться от него. Поэтому, когда вы записываете регулярное выражение с целью применять его в библиотеке **regex** (Глава 37), в этот момент классическая роль обратной наклонной черты в строковых литералах становится источником многочисленных ошибок. Вот наглядный пример сказанного: надо написать образец, представляющий два слова, отделенных друг от друга наклонной чертой:

```
string s = "\\w\\\\w";      // надеюсь не ошибся
```

Чтобы предотвратить подобного рода неприятности и избежать потенциальных ошибок в C++ введено понятие сырого строкового литерала. *Сырой строковый литерал* (*raw string literal*) — это строковый литерал, в котором обратная наклонная черта есть просто обратная наклонная черта (а двойные кавычки есть просто двойные кавычки), так что наш предыдущий пример превращается в нечто более удобочитаемое:

```
string s = R"(\w\\w)";      // уверен что не ошибся
```

Сырые строковые литералы применяют обозначение **R" (ccc)"** для последовательности символов **ccc**. Начальный символ **R** и служит здесь для того, чтобы указать на сырой строковый литерал. А круглые скобки порождают возможность прямолинейного использования двойных кавычек. Пример:

```
R"("quoted string")"      // содержание строки есть "quoted string"
```

При этом возникает вопрос о том, как ввести в сырой строковый литерал последовательность символов)", что в принципе является не часто встречающейся ситуацией. В таких случаях следует предварить открывающую круглую скобку некоторой заданной последовательностью символов, которую в конце следует повторить перед закрывающей круглой скобкой:

```
R"***("quoted string containing the usual terminator ("))")***"
    // "quoted string containing the usual terminator ("))"
```

Еще раз подчеркнем, что последовательность символов после) должна совпадать с таковой перед (.

Конечно, если вы не имеете дело с регулярными выражениями, то сырые строковые литералы могут показаться вам избыточным средством. Но регулярные выражение встречаются в реальной практике весьма часто и они очень полезны, так что «жертвы», связанные с изучением еще одного нового средства языка C++, не напрасны. Вот пример из реальной жизни:

```
"('(?:[^\\\\']|\\\\.)*'|\"  (?:[^\\\\\"]|\\\\.)*\")|"      // Пяти обратных наклонных черт достаточно?
```

С такими примерами даже эксперты испытают затруднения, не то что обычные смертные, которым сырые строковые литералы сослужат тут неплохую службу.

В отличие от несырых строковых литералов их «сырые собратья» допускают наличие физических разрывов на разные строки:

```
string counts {R"(1
22
333)"};
```

Это эквивалентно

```
string x {"1\n22\n333"};
```

7.3.2.2. Широкие (многобайтовые) символы

Строки с префиксом **L**, такие как например **L"angst"**, являются строками с широкими (многобайтовыми) символами (§6.2.3). Их тип есть **const wchar_t[]**. Аналогично, строки с префиксом **LR**, такие как **LR"(angst)"** — это сырые строки (§7.3.2.1) с широкими символами типа **const wchar_t[]**. Такие строки оканчиваются символом **L'\0'**.

Существует шесть разновидностей символьных литералов, поддерживающих кодировки UNICODE (*Unicode literals*). Это кажется избыточным, но при этом лишь три кодировки UNICODE являются основными: UTF-8, UTF-16 и UTF-32. Для каждого из этих вариантов существуют «сырые» и «обычные» строковые литералы. Все эти UTF-кодировки содержат все символы, так что предпочтение одной из них более всего зависит от конкретной системы, для которой ваш код предназначен. Основные интернет-приложения (браузеры и почтовые клиенты) поддерживает одну или все эти кодировки.

Кодировка UTF-8 является кодировкой с переменной длиной: основные символы требуют одного байта, менее распространенные укладываются в два байта, ну а совсем редкие — в три или четыре байта. Например, ASCII-символы требуют в кодировке UTF-8 один байт, как и в стандартной ASCII-кодировке (с теми же самыми числовыми кодами). Разные латинские алфавиты, кириллица, греческий алфавит и т.д. требуют двух байт на символ.

Строки в кодировке UTF-8 оканчиваются символом **'\0'**, UTF-16 строки — символом **u'\0'**, а UTF-32 — символом **U'\0'**.

Обычную английскую строку можно представить во множестве вариантов. Рассмотрим имя файла, используя символ \ в качестве разделителя:

```
"folder\\file"      // строка из символьного набора реализации
R"(folder\file)"    // сырая строка из символьного набора реализации
u8"folder\\file"    // UTF-8 строка
u8R"(folder\file)"  // UTF-8 сырая строка
u"folder\\file"     // UTF-16 строка
uR"(folder\file)"   // UTF-16 сырая строка
U"folder\\file"     // UTF-32 строка
UR"(folder\file)"   // UTF-32 сырая строка
```

Будучи напечатанными, эти строки выглядят одинаково. Но с точки зрения их внутреннего устройства, все они разные, за исключением UTF-8 и обычного ASCII-варианта.

Естественно, что Unicode-строки позволяют внедрять Unicode-символы. Например:

```
u8"The official vowels in Danish are: a, e, i, o, u, \u00E6, \u00F8, \u00E5 and y."
```

В напечатанном варианте получим

```
The official vowels in Danish are: a, e, i, o, u, æ, ø, å and y.
```

После символа **\u** здесь стоят соответствующие Unicode-коды (§ iso.2.14.3) [Unicode,1996]. Эти числовые коды внутри строковых литералов абсолютны, то есть не зависят от применяемых вариантов кодировок, но они имеют разные представления (битов в байтах). К примеру, **u'0430'** (кириллическая буква «а» в нижнем регистре) представима как двухбайтовое шестнадцатеричное число **D0B0** в UTF-8, как двухбайтовое шестнадцатеричное число **0403** в UTF-16 и как четырехбайтовое шестнадцатеричное число **00000403** в UTF-32. Эти шестнадцатеричные значения называются *универсальными именами символов* (*universal character names*).

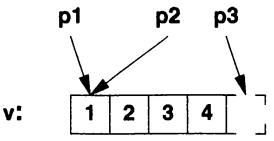
Последовательность всех этих **u** и **r** имеет значение: не допускаются префиксы **RU** и **Ur**.

7.4. Указатели на массивы

В языке C++ указатели и массивы тесно связаны. Имя массива можно использовать в качестве указателя на его первый элемент. Например:

```
int v[] = {1, 2, 3, 4};
int* p1 = v;          // указатель на начальный элемент (неявная конверсия)
int* p2 = &v[0];      // указатель на начальный элемент
int* p3 = v+4;        // указатель на элемент "за последним"
```

или графически:



Допускается указатель на элемент, следующий за последним в массиве, что очень важно для многих алгоритмов (§4.5, §33.1). Но так как такой указатель не смотрит ни на какой действительный элемент массива, то его нельзя использовать для чтения или записи. Попытки установить указатель на элемент, предшествующий первому элементу в массиве или на элемент, следующий за «следующим за последним элементом», имеют непредсказуемые последствия и их следует всячески избегать. Пример:

```
int* p4 = v-1; // перед началом, неопределено: не делайте так
int* p5 = v+7; // за концом, неопределено: не делайте так
```

Неявное преобразование имени массива в указатель на первый элемент широчайше используется в вызовах функций в С-стиле. Например:

```
extern "C" int strlen(const char*); // from <string.h>

void f()
{
    char v[] = "Annemarie";
    char* p = v;          // неявная конверсия char[] в char*
    strlen(p);
    strlen(v);            // неявная конверсия char[] в char*
    v = p;                // ошибка: невозможно присвоение массиву
}
```

Одно и то же значение передается в двух вызовах функции **strlen()** из стандартной библиотеки. Невозможно избежать неявного преобразования, то есть невозможно передать массив по значению (копированием массива **v**). Хорошо еще, что отсутствует неявное преобразование из указателя в массив.

Неявное преобразование имени массива в указатель означает, что размер массива теряется при вызове функции. В то же время, код функции так или иначе должен знать размер, чтобы работать с массивом разумным образом. Как и все другие стандартные функции языка С, **strlen()** полагается на терминальный нуль, сигнализирующий о конце строки; **strlen(p)** возвращает длину переданной строки (без учета терминального нуля). Все это довольно низкоуровневые детали. Такие типы стандартной библиотеки, как **vector** (§4.4.1, §13.6, §31.4), **array** (§8.2.4, §34.2.1) и **string** (§4.2) не страдают от такого обременения. Все эти типы предоставляют метод **size()** с этой целью, избавляя от необходимости то и дело подсчитывать число элементов.

7.4.1. Навигация по массивам

Эффективный и элегантный доступ к массивам (и аналогичным структурам) является ключевым для многих алгоритмов (§4.5, Глава 32). Доступ можно реализовать посредством указателя на массив плюс индекс элемента или же сразу указателем на элемент. Пример:

```
void fi(char v[])
{
    for (int i = 0; v[i]!=0; ++i)
        use(v[i]);
}

void fp(char v[])
{
    for (char* p = v; *p!=0; ++p)
        use(*p);
}
```

Префиксный знак * разыменовывает указатель, так что ***p** и есть элемент, на который «смотрит» **p**; операция **++** продвигает указатель так, что после этого он смотрит на следующий элемент массива.

Оба способа доступа практически идентичны в плане эффективности, и многие современные компиляторы генерируют при этом практически одинаковый код. Программисты осуществляют свой выбор иногда из логических, а иногда и из эстетических предпочтений.

Индексирование встроенных массивов определяется в терминах операций с указателями. Для любого встроенного массива **a** и целого значения **j** в пределах корректного диапазона для **a**, имеем:

```
a[j] == *(&a[0]+j) == *(a+j) == *(j+a) == j[a]
```

Обычно людей крайне удивляет такое вот тождество: **a[j]==j[a]**. Например, **2["Texas"] =="Texas" [2]=='x'**. Естественно, что в реальном практическом коде демонстрировать свою избыточную «умность» нет никакого резона. Все это слишком мелкие и низкоуровневые детали, и к тому же, такая перестановочность не имеет места в случае стандартных контейнеров вроде **array** и **vector**.

Результат применения операций **+**,**-**, **++** или **--** к указателям зависит от типа указуемого объекта. Когда арифметическая операция применяется к указателяю **p** типа **T***, то предполагается, что **p** указывает на элемент массива объектов типа **T**; **p+1** указывает на следующий элемент массива, а **p-1** указывает на предыдущий элемнт. Это означает, что значение **p+1** на **sizeof(T)** больше целочисленного значения переменной **p**. Например:

```
template<typename T>
int byte_diff(T* p, T* q)
{
    return reinterpret_cast<char*>(q)-reinterpret_cast<char*>(p);
}

void diff_test()
{
    int vi[10];
    short vs[10];
    cout << vi << ' ' << &vi[1] << ' ' << &vi[1]-&vi[0] << ' ' << byte_diff(&vi[0],&vi[1]) << '\n';
    cout << vs << ' ' << &vs[1] << ' ' << &vs[1]-&vs[0] << ' ' << byte_diff(&vs[0],&vs[1]) << '\n';
}
```

Вывод этого кода будет следующим:

```
0x7fffaef0 0x7fffaef4 1 4
0x7fffaedc 0x7fffaede 1 2
```

Значения указателей были выведены с использованием шестнадцатеричной нотации. Отсюда видно, что на моей машине **sizeof(short)** равен **2**, а **sizeof(int)** равен **4**.

Вычитание указателей допускается только для указателей, «глядящих» на элементы одного и того же массива. В этом случае **q-p** (где **p** и **q** — указатели) равно числу элементов в последовательности **[p:q)**. К указателю можно прибавить целое число или вычесть целое число; в обоих случаях в результате имеем значение указателя. Если новое значение указателя не указывает на элементы того же самого массива, на который указывает исходный указатель, или же на элемент, следующий за последним, то результат применения такого указателя не определен. Пример:

```
void f()
{
    int v1[10];
    int v2[10];
    int i1 = &v1[5]-&v1[3];    // i1 = 2
    int i2 = &v1[5]-&v2[3];    // результат неопределен

    int* p1 = v2+2;            // p1 = &v2[2]
    int* p2 = v2-2;            // *p2 неопределен
}
```

Обычно нет необходимости в слишком сложной арифметике указателей, и ее лучше всего избегать. Сложение указателей вообще не имеет смысла, и оно запрещено.

Массивы не описывают себя полностью, ибо размер массива не хранится где-то в связке с переменной-именем массива. Из этого следует, что при передаче массива, для которого отсутствует внятный терминальный элемент (как у строк в С-стиле), нам нужно как-то передавать и число элементов в массиве. Пример:

```
void fp(char v[], int size)
{
    for (int i=0; i!=size; ++i)
        use(v[i]);             // надеемся, что v имеет по крайней мере size элементов
    for (int x : v)
        use(x);                // ошибка: range-for не работает для указателей

    const int N = 7;
    char v2[N];
    for (int i=0; i!=N; ++i)
        use(v2[i]);
    for (int x : v2)
        use(x);                // range-for работает для массивов известного размера
}
```

Концепция встроенных массивов довольно низкоуровневая. Большинство преимуществ встроенных массивов (а также ряд их недостатков) можно получить от стандартного контейнера **array** (§8.2.4, §34.2.1). Некоторые реализации C++ предлагают опционально проверку диапазона для массивов. Но такая проверка может при этом сильно влиять на эффективность, так что даже в случае наличия такой проверки ее стоит использовать лишь на этапе отладки, и исключить из окончательно отлаженного кода. Если не применять проверки диапазона при доступе к индивидуальным элементам массива, то следует придерживаться общей согласованной политики доступа, которая обеспечит возможность досту-

па лишь в правильном диапазона индексов. Это удачно получается при использовании интерфейсов более высокоуровневых контейнерных типов, таких как **vector**, где намного труднее отклониться от корректного диапазона элементов.

7.4.2. Многомерные массивы

Многомерные массивы представляются в виде массива массивов; массив 3-на-5 объявляется следующим образом:

```
int ma[3][5];    // 3 массива по 5 int каждый
```

Теперь инициализируем массив **ma**:

```
void init_ma()
{
    for (int i = 0; i!=3; i++)
        for (int j = 0; j!=5; j++)
            ma[i][j] = 10*i+j;
}
```

или графически:

ma:	00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

Массив **ma** есть просто массив из 15 целых, к которому мы обращаемся так, как будто бы это 3 массива по 5 целых. В частности, нет никакого специального объекта в памяти, представляющего многомерный массив **ma** — хранятся лишь элементы. Размеры **3** и **5** представлены лишь в исходном коде. Нам самим предстоит помнить о них и предоставлять всюду, где это требуется. Например, можно распечатать массив **ma** следующим образом:

```
void print_ma()
{
    for (int i = 0; i!=3; i++) {
        for (int j = 0; j!=5; j++)
            cout << ma[i][j] << '\t';
        cout << '\n';
    }
}
```

Распространенную нотацию с применением запятой, обычную для некоторых языков программирования, невозможно использовать в C++, ибо в нем этот знак зарезервирован за операцией следования (§10.3.2). К нашей радости, большинство ошибок с многомерными массивами отлавливаются компилятором. Пример:

```
int bad[3,5];              // ошибка: запятая запрещена в константных выражениях
int good[3][5];            // 3 массива по 5 int каждый
int ouch = good[1,4]; // ошибка: int инициализир. через int* (good[1,4] значит good[4], что есть int*)
int nice = good[1][4];
```

7.4.3. Передача массивов

Массивы невозможно передать по значению. Вместо этого, массив передается как указатель на его первый элемент. Например:

```
void comp(double arg[10])          // arg is a double*
{
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}
```

```
void f()
{
    double a1[10];
    double a2[5];
    double a3[100];

    comp(a1);
    comp(a2);     // катастрофа!
    comp(a3);     // используются только 10 первых элементов
};
```

Этот код компилируется, но вызов **comp(a2)** вызовет запись вне корректного диапазона элементов массива **a2**. Массив не передается по значению, и любой, кто так думает, будет разочарован: запись в **arg[i]** это прямая запись в элементы аргумента функции **comp()**, а не в копию аргумента. Функцию можно эквивалентно переписать так:

```
void comp(double* arg)
{
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}
```

В такой форме все более очевидно. Будучи аргументом функции, первое измерение массива трактуется как указатель. А размер массива при этом просто игнорируется. Из этого следует, что если вам нужно передать в функцию последовательность элементов без потери информации о количестве элементов (размере), то следует отказаться от встроенного массива. Вы можете, например, поместить массив внутрь класса (в качестве члена, как это делается в случае **std::array**) или определить класс, функционирующий как дескриптор (как это делается в случае **std::string** или **std::vector**).

Настаивая на прямом использовании встроенных массивов, вы ничего существенно не выигрываете в производительности, теряя при этом в надежности кодирования (очень легко пропустить малозаметные, на первый взгляд, ошибки). Рассмотрим к примеру функцию, которая оперирует над двумерным массивом (матрицей). Если размеры известны во время компиляции, никаких проблем:

```
void print_m35(int m[3][5])
{
    for (int i = 0; i!=3; i++) {
        for (int j = 0; j!=5; j++)
            cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

Матрица, представимая многомерным массивом, передается как указатель (а не копируется; §7.4). Первый из указанных размеров не имеет отношения к позициям элементов; это просто количество элементов (здесь **3**) соответствующего типа (здесь **int[5]**). Для массива **ma**, представленного в коде примера, знание второго размера (это **5**) позволяет определить расположение **ma[i][5]** для любого **i**. А первый размер можно передать и вторым аргументом:

```
void print_mi5(int m[][5], int dim1)
{
    for (int i = 0; i!=dim1; i++) {
        for (int j = 0; j!=5; j++)
            cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

Передавая оба размера отдельными аргументами, получаем неработоспособное решение:

```
void print_mij(int m[][], int dim1, int dim2)  // иное поведение, чем многие думают
{
    for (int i = 0; i!=dim1; i++) {
        for (int j = 0; j!=dim2; j++)
            cout << m[i][j] << '\t';          // сюрприз!
        cout << '\n';
    }
}
```

По счастью, объявление аргумента **m[][]** запрещено, ибо без знания второго размера невозможно определить расположение элементов. Тем не менее, выражение **m[i][j]** корректно интерпретируется как ***(*(m+i)+j)**, хотя это и не то, что думал программист. Вот корректное решение:

```
void print_mij(int* m, int dim1, int dim2)
{
    for (int i = 0; i!=dim1; i++) {
        for (int j = 0; j!=dim2; j++)
            cout << m[i*dim2+j] << '\t';    // неясно
        cout << '\n';
    }
}
```

Выражение для доступа к элементам, использованное в функции **print_mij()**, эквивалентно тому, что генерирует компилятор в случае, когда ему известен последний размер.

Вызывая эту функцию, передаем ей матрицу как обычный указатель:

```
int test()
{
    int v[3][5] = {
        {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24}
    };
    print_m35(v);
    print_mi5(v,3);
    print_mij(&v[0][0],3,5);
}
```

Обратите внимание на **&v[0][0]**; **v[0]** также подойдет, так как это просто эквивалентно, а вот **v** будет ошибкой типа. Подобного рода низкоуровневый и неряшливый код лучше всего скрывать где-нибудь поглубже в коде проекта. Если уж нужно напрямую работать с многоуровневыми массивами, лучше всего инкапсулировать зависящий от этого код. Этим вы поможете программистам, которые впоследствии будут работать с вашим кодом. Если вы предоставите сообществу высокоуровневый код с более-менее конвенциальными операциями индексирования, то пользователям не придется беспокоиться о деталях расположения данных в памяти (§29.2.2, §40.5.2).

Стандартный тип **vector** (§31.4) свободен от подобного рода неприятностей.

7.5. Указатели и модификатор const

Язык C++ предлагает два понятия «константности»:

- **constexpr**: вычислить во время компиляции (§2.2.3, §10.4),
- **const**: неизменно в данной области видимости (§2.2.3).

Роль **constexpr** состоит в том, чтобы обеспечить и гарантировать вычисление на этапе компиляции, в то время как **const** специфицирует неизменность (неизменяемость, константность) в интерфейсах. Данный раздел фокусируется на второй роли — неизменности в спецификациях интерфейсов.

Многие объекты не подлежат изменению после инициализации:

- Символические константы значительно улучшают возможности поддержки кода, нежели применение буквальных значений непосредственно в коде.
- По многим указателям выполняется лишь чтение данных, а не запись.
- Многие параметры функции используются лишь для чтения.

Для отражения этих понятий неизменности после инициализации, мы добавляем модификатор **const** в определении объекта. Например:

```
const int model = 90;          // model есть константа
const int v[] = {1, 2, 3, 4};  // v[i] есть константа
const int x;                   // ошибка: нет инициализатора
```

Поскольку объекту с модификатором **const** нельзя ничего присваивать, то он должен быть в обязательном порядке проинициализирован в определении.

Объявляя что-либо с модификатором **const**, мы гарантируем его неизменность во всей области видимости:

```
void f()
{
    model = 200;  // ошибка
    v[2] = 3;     // ошибка
}
```

Обратите внимание на то, что **const** изменяет тип; он ограничивает варианты использования объекта, а не уточнет, как объект должен аллоцироваться в памяти. Например:

```
void g(const X* p)
{
    // невозможно модифицировать *p здесь
}

void h()
{
    X val;    // val можно здесь модифицировать
    g(&val);
    // ...
}
```

Используя указатель, вы имеете дело с двумя сущностями: непосредственно с указателем и с объектом, на который он указывает. Если указатель объявлен с модификатором **const**, то константным становится объект, на который смотрит указатель, а не сам указатель. Чтобы сделать константным (неизменяемым) именно указатель, его нужно объявить с модификатором ***const**. Например:

```
void f1(char* p)
{
    char s[] = "Gorm";

    const char* pc = s;        // указатель на константу
    pc[3] = 'g';               // ошибка: pc указывает на константу
    pc = p;                    // OK
```

```
    char *const cp = s;          // константный указатель
    cp[3] = 'a';                 // OK
    cp = p;                      // ошибка: cp есть константа

    const char *const cpc = s;   // константный указатель на константу
    cpc[3] = 'a';                // ошибка: cpc указывает на константу
    cpc = p;                     // ошибка: cpc есть константа
}
```

Итак, указатель константным делает декларатор ***const**, а декларатора **const*** не существует. Пример:

```
char *const cp;      // константный указатель на char
char const* pc;      // указатель на const char
const char* pc2;     // указатель на const char
```

Некоторые люди находят полезным читать такие объявления справа-налево, например, «**cp** это константный (**const**) указатель на **char**» и «**pc2** это указатель на **char** который **const**».

Объект может быть константным при доступе с одного указателя и неконстантным при доступе с другого указателя. Это очень полезно в случае аргументов функции. Объявляя указатель-аргумент функции константным, мы гарантируем, что функция не изменит объект, адресуемый этим указателем. Пример:

```
const char* strchr(const char* p, char c);    // ищем первое вхождение c в p
char* strchr(char* p, char c);                // ищем перве вхождение c в p
```

Первая версия функции используется для немодифицируемых строк и возвращает указатель, по которому модификация тоже запрещена. Вторая версия функции используется для изменяемых строк.

Можно присваивать адрес неконстантного объекта указателю на константу, поскольку в этом нет ничего опасного. В то же время адрес константного объекта нельзя присваивать указателю на переменные величины, ибо в таком случае возможна модификация значения объекта. Пример:

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;      // OK
    const int* p2 = &a;      // OK
    int* p3 = &c;            // ошибка: инициализация int* посредством const int*
    *p3 = 7;                 // попытка изменить значение c
}
```

Возможно, но неразумно снимать ограничения на указатели на **const** посредством явного приведения типа (§16.2.9, §11.5).

7.6. Указатели и владение объектами

Ресурс это нечто, что должно быть захвачено, а позже освобождено (§5.2). Память захватывается операцией **new** и позже освобождается операцией **delete** (§11.2), файлы открываются вызовом функции **fopen()** и закрываются вызовом функции **fclose()** (§43.2) — это все примеры ресурсов, для которых самый прямой доступ реализуется через указатель. Это является источником ошибок, ибо указатель можно свободно передавать в функции,

которые не знают как различать указатели, одни из которых «владеют» захваченным ресурсом, а другие нет. Рассмотрим следующий код:

```
void confused(int* p)
{
    // удалить p?
}

int global {7};

void f()
{
    X* pn = new int{7};
    int i {7};
    int q = &i;
    confused(pn);
    confused(q);
    confused(&global);
}
```

Если функция **confused()** будет использовать **delete p**, то программа будет вести себя неадекватно для двух последних вызовов, поскольку нельзя при помощи операции **delete** удалять объекты, которые ранее не были созданы в памяти при помощи **new** (§11.2). А если **confused()** не будет использовать **delete p**, то тогда возникнет утечка памяти (§11.2.1). В таком случае функция **f()** сама должна контролировать время жизни объектов, которые она размещает в памяти посредством **new**, но для больших программ в этом вопросе требуется единая согласованная стратегия.

Обычно хорошей идеей будет поместить указатель, через который осуществляется владение ресурсом, в класс дескриптора ресурса, вроде **vector**, **string** и **unique_ptr**. В этом случае мы можем полагать, что любой указатель, не входящий в дескриптор ресурса, не владеет никаким ресурсом и к нему не нужно применять операцию **delete**. Глава 13 посвящена более детальному обсуждению проблематики управления ресурсами.

7.7. Ссылки

Указатели позволяют передавать потенциально большие объемы данных без необходимости копирования данных: вместо этого передается адрес, являющийся значением указателя. Тип указателя говорит о том, что можно делать с указуемыми им данными. Использование указателя отличается от использования имен объектов следующими моментами:

- Иной синтаксис, например ***p** вместо **obj** или **p->m** вместо **obj.m**.
- Посредством указателя можно адресовать разные объекты в разные моменты времени.
- С указателями нужно быть более внимательными, ибо указатель может иметь значение **nullptr**, или указывать не на тот объект, который ожидается.

Эти отличия могут сильно навредить в самый неподходящий момент, к тому же многие программисты записи вроде **f(&x)** считают «безобразно выглядящими» по сравнению с **f(x)**. Работа с указателями, которые могут изменяться, весьма трудоемка, ибо нужно то и дело проверять их на **nullptr**. Наконец, при перегрузке, например, операции **+** хотелось бы писать привычное **x+y**, а не **&x+&y**. Язык C++ откликается на эти проблемы механизмом *ссылок* (*references*). Как и указатель, ссылка служит синонимом для объекта, чаще всего реализуется с целью содержать адрес объекта, также не накладывает никаких дополнительных накладных расходов (как и в случае указателей), но отличается от указателей в следующих чертах:

- Доступ к объекту по ссылке выполняется тем же синтаксисом, что и для имени объекта.
- Ссылка всегда настроена на один и тот же объект, для которого она была проинициализирована.
- Нет никаких нулевых ссылок, и всегда можно считать, что ссылка ссылается на какой-либо объект (§7.7.4).

Ссылка является, таким образом, альтернативным именем объекта (синонимом). Преимущественным использованием ссылок служит объявление аргументов функций, возвращаемых функциями значений и проблематика перегружаемых операций в целом (Глава 18). Пример:

```
template<class T>
class vector {
  T* elem;
  // ...
public:
  T& operator[](int i) {return elem[i];}                  // возврат ссылки на элемент
  const T& operator[](int i) const {return elem[i];}      // возврат ссылки на константный элемент

  void push_back(const T& a);                             // передать элемент по ссылке
  // ...
};
void f(const vector<double>& v)
{
  double d1 = v[1]; // скопировать знач. типа double, на кот. ссылаются посредством v.operator[](1), в d1
  v[2] = 7;         // поместим 7 в переменную типа double, на которую ссылается v.operator[](2)
  v.push_back(d1); // дадим push_back() ссылку на d1
}
```

Идея передачи функциям аргументов по ссылке является чрезвычайно старой (например, самые старые версии языка Fortran уже использовали этот механизм).

В контексте разграничения lvalue/rvalue и **const**/не-**const** ссылок имеются три вида ссылок:

- *lvalue-ссылки*: чтобы ссылаться на объекты, которые мы хотим изменять
- **const**-*ссылки*: чтобы ссылаться на объекты, которые мы не хотим изменять (то есть на константы)
- *rvalue-ссылки*: чтобы ссылаться на объекты, значения которых нам не нужны после того, как мы их использовали (например, временные объекты)

Собирательно все эти три вида называются просто *ссылками*. Первые два вида характеризуют как *lvalue* ссылки.

7.7.1. lvalue-ссылки

В имени типа запись **X&** означает «ссылка на **X**». Такие обозначения используются для *lvalue-ссылок*, которые ссылаются на lvalue-объекты. Пример:

```
void f()
{
    int var = 1;
    int& r {var};   // r и var теперь ссылаются на один и тот же int
    int x = r;      // x становится 1
    r = 2;          // var становится 2
}
```

Чтобы гарантировать «настроенность ссылки» на какой-либо объект, мы обязаны инициализировать ссылку. Например:

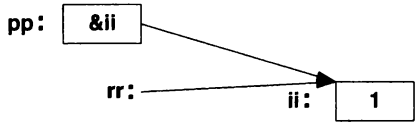
```
int var = 1;
int& r1 {var};        // OK: r1 инициализирован
int& r2;              // ошибка: отсутствует инициализатор
extern int& r3;       // OK: r3 инициализируется в ином месте
```

Инициализация ссылки — это не то же самое, что присваивание ссылке значения, ибо при этом над ссылкой не выполняется никакая операция. Пример:

```
void g()
{
    int var = 0;
    int& rr {var};
    ++rr;             // var инкрементируется на 1
    int* pp = &rr;    // pp указывает на var
}
```

Здесь ++ не инкрементирует ссылку; скорее эта операция инкрементирует объект типа **int**, на который ссылка смотрит, то есть **var**. Таким образом, значение ссылки неизменяемо после ее инициализации; она всегда смотрит на тот объект, для которого была инициализирована. Чтобы получить адрес объекта, на который смотрит ссылка **rr**, мы используем довольно очевидный синтаксис **&rr**. Таким образом, мы не можем иметь указатель на ссылку, и значит невозможно создать массив ссылок. В этом смысле, ссылка не является объектом.

Очевидной реализацией ссылки служит указатель (константный), который автоматически разыменовывается при его использовании. В принципе, нет никакого вреда в том, чтобы представлять себе ссылки таким образом до тех пор, пока в голове остается понимание того, что ссылками нельзя манипулировать так же, как и указателями:



В некоторых случаях компиляторы могут оптимизировать машинный код таким образом, что в нем не остается никаких следов объектов, используемых для реализации ссылок во время исполнения программы.

Инициализация ссылки тривиальна, когда инициализатор относится к категории lvalue (объект с адресом, до которого вы можете добраться; §6.4). Инициализатором для **T&** должен быть lvalue типа **T**.

А вот инициализатор для **const T&** не обязан быть lvalue и даже не обязан быть типа **T**. В таких случаях:

1. Неявное приведение к типу **T** применяется в случае необходимости (§10.5).
2. После этого результирующая величина помещается во временную переменную типа **T**.
3. Наконец, эта временная переменная используется как значение инициализатора.

Рассмотрим следующее:

```
double& dr = 1;          // ошибка: требуется lvalue
const double& cdr {1};   // OK
```

Последнюю инициализацию можно интерпретировать следующим образом:

```
double temp = double{1};      // создаем временную переменную
const double& cdr {temp};     // затем используем ее как инициализатор для cdr
```

Временная переменная, созданная для того, чтобы хранить инициализатор ссылки, актуальна до конца области видимости связанной с ней ссылки.

Ссылки на переменные и ссылки на константы весьма различаются, так как введение временной переменной для исходной переменной может служить источником серьезной ошибки — присваивание переменной стало бы присваиванием временной переменной, которая вскоре может исчезнуть. Такой проблемы не существует для ссылок на константы, и ссылки на константы имеют важное применение в качестве аргументов функций (§18.2.4).

Ссылки часто используются для индикации того факта, что функция может изменять значение передаваемого ей объекта. Например:

```
void increment(int& aa)
{
    ++aa;
}

void f()
{
    int x = 1;
    increment(x);  // x = 2
}
```

Семантика передачи аргумента идентична семантике инициализации, так что при вызове **increment()** ее аргумент **aa** становится другим именем для **x**. Программы бывает легче читать (и понимать), если избегать изменения функциями значений своих аргументов. Вместо этого, можно возвращать новое значение из функции явным образом:

```
int next(int p) {return p+1;}

void g()
{
    int x = 1;
    increment(x);  // x = 2
    x = next(x);   // x = 3
}
```

Семантика вызова **increment(x)** не позволяет просто так читателю догадаться, что величина **x** будет изменена, в то время как выражение **x=next(x)** явно указывает на это. Следовательно, применение просто ссылок (без модификаторов) в качестве аргументов функции целесообразно снабжать крайне информативными именами функций, указывающими на то, что значение аргумента будет изменяться в функции.

Ссылки можно использовать и для возврата из функций. Во многих случаях так поступают, если вызов функции может стоять как в левой части присваивания, так и в правой части. Вот наглядный пример:

```
template<class K, class V>
class Map {   // простой класс
public:
    V& operator[](const K& v);   // возврат значения для ключа v

    pair<K, V>* begin() {return &elem[0];}
    pair<K, V>* end() {return &elem[0]+elem.size();}
```

```
private:
    vector<pair<K, V>> elem;      // {key, value} pairs
};
```

Тип **map** из стандартной библиотеки (§4.4.3, §31.4.3) в типичном случае реализуется как красно-черное дерево, но чтобы избежать ненужных осложнений при изложении, я покажу реализацию на базе линейного поиска совпадения ключа:

```
template<class K, class V>
V& Map<K, V>::operator[](const K& k)
{
    for (auto& x : elem)
        if (k == x.first)
            return x.second;

    elem.push_back({k, V{}});      // добавляем пару в конец (§ 4.4.2)
    return elem.back().second;     // возвращаем (умолчательное) значение нового эл-та
}
```

Я передаю ключевой аргумент, **k**, по ссылке, потому что он может оказаться типом, для которого копирование обойдется слишком дорого. По аналогичной причине здесь и возврат реализуется по ссылке. Я использую **const**-ссылку для **k**, потому что, во-первых, не собираюсь его модифицировать, а во-вторых, он может задаваться в буквальной форме или как временный объект. А вот возврат уже не является константной ссылкой, поскольку пользователь контейнера **Map** возможно захочет модифицировать найденное значение. Например:

```
int main() // считаем количество вхождений каждого слова на входе
{
    Map<string, int> buf;

    for (string s; cin>>s;) ++buf[s];

    for (const auto& x: buf)
        cout << x.first << " : " << x.second << '\n';
}
```

В каждой итерации ввода считывается одно слово из входного потока **cin** в строку **s** (§4.3.2) и счетчик, связанный со словами, обновляется. В конце распечатывается результирующая таблица введенных слов вместе с числом повторов для каждого слова. Допустим, что ввод был таким

```
aa bb bb aa aa bb aa aa
```

Тогда программа произведет вот такой вывод:

```
aa: 5
bb: 3
```

Здесь цикл range-**for** проходит потому, что **Map** определяет методы **begin()** и **end()** так же, как это делает **map** из стандартной библиотеки.

7.7.2. rvalue-ссылки

Основная идея, лежащая в основе введения разных видов ссылок, заключается в поддержке разной работы с объектами:

- Не-**const** lvalue-ссылки ссылаются на объект, в который пользователь ссылки может писать.

- **const** lvalue-ссылки ссылаются на константу, которая с точки зрения пользователя ссылки представляет собой неизменную сущность.
- rvalue-ссылки ссылаются на временный объект, который пользователь ссылки может (и часто хочет) модифицировать, предполагая что объект никогда снова не будет использоваться.

Нам неплохо бы знать, ссылается ли ссылка на временный объект, потому что в таком случае мы иногда могли бы вместо дорогой операции копирования перейти к более дешевой операции перемещения (§3.3.2, §17.1, §17.5.2). Объект (например, типа **string** или **list**), представимый небольшим дескриптором, указывающим на потенциально огромный объем информации, мог бы просто и дешево перемещен, если мы уверены, что источник не будет далее использоваться. Классическим примером служит возврат функции, в рамках которого компилятор точно значет, что локальная переменная, использованная для возврата, никогда далее не будет использована (§3.3.2).

Инициализировать rvalue-ссылку можно лишь посредством инициализатора со значением категории rvalue, но не lvalue. Пример:

```
string var {"Cambridge"};
string f();

string& r1 {var};          // lvalue ссылка, привязка r1 к var (lvalue)
string& r2 {f()};          // lvalue ссылка, ошибка: f() есть rvalue
string& r3 {"Princeton"};  // lvalue ссылка, ошибка: нет привязки ко временной переменной
string&& rr1 {f()};        // rvalue ссылка, OK: привязка rr1 к rvalue (времен. объект)
string&& rr2 {var};        // rvalue ссылка, ошибка: var есть lvalue
string&& rr3 {"Oxford"};   // rr3 ссылается на времен. объект "Oxford"

const string cr1& {"Harvard"}; // OK: создаем времен. объект и привязываем к cr1
```

Операция объявления **&&** означает «rvalue-ссылка». Мы не применяем константные rvalue-ссылки; основное преимущество rvalue-ссылок состоит в возможности писать в объекты, на которые они ссылаются. Как **const** lvalue-ссылки, так и rvalue-ссылки могут быть связаны с объектами, относящимися к категории rvalue. Но цели при этом фундаментально разные:

- Мы используем rvalue-ссылки для «разрушающего чтения» ради оптимизации кода, в котором иначе пришлось бы выполнять копирование.
- Мы используем **const** lvalue-ссылки для предотвращения модификации аргумента.

Доступ к объекту, адресуемому посредством rvalue-ссылки, выполняется точно так же, как и в случае доступа к объекту по имени или по lvalue-ссылке. Пример:

```
string f(string&& s)
{
    if (s.size())
        s[0] = toupper(s[0]);
    return s;
}
```

Иногда программист знает, что объект никогда далее не будет использоваться, в то время как компилятор об этом не может догадаться. Рассмотрим следующий код:

```
template<class T>
swap(T& a, T& b)  // "swap" в старом стиле
{
    T tmp {a};    // теперь две копии a
    a = b;        // теперь две копии b
    b = tmp;      // теперь две копии tmp
}
```

Если **T** является типом, для которого копирование всех элементов может быть дорогой операцией, вроде типов **string** или **vector**, то представленный вариант функции **swap()** окажется дорогостоящей операцией. Обратим внимание на курьезный момент: нам в данном случае не нужны никакие копирования; мы лишь хотим переместить по кругу значения переменных **a**, **b** и **tmp**. Мы можем сказать об этом компилятору:

```
template<class T>
void swap(T& a, T& b)  // "идеальный swap" (почти)
{
    T tmp {static_cast<T&&>(a)}; // при инициализации можно писать в a
    a = static_cast<T&&>(b);     // присвоение может писать в b
    b = static_cast<T&&>(tmp);   // присвоение может писать в tmp
}
```

Значением выражения **static_cast<T&&>(x)** будет **rvalue** типа **T&&** для **x**; и теперь любые операции, оптимизированные под rvalue, могут использовать эту оптимизацию для **x**. В частности, если тип **T** имеет перемещающий конструктор (§3.3.2, §17.5.2) или перемещающее присваивание, то это и будет использовано. Рассмотрим тип **vector**:

```
template<class T> class vector {
    // ...
    vector(const vector& r); // копирующий конструктор (копирует представление у r)
    vector(vector&& r);      // перемещающий конструктор ("крадет" представление у r)
};

vector<string> s;
vector<string> s2 {s};          // s есть lvalue, используем копирующий конструктор
vector<string> s3 {s+"tail"};   // s+"tail" есть rvalue, используем перемещающий конструктор
```

Применение **static_cast** в функции **swap()** несколько громоздко и поэтому стандартная библиотека предоставляет функцию **move()**: **move(x)** означает **static_cast<X&&>(x)**, где **X** это тип для **x**. Воспользуемся новой возможностью и несколько упростим определение функции **swap()**:

```
template<class T>
void swap(T& a, T& b)  // "идеальный swap" (почти)
{
    T tmp {move(a)};   // перемещаем из a
    a = move(b);       // перемещаем из b
    b = move(tmp);     // перемещаем из tmp
}
```

В отличие от самой первой версии функции **swap()** последняя версия свободна от копирования; она использует перемещающие операции везде, где это возможно. Я полагаю этот вариант функции **swap()** почти идеальным, поскольку он будет обменивать значения сущностей, относящихся только к категории lvalue.

А теперь рассмотрим следующий пример:

```
void f(vector<int>& v)
{
    swap(v, vector<int>{1,2,3});   // заменяем элементы у v на 1,2,3
    // ...
}
```

Ясно, что на практике часто действительно бывает нужным заменить содержимое контейнера некими значениями по умолчанию, но наш вариант функции **swap()** не может этого сделать. Решением будет усиление нашего решения двумя перегрузками:

```
template<class T> void swap(T&& a, T& b);
template<class T> void swap(T& a, T&& b)
```

Представленный выше пример будет решен с помощью последнего из указанных вариантов. В стандартной библиотеке использован иной вариант решения, основанный на реализации функций **shrink_to_fit()** и **clear()** для типов **vector**, **string** и т.п. (§31.3.3), позволяющий справляться с общими случаями аргументов категории rvalue для функции **swap()**:

```
void f(string& s, vector<int>& v)
{
    s.shrink_to_fit();          // делаем s.capacity()==s.size()
    swap(s, string{s});         // делаем s.capacity()==s.size()
    v.clear();                  // делаем v пустым
    swap(v.vector<int>{});      // делаем v пустым
    v = {};                     // делаем v пустым
}
```

Все контейнеры стандартной библиотеки предоставляют перемещающий конструктор и перемещающее присваивание (§31.3.2). Их операции для вставки новых элементов, такие как **insert()** и **push_back()** имеют версии, которые принимают аргументы, относящиеся к категории rvalue.

7.7.3. Ссылки на ссылки

Определяя ссылку на ссылку на тип, вы получаете новый тип. Но какова категория таких ссылок — lvalue-ссылка или rvalue-ссылка? Рассмотрим следующее:

```
using rr_i = int&&;
using lr_i = int&;
using rr_rr_i = rr_i&&;   // "int && &&" есть int&&
using lr_rr_i = rr_i&;    // "int && &" есть int&
using rr_lr_i = lr_i&&;   // "int & &&" есть int&
using lr_lr_i = lr_i&;    // "int & &" есть int&
```

Иными словами, всегда побеждают lvalue-ссылки.

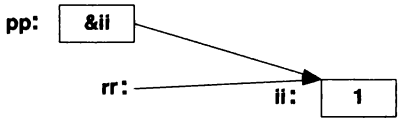
Синтаксисом запрещено

```
int && & r = i;
```

Ссылки на ссылки возникают в результате синонимизации (§3.4.5, §6.5) или в качестве аргумента шаблона (§23.5.2.1).

7.7.4. Указатели и ссылки

Указатели и ссылки являются двумя механизмами адресации объектов из разных частей программы без копирования. Это можно отразить графически:



У каждого из этих двух способов есть сильные и слабые стороны.

Если нужно менять объекты, на которые вы ссылаетесь, выбирайте указатель. Можно использовать операции =, +=,-=, ++ и -- для изменения значения переменной-указателя (§11.1.4). Пример:

```
void fp(char* p)
{
    while (*p)
        cout << ++*p;
}

void fr(char& r)
{
    while (r)
        cout << ++r;   // oops: инкрементируем char
                       // почти бесконечный цикл!
}

void fr2(char& r)
{
    char* p = &r;      // получаем указатель на объект, на который ссылаемся
    while (*p)
        cout << ++*p;
}
```

В ином случае, когда вы хотите быть уверенными в том, что имя ссылается всегда на один и тот же объект, выбирайте ссылку. Пример:

```
template<class T> class Proxy {  // Proxy ссылается на объект, которым проинициализирован
    T& m;
public:
    Proxy(T& mm): m{mm} {}
    // ...
};

template<class T> class Handle { // Handle ссылается на текущий объект
    T* m;
public:
    Proxy(T* mm): m{mm} {}
    void rebind(T* mm) {m = mm;}
    // ...
};
```

Наконец, если вы хотите иметь пользовательскую (перегруженную) операцию (§18.1) для чего-то, что ссылается на объект, используйте ссылку:

```
Matrix operator+(const Matrix&, const Matrix&);   // OK
Matrix operator-(const Matrix*, const Matrix*);   // ошибка: нет аргумента пользоват. типа

Matrix y, z;
// ...
Matrix x = y+z;      // OK
Matrix x2 = &y-&z; // ошибка
```

Невозможно переопределить операцию для пары встроенных типов, таких как указатели (§18.2.3).

Если вам нужен набор (коллекция) чего-либо, что ссылается на объекты, то вам нужны указатели:

```
int x, y;
string& a1[] = {x, y};              // ошибка: массив ссылок
string* a2[] = {&x, &y};            // OK
vector<string&> s1 = {x, y};        // ошибка: вектор ссылок
vector<string*> s2 = {&x, &y};      // OK
```

Иные случаи, где возможен выбор между указателями и ссылками, остаются на усмотрение программистов и часто относятся к вопросам эстетики. В идеале, нам надо бы

делать выбор опираясь, в первую очередь, на минимизацию потенциально возможных ошибок и особенно на увеличение ясности и прозрачности кода.

Роль понятия «нет значения» для указателей играет **nullptr**. Для ссылок же никакого эквивалента вроде «нулевой ссылки» просто нет. Поэтому, если в контексте вашей задачи имеется необходимость в величине «нет значения», то указатели будут лучшим выбором. Пример:

```
void fp(X* p)
{
      if (p == nullptr) {
          // нет значения
      }
      else {
          // используем *p
      }
}

void fr(X& r)   // общепринятый стиль
{
      // полагаем r корректным и используем его
}
```

Если уж очень нужно, то можно сконструировать (а затем проверять на совпадение) некий вариант «нулевой ссылки» для частного типа:

```
void fr2(X& r)
{
      if (&r == &nullX) {  // или может быть r==nullX
          // нет значения
      }
      else {
          // используем r
      }
}
```

Естественно, необходимо подходящим образом определить **nullX**. Но, конечно, это не слишком хороший стиль программирования. Программист может рассчитывать, что ссылка корректна (действительна). Вы можете попробовать сконструировать некорректную ссылку, что-то вроде следующего варианта:

```
char* ident(char * p) {return p;}

char& r {*ident(nullptr)}; // неправильный код
```

Этот код не является корректным C++-кодом. Не делайте так, даже если ваша реализации не отловит ошибки.

7.8. Советы

[1] Используйте указатели в простой и стандартной манере; §7.4.1.

[2] Избегайте нетривиальной арифметики указателей; §7.4.

[3] Следите за тем, чтобы не писать за границы массивов; §7.4.1.

[4] Избегайте многомерных массивов; применяйте вместо них подходящие контейнеры; §7.4.2.

[5] Используйте **nullptr** вместо **0** или **NULL**; §7.2.2.

[6] Используйте контейнеры (например, **vector**, **array** или **valarray**) вместо встроенных массивов в С-стиле; §7.4.1.

[7] Применяйте **string** вместо нуль-терминированных массивов символов типа **char**; §7.4.

[8] Используйте «сырые» строки в качестве литералов со множественными обратными наклонными чертами; §7.3.2.1.

[9] Предпочитайте использовать вместо простых ссылочных аргументов ссылки с модификатором **const**; §7.7.3.

[10] Используйте rvalue-ссылки для семантики перемещения; §7.7.2.

[11] Храните указатели, отражающие факт владения, внутри дескрипторных классов; §7.6.

[12] Избегайте указателей типа **void*** всюду, кроме самого низкоуровневого кода; §7.2.1.

[13] Используйте указатели и ссылки с модификатором **const** для отражения факта неизменности в интерфейсах; §7.5.

[14] В аргументах предпочитайте ссылки указателям за исключением случая, когда допускается опция «нет объекта»; §7.7.4.

8

Структуры, объединения и перечисления

Сформируйте более совершенное Объединение.
— Люди

- Введение
- Структуры

  Раскладка структурных объектов в памяти; Имена структур; Структуры и классы; Структуры и массивы; Эквивалентность типов; POD; Битовые поля
- Объединения

  Объединения и классы; Анонимные объединения
- Перечисления

  enum class; «Простой» **enum**; Безымянные перечисления
- Советы

8.1. Введение

Ключом к эффективному использованию C++ служат определение и применение пользовательских типов. В данной главе мы рассмотрим набор из трех наиболее простых пользовательских типов:

- Структуры (**struct**) — последовательность элементов (членов) произвольных типов.
- Объединения (**union**) — структура, которая в каждый момент времени содержит лишь один из ее элементов.
- Перечисления (**enum**) — это тип с набором именованных констант (энумераторов).
- **enum class** — это перечисление, инкапсулирующее имена констант в своей собственной области видимости и не допускающее неявных преобразований к другим типам.

Некие варианты этих простых типов существовали в языке C++ с самых ранних версий языка. Они фокусировались на специфических представлениях данных и служили твердой основой для программирования в С-стиле. Понятие структуры (**struct**) является простой формой понятия класс (**class**) (§3.2, Глава 16).

8.2. Структуры

Массив — это агрегат элементов одного и того же типа. В простейшей форме структура — это агрегат элементов произвольных типов. Например:

```
struct Address {
    const char* name;       // "Jim Dandy"
    int number;             // 61
```

```
    const char* street;     // "South St"
    const char* town;       // "New Providence"
    char state[2];          // 'N' 'J'
    const char* zip;        // "07974"
};
```

Здесь дается определение типа с именем **Address**, состоящего из элементов, необходимых для отправки письма в пределах США. Обратите внимание на терминальную точку с запятой.

Переменные типа **Address** объявляются точно так же, как и все иные переменные, а доступ к индивидуальным членам структуры выполняется с помощью операции точка. Пример:

```
void f()
{
    Address jd;
    jd.name = "Jim Dandy";
    jd.number = 61;
}
```

Переменные структурных типов можно инициализировать в {}-стиле (§6.3.5). Например:

```
Address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence",
    {'N','J'}, "07974"
};
```

Обратите внимание на то, что **jd.state** не может быть проинициализирован строкой **"NJ"**. Строки содержат терминальный нуль, **'\0'**, так что строка **"NJ"** состоит из трех символов — на один больше, чем умещается в **jd.state**. Я намеренно применяю такие довольно низкоуровневые члены для иллюстрации того, как все может реализовываться и какие проблемы это вызывает.

К структурам часто обращаются по указателям с помощью операции **->** (операция разыменования указателя на структуру). Например:

```
void print_addr(Address* p)
{
    cout << p->name << '\n'
         << p->number << ' ' << p->street << '\n'
         << p->town << '\n'
         << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

Для указателя **p** выражение **p->m** эквивалентно выражению **(*p).m**.

Если к структуре обращаться посредством ссылки, то доступ к членам выполняется операцией точка:

```
void print_addr2(const Address& r)
{
    cout << r.name << '\n'
         << r.number << ' ' << r.street << '\n'
         << r.town << '\n'
         << r.state[0] << r.state[1] << ' ' << r.zip << '\n';
}
```

Передача аргументов в функции обсуждается в §12.2.

Объекты структурных типов могут присваиваться, передаваться функциям и возвращаться из функций. Пример:

```
Address current;

Address set_current(Address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

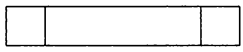
Другие желательные операции, вроде сравнений (== и!=), не предоставляются по умолчанию. Естественно, что пользователь может сам определить такие операции (§3.2.1.1, Глава 18).

8.2.1. Раскладка структурных объектов в памяти

Объект структурного типа содержит члены в порядке их объявления. Например, мы могли бы хранить информацию об оборудовании в структуре:

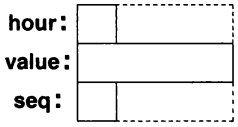
```
struct Readout {
    char hour;      // [0:23]
    int value;
    char seq;       // отметка последовательности ['a':'z']
};
```

Раскладку объектов типа **Readout** в памяти можно представить графически:



Члены структуры располагаются в памяти в порядке объявления, так что адрес члена **hour** меньше адреса члена **value**. Смотри также §8.2.6.

В то же время, суммарный размер объекта структурного типа не обязательно равен сумме размеров его членов. Это связано с тем, что на многих машинных архитектурах (аппаратуре) требуется располагать объекты определенных типов по границам, специфичным для конкретной архитектуры, или же по причине более эффективного к ним доступа в случае выравнивания по определенным границам. К примеру, целые значения часто выгодно располагать по границам машинных слов. Об этом говорят как о необходимости правильно выравнивать расположение (раскладку) объектов в памяти (§6.2.9). Это может приводить к «дырам» внутри объектов структурных типов. В связи с этим, приведем более реалистичную раскладку для объектов типа **Readout** на машинах с 4-байтовыми **int**:

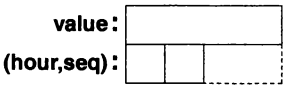


Поэтому на многих машинах **sizeof(Readout)** равен **12**, а не **6**, как многие могли бы наивно предположить, сложив размеры индивидуальных членов.

Можно минимизировать потери памяти путем изменения порядка следования членов структуры (большего размера члены — вперед). Например:

```
struct Readout {
    int value;
    char hour;      // [0:23]
    char seq;       // отметка последовательности ['a':'z']
};
```

Теперь имеем следующее:



Заметим, что теперь у нас остается «дыра» в **Readout** (неиспользуемое пространство) в 2 байта, а **sizeof(Readout)==8**. Это еще важнее для массивов, так как размер массива из 10 объектов типа **Readout** будет **10*sizeof(Readout)**.

Нужно также учитывать, что все-таки желательно располагать члены структур в первую очередь в плане улучшения читаемости программы, а переставлять их только в случае большой нужды в оптимизации.

Применение спецификаторов доступа (**public**, **private** или **protected**) может оказывать влияние на раскладку (§20.5).

8.2.2. Имена структур

Имя типа становится доступным сразу же, как только оно появляется в программе, а не после того, как станет видимым полное объявление. Пример:

```
struct Link {
    Link* previous;
    Link* successor;
};
```

Однако ясно, что объявлять объекты структурного типа возможно лишь после того, как будет дано полное определение структурного типа. Например:

```
struct No_good {
    No_good member; // ошибка: рекурсивное определение
};
```

Здесь ошибка возникает из-за того, что компилятор заранее не знает полного размера для типа **No_good**. Чтобы две (или более) структуры могли ссылаться друг на друга, нужно объявить соответствующие имена именами структур. Например:

```
struct List;    // объявление имени структуры: будет определена позже

struct Link {
    Link* pre;
    Link* suc;
    List* member_of;
    int data;
};

struct List {
    Link* head;
};
```

Без предварительного объявления для **List** применение указателя типа **List*** в объявлении для **Link** было бы синтаксической ошибкой.

Имя структурного типа может использоваться до того, как этот тип определен, если только не используются при этом имена членов и если не нужно знать полного размера

для этого типа. Однако, до того момента, пока структурный тип не будет полностью определен, имя типа трактуется как имя незавершенного типа. Пример:

```
struct S; // "S" это имя некоторого типа

extern S a;
S f();
void g(S);
S* h(S*);
```

Многие из таких объявлений не могут использоваться, пока тип **S** не будет определен:

```
void k(S* p)
{
    S a;          // ошибка: S не определена; нужен размер для выделения памяти

    f();          // ошибка: S не определена; нужен размер для возврата
    g(a);         // ошибка: S не определена; нужен размер для передачи аргумента
    p->m = 7;     // ошибка: S не определена; не известно имя члена

    S* q = h(p);  // ok: указатели можно аллоцировать и передать
    q->m = 7;     // ошибка: S не определена; неизвестное имя члена
}
```

По историческим причинам, восходящим к языку С, возможно объявление структурного и неструктурного типов с одинаковым именем в пределах одной и той же области видимости. Пример:

```
struct stat {/* ... */};
int stat(char* name, struct stat* buf);
```

В этом случае имя **stat** будет относиться к неструктурному типу, а на структурный тип нужно ссылаться с префиксом **struct**. Аналогично, ключевые слова **class**, **union** (§8.3) и **enum** (§8.4) следует использовать как префиксы для устранения неоднозначностей. В то же время, лучше вообще не перегружать имена, чтобы не порождать такие неоднозначности.

8.2.3. Структуры и классы

Вообще-то, структура — это класс, в котором все члены по умолчанию имеют статус доступа **public**. Отсюда вытекает, что в структурах могут быть члены-функции (§2.3.2, Глава 16). В частности, структуры могут иметь конструкторы. Пример:

```
struct Points {
    vector<Point> elem;      // должен содержать минимум один Point
    Points(Point p0) {elem.push_back(p0);}
    Points(Point p0, Point p1) {elem.push_back(p0); elem.push_back(p1);}
    // ...
};

Points x0;                   // ошибка: нет умолчательного конструктора
Points x1{{100,200}};        // один Point
Points x1{{100,200}, {300,400}};  // два Point
```

Нет необходимости определять конструктор лишь для того, чтобы инициализировать члены по-очереди. Пример:

```
struct Point {
    int x, y;
};

Point p0;          // опасно: не инициализировано в локальной области (§ 6.3.5.1)
Point p1 {};       // умолчательное конструирование: {{},{}}; that is {0.0}
```

```
Point p2 {1};       // второй член конструируется умолчательно: {1,{}}; that is {1,0}
Point p3 {1,2};     // {1,2}
```

Конструкторы нужны, если нужно переставлять аргументы, проверять аргументы на корректность, модифицировать аргументы, устанавливать инварианты (§2.4.3.2, §13.4) и т.д. Например:

```
struct Address {
    string name;    // "Jim Dandy"
    int number;     // 61
    string street;  // "South St"
    string town;    // "New Providence"
    char state[2];  // 'N' 'J'
    char zip[5];    // 07974

    Address(const string n, int nu, const string& s, const string& t, const string& st, int z);
};
```

Здесь я добавил конструктор, чтобы гарантировать инициализацию всех членов и чтобы можно было использовать **string** и **int** для почтового кода вместо того, чтобы возиться с одиночными символами. Например:

```
Address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence",
    "NJ", 7974          // (07974 будет восьмеричным; § 6.2.4.1)
};
```

Конструктор для **Address** мог бы быть определен следующим образом:

```
Address::Address(const string& n, int nu, const string& s, const string& t, const string& st, int z)
            // проверяем почтовый код
    : name{n},
    number{nu},
    street{s},
    town{t}
{
    if (st.size()!=2)
        error("State abbreviation should be two characters")
    state = {st[0], st[1]};     // отметка хранения в виде символов
    ostringstream ost;          // поток вывода строк; см. § 38.4.2
    ost << z;                   // извлекаем символы из int
    string zi {ost.str()};
    switch (zi.size()) {
    case 5:
        zip = {zi[0], zi[1], zi[2], zi[3], zi[4]};
        break;
    case 4:  // начинается с '0'
        zip = {'0', zi[0], zi[1], zi[2], zi[3]};
        break;
    default:
        error("unexpected ZIP code format");
    }
    // ... проверить смысл кода ...
}
```

8.2.4. Структуры и массивы

Естественно, что можно создавать массивы с элементами структурного типа и можно иметь структуры, содержащие массивы. Например:

```
struct Point {
    int x, y
};

Point points[3] {{1,2},{3,4},{5,6}};
int x2 = points[2].x;
struct Array {
    Point elem[3];
};

Array points2 {{1,2},{3,4},{5,6}};
int y2 = points2.elem[2].y;
```

Помещая встроенный массив в структуру, мы достигаем того, что можем трактовать массив как объект: мы можем копировать содержащую массив структуру в инициализации и присваивании. Например:

```
Array shift(Array a, Point p)
{
    for (int i=0; i!=3; ++i) {
        a.elem[i].x += p.x;
        a.elem[i].y += p.y;
    }
    return a;
}

Array ax = shift(points2,{10,20});
```

Вся эта работа с **Array** несколько примитивная: почему **i!=3**? Почему элементы лишь типа **Point**? Стандартная библиотека предоставляет тип **std::array** (§34.2.1), как более полный и элегантный вариант реализации в виде структуры идеи о массиве фиксированного размера:

```
template<typename T, size_t N >
struct array {  // упрощение (см. § 34.2.1)
    T elem[N];

    T* begin() noexcept {return elem;}
    const T* begin() const noexcept {return elem;}
    T* end() noexcept {return elem+N;}
    const T* end() const noexcept {return elem+N;}

    constexpr size_t size() noexcept;

    T& operator[](size_t n) {return elem[n];}
    const T& operator[](size_type n) const {return elem[n];}

    T * data() noexcept {return elem;}
    const T * data() const noexcept {return elem;}

    // ...
};
```

Этот **array** является шаблоном, допускающим произвольное число элементов произвольного типа. Он также работает с исключениями (§13.5.1) и **const**-объектами (§16.2.9.1). Применяя **array**, мы теперь пишем следующее:

```
struct Point {
    int x, y
};

using Array = array<Point,3>;       // массив из 3 Point
```

```
Array points {{1,2},{3,4},{5,6}};
int x2 = points[2].x;
int y2 = points[2].y;
Array shift(Array a, Point p)
{
    for (int i=0; i!=a.size(); ++i) {
        a[i].x += p.x;
        a[i].y += p.y;
    }
    return a;
}

Array ax = shift(points,{10,20});
```

Основным преимуществом **std::array** перед встроенными массивами служит то, что он является вполне объектным типом (допускает присваивание и т.д.) и не допускает неявного приведения к типу указателя на индивидуальный элемент:

```
ostream& operator<<(ostream& os, Point p)
{
    cout << '{' << p[i].x << ',' << p[i].y << '}';
}

void print(Point a[], int s)    // требуется указать число элементов
{
    for (int i=0; i!=s; ++i)
        cout << a[i] << '\n';
}

template<typename T, int N>
void print(array<T, N>& a)
{
    for (int i=0; i!=a.size(); ++i)
        cout << a[i] << '\n';
}

Point point1[] = {{1,2},{3,4},{5,6}};          // 3 элемента
array<Point,3> point2 = {{1,2},{3,4},{5,6}};   // 3 элемента

void f()
{
    print(point1,4); // 4 это ошибка
    print(point2);
}
```

Но по сравнению со встроенным массивом тип **std::array** не позволяет вывести число элементов из инициализатора:

```
Point point1[] = {{1,2},{3,4},{5,6}};          // 3 элемента
array<Point,3> point2 = {{1,2},{3,4},{5,6}};   // 3 элемента
array<Point> point3 = {{1,2},{3,4},{5,6}};     // ошибка: не дано число элементов
```

8.2.5. Эквивалентность типов

Две структуры суть разные типы даже в случае, когда у них одинаковые члены. Пример:

```
struct S1 {int a;};
struct S2 {int a;};
```

Здесь **S1** и **S2** — это два разных типа, так что:

```
S1 x;
S2 y = x;      // ошибка: несоответствие типов
```

Тип структуры также отличается и от типа члена. Например:

```
S1 x;
int i = x;     // ошибка: несоответствие типов
```

Каждая структура должна иметь уникальное определение в пределах программы (§15.2.3).

8.2.6. POD (Plain Old Data)

Иногда может потребоваться трактовать объект как POD (Plain Old Data — Простые Данные, как в старые «добрые» времена; набор смежных байт в памяти) и не беспокоиться о продвинутых семантических понятиях, вроде полиморфизма времени исполнения (§3.2.3, §20.3.2), семантике определяемого пользователем копирования (§3.3, §17.5) и т.п. Часто причиной служит желание обходиться с данными в памяти способом, наиболее эффективным для конкретной аппаратуры. Например, копирование 100-элементного массива посредством ста вызовов копирующего конструктора вряд ли будет столь же эффективным, как однократный вызов функции **std::memcpy()**, которая в типичном случае опирается на команды процессора по перемещению блока памяти. Даже если конструктор встраиваемый, компилятору было бы нелегко выявить тут какой-либо способ заметной оптимизации. Необходимость в такого рода трюках не столь уж и редка; они используются при оптимизации таких контейнеров, как **vector**, а также в низкоуровневых процедурах ввода-вывода. В высокоуровневом коде лучше избегать столь низкоуровневых деталей.

Итак, под концепцией POD мы понимаем объект, которым мы манипулируем просто как данными, избегая осложнений с подробностями классов и пользовательской семантики конструирования, копирования или перемещения. Например:

```
struct S0 {};                                          // POD
struct S1 {int a;};                                    // POD
struct S2 {int a; S2(int aa): a(aa) {}};               // не POD (нет умолчат. конструктора)
struct S3 {int a; S3(int aa): a(aa) {} S3() {}};       // POD (пользователь. умолчат. конструктор)
struct S4 {int a; S4(int aa): a(aa) {} S4() = default;}; // POD
struct S5 {virtual void f(); /* ... */};               // не POD (есть виртуальная функция)

struct S6: S1 {};          // POD
struct S7: S0 {int b;};    // POD
struct S8: S1 {int b;};    // не POD (данные в S1 и S8)
struct S9: S0, S1 {};      // POD
```

Чтобы манипулировать объектом «просто как данными» (концепция POD), объект должен:

- не иметь сложной структуры в памяти (например, **vptr**; §3.2.3, §20.3.2),
- не иметь нестандартной семантики копирования,
- иметь тривиальный конструктор по умолчанию.

Естественно, мы должны иметь в виду, что все эти разговоры о простых данных и оптимизациях, связанных с ними, не должны нарушать никаких языковых гарантий. С формальной точки зрения (§ iso.3.9, § iso.9) объект категории POD должен быть

- типа со стандартной раскладкой в памяти,
- типа с тривиальным копированием,
- типа с тривиальным конструктором по умолчанию.

Близкой концепцией является концепция *тривиального типа*, означающей тип с

- тривиальным конструктором по умолчанию и с
- тривиальными операциями копирования и перемещения

С неформальной точки зрения конструктор по умолчанию является тривиальным, если ему не нужно выполнять никакую работу (используйте **=default** для явного определения такой особенности; §17.6.1).

У типа раскладка в памяти стандартная, если он

- не имеет нестатического члена или базового типа с нестандартными раскладками,
- не имеет виртуальных функций (§3.2.3, §20.3.2),
- не имеет виртуального базового типа (§21.3.5),
- не имеет члена с типом ссылки (§7.7),
- не имеет множественных спецификаторов доступа для нестатических членов-данных (§20.5), и
- не препятствует оптимизации раскладки посредством того, что
  - имеет нестатические члены-данных в более чем одном базовом классе или одновременно в производном и базовом классах
  - имеет базовый класс того же типа, что и первый нестатический член-данных.

Фундаментально, стандартная раскладка типа должна иметь эквивалент в языке C и с ней должны быть совместимы общие бинарные интерфейсы приложений языка C++ (ABIs — Application Binary Interfaces).

Тип соответствует концепции тривиального копирования, если только у него нет нетривиальных операций копирования, перемещения и нетривиального деструктора (§3.2.1.2, §17.6). С неформальной точки зрения операция копирования тривиальная, если она может быть выполнена побитовым копированием. Итак, а что делает копирование, перемещение или деструктор нетривиальными?

- Они определены пользователем.
- Их класс имеет виртуальные функции.
- Их класс имеет виртуальный базовый класс.
- Их класс имеет базовый класс или член, которые не являются тривиальными.

Объекты встроенных типов копируются тривиально и имеют тривиальную раскладку. Рассмотрим пример:

```
template<typename T>
    void mycopy(T* to, const T* from, int count);
```

Хотелось бы оптимизировать простейший вариант, в котором тип **T** относится к категории POD. Но тогда вызывать **mycopy()** нужно будет только для типов POD: а можно ли при этом рассчитывать на то же самое, если впоследствии код будет поддерживать другой человек? В общем случае, вряд ли. Конечно, можно опереться и на функцию **std::copy()**, код которой наверняка оптимизирован в этом отношении. Так или иначе, но вот общий и оптимизированный код:

```
template<typename T>
void mycopy(T* to, const T* from, int count)
{
    if (is_pod<T>::value)
        memcpy(to, from, count*sizeof(T));
    else
        for (int i=0; i!=count; ++i)
            to[i]=from[i];
}
```

Здесь **is_pod** — это предикат свойств типа из стандартной библиотеки (§35.4.1), определенный в **<type_traits>**, который позволяет нам задавать вопрос «Относится ли тип **T** к POD»? Самое главное состоит в том, что **is_pod<T>** избавляет нас от необходимости знания точных правил определения типов категории POD.

Обратите внимание на то, что добавление (или удаление) недефолтных конструкторов не влияет на раскладку или производительность (что не было верным в случае C++98).

Если вам все же очень хочется непременно стать экспертом-адвокатом в языке, тщательно изучите концепции раскладки и тривиальности в рамках стандарта (§ iso.3.9, § iso.9), а также попробуйте представить себе их влияние на программистов и на компиляторы. После долгой работы в этом направлении вы, возможно, излечитесь от такого желания.

8.2.7. Битовые поля

Кажется несколько экстравагантным использовать целый байт (типы **char** или **bool**) для представления двоичной (бинарной) переменной — вроде переключателя вкл/выкл — но **char** это наименьший тип для аллоцирования (позиционирования) объекта в памяти и адресации его в языке C++ (§7.2). Но возможно несколько таких мельчайших величин связать в один «пучок» в качестве *битовых полей* в структуре. Член структуры при этом должен специфицироваться числом бит, которое он хочет занимать. Допускаются и неименованные поля (они могут улучшить раскладку в зависимости от особенностей аппаратуры):

```
struct PPN {                    // R6000 номер физической страницы
    unsigned int PFN: 22;       // номер кадра страниц
    int: 3;                     // не используется
    unsigned int CCA: 3;        // алгоритм когерентности кэша
    bool nonreachable: 1;
    bool dirty: 1;
    bool valid: 1;
    bool global: 1;
};
```

Данный пример иллюстрирует еще одно применение битовых полей — это именование частей раскладки, заданной нам извне. Поля имеют интегральный тип или тип перечисления (§6.2.1). Нельзя запросить адрес битового поля. В остальном, они ведут себя так же, как и другие члены. Битовое поле типа **bool** реально может быть представлено одним битом. В ядре операционной системы или в отладчике тип **PPN** мог бы быть использован следующим образом:

```
void part_of_VM_system(PPN* p)
{
    // ...
    if (p->dirty) { // содержимое изменилось
        // копируем на диск
        p->dirty = 0;
    }
}
```

Как ни странно, но применение битовых полей для упаковки нескольких величин в единственный байт не всегда экономит общее пространство памяти. Размер памяти под данные действительно уменьшается, но зато возрастает объем кода, который должен с этими полями работать. К тому же, доступ к **int** или **char** выполняется в типичном случае намного быстрее, чем к битовым полям. Битовые поля служат удобной формой записи для побитовых логических операций (§11.1.1), извлекающих или внедряющих информацию в некоторые части машинных слов.

8.3. Объединения

Объединение (ключевое слово **union**) — это структура, члены которой аллоцируются в памяти по одному и тому же адресу с объемом, достаточным для хранения самого большого типа членов структуры. Естественно, что в каждый момент времени в объединении хранится лишь один из его членов. Для примера рассмотрим таблицу, содержащую имя и значение:

```
enum Type {str, num};

struct Entry {
    char* name;
    Type t;
    char* s;  // используем s если t==str
    int i;    // используем i если t==num
};

void f(Entry* p)
{
    if (p->t == str)
        cout << p->s;
    // ...
}
```

Члены **s** и **i** не могут использоваться одновременно, так что независимые участки памяти под них только зря расходуют память. Это можно исправить, объявив их членами объединения:

```
union Value {
    char* s;
    int i;
};
```

Со стороны языка программирования нет никакого слежения за тем, что именно в данный момент хранится в объединении, так что это зона ответственности самого программиста:

```
struct Entry {
    char* name;
    Type t;
    Value v;  // используем v.s если t==str; используем v.i если t==num
};

void f(Entry* p)
{
    if (p->t == str)
        cout << p->v.s;
    // ...
}
```

Во избежание ошибок следует инкапсулировать объединение так, чтобы соответствие между полем типа и доступом к членам объединения могло гарантироваться (§8.3.2).

Иногда объединения нежелательным образом используются для «приведения типов». Это обычно практикуется программистами, работавшими до этого с языками, в которых отсутствует возможность явного приведения типов, так что им приходилось выполнять такие трюки. К примеру, следующий код выполняет преобразование из **int** в **int***, просто полагаясь на побитовую эквивалентность:

```
union Fudge {
    int i;
    int* p;
};

int* cheat(int i)
{
    Fudge a;
    a.i = i;
    return a.p;      // плохо
}
```

Конечно, это не замена настоящему приведению типов. На некоторых машинах типы int и int* требуют памяти разного размера для хранения, а на других целые переменные не могут иметь нечетного адреса. Таким образом, показанное выше использование объединений для псевдоприведения типов опасно и непереносимо. Вместо этого следует применять операции явного приведения типов (§11.5.2) с тем, чтобы читатель мог четко видеть, что происходит. Пример:

```
Int* cheat2(int i)
{
    return reinterpret_cast<int*>(i);      // опасно
}
```

Здесь, по крайней мере, компилятор предупредит нас о том, что размеры объектов разные.

Применение объединений способствует компактизации кода и может улучшить производительность. Но большинство программ мало что выиграют от применения объединений, которые, кстати, могут порождать ошибки в программе. Я считаю, что объединения на практике применяются неоправданно часто; по-возможности их следует избегать.

8.3.1. Объединения и классы

Во многих нетривиальных объединениях имеется самый большой член, который при этом используется реже остальных членов. В результате большую часть времени память расходуется напрасно.Эту проблему можно решить, заменив объединение набором производных классов (§3.2.2, Глава 20).

С технической точки зрения объединение — это вариант структуры (§8.2), которая, в свою очередь, есть вариант класса (Глава 16). Однако следует иметь в виду, что многие средства работы с классами для объединений не уместны, так что в этом плане для объединений имееется ряд ограничений:

1. Объединение не может иметь виртуальных методов (функций).
2. Объединение не может иметь члены ссылочного типа.
3. Объединение не может иметь базового класса.
4. Если у объединения есть член с определяемым пользователем конструктором, копирующими и перемещающими операциями, тогда эти специальные функции подавляются (§3.3.4, §17.6.4) для такого объединения.
5. Максимум один член объединения может иметь внутриклассный инициализатор (§17.4.4).

Объединение не может выступать в роли базового класса.

Эти ограничения блокируют возможность тончайших ошибок и упрощают реализацию объединений. Последнее очень важно, ибо часто обьединения используют как раз для оптимизации, и мы не хотели бы скомпрометировать эту возможность.

Так как у представленного выше типа **Entry** нет членов с конструкторами, деструкторами и пользовательским присваиванием, то мы можем свободно создавать и копировать объекты типа **Entry**. Например:

```
void f(Entry a)
{
    Entry b = a;
};
```

Если бы такое было допустимо и для более сложных объединений, то могли бы возникнуть трудности с реализацией и ошибки:

```
union U {
    int m1;
    complex<double> m2;    // complex имеет конструктор
    string m3;             // string имеет конструктор (с инвариантом)
};
```

Для копирования **U** нужно было бы решить, какую операцию копирования использовать. Например:

```
void f2(U x)
{
    U u;                   // ошибка: что с умолчательным конструктором?
    U u2 = x;              // ошибка: который копирующий конструктор?
    u.m1 = 1;              // присвоение целочисленному члену
    string s = u.m3;       // катастрофа: читаем из строкового члена
    return;                // ошибка: какие деструкторы вызывать для x, u, и u2?
}
```

Не допускается запись в один член объединения, после чего выполняется попытка чтения другого члена, но многие люди так пишут (часто просто по ошибке). В данном случае копирующий конструктор для **string** будет вызван с неправильным аргументом. Здесь нас спасет то, что тип **U** не пройдет компиляцию. В случае необходимости, пользователь может определить класс, содержащий объединение, и который надлежащим образом обращается с членами, имеющими конструкторы, деструкторы и присваивания (§8.3.2). Он же может предотвратить запись в один член и чтение из другого.

Можно определить не более одного внутриклассного инициализатора члена. В таком случае этот инициализатор будет применяться для инициализации по умолчанию. Например:

```
union U2 {
    int a;
    const char* p {""};
};

U2 x1;          // умолчательно проинициализирован x1.p == ""
U2 x2 {7};      // x2.a == 7
```

8.3.2. Анонимные объединения

Посмотрим, как определить класс, помогающий обойти проблемы с неправильным использованием объединений: рассмотрим новый вариант для **Entry** (§8.3):

```
class Entry2 {      // два альтернативных представления в виде объединения
private:
    enum class Tag {number, text};
    Tag type;       // дискриминант
```

```
    union {          // представление
        int i;
        string s;  // string имеет умолчат. конструктор, копирующие операции и деструктор
    };
public:
    struct Bad_entry {};       // для исключений

    string name;

    ~Entry2();
    Entry2& operator=(const Entry2&);       // требуется из-за варианта string
    Entry2(const Entry2&);
    // ...

    int number() const;
    string text() const;

    void set_number(int n);
    void set_text(const string&);
    // ...
};
```

Я не сторонник get/set-функций (для чтения и записи, соответственно), но здесь мы действительно нуждаемся в выполнении нетривиальных пользовательских действий для каждого из этих видов доступа. Для четкой индикации set-функций я применяю **set**_префикс. Такой стиль именования мне нравится больше, чем иные варианты.

Функции чтения (get-функции) можно определить следующим образом:

```
int Entry2::number() const
{
    if (type!=Tag::number) throw Bad_entry{};
    return i;
};

string Entry2::text() const
{
    if (type!=Tag::text) throw Bad_entry{};
    return s;
};
```

Эти функции проверяют значение «ярлыка» **type**, и если оно соответствует виду доступа, то возвращается ссылка на значение (**i** и **s**, соответственно); иначе генерируется исключение. Такие объединения часто называют *объединениями с ярлыком* (*tagged union*).

Функции доступа для записи (set-функции) также проверяют значение ярлыка **type**:

```
void Entry2::set_number(int n)
{
    if (type==Tag::text) {
        s.~string();      // явное уничтожение string (§ 11.2.4)
        type = Tag::number;
    }
    i = n;
}

void Entry2::set_text(const string& ss)
{
    if (type==Tag::text)
        s = ss;
    else {
        new(&s) string{ss}; // размещающее new: явно конструируем string (§ 11.2.4)
        type = Tag::text;
    }
}
```

Использование объединений заставляет нас применять низкоуровневые средства языка для контроля времени жизни элементов объединения, которыми в иных случаях лучше не пользоваться. Это еще одна причина быть осторожными в вопросах применения объединений.

Обратите внимание на то, что объединение из объявления **Entry2** не имеет имени. Такие объединения принято называть *анонимными* (*anonymous*). По сути анонимное объединение реализуется лишь в виде объекта (участка памяти) и не является типом (именито нет). Доступ к его членам выполняется так же, как и к членам объемлющего класса (не забывая, конечно, о том, что возможен одномоментный доступ лишь к одному члену объединения).

Так как **Entry2** содержит член с пользовательской операцией присваивания (член типа **string**), то операция присваивания для **Entry2** удаляется (§3.3.4, §17.6.4). Если нам нужно присваивание для **Entry2**, то мы должны сами определить **Entry2::operator=()**. Присваивание вбирает в себя сложности как чтения, так и записи, но по сути эквивалентно функциям доступа:

```
Entry2& Entry2::operator=(const Entry2& e) // требуется из-за варианта string
{
    if (type==Tag::text && e.type==Tag::text) {
        s = e.s;  // обычное присваивание строк
    return *this;
    }

    if (type==Tag::text) s.~string(); // явное уничтожение (§ 11.2.4)

    switch (e.type) {
    case Tag::number:
        i = e.i;
        break;
    case Tag::text:
        new(&s)(e .s); // размещающее new: явное конструирование (§ 11.2.4)
        type = e.type;
    }

    return *this;
}
```

Аналогично можно определить конструкторы и операцию перемещения. Нам нужны один или два конструктора для установления соответствия между тегом (ярлыком **type**) и значением. Деструктор же должен обработать ситуацию с членом типа **string**:

```
Entry2::~Entry2()
{
    if (type==Tag::text) s.~string();      // явное уничтожение (§ 11.2.4)
}
```

8.4. Перечисления

Перечисление является типом, предназначенным для хранения набора целых значений, определяемых пользователем (§ iso.7.2). Некоторые из возможных значений перечисления имеют имя и называются *энумераторами*. Пример:

```
enum class Color {red, green, blue};
```

Здесь определяется перечисление **Color** с энумераторами **red**, **green** и **blue**.

Существует два вида перечислений:

1. **enum class** — в перечислениях этого вида имена энумераторов локальны в самом перечислении и их значения не допускают неявного приведения к другим типам.
2. Простые перечисления (просто **enum**) — имена энумераторов видны вне перечисления и допускается неявное приведение их к целым.

В общем случае предпочитайте **enum class**, так как они порождают меньше «сюрпризов».

8.4.1. enum class

С помощью этих ключевых слов определяется перечисление с собственной областью видимости и без неявных приведений типов:

```
enum class Traffic_light {red, yellow, green};
enum class Warning {green, yellow, orange, red}; // уровни предупреждений

Warning a1 = 7;                 // ошибка: нет конверсии int->Warning
int a2 = green;                 // ошибка: green не в области видимости
int a3 = Warning::green;        // ошибка: нет конверсии Warning->int
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) {/* ... */}                 // ошибка: 9 не есть Traffic_light
    if (x == red) {/* ... */}               // ошибка: нет red в области видимости
    if (x == Warning::red) {/* ... */}      // ошибка: x не есть Warning
    if (x == Traffic_light::red) {/* ... */}// OK
}
```

Энумераторы, определеные в обоих перечислениях, не конфликтуют по именам, поскольку каждое имя находится в своей собственной области видимости.

Нижележащий тип (*underlying type*), используемый для представления перечислений, является целым типом (знаковым или беззнаковым; §6.2.4) — по умолчанию это **int**. Его можно задать и явным образом:

```
enum class Warning: int {green, yellow, orange, red}; // sizeof(Warning)==sizeof(int)
```

Если такое решение покажется неэффективным из-за лишней траты памяти, то можно явно назначить тип **char**:

```
enum class Warning: char {green, yellow, orange, red}; // sizeof(Warning)==1
```

По умолчанию энумераторам присваиваются возрастающие целые значения, начиная с нуля, так что справедливо следующее:

```
static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3
```

Снабжая переменную типом **Warning** вместо **int**, мы подсказываем как пользователю, так и компилятору о вероятных сценариях ее использования. Например:

```
void f(Warning key)
{
    switch (key) {
    case Warning::green:
        // что-нибудь делаем
        break;
    case Warning::orange:
        // что-нибудь делаем
        break;
```

```
        case Warning::red:
            // что-нибудь делаем
            break;
        }
}
```

Человек, например, может сразу отметить, что значение **yellow** не использовано, а компилятор может выпустить предупреждающее сообщение на эту тему.

Энумераторы можно инициализировать константным выражением (§10.4) интегрального типа (§6.2.1). Пример:

```
enum class Printer_flags {
    acknowledge=1,
    paper_empty=2,
    busy=4,
    out_of_black=8,
    out_of_color=16,
    //
};
```

Значения для энумераторов перечисления **Printer_flags** выбраны так, что их можно комбинировать при помощи побитовых операций. Например:

```
constexpr Printer_flags operator|(Printer_flags a, Printer_flags b)
{
    return static_cast<Printer_flags>(static_cast<int>(a))|static_cast<int>(b));
}

constexpr Printer_flags operator&(Printer_flags a, Printer_flags b)
{
    return static_cast<Printer_flags>(static_cast<int>(a))&static_cast<int>(b));
}
```

Явные приведения типа необходимы, ибо **class enum** не поддерживает неявных преобразований. Исходя из приведенных определений операций **|** и **&** для типа **Printer_flags**, мы можем написать следующее:

```
void try_to_print(Printer_flags x)
{
    if (x&Printer_flags::acknowledge) {
        // ...
    }
    else if (x&Printer_flags::busy) {
        // ...
    }
    else if (x&(Printer_flags::out_of_black|Printer_flags::out_of_color)) {
        // либо мы не в black, либо не в color
        // ...
    }
    // ...
}
```

Я определил **operator|()** и **operator&()** как **constexpr**-функции (§10.4, §12.1.6), поскольку кому-нибудь может потребоваться использовать эти операции в константных выражениях. Например:

```
void g(Printer_flags x)
{
    switch (x) {
    case Printer_flags::acknowledge:
        // ...
        break;
```

```
        case Printer_flags::busy:
            // ...
            break;
        case Printer_flags::out_of_black:
            // ...
            break;
        case Printer_flags::out_of_color:
            // ...
            break;
        case Printer_flags::out_of_black&Printer_flags::out_of_color:
            // мы не в black и не в color
            // ...
            break;
        }

        // ...
}
```

Имеется возможность объявить перечисление вида **enum class**, отложив полное определение (§6.3) на потом:

```
enum class Color_code: char;      // объявление
void foobar(Color_code* p);       // использование объявления
// ...
enum class Color_code: char {     // определение
    red, yellow, green, blue
};
```

Значение интегрального типа можно явным образом привести к типу перечисления. Результат такого приведения будет неопределен, кроме как в случае, когда значение входит в диапазон нижележащего типа. Пример:

```
enum class Flag: char{x=1, y=2, z=4, e=8};

Flag f0{};                        // f0 получает умолчательное значение 0
Flag f1 = 5;                      // ошибка типа: 5 не есть Flag
Flag f2 = Flag{5};                // ошибка: нет сужающей конверсии в enum class
Flag f3 = static_cast<Flag>(5);   // грубая сила
Flag f4 = static_cast<Flag>(999); // ошибка: 999 это не значение char (не перехвачено)
```

Последние из приведенных присваиваний показывают, почему нет неявных приведений от целого к перечислению; большая часть целых значений не представлены в перечислении.

Каждый энумератор имеет целое значение. Можно извлечь это целое значение явным образом. Пример:

```
int i = static_cast<int>(Flag::y);      // i становится 2
char c = static_cast<char>(Flag::e);    // c становится 8
```

Понятие диапазона значений для перечислений отличается от такового в языке Pascal. Тем не менее, битовые манипуляции, требующие, чтобы были четко определены величины вне набора энумераторов (как в **Printer_flags**), имеют долгую историю в языках C и C++.

Результат **sizeof** для **enum class** равен таковому для нижележащего типа. В частности, если нижележащий тип явно не специфицируется, то размер равен **sizeof(int)**.

8.4.2. «Простой» enum

«Простой» **enum**, грубо говоря, соответствует тому, что предлагал C++ до появления в нем **enum class**. Поэтому примеры его использования широко представлены в програм-

мах, написанных в стиле языков C и C++98. Их энумераторы видны в области видимости, в которой существует само перечисление, а также допускается неявное приведение к величинам некоторого целого типа. Рассмотрим примеры из §8.4.1, из которых ключевое слово **class** устранено:

```
enum Traffic_light {red, yellow, green};
enum Warning {green, yellow, orange, red}; // уровни предупреждений

// ошибка: два определения yellow (с одним значением)
// ошибка: два определения red (с разными значениями)

Warning a1 = 7;                    // ошибка: нет конверсии int->Warning
int a2 = green;                    // OK: green в области видимости и конвертируется в int
int a3 = Warning::green;           // OK: конверсия Warning->int
Warning a4 = Warning::green;       // OK

void f(Traffic_light x)
{
    if (x == 9) {/* ... */}                  // OK (но у Traffic_light нет значения 9)
    if (x == red) {/* ... */}                // ошибка: два red в области видимости
    if (x == Warning::red) {/* ... */}       // OK (Ух!)
    if (x == Traffic_light::red) {/* ... */} // OK
}
```

Нам «повезло», что определение **red** в двух перечислениях в одной и той же области видимости, уберегло нас от более труднолокализуемых ошибок. Рассмотрим следующее «очищение» кода посредством устранения неоднозначностей в простых перечислениях (что довольно просто выполняется в небольших программах, но весьма затруднительно в сегментах кода большого объема):

```
enum Traffic_light {tl_red, tl_yellow, tl_green};
enum Warning {green, yellow, orange, red}; // уровни предупреждений

void f(Traffic_light x)
{
    if (x == red) {/* ... */}                // OK (Ух!)
    if (x == Warning::red) {/* ... */}       // OK (Ух!)
    if (x == Traffic_light::red) {/* ... */} // ошибка: red не есть значение Traffic_light
}
```

Компилятор принимает **x==red**, что почти наверняка является ошибкой. Проникновение имен в объемлющие области видимости *загрязняет* их (*namespace pollution*) и является серьезной проблемой в больших программах (Глава 14).

Вы можете явно специфицировать нижележащий тип для простых перечислений так же, как ранее делали для **enum class**. В этом случае вы можете объявлять перечисления без последующего их определения. Пример:

```
enum Traffic_light: char {tl_red, tl_yellow, tl_green};   // нижележащий тип есть char

enum Color_code: char;      // объявление
void foobar(Color_code* p); // используем объявление
// ...
enum Color_code: char {red, yellow, green, blue};         // определение
```

Если нижележащий тип не специфицирован, то уже нельзя обойтись без определения перечисления. При этом нижележащий тип определяется по довольно сложному алгоритму: если все энумераторы неотрицательные, тогда диапазон для перечисления есть [0:2k-1], где 2k — это наименьшая степень двойки, для которой все энумераторы попада-

ют в диапазон. Если же есть отрицательные энумераторы, то диапазон есть [-2k:2k-1]. Все это задает наименьшее битовое поле, способное вместить значения всех энумераторов (в стандартном представлении с дополнением до двух). Пример:

```
enum E1 {dark, light};                  // диапазон 0:1
enum E2 {a = 3, b = 9};                 // диапазон 0:15
enum E3 {min = -10, max = 1000000};     // диапазон –1048576:1048575
```

Правило для явного приведения целого к простому **enum** такое же, как для **class enum**, за исключением того, что в отсутствие явно определенного нижележащего типа результат приведения неопределен, кроме как в случае попадания значения в диапазон перечисления. Пример:

```
enum Flag {x=1, y=2, z=4, e=8}; // диапазон 0:15

Flag f0{};                      // f0 получает умолчательное значение 0
Flag f1 = 5;                    // ошибка типа: 5 не типа Flag
Flag f2 = Flag{5};              // ошибка: нет явной конверсии из int во Flag
Flag f2 = static_cast<Flag>(5); // OK: 5 в диапазоне Flag
Flag f3 = static_cast<Flag>(zle); // OK: 12 в диапазоне Flag
Flag f4 = static_cast<Flag>(99); // не определено: 99 не в диапазоне Flag
```

Поскольку имеется неявное приведение из простого **enum** в нижележащий тип, то нет необходимости определять операцию **|** с тем, чтобы этот пример работал: **z** и **e** преобразуются в **int**, так что **zle** может быть вычислено. Результат **sizeof** для перечисления такой же, как и результат **sizeof** для нижележащего типа. Если нижележащий тип явно не специфицирован, то это один из интегральных типов, способный вместить диапазон перечисления, и который не больше **sizeof(int)**, кроме как если энумератор не может быть представлен типом **int** или **unsigned int**. Например, **sizeof(e1)** может быть **1** или может быть **4**, но не **8**, на машине, где **sizeof(int)==4**.

8.4.3. Безымянные перечисления

Простой **enum** может быть безымянным. Пример:

```
enum {arrow_up=1, arrow_down, arrow_sideways};
```

Такие перечисления применяются тогда, когда все, что нам нужно, это набор целых констант, а не тип для переменных.

8.5. Советы

[1] Когда важна компактность данных, располагайте члены структур большого размера перед малыми членами; §8.2.1.

[2] Используйте битовые поля для представления данных, навязываемых аппаратурой; §8.2.7.

[3] Не пытайтесь наивно оптимизировать расход памяти, упаковывая несколько значений в один байт; §8.2.7.

[4] Применяйте объединения для экономии памяти и никогда для преобразования типов; §8.3.

[5] Используйте перечисления для представления набора именованных констант; §8.4.

[6] Для минимизации сюрпризов используйте **class enum** вместо «простого» **enum**; §8.4.

[7] Определяйте операции над перечислениями ради простоты и безопасности; §8.4.1.

9

Операторы

Программист — это машина
по переработке кофеина в код.
— Программист

- Введение
- Сводка операторов
- Объявления как операторы
- Операторы выбора

  Оператор **if**; Оператор **switch**; Объявления в ветвях **switch**-оператора
- Операторы итераций

  Оператор range-**for**; Оператор **for**; Оператор **while**; Оператор **do**; Выход из цикла
- Оператор **goto**
- Комментарии и отступы
- Советы

9.1. Введение

В языке C++ имеются все традиционные операторы. По сути, все, что является особо интересным или же сложным, сосредоточено в объявлениях и в выражениях. Со своей стороны, объявление является оператором, а выражение превращается в оператор, если его завершить точкой с запятой.

В отличие от выражения оператор не вырабатывает результирующих значений. Операторы предназначены для формулирования порядка выполнения действий. Например:

```
a = b+c;        // оператор-выражение
if (a==7)       // оператор
    b = 9;      // только и если только a==9
```

Здесь **a=b+c** вычисляется раньше, чем начинает работу оператор **if**, что очевидно каждому. Компилятор может впоследствии реорганизовывать порядок исполнения машинных команд с тем, чтобы повысить эффективность, но при этом сохраняет все значимые результаты работы программы.

9.2. Сводка операторов

Представим сводную информацию об операторах языка C++:

оператор:
    объявление
    выражение$_{opt;}$
    {*список-операторов*$_{opt}$}

try {*список-операторов* $_{opt}$} *список-обработчиков*
case *const-выражение*: *оператор*
default: *оператор*
break ;
continue ;

return *выражение* $_{opt}$;

goto *id*; *id*:
оператор

оператор-выбора
оператор-итераций

оператор-выбора:
if (*условие*) *оператор*
if (*условие*) *оператор* **else** *оператор*
switch (*условие*) *оператор*

оператор-итерации:
while (*условие*) *оператор*
do *оператор* **while** (*выражение*);
for (*for-init-statement* *условие* $_{opt}$; *выражение*) *оператор*
for (*for-объявление: выражение*) *оператор*

список-операторов:
оператор список-операторов $_{opt}$

условие:
выражение
спецификатор-типа декларатор = *выражение*
спецификатор-типа декларатор {*выражение*}

список-обработчиков:
обработчик список-обработчиков $_{opt}$

обработчик:
catch (*объявление-исключение*) {*список-операторов* $_{opt}$}

Можно сказать, что точка с запятой сама по себе является оператором — *пустым оператором*.

Последовательность (возможно пустая) операторов внутри фигурных скобок называется *блоком* или *составным оператором*. Имя, объявленное внутри блока, выходит из области видимости по выходу из блока.

Объявление является оператором; отсутствуют операторы присваивания и вызова функций; присваивания и вызовы функций являются выражениями.

Секция инициализации в операторе **for** является объявлением или же выражением-оператором. Оба варианта оканчиваются точкой с запятой.

Объявление в секции инициализации оператора **for** должно быть объявлением одной неинициализированной переменной.

Операторы для обработки исключений, try-блоки, рассматриваются в §13.5.

9.3. Объявления как операторы

Объявление является оператором. Если только переменная не объявляется как **static**, то ее инициализатор выполняется во время прохода потока исполнения через объявление (см. также §6.4.2). Причина, по которой объявления допускаются всюду, где могут присутствовать операторы, заключается в желании минимизировать проблемы, порождаемые неинициализированными переменными, а также для лучшей локализации в коде. Редко когда возникает необходимость в объявлении переменной до того момента, как для нее находится подходящее инициализирующее значение. Например:

```
void f(vector<string>& v, int i, const char* p)
{
    if (p==nullptr) return;
    if (i<0 || v.size()<=i)
        error("bad index");
    string s = v[i];
    if (s == p) {
        // ...
    }
    // ...
}
```

Возможность объявлять переменные после исполняемого кода весьма существенна для многих констант и для стилей программирования с единичными присваиваниями, когда значение переменной не изменяется после ее инициализации. Для типов, определяемых пользователем, возможность отложить определение переменной до момента возникновения подходящего инициализатора улучшает производительность кода. Пример:

```
void use()
{
    string s1;
    s1 = "The best is the enemy of the good.";
    // ...
}
```

Этот код порождает инициализацию умолчательным значением (пустая строка), после чего идет присваивание. Это может снизить эффективность по сравнению с инициализацией целевым значением:

```
string s2 {"Voltaire"};
```

Причиной объявления переменной без инициализатора является необходимость вычисления целевого значения посредством отдельного оператора. Здесь резонным примером служит ввод значений с того или иного устройства ввода:

```
void input()
{
    int buf[max];
    int count = 0;
    for (int i; cin>>i;) {
        if (i<0) error("unexpected negative value");
        if (count==max) error("buffer overflow");
        buf[count++] = i;
    }
    // ...
}
```

Для этого примера лучшей альтернативой было бы применение стандартной библиотечной функции **push_back()** (§3.2.1.3, §13.6, §31.3.6).

9.4. Операторы выбора

Значение может быть протестировано (на его величину) **if**-операторами или **switch**-операторами:

if (*условие*) *оператор*
if (*условие*) *оператор* **else** *оператор*
switch (*условие*) *оператор*

Условие (*condition*) может быть выражением или объявлением (§9.4.3).

9.4.1. Оператор if

В управляющем операторе **if** первый из подчиненных операторов выполняется, когда условие истинно (**true**), а второй подчиненный оператор (если он присутствует) — в противном случае. Если вычисление условия приводит к результату, отличному от логического типа, то оно по-возможности неявно преобразуется к типу **bool**. Отсюда следует, что любые арифметические выражения или выражения с указателями могут применяться в качестве условий. Например, если **x** целое, тогда

```
if (x) // ...
```

означает

```
if (x != 0) // ...
```

Для указателя **p** запись

```
if (p) // ...
```

является аналогом вопроса «На самом ли деле указатель **p** указывает на действительный объект?» и эквивалентна следующему

```
if (p != nullptr) // ...
```

Нужно отметить, что «простой» **enum** может быть неявно приведен к целому типу, а затем к типу **bool**, в то время как **enum class** этого не допускает (§8.4.1). Пример:

```
enum E1 {a, b};
enum class E2 {a, b};

void f(E1 x, E2 y)
{
    if (x)              // OK
        // ...
    if (y)              // ошибка: нет конверсии в bool
        // ...
    if (y==E2::a)       // OK
        // ...
}
```

Логические операции

```
&& || !
```

часто применяются внутри условий. Операции **&&** и **||** не вычисляют своего второго аргумента в случае отсутствия необходимости (когда результат уже предопределен первым аргументом). Пример:

```
if (p && 1<p->count) // ...
```

Здесь к выражению **1<p->count** приступают только в том случае, когда **p** не равен **nullptr**.

В случае выбора из двух альтернатив, каждая из которых производит значение, предпочтение следует отдать условному выражению (§11.1.3), так как оно яснее отражает намерение, чем **if**-оператор. Пример:

```
int max(int a, int b)
{
    return (a>b)?a:b;    // return the larger of a and b
}
```

Любое имя может использоваться лишь в области видимости, в котором оно объявлено. В частности, его нельзя использовать в другой ветви **if**-оператора. Пример:

```
void f2(int i)
{
    if (i) {
        int x = i+2;
        ++x;
        // ...
    }
    else {
        ++x;        // ошибка: x не в области видимости
    }
    ++x;            // ошибка: x не в области видимости
}
```

Ветвь **if**-оператора не может состоять лишь из объявления. Если нужно объявить имя в ветви **if**-оператора, его нужно погрузить в блок (§9.2). Например:

```
void f1(int i)
{
    if (i)
        int x = i+2;    // ошибка: объявление в ветви if-оператора
}
```

9.4.2. Оператор switch

С помощью **switch**-операторов производится выбор из набора альтернатив. Выражения в **case**-ветвях должны быть константными выражениями интегральных типов или же перечислениями. Например:

```
void f(int i)
{
    switch (i) {
    case 2.7:       // ошибка: вещественное число
        // ...
    case 2:
        // ...
    case 4-2:       // ошибка: 2 используется дважды в ветвях switch-оператора
        // ...
};
```

Альтернативно **switch**-оператор можно записать в виде набора **if**-операторов. Например, следующий код

```
switch (val) {
case 1:
    f();
    break;
case 2:
    g();
    break;
default:
    h();
    break;
}
```

эквивалентен коду с операторами **if**:

```
if (val == 1)
    f();
else if (val == 2)
    g();
else
    h();
```

Смысл один и тот же, но первый вариант (с оператором **switch**) более предпочтителен, ибо в нем семантика (сравнение единственного значения с набором констант) отражена точнее. Поэтому в более сложных случаях читать и понимать код со **switch**-оператором легче. Кроме того, компилятор может сгенерировать более эффективный код (может быть применена таблица переходов).

Не забывайте о том, что каждая отдельная ветвь в **switch**-операторе подлежит явному завершению, или же будут далее выполняться и другие (последующие) ветви. Рассмотрим следующий пример:

```
switch (val) {              // осторожно
case 1:
     cout << "case 1\n";
case 2:
     cout << "case 2\n";
default:
     cout << "default: case not found\n";
}
```

В случае **val==1**, вывод будет следующим (неожиданным для непосвященных):

```
case 1
case 2
default: case not found
```

Будет хорошей идеей комментировать редкие случаи, когда мы намеренно собираемся «провалиться» в соседнюю ветвь **switch**-оператора. Тогда такие случаи без комментария будет легко обнаружить как ошибочные. Например:

```
switch (action) {           // обработка пары (action, value)
case do_and_print:
     act(value);
     // отсутствует break: проход в print
case print:
     print(value);
     break;
// ...
}
```

Чаще всего ветви **switch**-операторов завершают с помощью **break**, но можно это делать и при помощи **return** (§10.2.1).

Когда следует внедрять в **switch**-оператор **default**-ветвь? Нет однозначного ответа на этот вопрос. Здесь можно выделить два противоположных случая. Один соответствует наиболее частому варианту — его мы и относим к **default**-ветви. Другой — когда **default**-ветвь относится к исключительному случаю для обработки ошибочных ситуаций. Существуют и резоны вообще отказаться от **default**-ветви — когда с помощью **switch**-оператора обрабатываются все энумераторы перечисления. В таком случае **default**-ветвь с большой вероятностью будет просто ошибкой:

```
enum class Vessel {cup, glass, goblet, chalice};

void problematic(Vessel v)
{
     switch (v) {
     case Vessel::cup:      /* ... */   break;
     case Vessel::glass:    /* ... */   break;
     case Vessel::goblet:   /* ... */   break;
     }
}
```

Такая ошибка легко может возникнуть при добавлении нового энумератора в процессе поддержки кода.

9.4.2.1. Объявления в ветвях switch-оператора

Допускается объявлять переменные в блоке **switch**-оператора. При этом не допускается обход инициализации. Например:

```
void f(int i)
{
    switch (i) {
    case 0:
        int x;          // не инициализировано
        int y = 3;      // ошибка: возможный обход объявления
        string s;       // ошибка: возможный обход объявления
    case 1:
        ++x;            //ошибка: использование неинициализированного объекта
        ++y;
        s = "nasty!";
    }
}
```

Здесь, если **i==1**, поток управления пройдет мимо инициализации переменных **y** и **s**, и по этой причине функция **f()** не пройдет компиляцию. К несчастью, переменная типа **int** не требует инициализации и ее объявление не является синтаксической ошибкой. Тем не менее, это фактическая ошибка, ибо возможно чтение неинициализированной переменной. К сожалению, компиляторы или вообще пропускают такие случаи, или реагируют на них лишь предупреждением. Поэтому совет — избегайте неинициализированных переменных (§6.3.5.1).

Если уж нам действительно нужно объявить переменную внутри **switch**-оператора, то ее нужно поместить в блок и им ограничить область видимости. Смотри, к примеру, **prim()** в §10.2.1.

9.4.3. Объявления в условиях

Чтобы избежать нечаянного применения ненадлежащей переменной, желательно объявлять переменные в наиболее узких областях видимости. В частности, желательно откладывать определение локальных переменных до того момента, когда возникнут разумные величины для их инициализации.

Применяя высказанные принципы, помещаем объявление переменной прямо в условие **if**-оператора. Рассмотрим вот такой код:

```
if (double d = prim(true)) {
    left /= d;
    break;
}
```

Здесь **d** объявляется и инициализируется прямо в условии, в котором эта величина и проверяется на истинность. Область видимости переменной **d** начинается в точке ее объявления и распространяется до конца оператора, который контролируется данным **if**. Так что если бы здесь была и **else**-ветвь, то переменная **d** была бы доступна и в этой ветви.

Традиционной альтернативой служит решение, когда **d** объявляется перед условием. Однако ж, это открывает возможность использования переменной **d** до ее инициализации и после того, как заканчивается зона ее разумного применения:

```
double d;
// ...
d2 = d; // oops!
// ...
if (d = prim(true)) {
    left /= d;
    break;
}
// ...
d = 2.0; // две несвязанные манипуляции
```

Объявление в условии должно вводить и инициализировать единственную переменную или константу.

9.5. Операторы итераций

Операторы, реализующие итерации — это операторы цикла **for**, **while** и **do**:

while (*условие*) *оператор*
do *оператор* **while** (*выражение*);
for (*for-init-statement условие*$_{opt}$; *выражение*$_{opt}$) *оператор*
for (*for-объявление: выражение*) *оператор*

Здесь *for-init-statement* — это либо объявление, либо выражение-оператор. Оба этих варианта оканчиваются точкой с запятой (что очевидно).

Выполнение оператора *statement*, котролируемого **for**-оператором, повторяется до тех пор, пока условие *condition* не станет ложным (**false**) или программист не осуществит силовой выход из цикла посредством **break**, **return**, **throw** или **goto**.

Более сложные циклы можно организовать посредством алгоритмов и лямбда-выражений (§11.4.2).

9.5.1. Оператор range-for

Простейший оператор цикла — это оператор range-**for**; он просто предоставляет программисту доступ к каждому элементу некоторого *диапазона* (*range*). Пример:

```
int sum(vector<int>& v)
{
    int s = 0;
    for (int x : v)
        s+=x;
    return s;
}
```

Запись **for(int x : v)** может быть прочитана так: «для каждого элемента **x** из диапазона **v**». Элементы из **v** выбираются по-порядку от первого до последнего.

Область видимости элемента **x** ограничивается этим **for**-оператором.

Выражение, стоящее после двоеточия, должно обозначать последовательность (диапазон), то есть оно должно порождать сущность, для которой можно вызывать **v.begin()**, **v.end()**, или же **begin(v)** и **end(v)** для получения итераторов (§4.5):

1. Компилятор сначала ищет члены (методы) **begin** и **end** и пытается их использовать. Если **begin** и **end** не формируют диапазон (например, **begin** является членом данных, а не функцией), то применение оператора range-**for** ошибочно.
2. В противном случае (не находит **begin** и **end**) он пытается искать их в объемлющей области видимости. Если они там не находятся или же их нельзя использовать (например, **begin** не принимает аргумента типа последовательности), то оператор range-**for** порождает ошибку компиляции.

Для встроенного массива **T v[N]** компилятор вместо **begin(v)** и **end(v)** использует **v** и **v+N**. Заголовочный файл **<iterator>** определяет **begin(c)** и **end(c)** для встроенных массивов и для всех контейнеров стандартной библиотеки. Для определяемых пользователем последовательностей можно определить **begin()** и **end()** в той же манере, как это сделано для контейнеров стандартной библиотеки (§4.4.5).

Контролируемая оператором переменная, то есть **x** в данном примере, ссылающаяся на текущий элемент, эквивалентна выражению ***p** в эквивалентном **for**-операторе:

```
int sum2(vector<int>& v)
{
    int s = 0;
    for (auto p = begin(v); p!=end(v); ++p)
        s+=*p;
    return s;

}
```

Если нужно модифицировать элементы в цикле range-**for**, то переменная, обозначающая элементы последовательности, должна быть ссылкой. Вот пример, в котором мы инкрементируем каждый элемент контейнера **vector**:

```
void incr(vector<int>& v)
{
    for (int& x : v)
        ++x;
}
```

Ссылки также хороши в случаях, когда элементы последовательности имеют большой размер, так что их копирование накладно. Пример:

```
template<class T> T accum(vector<T>& v)
{
    T sum = 0;
    for (const T& x : v)
        sum += x;
    return sum;
}
```

Цикл range-**for** имеет весьма простую структуру. В нем вы не сможете обратиться к двум элементам в одно и то же время, и не сможете эффективно проходить два диапазона (последовательности) одновременно. Для этого вам придется обратиться к более общему оператору **for**.

9.5.2. Оператор for

Переменная цикла **for**, условие окончания цикла и выражение, которое обновляет переменную цикла, присутствуют в явном виде в одном месте. Вот пример:

```
void f(int v[], int max)
{
    for (int i = 0; i!=max; ++i)
        v[i] = i*i;
}
```

Это эквивалентно следующему:

```
void f(int v[], int max)
{
    int i = 0;          // переменная цикла
    while (i!=max) {    // условие окончания
```

```
            v[i] = i*i;         // выполняем тело цикла
            ++i;                // инкрементируем переменную цикла
        }
}
```

Переменную цикла можно объявлять непосредство в инициализирующей части оператора **for**; ее область видимости распространяется до конца оператора **for**.

Когда не очевиден подходящий тип для переменной цикла, можно использовать ключевое слово **auto**:

```
for (auto p = begin(c); c!=end(c); ++p) {
    // используем итератор p для элементов в контейнере c ...
}
```

Если нужно запомнить значение индекса после выполнения цикла, то индексную переменную следует объявить вне цикла **for** (см., например, §9.6).

Инициализирующая часть в операторе **for** может оставаться пустой, если в такой инициализации нет необходимости.

Если опущена часть, отвечающая за обновление переменной цикла, то тогда ее наращивание нужно реализовать в теле цикла. Если ваш итерационный код имеет более сложную логику, чем просто «ввести переменную цикла, проверить условие окончания, обновить переменную цикла», то в таком случае его лучше выразить посредством оператора **while**. В то же время, и с оператором **for** в не слишком стандартных вариантах тоже возможны элегантные решения:

```
for (string s; cin>>s;)
    v.push_back(s);
```

Здесь чтение строки **s** комбинируется с тестом на окончание цикла — **cin>>s** — так что можно обойтись вообще без отдельной переменной цикла. Дополнительно, оператор цикла **for** помогает ограничить область видимости текущей переменной **s** собственно циклом.

Часто оператор **for** применяется в варианте, когда явным образом не формулируется условие окончания цикла:

```
for (;;) {// "всегда"
    // ...
}
```

Некоторые программисты, тем не менее, считают показанный код несколько «темноватым», и предпочитают другой вариант:

```
while(true) {   // "всегда"
    // ...
}
```

9.5.3. Оператор while

Оператор **while** повторяет итерации (выполнение своего контролируемого оператора) до тех пор, пока условие не перестанет быть истинным (то есть не станет **false**). Например:

```
template<class Iter, class Value>
Iter find(Iter first, Iter last, Value val)
{
    while (first!=last && *first!=val)
        ++first;
    return first;
}
```

Я предпочитаю оператор цикла **while** оператору цикла **for** тогда, когда отсутствует очевидная переменная цикла или когда наращивание такой переменной естественнее всего выполнять где-нибудь в середине тела цикла.

Оператор **for** (§9.5.2) легко переписать в эквивалентный оператор **while**, и наоборот.

9.5.4. Оператор do

Оператор **do** аналогичен оператору **while** за исключением того, что в нем условие прекращение цикла расположено после тела цикла. Пример;

```
void print_backwards(char a[], int i)     // i должно быть положительным
{
    cout << '{';
    do {
        cout << a[--i];
    } while (i);
    cout << '}';
}
```

Это может быть вызвано вот так: **print_backwards(s, strlen(s));** при этом возможна ужасная ошибка. А что если **s** будет пустой строкой?

По моему опыту, оператор **do** — «хороший» источник ошибок и недоразумений. Причина состоит в том, что его тело выполняется прежде, чем просматривается условие его выполнения. В итоге, чтобы код работал корректно, требуется, чтобы перед циклом была реализована та или иная форма проверки на допустимость выполнения тела последующего цикла. Вообще-то неплохо, когда такое важное условие формулируется где-то пораньше, чтобы его можно было легко видеть и принимать во внимание. Как следствие, я предпочитаю избегать применения оператора цикла **do**.

9.5.5. Выход из цикла

Если опущено условие выхода из цикла (*condition*) в операторах **for**, **while** или **do**, то цикл будет выполняться сколь угодно долго, пока не сработает силовой выход посредством **break**, **return** (§12.1.4), **goto** (§9.6), **throw** (§13.5) или каким-нибудь менее очевидным способом, вроде вызова **exit()** (§15.4.3). Оператор **break** осуществляет выход из наиболее близкого (в случае вложенных операторов) оператора **switch** (§9.4.2) или оператора итераций. Например:

```
void f(vector<string>& v, string terminator)
{
    char c;
    string s;
    while (cin>>c) {
        // ...
        if (c == '\n') break;
        // ...
    }
}
```

Оператор **break** используется тогда, когда нам нужно выйти где-нибудь из «середины» цикла. Но все же лучше по-возможности формулировать полное условие завершения цикла в форме *condition* в операторах **while** или **for**.

Иногда не нужно полностью выходить из цикла, а нужно лишь пропустить выполнение части тела цикла и перейти к следующей итерации. Этому служит оператор **continue**. Пример:

```
void find_prime(vector<string>& v)
{
    for (int i = 0; i!=v.size(); ++i) {
        if (!prime(v[i]) continue;
        return v[i];
    }
}
```

После оператора **continue** выполняется инкрементирующая часть цикла (если присутствует в операторе), а за ней — проверяется условие (при его наличии в операторе). Таким образом, **find_prime()** можно эквивалентно представить в таком виде:

```
void find_prime(vector<string>& v)
{
    for (int i = 0; i!=v.size(); ++i) {
        if (!prime(v[i]) {
            return v[i];
        }
    }
}
```

9.6. Оператор goto

В языке C++ пресловутый оператор **goto** присутствует:

goto *id; id:*
оператор

У оператора **goto** имеются некоторые стандартные варианты применения в высокоуровневом коде, но он более полезен тогда, когда код пишет программа, а не человек; например, **goto** используется в парсерах, генерируемых при автоматическом разборе грамматики.

Областью видимости метки служит функция, в которой она находится (§6.3.4). Отсюда следует, что вы можете использовать **goto** для прыжков внутри и вовне блоков. Здесь имеются следующие ограничения: нельзя «прыгать» в обход инициализации и в обработчик исключений (§13.5).

Одним из немногих случаев разумного применения оператора **goto** в обычном коде является необходимость осуществить выход наружу из вложенного цикла или **switch**-оператора (**break** порождает выход только из ближайшего цикла).

Например:

```
void do_something(int i, int j)
    // делаем что-нибудь с двумерной матрицей matrix по имени mn
{
    for (i = 0; i!=n; ++i)
        for (j = 0; j!=m; ++j)
            if (nm[i][j] == a)
                goto found;
    // не найдено
    // ...
found:
    // nm[i][j] == a
}
```

Здесь **goto** просто реализует бросок вперед наружу из циклов. Такое применение **goto** относительно безопасно и не смущает читателя кода.

9.7. Комментарии и отступы

Разумное применение комментариев и отступов в коде может сделать чтение и понимание программного кода занятием более приятным, чем в отсутствие оных. На практике широко используется несколько разных стилей внесения отступов в код программы. Я не вижу никаких разумных доказательств преимущества одного стиля над другими. Это же верно и в отношении комментариев.

Комментарии можно выполнить так, что понимание программы только ухудшится. Компиляторы совершенно не понимают комментариев, и не существует автоматических средств проверить, что

- комментарий имеет смысл,
- описывает данную программу,
- комментарий не устарел.

Во многих программах комментарии неполные, противоречивые и даже просто неверные. Плохие комментарии хуже, чем их отсутствие.

Если что-то может быть ясным образом отражено непосредственно в коде программы, то так и нужно сделать. Это замечание нацелено на комментарии следующего вида:

```
// переменная "v" должна быть проинициализирована

// переменная "v" должна использоваться только функцией "f()"

// вызовите функцию  "init()" вызовом других функций из этого файла

// вызовите функцию  "cleanup()" в конце программы

// не используйте функцию "weird()"

// функция "f(int ...)" принимает два или три аргумента
```

Такие комментарии при надлежащем использовании C++ просто не нужны.

Если что-то ясно отражено в самом коде, нет никакой нужды дублировать это в комментарии. Пример:

```
a = b+c; // a становится b+c
count++; // инкрементируем счетчик
```

Такие комментарии даже хуже, чем просто ненужное повторение. Они увеличивают объем кода, который читатель должен просмотреть, они делают менее ясной и наглядной структуру кода, и могут быть просто неверными. Но следует отметить, что в учебных целях они используются часто, в том числе и в настоящей книге. В этом, а также во многих иных отношениях реальный код отличается от учебного кода.

Хороший комментарий должен формулировать идею о том, для чего предназначен данный кусок кода, в то время как сам код отражает только то, что собственно он фактически делает. Предпочтительно, чтобы комментарий формулировался на максимально возможном уровне абстракции, чтобы читатель мог быстро вникнуть в него, не зарываясь глубоко в дебри мелких и многочисленных деталей.

Вот мои предпочтения в этом вопросе:

- Комментарий к каждому файлу с исходным кодом должен обяснять, что общего между собой имеют располагающиеся в этом файле объявления, ссылки на источники, имя программиста, намеки на возможности поддержки и т.д.
- Комментарии к каждому классу, шаблону и пространству имен
- Комментарии к каждой нетривиальной функции о ее предназначении, использованном алгоритме и о предположениях о контексте ее работы

- Комментарий к каждой глобальной переменной или переменной из пространства имен (а также к константам)
- Комментарии к неясным или непереносимым частям кода
- И очень мало иных комментариев

Например:

```
// tbl.c: Реализация символьной таблицы.

/*
    Gaussian замена с частичным отражением.
    См. Ralston: "A first course ..." pg 411.
*/

// scan(p, n, c) требует чтобы p указывал на массив из минимум n элементов

// sor t(p, q) сортирует элементы последовательности [p: q) используя < для сравнения.

// Пересмотрено обращение с некорректными датами. Bjarne Stroustr up, Feb 292013
```

Хорошо отобранные и хорошо написанные комментарии являются необходимым атрибутом любой хорошей программы. Написание хороших комментариев может быть столь же сложной задачей, что и написание самой программы. Это искусство, которое нужно лелеять и культивировать.

Отметим, что стиль комментирования в виде /* */ не допускает вложений. Пример:

```
/*
    удаляем дорогостоящую проверку
    if (check(p, q)) error("bad p q") /* никогда не встречается  */
*/
```

Этот вложенный комментарий приведет к ошибке компиляции.

9.8. Советы

[1] Не объявляйте переменную до того момента, когда для нее возникнет подходящее инициализирующее значение; §9.3, §9.4.3, §9.5.2.

[2] По-возможности предпочитайте **switch**-оператор набору **if**-операторов; §9.4.2.

[3] По-возможности предпочитайте range-**for** оператор **for**-оператору; §9.5.1.

[4] Предпочитайте **for**-оператор **while**-оператору там, где имеется очевидная переменная цикла; §9.5.2.

[5] Предпочитайте **while**-оператор **for**-оператору, если нет очевидной переменной цикла; §9.5.3.

[6] Избегайте **do**-операторов; §9.5.

[7] Избегайте **goto**; §9.6.

[8] Делайте комментарии полезными и сохраняйте их актуальными; §9.7.

[9] Не сообщайте в комментариях того, что и так ясно из кода; §9.7.

[10] В комментариях отражайте намерения; §9.7.

[11] Применяйте согласованный стиль отступов; §9.7.

10

Выражения

Программирование это как секс:
может дать некоторый конкретный результат,
но занимаемся мы этим по другой причине.
— апология Ричарду Фейнману

- Введение
- Проект «Настольный калькулятор»
  Парсер; Низкоуровневый ввод; Обработка ошибок; Главная программа; Заголовочные файлы; Аргументы командной строки; Замечание по стилю
- Сводка операций
  Результаты; Порядок вычислений; Приоритет операций; Временные объекты
- Константные выражения
  Символические константы; Ключевое слово **const**; Литеральные типы; Аргументы-ссылки; Адресные константные выражения
- Неявное приведение типов
  Продвижения; Преобразования; Обычные арифметические преобразования
- Советы

10.1. Введение

В данной главе в деталях рассматриваются выражения языка C++. В этом языке присваивание является выражением, вызов функции также выражение, выражением является и конструирование объекта, а также многие иные операции, выходящие далеко за рамки традиционных арифметических выражений. Чтобы дать ясное представление о том, как в языке C++ используются выражения и каков контекст их применения, я предлагаю рассмотреть простой проект «Настольный калькулятор». Затем я представлю полный список операций и их смысл по отношению ко встроенным типам. Операции, требующие более интенсивного рассмотрения и более скрупулезного изучения, отложены до Главы 11.

10.2. Проект «Настольный калькулятор»

Рассмотрим проект по созданию программы «настольного калькулятора», реализующего четыре арифметических действия над числами с плавающей запятой. Пользователь калькулятора также сможет определять и использовать переменные. Вот иллюстрация того, как на ввод пользователя

```
r = 2.5
area = pi * r * r
```

(число **pi** является предопределенным) программа калькулятора выдаст

```
2.5
19.635
```

где **2.5** — это результат первой строки ввода, а **19.635** — результат для второй строки ввода.

Калькулятор состоит из четырех основных частей: парсера, функции ввода, таблицы символов и главной (стартовой) программы. По сути, это компилятор в миниатюре, в котором парсер выполняет синтаксический анализ, функция ввода ответственна за ввод информации от пользователя и лексический анализ, таблица символов содержит постоянную информацию, а главная программа собирает все воедино, а также выводит результаты и сообщения об ошибках. Конечно, можно было бы добавить в калькулятор много другой полезной функциональности, но код и так достаточно большой и, кроме того, такое расширение мало что дало бы дополнительно в отношении изучения языка C++.

10.2.1. Парсер

Представим грамматику языка, с которым имеет дело калькулятор:

```
program:
    end                          // end это конец ввода
    expr_list end

expr_list:
    expression print             // print это новая строка или точка с запятой
    expression print expr_list

expression:
    expression + term
    expression - term
    term

term:
    term / primary
    term * primary
    primary

primary:
    number                       // number это литерал с плавающей запятой
    name                         // name это идентификатор
    name = expression
    - primary
    ( expression )
```

Иными словами можно сказать, что программа состоит из последовательности выражений, разделенных точками с запятой. Базовыми единицами выражений служат числа, имена и знаки операций *, /, +, - (как унарный, так и бинарный) и знак = (операция присваивания). Имена не требуется объявлять перед употреблением.

Я применяю рекурсивно нисходящий стиль синтаксического анализа; это простой и прямолинейный стиль анализа «сверху-вниз». Для языка C++, в котором вызов функции относительно недорог, он также и эффективен. Для любого грамматического результата имеется функция, которая вызывает остальные функции. Терминальные символы (например, **end**, **number**, + и -) распознаются лексическим анализатором, а нетерминальные символы распознаются функциями синтаксического анализа, **expr()**, **term()** и **prim()**. Как только

оба операнда (под)выражения известны, выражение вычисляется; в реальном компиляторе в этой точке генерируется код.

Для ввода парсер использует **Token_stream**, который инкапсулирует чтение символов и составляет из них объекты типа **Token**. Здесь **Token** (лексема) это пара {вид лексемы, значение}, такая как например {**number**,**123.45**}, где **123.45** — это значение с плавающей запятой. Основной части парсера нужно знать лишь имя **ts**, для **Token_stream**, и как получить **Token** с его помощью. Каждый следующий **Token** получается посредством вызова **ts.get()**. Самый последний прочитанный **Token** (текущий) запрашивается с помощью вызова **ts.current()**. Дополнительно, **Token_stream** скрывает истинный источник символов: они могут поступать непосредственно с пользовательского ввода в стандартный объект **cin**, из командной строки или из любого иного потока ввода (§10.2.7).

Вот определение типа **Token**:

```
enum class Kind : char {
    name, number, end,
    plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
};

struct Token {
    Kind kind;
    string string_value;
    double number_value;
};
```

Представление лексем целым числом эффективно, удобно и может помочь при поиске ошибок. Это работает до тех пор, пока никакой символ из входного потока не имеет числового значения, равного числовому значению энумератора из представленного выше перечисления (в известных мне наборах символов нет печатных символов со значениями из одной десятичной цифры).

Тип **Token_stream** выглядит следующим образом:

```
class Token_stream {
public:
    Token get();                    // прочитать и вернуть следующую лексему
    const Token& current();         // недавно прочитанная лексема
    // ...
};
```

Здесь представлен интерфейс класса. Реализация рассматривается в §10.2.2.

Каждая функция парсера принимает аргумент типа **bool** (§6.2.2), именуемый **get**, указывающий, требуется ли функции вызывать **Token_stream::get()** для получения следующей лексемы. Каждая функция парсера вычисляет свое выражение и возвращает значение. Функция **expr()** выполняет сложение и вычитание. Она состоит из единственного цикла, в котором разыскиваются операнды для сложения или вычитания:

```
double expr(bool get)    // сложение и вычитание
{
    double left = term(get);

    for (;;) { // "вечно"
        switch (ts.current().kind) {
        case Kind::plus:
            left += term(true);
            break;
        case Kind::minus:
            left -= term(true);
            break;
```

```
            default:
                return left;
            }
        }
    }
```

Эта функция фактически мало что делает сама. В манере распорядительных функций из больших программ она вызывает другие функции для выполнения работы.

В операторе **switch** (§2.2.4, §9.4.2) значение условного выражения сравнивается с набором констант. Выход из **switch**-оператора реализуется посредством операторов **break**. В случае отсутствия совпадения с константами выполняется **default**-ветвь.

Заметим, что выражение вроде **2-3+4** вычисляется как **(2-3)+4** в соответствии с грамматикой.

Забавная нотация **for(;;)** формирует бесконечный цикл; можете транскрибировать это как «всегда» (§9.5); альтернативной реализацией является **while(true)**. Оператор **switch** выполняется снова и снова до тех пор, пока не встретится что-нибудь отличное от **+** или **-**. Тогда исполнится оператор **return** из **default**-ветви.

Операции **+=** и **-=** используются для обработки случаев сложения или вычитания; без изменения смысла программы можно было бы использовать **left=left+term(true)** и **left=left-term(true)**. Однако выбранный нами вариант и короче, и точнее описывает наши намерения. Каждая операция присваивания — это отдельная лексема, так что **a + = 1;** является синтаксической ошибкой из-за пробела между знаками **+** и **=**.

В языке C++ для бинарных операций

```
+ - */ % & | ^ << >>
```

присутствуют и соответствующие операции присваивания:

```
= += -= *= /= %= &= |= ^= <<= >>=
```

Знак **%** отвечает операции взятия остатка от деления; **&**, **|** и **^** обозначают логические операции И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ; **<<** и **>>** — это операции левого сдвига и правого сдвига; в §10.3 суммируются сведения об этих операциях и их смысле. Для любой бинарной операции **@**, примененной к операндам встроенных типов, выражение **x@=y** означает **x=x@y** за исключением того обстоятельства, что **x** вычисляется лишь однажды.

Функция **term()** обрабатывает умножение и деление в той же манере, в которой функция **expr()** обрабатывает сложение и вычитание:

```
double term(bool get)          // умножение и деление
{
    double left = prim(get);

    for (;;) {
        switch (ts.current().kind) {
        case Kind::mul:
            left *= prim(true);
            break;
        case Kind::div:
            if (auto d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");
        default:
            return left;
        }
    }
}
```

Результат деления на нуль не определен и обычно катастрофичен. Мы поэтому проверяем на нуль перед выполнением деления и в случае обнаружения нуля вызываем **error()**. Функция **error()** рассматривается в §10.2.4.

Переменная **d** вводится в программу аккурат в том месте, где она нужна, и тут же инициализируется. Область видимости переменной, объявляемой в условии, распространяется на оператор, который относится к проверяемому условию. Деление и присваивание **left/=d** выполняются только в случае, если **d** не равно нулю.

Функция **prim()**, лежащая в иерархии вызовов ниже функций **expr()** и **term()**, самостоятельно выполняет, в отличие от этих функций, некоторую конкретную работу:

```
double prim(bool get)          // обработка примитивов
{
    if (get) ts.get();         // прочитать следующую лексему

    switch (ts.current().kind) {
    case Kind::number:   // константа с плавающей запятой
    {   double v = ts.current().number_value;
        ts.get();
        return v;
    }
    case Kind::name:
    {   double& v = table[ts.current().string_value];      // найти соответствие
        if (ts.get().kind == Kind::assign) v = expr(true); // '=' увидели: присваивание return v;
        return v;
    }
    case Kind::minus:          // унарный минус
        return -prim(true);
    case Kind::lp:
    {   auto e = expr(true);
        if (ts.current().kind != Kind::rp) return error("')' expected");
        ts.get();              // съели ')
        return e;
    }
    default:
        return error("primary expected");
    }
}
```

Когда **Token** видится как **number** (то есть, целый литерал или литерал с плавающей запятой), его значение помещается в **number_value**. Аналогично, если **Token** это **name**, то его значение помещается в **string_value**.

Заметьте, что функция **prim()** дополнительно читает еще один **Token**, кроме того, что используется для анализа первичного выражения. Причина в том, что так нужно делать в некоторых случаях (например, с целью удостовериться, что имени что-то присвоено), и поэтому для согласованности кода делаем так во всех случаях. Когда парсер просто хочет перейти к следующей лексеме, он не использует возврат от вызова **ts.get()**. Тут нет проблем, ибо мы можем узнать результат из **ts.current()**.

Прежде, чем делать что-либо с именем, калькулятор должен удостовериться, что ему что-то присвоено или для него выполнен ввод. В обоих случаях справляются в таблице символов. Таблица символов реализуется как **map** (§4.4.3, §31.4.3):

```
map<string,double> table;
```

Это означает, что при индексировании таблицы по типу **string**, результирующее значение будет типа **double**. Например, если пользователь введет

```
radius = 6378.388;
```

калькулятор попадет на ветвь **case Kind::name** и выполнит код

```
double& v = table["radius"];
// ... expr() вычисляет присваиваемое значение ...
v = 6378.388;
```

Ссылка **v** предназначена для того, чтобы содержать числовое значение типа **double**, ассоциированное с **radius**, в то время как **expr()** вычисляет значение **6378.388** из символов ввода.

В Главе 14 и Главе 15 обсуждается вопрос о том, как организовать программу в виде набора модулей. Тем не менее, у нас за одним исключением все объявления упорядочены: все объявляется ровно один раз и строго перед тем, как используется. Единственное исключение — это функция **expr()**, которая вызывает **term()**, которая вызывает **prim()**, а та, в свою очередь, вызывает **expr()**. Эту петлю вызовов нужно как-то разорвать. Объявление

```
double expr(bool);
```

позиционированное перед определением функции **prim()**, решает проблему.

10.2.2. Ввод

Часто чтение входной информации порождает наиболее запутанную часть программы. В самом крайнем варианте коммуникация с человеком «требует» от программы учитывать человеческие причуды, привычки и случайные ошибки. Попытка потребовать от человека действовать в манере, более удобной для машины (программиста), рассматривается чуть ли не как оскорбление.

Задачей низкоуровневого ввода служит чтение символов и сборка лексем из этих символов. Лексемы затем работают как единицы входа для более высокоуровневых частей программы. У нас низкоуровневый ввод выполняется в функции **ts.get()**. Низкоуровневый ввод не является рутинной задачей для среднестатистического программиста. Многие системы помогают в этой задаче, предоставляя стандартные решения.

Сначала представим полное определение для **Token_stream**:

```
class Token_stream {
public:
    Token_stream(istream& s) : ip{&s}, owns{false} { }
    Token_stream(istream* p) : ip{p}, owns{true} { }

    ~Token_stream() { close(); }

    Token get();        // прочитать и вернуть следующую лексему
    Token& current();   // недавно прочитанная лексема

    void set_input(istream& s) { close(); ip = &s; owns=false; }
    void set_input(istream* p) { close(); ip = p; owns = true; }

private:
    void close() { if (owns) delete ip; }

    istream*ip;             // указатель на поток ввода
    bool owns;              // владеет ли Token_stream потоком istream?
    Token ct {Kind::end} ;  // текущая лексема
};
```

Инициализация **Token_stream** выполняется потоком ввода (§4.3.2, Глава 38), из которого получают символы. **Token_stream** поддерживает соглашение, что он владеет объектом типа **istream** (и впоследствии удаляет; §3.2.1.2, §11.2), передаваемым посредством указа-

теля, но не посредством ссылки. Может, для данной программы это и несколько излишне, но это полезно для общей техники классов, хранящих указатели ради уничтожения (освобождения) ресурсов.

В **Token_stream** содержатся три величины: указатель на поток ввода (**ip**), булево (логическое) значение (**owns**), отражающие факт владения потоком ввода, и текущая лексема (**ct**).

Я присваиваю полю **ct** начальное значение, ибо иначе код выглядел бы неряшливым. Не нужно вызывать **current()** до вызова **get()**, но даже в таком случае будет получен вполне определенный **Token**. Я выбрал **Kind::end** в качестве начального значения для **ct**.

Я представлю **Token_stream::get()** в два захода. Сначала это будет простая версия, возлагающая на пользователя большее бремя. Затем я модифицирую этот вариант в визуально более громоздкий (и менее элегантный), но более удобный на практике. Идея для **get()** проста: читаем символ и решаем, какая лексема должна формироваться из этого символа, читаем при необходимости дополнительные символы, а затем возвращаем **Token**, представляющий прочитанные символы.

Сначала операторы вычитывают первый непробельный символ из ***ip** в **ch** и проверяют успешность операции чтения:

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end};    // присвоить и вернуть
```

По умолчанию, операция >> пропускает пробельные символы (пробелы, символы табуляции, символы перехода на новую строку и т.д.) и оставляет значение переменной **ch** неизменным в случае неуспеха операции ввода. Следовательно, **ch==0** указывает на конец ввода.

Присваивание — это операция, и результат присваивания совпадает со значением переменной, которой было присвоено это значение. Из-за этого я присваиваю величину **Kind::end** переменной **curr_tok** и возвращаю это значение в одном и том же операторе. Один оператор вместо двух — это всегда полезно в дальнейшем при поддержке кода. Например, в случае двух операторов поддерживающий программист может обновить один оператор, но забыть скорректировать второй оператор.

Обратите внимание также и на то, как **{}**-список (§3.2.1.3, §11.3) используется в правой части операции присваивания, так как это выражение. Я мог бы написать и вот так

```
ct.kind = Kind::end;      // присвоить
return ct;                // вернуть
```

Однако, я считаю, что присваивание полного объекта **{Kind::end}** точнее отражает суть дела, чем возня с отдельными членами **ct**. Выражение **{Kind::end}** эквивалентно **{Kind::end,0,0}**.

Рассмотрим отдельные случаи перед рассмотрением полного варианта функции. Терминатор для выражений, ;, круглые скобки и знаки операций обрабатываются проще некуда — просто возвращаются их значения:

```
case ';':  // конец выражения; print
case '*':
case '/':
```

```
case '+':
case '-':
case '(':
case ')':
case '=':
    return ct={static_cast<Kind>(ch)};
```

Здесь **static_cast** (§11.5.2) необходим, потому что отсутствует неявное приведение типа от **char** к **Kind** (§8.4.1); лишь некоторые символы корреспондируют со значениями **Kind**, так что нужно явно подтвердить, что в данном случае **ch** подходит для этого.

С числами поступаем следующим образом:

```
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
case '.':
    ip->putback(ch);                // вернуть первую цифру (или .) назад в поток ввода
    *ip >> ct.number_value;         // прочитать число в ct
    ct.kind=Kind::number;
    return ct;
```

Располагая метки ветвей горизонтально вместо вертикальной записи, мы вряд ли поступаем правильно, ибо такая запись читается хуже. В то же время, довольно нудно под каждую цифру аккуратно выделять отдельную строку записи. Поскольку операция >> уже определена для чтения значений с плавающей запятой одинарной точности в **double**, то наш код тривиален. Сначала первый прочитанный символ (цифра или точка) помещается назад в **cin**, а затем значение с плавающей запятой читается в **ct.number_value**.

Если лексема не совпадает ни с концом ввода, ни с операцией, ни с символом пунктуации или числом, то она должна быть именем. Имя обрабатывается так же, как и число:

```
default:                            // имя, имя =, или ошибка
    if (isalpha(ch)) {
        ip->putback(ch);            // вернуть первый символ назад в поток ввода
        *ip>>ct.string_value;       // прочитать строку в ct
        ct.kind=Kind::name;
        return ct;
    }
```

Наконец, у нас может встретиться ошибка. Обрабатываем эту ситуацию следующим образом:

```
error("bad token");
return ct={Kind::print};
```

Чтобы избежать явного перечисления всех буквенных символов в отдельных ветвях оператора **switch**, применяем функцию **isalpha()** (§36.2.1) из стандартной библиотеки. Операция >> над строками (в данном случае над типом **string_value**) вычитывает до пробельного символа. Поэтому пользователь должен после имени и перед операцией (над этим именем) ставить пробел. Это далеко от идеала, так что мы вернемся к этой проблеме в §10.2.3.

И вот, наконец, полный вариант функции ввода:

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end};      // присвоить и вернуть
```

```
        case ';':       // конец выражения; print
        case '*':
        case '/':
        case '+':
        case '-':
        case '(':
        case ')':
        case '=':
            return ct={static_cast<Kind>(ch)};
        case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
        case '.':
            ip->putback(ch);                // вернуть первую цифру (или .) назад в поток ввода
            *ip >> ct.number_value;         // прочитать число в ct
            ct.kind=Kind::number;
            return ct;
        default:                // имя, имя =, или error
            if (isalpha(ch)) {
                ip->putback(ch);            // вернуть первый символ назад в поток ввода
                *ip>>ct.string_value;       // прочитать строку в ct
                ct.kind=Kind::name;
                return ct;
            }

            error("bad token");
            return ct={Kind::print};
        }
}
```

Преобразование операции в **Token** тривиально, поскольку для каждой операции определен **kind** как целое значение (§10.2.1).

10.2.3. Низкоуровневый ввод

Практическое использование калькулятора в его текущем состоянии обнаруживает несколько неудобств. Утомительно все время помнить про точку с запятой после выражения ради получения вывода его значения, да и завершение имени исключительно пробелом изрядно надоедает (и раздражает). Например, **x=7** является идентификатором — а не идентификатором **x**, за которым следуют операция **=** и число **7**. Чтобы добиться в данном случае желаемого, мы должны после **x** поставить пробел: **x =7**. Обе проблемы решаются заменой в функции **get()** умолчательных операций ввода, ориентированных на тип данных, на код, который читает и работает с индивидуальными символами.

Сначала мы точку с запятой заменим символом перехода на новую строку в качестве маркировки конца выражения:

```
Token Token_stream::get()
{
    char ch;

    do {    // пропустить пробельные символы кроме '\n'
        if (!ip->get(ch)) return ct={Kind::end};
    } while (ch!='\n' && isspace(ch));

    switch (ch) {
    case ';':
    case '\n':
        return ct={Kind::print};
```

Здесь я использую оператор **do**, который эквивалентен оператору **while**, за исключением того, что оператор под его управлением всегда выполняется хотя бы один раз. Вызов

ip->get(ch) вычитывает один символ из потока ввода ***ip** в **ch**. По умолчанию, **get()** не пропускает пробелы так, как это делает операция **>>**. Проверка **if (!ip->get(ch))** успешна в случае, когда ни один символ не может быть прочитан из **cin**; в этом случае возвращается **Kind::end** для завершения сессии работы калькулятора. Операция **!** используется из-за того, что **get()** возвращает **true** в случае успеха.

Функция **isspace()** из стандартной библиотеки выполняет проверку на пробельные символы (§36.2.1); **isspace()** возвращает ненулевое значение, если **c** является пробельным символом, и нуль в противном случае. Аналогичные функции выполняют проверку на цифры (**isdigit()**), на буквы (**isalpha()**) и на буквенно-цифровые символы (**isalnum()**).

После пропуска пробельных символов последующий символ используется для того, чтобы определить, какой тип лексемы на подходе.

Проблема с необходимостью искуственного пробела после имени решается путем чтения по одному символу за один раз до тех пор, пока не встретится символ, не являющийся ни цифрой, ни буквой:

```
default:                // NAME, NAME=, or error
    if (isalpha(ch)) {
        string_value = ch;
        while (ip->get(ch) && isalnum(ch))
            string_value += ch;      // добавить ch к концу string_value
        ip->putback(ch);
        return ct={Kind::name};
    }
```

По счастью, обе проблемы решаются локальными модификациями кода, что вообще является знаком неплохой организации программы.

Вы можете сомневаться в эффективности решения читать символы друг за другом, добавляя их к концу строкового значения (типа **string**). Это может иметь место лишь для сверхбольших строковых значений, а для небольших строк все современные реализации типа **string** включают в себя отдельные оптимизации для «малых строк» («small string optimization»; §19.3.3). Это значит, что наша работа со строками в программе калькулятора достаточно эффективна (и она не требует обращения к свободной памяти). Максимальное число символов в «коротких строках» зависит от реализации, но число 14 является хорошим ориентиром.

10.2.4. Обработка ошибок

Всегда полезно выявлять ошибки и сообщать о них. В нашей программе функция **error()** подсчитывает ошибки, выводит сообщение и выполняет возврат:

```
int no_of_errors;

double error(const string& s)
{

    no_of_errors++;
    cerr << "error: " << s << '\n';
    return 1;
}
```

Небуферированный поток вывода **cerr** традиционно используется для сообщений об ошибках (§38.1).

Причина возврата значения состоит в том, что обычно ошибки появляются где-то в середине процесса обработки выражения, так что нам остается либо прервать этот

процесс, или же вернуть значение, которое с большой долей вероятности не породит последующих ошибок. Этого достаточно для простого варианта калькулятора. Если бы **Token_stream::get()** отслеживал бы номера строк, функция **error()** могла бы информировать пользователя о примерном местоположении ошибки.

Более корректный стиль обработки ошибок предполагает отделение выявления ошибки от ее обработки и принятии решения об исправлении ситуации. Это можно реализовать посредством исключений (см. §2.4.3.1, Глава 13), но в нашем 180-строчном примере достаточно того, что мы уже реализовали.

10.2.5. Главная программа

Теперь, когда готовы все основные части программы, нам осталось написать главную программу — которая все запускает в работу. Я предусмотрел тут две функции: **main()** для запуска процесса работы и сообщения об ошибках и **calculate()** для реализации собственно вычислений:

```
Token_stream ts {cin};          // ввод из cin

void calculate()
{
    for (;;) {
        ts.get();
        if (ts.current().kind == Kind::end) break;
        if (ts.current().kind == Kind::print) continue;
        cout << expr(false) << '\n';
    }
}

int main()
{
    table["pi"] = 3.1415926535897932385;    // предопределенные имена
    table["e"] = 2.7182818284590452354;

    calculate();

    return no_of_errors;
}
```

По традиции **main()** возвращает нуль, если программа завершается штатным образом, и ненулевое значение в противном случае (§2.2.1). Возврат числа ошибок подойдет здесь наилучшим образом. Инициализация у нас сводится к созданию таблицы предопределенных имен.

Задачей цикла в функции **calculate()** является чтение выражений и вывод ответа. Это реализуется в следующей строке:

```
cout << expr(false) << '\n';
```

Аргумент **false** говорит функции **expr()**, что она не должна вызывать **ts.get()** для чтения лексемы, над которой предстоит работа.

Проверка на **Kind::end** гарантирует, что из цикла будет корректный выход, когда **ts.get()** столкнется с ошибкой ввода или же будет достигнут конец ввода. Оператор **break** выполняет выход из ближайшего оператора **switch** или оператора цикла (§9.5). Проверка на **Kind::print** (то есть на '\n' и ;) освобождает функцию **expr()** от ответственности за корректную обработку пустых выражений. Оператор **continue** эквивалентен переходу к концу тела цикла.

10.2.6. Заголовочные файлы

Калькулятор использует средства стандартной библиотеки. Поэтому нужно включать соответствующие заголовочные файлы:

```
#include<iostream> // I/O
#include<string>   // строки
#include<map>      // map
#include<cctype>   // isalpha(), и т.д.
```

Все эти заголовочные файлы определяют имена в пространстве имен **std**, так что для их использования далее в программе нужно либо их явно квалифицировать префиксом **std::** или же внести их в глобальное пространство имен:

```
using namespace std;
```

Далее в настоящей книге я рассматриваю вопросы о том, как организовать данную программу-калькулятор в виде модулей, используя пространства имен, и как все разместить по исходным файлам.

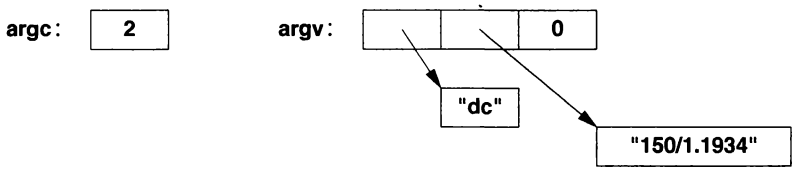
10.2.7. Аргументы командной строки

После того, как программа была написана, я обнаружил, что запускать программу, затем вводить выражение и наблюдать выход из программы несколько неудобно. Ведь штатное применение программы состоит в оценке единственного выражения. Поэтому целесообразно вводить выражение в виде аргумента командной строки. Для этого нам потребуются совсем небольшие изменения в программе.

Программа стартует с функции **main()** (§2.2.1, §15.4). При этом ей передаются два параметра: количество аргументов в командной строке (**argc**) и массив этих аргументов (**argv**). Аргументы имеют тип С-строк (§2.2.5, §7.3), то есть тип **argv** есть **char*[argc+1]**. В командной строке имя самой программы передается первым аргументом, то есть **argv[0]**, и поэтому **argc** всегда не меньше единицы. Список аргументов является нуль-терминированным, то есть **argv[argc]==0**. Например, для команды

```
dc 150/1.1934
```

аргументы следующие:



Поскольку соглашение о вызове функции **main()** одинаковое с языком С, то и массивы и строки также совпадают (то есть имеют С-стиль).

Наша идея состоит в том, чтобы читать из командной строки так же, как из потока ввода. Поток для чтения из строки называется **istringstream** (§38.2.2). Так что мы должны заставить наш **Token_stream** читать из соответствующего **istringstream**:

```
Token_stream ts {cin};

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:                 // чтение из стандартного ввода
        break;
```

```
        case 2:                         // чтение из строки аргумента
            ts.set_input(new istringstream{argv[1]});
            break;
        default:
            error("too many arguments");
            return 1;
        }

        table["pi"] = 3.1415926535897932385;    // предопределенные имена
        table["e"] = 2.7182818284590452354;

        calculate();

        return no_of_errors;
    }
```

Чтобы можно было воспользоваться **istringstream**, нужно включить заголовочный файл **<sstream>**.

Было бы нетрудно модифицировать функцию **main()** так, чтобы можно было читать сразу несколько аргументов из командной строки, но в этом нет необходимости, так как несколько выражений можно передать единственным аргументом:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Я применяю двойные кавычки потому, что на моей UNIX-системе точки с запятой отделяют команды друг от друга. На других системах могут встретиться их собственные ограничения.

Сами по себе, **argc** и **argv** могут послужить источником небольших, но раздражающих ошибок. Поэтому, чтобы облегчить передачу аргументов внутри программы я применяю простую функцию, создающую некое хранилище типа **vector<string>**:

```
vector<string> arguments(int argc, char* argv[])
{
    vector<string> res;
    for (int i = 0; i!=argc; ++i)
        res.push_back(argv[i]);
    return res;
}
```

Более сложные решения для обработки аргументов не так уж и нужны и встречаются редко.

10.2.8. Замечание по стилю

Программистам, не знакомым с ассоциативными массивами, применение типа **map** для таблицы символов может показаться излишним и нарочитым. Это не так. Стандартная библиотека для того и создана, чтобы ее на самом деле применяли, ибо ее дизайн и реализация выполнены настолько продуманно и тщательно, что рядовому программисту сравниться в этих показателях будет крайне сложно (тем более, что нужно при этом перенапрячься ради единственной программы).

Если посмотреть на наш калькулятор, особенно на его первую версию, то можно заметить, что не так уж и много мы использовали конструкций в стиле языка C. Многие известные для языка C трюки были заменены классами стандартной библиотеки, такими как **ostream**, **string** и **map** (§4.3.1, §4.2, §4.4.3, §31.4, Глава 36, Глава 38).

Также относительно немного циклов, арифметических вычислений и присваиваний. К этому и нужно стремиться в любом коде, который не имеет дело с аппаратурой напрямую и не реализует низкоуровневые абстракции.

10.3. Сводка сведений по операциям

В данном разделе мы представим в табличном виде суммирующую информацию об особенностях операций в языке C++. В этих таблицах:

- *name* (*имя*) — идентификатор (например, **sum** или **map**), имя операции (например, **operator int**, **operator+** или **operator"" km**) или имя специализации шаблона (например, **sort<Record>** и **array<int,10>**), возможно с квалификацией посредством :: (например, **std::vector** или **vector<T>::operator[]**).
- *class-name* (*имя-класса*) это имя класса (включая **decltype(expr)**, где **expr** обозначает класс).
- *member* (*член*) это имя члена (включая имя деструктора или члена-шаблона).
- *object* (*объект*) это выражение, порождающее объект класса.
- *pointer* (*указатель*) это выражение, порождающее указатель (включая **this** и объект этого класса, поддерживающий операции над указателями).
- *expr* (*выр-е*) это выражение, включая литерал (например, **17**, **"mouse"** или **true**).
- *expr-list* (*список выражений*) это возможно пустой список выражений.
- *lvalue* — выражение, обозначающее модифицируемый объект (§6.4.1).
- *type* (*тип*) может быть полностью общим именем типа (с *, () и т.д.) только когда появляется в круглых скобках; в остальных случаях имеются ограничения (§iso.A).
- *lambda-declarator* — это возможно пустой (с запятыми в качестве разделителей) список параметров, за которым опционально следует спецификатор **mutable**, за которым опционально идет **noexcept** спецификатор и далее опционально тип возврата (§11.4).
- *capture-list* — список (возможно пустой), специфицирующий контекстные зависимости (§11.4).
- *stmt-list* — это возможно пустой список операторов (§2.2.4, Глава 9).

Синтаксис выражений не зависит от типов операндов. Смысл представленной информации применим к операндам встроенных типов (§6.2.1). Вы можете дополнительно опеределять смысл операций, применимых к операндам пользовательских типов (§2.3, Глава 18).

Таблица передает правила грамматики не со стопроцентной полнотой. За большими деталями обращайтесь к §iso.5 и §iso.A.

Сводка операций (§iso.5.1)		
Выражение в скобках	(*expr*)	
Лямбда	[*capture-list*] *lambda-declarator* { *stmt-List* }	§11.4
Разрешение области видимости	*class-name* :: *member*	§16.2.3
Разрешение области видимости	*namespace-name* :: *member*	§14.2.1
Глобальное имя	:: *name*	§14.2.1

В каждой секции таблицы расположены операции с одинаковыми уровнями приоритетов. Операции в вышерасположенных секциях имеют более высокий приоритет. Например, **N::x.m** означает **(N::x).m**, а не (некорректное) **N::(x.m)**.

Продолжение

Сводка операций		
Выбор члена	*object. member*	§16.2.3
Выбор члена	*pointer* -> *member*	§16.2.3
Индексирование	*pointer* [*expr*]	§7.3
Вызов функции	*expr* (*expr-list*)	§12.2
Конструирование значения	*type* { *expr-list* }	§11.3.2
Конверсия в функцион. стиле	*type* (*expr-list*)	§11.5.4
Постинкремент	*lvalue* ++	§11.1.4
Постдекремент	*lvalue* --	§11.1.4
Идентификация типа	**typeid** (*type*)	§22.5
RTTI	**typeid** (*expr*)	§22.5
Проверка типа времени выполнения	**dynamic_cast** < *type* > (*expr*)	§22.2.1
Проверка типа во время компиляции	**static_cast** < *type* > (*expr*)	§11.5.2
Конверсия без проверки	**reinterpret_cast** < *type* > (*expr*)	§11.5.2
const конверсия	**const_cast** < *type* > (*expr*)	§11.5.2
Размер объекта	**sizeof** *expr*	§6.2.8
Размер типа	**sizeof** (*type*)	§6.2.8
Размер пакета параметров	**sizeof...** *name*	§28.6.2
Выравнивание типа	**alignof** (*type*)	§6.2.9
Преинкремент	++ *lvalue*	§11.1.4
Предекремент	-- *lvalue*	§11.1.4
Дополнение	~ *expr*	§11.1.2
НЕ	! *expr*	§11.1.1
Унарный минус	- *expr*	§2.2.2
Унарный плюс	+ *expr*	§2.2.2
Адрес	& *lvalue*	§7.2
Разыменование	* *expr*	§7.2
Создать (аллоцировать)	**new** *type*	§11.2
Создать (аллоцировать с инициализацией)	**new** *type* (*expr-list*)	§11.2
Создать (аллоцировать с инициализацией)	**new** *type* { *expr-list* }	§11.2
Создать (разместить)	**new** (*expr-list*) *type*	§11.2.4
Создать (разместить и инициализировать)	**new** (*expr-list*) *type* (*expr-list*)	§11.2.4
Создать (разместить и инициализировать)	**new** (*expr-list*) *type* { *expr-list* }	§11.2.4
Уничтожить (деаллоцировать)	**delete** *pointer*	§11.2
Уничтожить массив	**delete** [] *pointer*	§11.2.2
Возможна генерация исключения в выражении?	**noexcept** (*expr*)	§13.5.1.2
Приведение типа	(*type*) *expr*	§11.5.3
Выбор члена	*object* .* *pointer-to-member*	§20.6
Выбор члена	*pointer* ->* *pointer-to-member*	§20.6

Например, постфиксная операция ++ имеет более высокий приоритет, чем унарный *, так что ***p++** означает ***(p++)**, а не **(*p)++**.

Продолжение

Сводка операций		
Умножить	*expr* * *expr*	§10.2.1
Делить	*expr* / *expr*	§10.2.1
Остаток по модулю	*expr* % *expr*	§10.2.1
Сложение	*expr* + *expr*	§10.2.1
Вычитание	*expr* - *expr*	§10.2.1
Сдвиг влево	*expr* << *expr*	§11.1.2
Сдвиг вправо	*expr* >> *expr*	§11.1.2
Меньше чем	*expr* < *expr*	§2.2.2
Меньше чем или равно	*expr* <= *expr*	§2.2.2
Больше чем	*expr* > *expr*	§2.2.2
Больше чем или равно	*expr* >= *expr*	§2.2.2
Равно	*expr* == *expr*	§2.2.2
Не равно	*expr* != *expr*	§2.2.2
Побитовое ИЛИ	*expr* & *expr*	§11.1.2
Побитовое исключающее ИЛИ	*expr* ^ *expr*	§11.1.2
Побитовое инклюзивное ИЛИ	*expr* | *expr*	§11.1.2
Логическое И	*expr* && *expr*	§11.1.1
Логическое инклюзивное ИЛИ	*expr* || *expr*	§11.1.1
Условное выражение	*expr* ? *expr* : *expr*	§11.1.3
Список	{ *expr-list* }	§11.3
Генерация исключения	**throw** *expr*	§13.5
Простое присваивание	*lvalue* = *expr*	§10.2.1
Умножить и присвоить	*lvalue* *= *expr*	§10.2.1
Поделить и присвоить	*lvalue* /= *expr*	§10.2.1
Деление по модулю и присвоение	*lvalue* %= *expr*	§10.2.1
Сложить и присвоить	*lvalue* += *expr*	§10.2.1
Вычесть и присвоить	*lvalue* -= *expr*	§10.2.1
Сдвиг влево и присвоение	*lvalue* <<= *expr*	§10.2.1
Сдвиг вправо и присвоение	*lvalue* >>= *expr*	§10.2.1
Побитовое И и присвоение	*lvalue* &= *expr*	§10.2.1
Побитовое инклюзивное ИЛИ и присвоение	*lvalue* |= *expr*	§10.2.1
Побитовое исключающее ИЛИ и присвоение	*lvalue* ^= *expr*	§10.2.1
Запятая (следование)	*expr* , *expr*	§10.3.2

Например, **a+b*c** означает **a+(b*c)**, а не **(a+b)*c**, потому что * имеет более высокий приоритет, чем +.

Унарные операции и операции присваивания правоассоциативны; все остальные являются левоассоциативными. К примеру, **a=b=c** означает **a=(b=c)**, в то время как **a+b+c** означает **(a+b)+c**.

Несколько грамматических правил не могут быть объяснены в рамках уровня приоритета и ассоциативности. Например, **a=b<c?d=e:f=g** означает **a=((b<c)?(d=e):(f=g))**, но за пояснениями обратитесь к §iso.A.

Перед тем, как применять правила грамматики, нужно собрать лексемы из отдельных символов. Лексему нужно сотавлять из максимально возможного числа символов. Например, **&&** это единый знак операции, а не две операции **&**, и **a+++1** означает **(a ++) + 1**.

Сводка лексем (§iso.2.7)		
Класс лексемы	Примеры	Ссылки
Идентификатор	**vector, foo_bar, x3**	§6.3.3
Ключевое слово	**int, for, virtual**	§6.3.3.1
Символьный литерал	**'x', \n', 'U'\UFADEFADE'**	§6.2.3.2
Целый литерал	**12, 012, 0x12**	§6.2.4.1
Вещественный литерал	**1.2, 1.2e-3, 1.2L**	§6.2.5.1
Строковый литерал	**"Hello!", R"("World"!)"**	§7.3.2
Операция	**+=, %, <<**	§10.3
Пунктуация	**;, ,, {, }, (,)**	
Нотация препроцессора	**#, ##**	§12.6

Пробельные символы (пробел, табуляция, новая строка) могут служить разделителями лексем (например, **int count** это ключевое слово, за которым следует идентификатор, а не **intcount**), а в иных случаях игнорируются.

Некоторые символы из базового набора (§6.1.2), вроде символа |, не всегда бывает удобно набирать на некоторых клавиатурах. Кроме того, отдельные программисты считают, что нехорошо использовать символы **&&** и ~ для базовых логических операций. Как следствие, предоставляются ключевые слова для альтернативного обозначения:

Альтернативное представление (§iso.2.12)										
and	**and_eq**	**bitand**	**bitor**	**compl**	**not**	**not_eq**	**or**	**or_eq**	**xor**	**xor_eq**
&	**&=**	**&**	**|**	**~**	**!**	**!=**	**|**	**|=**	**^**	**^=**

Например:

```
bool b = not (x or y) and z;
int x4 = ~ (x1 bitor x2) bitand x3;
```

эквивалентно

```
bool b = !(x || y) && z;
int x4 = ~(x1 | x2) & x3;
```

Отметим, что **and=** не эквивалентно **&=**; если вы предпочитаете ключевые слова, то вам нужно писать **and_eq**.

10.3.1. Результаты

Результирующий тип для арифметических операций определяется согласно набору правил, известному как «обычные арифметические преобразования» (§10.5.3). Главной целью является стремление произвести результат, имеющий тип «максимального» типа операндов. Например, если бинарная операция имеет операнд с плавающей запятой, то вычисления выполняются в арифметике с плавающей запятой и результат вычислений также имеет этот тип. Аналогично, если имеется операнд типа **long**, то вычисления выполняются как для типа **long** и результат имеет тип **long**. Операнды, тип которых «меньше» **int** (**bool** и **char**) преобразуются к **int** перед вычислениями.

Операции сравнения, **==**, **<=** и т.д., порождают результат булевского типа. Смысл и тип результата для операндов пользовательских типов определяются их объявлениями (§18.2).

Там, где это логически допустимо, результат операции над операндом категории lvalue также относится к lvalue, обозначающему этот операнд. Пример:

```
void f(int x, int y)
{
    int j = x = y;          // значение x=y есть значение x после присваивания
    int* p = &++x;          // p указывает на x
    int* q = &(x++);        // ошибка: x++ не lvalue (не значение, хранящееся в x)
    int* p2 = &(x>y?x:y);   // значение целого с большей величиной
    int& r = (x<y)?x:1;     // ошибка: 1 не lvalue
}
```

Если второй и третий операнды операции **?:** относятся к lvalue и имеют одинаковый тип, результат будет иметь тот же тип и относиться к lvalue. Сохранение lvalue описанным образом придает чрезвычайную гибкость в применении операций. Это особенно полезно в случае, когда код должен однородно и эффективно работать как со встроенными типами, так и с типами, определяемыми пользователем (например, при написании программ с шаблонами или программ, генерирующих C++ код).

Тип результата операции **sizeof** — это беззнаковый интегральный тип **size_t**, определяемый в файле **<cstddef>**. Тип разности указателей — это знаковый интегральный тип **ptrdiff_t**, определяемый в **<cstddef>**.

Реализации не обязаны следить за арифметическим переполнением и на практике едва ли делают это. Например:

```
void f()
{
    int i = 1;
    while (0 < i) ++i;
    cout << "i has become negative!" << i << '\n';
}
```

Здесь (в конце концов) будет попытка увеличить **i** сверх наибольшего целого. Что в этом случае произойдет, заранее неизвестно (неопределенное поведение), но в типичном случае значение будет отрицательным (на моей машине **-2147483648**). Аналогично, не определен результат деления на нуль, но чаще всего в таком случае происходит принудительное завершение работы программы. В частности, переполнение, исчезновение порядка и деление на нуль не генерируют стандартных исключений (§30.4.1.1).

10.3.2. Порядок вычислений

Порядок вычисления подвыражений выражения неопределен. В частности, вы не можете считать, что выражение вычисляется в порядке слева-направо. Например:

```
int x = f(2)+g(3);    // не ясно, f() или g() вызывается первым
```

Дело в том, что в отсутствии навязывания порядка вычислений возможно генерирование более эффективного кода. В то же время, отсутствие этого порядка может вести к неопределенным результатам. Пример:

```
int i = 1;
v[i] = i++;      // результат не определен
```

Присваивание может выполниться как **v[1]=1** или как **v[2]=1**, или же может быть реализовано нечто еще более странное. Компиляторы могут предупредить о таких неоднозначностях, но многие из них этого не делают. Поэтому старайтесь избегать выражений, которые читают или пишут в объект более одного раза, кроме случаев, когда выражение так

делает посредством единственной операции с однозначным поведением, такой как ++ или +=, или же в случае явного задания порядка посредством операции «запятая», && или ||.

Перечисленные операции гарантируют, что их левый операнд будет вычисляться до вычисления правого операнда. Например, выражение **b=(a=2,a+1)** присвоит значение **3** переменной **b**. Примеры использования операций || и && можно найти в §10.3.3. Для встроенных типов второй операнд операции && вычисляется только в том случае, если первый операнд имеет значение **true**, а второй операнд операции || вычисляется только, если первый операнд **false**. Обратите внимание на то, что операция следования (запятая) логически отличается от запятой, разделяющей аргументы в функциональном вызове. Например:

```
f1(v[i],i++);      // два аргумента
f2( (v[i],i++) ); // один аргумент
```

Функция **f1** вызывается с двумя аргументами, **v[i]** и **i++**, а порядок вычисления аргументов-выражений неопределен. Это плохой стиль и его следует избегать. Вызов **f2** имеет всего один аргумент — выражение с операцией следования, имеющее то же самое значение, что и **i++**. Все это, конечно, может смутить читающего программу, так что этого лучше избегать.

Круглые скобки применяются для группировки операций. Например, **a*b/c** означает **(a*b)/c**, так что круглые скобки нужны для варианта **a*(b/c)**. Нужно отметить, что вроде бы тождественные выражения **a*(b/c)** и **(a*b)/c** могут дать разные результаты для чисел с плавающей запятой, так что компилятор работает с такими выражениями в точности, как они написаны.

10.3.3. Приоритеты операций

Уровни приоритетов и правила ассоциативности отражают тенденции типового использования операций. Например:

```
if (i<=0 || max<i) // ...
```

означает «если **i** меньше или равно **0** или если **max** меньше чем **i**». Это эквивалентно

```
if ( (i<=0) || (max<i) ) // ...
```

а не бессмысленному

```
if (i <= (0||max) < i) // ...
```

Тем не менее, всегда лучше применить круглые скобки, если возникают хоть какие-то сомнения. Чем сложнее становятся подвыражения, тем чаще применяются круглые скобки, но применение сложных подвыражений является типовым источником ошибок. Поэтому, если вы почувствовали, что хотите применить круглые скобки из-за сложности выражения, то возможно лучше ввести лишнюю переменную для хранения подвыражения.

В ряде случаев приоритеты операций не дают очевидной интерпретации выражения. Вот пример:

```
if (i&mask == 0) // упс! == выражение как операнд для &
```

Здесь не имеет место наложения маски на **i** и затем сравнение результата с нулем. Поскольку операция == имеет более высокий приоритет по сравнению с операцией **&**, то выражение интерпретируется как **i&(mask==0)**. К счастью, компиляторы чаще всего выдают предупреждение в таких случаях, и скобки требуется применить для получения нужного результата:

```
if ((i&mask) == 0) // ...
```

Ясно, что следующий фрагмент не работает так, как этого может ожидать математик:

```
if (0 <= x <= 99) // ...
```

Здесь нет никаких синтаксических ошибок, но интерпретируется это так: **(0<=x)<=99**. Результат первого сравнения дает **true** или **false**. Затем такие булевские значения неявно приводятся к **1** или **0** и сравниваются с **99**, что дает **true**. Чтобы корректно выяснить, лежит ли **x** в диапазоне **0..99**, можно воспользоваться следующим кодом:

```
if (0<=x && x<=99) // ...
```

Типичной ошибкой начинающих программистов является использование операции = вместо == в записи условия:

```
if (a = 7)      // упс! Присваивание константы в условии
```

В этом нет ничего удивительного, ибо знак = используется во многих других языках программирования для сравнения на равенство. Компиляторы чаще всего предупреждают программиста в таких случаях. И я не думаю, что стоит коверкать чистоту записи, не доверяя компилятору:

```
if (7 == a)     // попытка защиты от неправильного =; не рекомендуется
```

10.3.4. Временные объекты

Часто компилятору приходится создавать объект для хранения промежуточного результата вычисления выражения. Например, для **v=x+y*z** результат вычисления **y*z** нужно где-то на время сохранить прежде, чем он будет прибавлен к значению переменной **x**. Для встроенных типов это реализуется посредством введения невидимого для программиста *временного объекта* (*temporary object*). В то же время, для пользовательского типа, хранящего некий ресурс, важно знать время жизни временного объекта. Не будучи связанным ссылкой или не в случае его применения для инициализации именованного объекта временный объект уничтожается в конце полного выражения, в рамках которого он создается. *Полное выражение* (*full expression*) — это выражение, которое не входит подвыражением ни в какое другое выражение.

Тип **string** из стандартной библиотеки имеет член **c_str()** (§36.3), который возвращает указатель в С-стиле на массив символов с терминальным нулем (§2.2.5, §43.4). Также для этого типа определена операция +, означающая конкатенацию. Все это полезные вещи для типа **string**. Но вот в их комбинации возможна неясность. Пример:

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
        // cs используется здесь
    }
}
```

Возможно, вашей первой реакцией на этот код будет восклицание «Не надо так делать!». Тем не менее, полезно проанализировать, как такой код интерпретируется.

Создается временная строка типа **string** для хранения «суммы» **s1+s2**. Затем из этого объекта извлекается указатель на строку в С-стиле. Наконец, в самом конце выражения временный объект удаляется. Но строка в С-стиле, которая возвращается членом **c_str()**, была частью временного объекта, хранящего сумму **s1+s2**, и нет никакой гарантии, что она существует после того, как временный объект уничтожается. Следовательно, указатель **cs**

указывает на скомпрометированный блок памяти. Операция вывода **cout<<cs** может сработать штатно, но только в случае большой удачи. Обычно компилятор в состоянии выявить и предупредить о подобного рода проблемах.

Проблема с оператором **if** более тонкая. В условии все нормально, ибо временный объект для хранения **s2+s3** создается в рамках условия. Но он уничтожается до того, как управление перейдет к оператору, контролируемому посредством **if**, так что применение в нем **cs** не гарантировано от ошибок.

Можно обратить внимание на то, что эти проблемы возникают в том случае, когда высокоуровневый тип данных используется в весьма низкоуровневом стиле. Если программировать в более чистом стиле, то программный фрагмент станет более понятным и избежит проблем с временными объектами. Например:

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;
    if (s.length()<8 && s[0]=='a') {
        // s используется здесь
    }
}
```

Временный объект может использоваться в качестве инициализатора для константной ссылки или именованного объекта. Например;

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
    const string& s = s1+s2;
    string ss = s1+s2;

    g(s,ss);   // можно использовать s and ss здесь
}
```

Здесь все замечательно. Временный объект уничтожается, когда его ссылка или именованный объект выходит из области видимости. Помните, что возврат ссылки на локальный объект является ошибкой (§12.1.4) и что временный объект нельзя связывать с неконстантной ссылкой категории lvalue (§7.7).

Временный объект может быть создан явным образом в выражении посредством конструктора (§11.5.1). Например:

```
void f(Shape& s, int n, char ch)
{
    s.move(string{n,ch}); // конструирование string с n копиями ch для передачи Shape::move()
    // ...
}
```

Такие временные объекты уничтожаются так же, как и неявно генерируемые временные объекты.

10.4. Константные выражения

Язык C++ предлагает два значения для понятия «константность»:

- **constexpr**: означает «Вычислить во время компиляции» (§2.2.3).
- **const**: означает «Не изменять в данной области видимости» (§2.2.3, §7.5).

По самой сути, **constexpr** просто гарантирует вычисление на этапе компиляции, в то время как **const** специфицирует неизменяемость в интерфейсах. Сейчас мы уделим внимание первой роли: вычислению на этапе компиляции.

Константное выражение — это выражение, которое компилятор может вычислить во время компиляции. Оно не может содержать величин, значение которых во время компиляции неизвестно и оно не может иметь побочных эффектов. В итоге, константное выражение должно стартовать с интегрального значения (§6.2.1), со значения с плавающей запятой (§6.2.5) или с энумератора (§8.4), и мы можем комбинировать перечисленное с помощью операций и **constexpr**-функций, что также возвращает некоторое значение. Кроме того, в некоторых вариантах константных выражений можно также использовать и некоторые адреса. Для простоты я откладываю отдельное обсуждение этого вопроса до §10.4.5.

Существует ряд причин, по которым кто-то может предпочесть именованную константу литералу или значению переменной:

1. Именованная константа делает код более понятным и помогает его поддерживать в дальнейшем.
2. Переменную можно изменять (так что нам нужно четко обосновать, зачем применяем именно переменную).
3. Язык требует константных выражений для размеров массивов, для меток **case**-веток и для аргументов-значений у шаблонов.
4. Программисты встраиваемых систем любят помещать неизменяемые данные в память «только для чтения», поскольку такая память дешевле динамической памяти и более доступна на рынке. Кроме того, данные в памяти «только для чтения» часто невосприимчивы к большинству типичных вариантов отказа техники.
5. Если инициализация выполняется на этапе компиляции, то доступ к таким данным свободен от «гонки за данными» в многопоточных системах.
6. Одноразовое вычисление на этапе компиляции приводит к большому выигрышу в производительности по сравнению с многократными вычисления во время выполнения программы.

Мы не применяем константные выражения исключительно из-за помешательства на почве производительности. Часто причиной служит стремление точнее отразить системные требования.

Если инициализатор для **constexpr**-выражения не может быть вычислен на этапе компиляции, то компилятор выдаст сообщение об ошибке. Например:

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1;        // ошибка: инициализатор не константное выражение
constexpr int x4 = x2;        // OK

void f()
{
    constexpr int y3 = x1;    // ошибка: инициализатор не константное выражение
    constexpr int y4 = x2;    // OK
    // ...
}
```

Умный компилятор мог бы догадаться, что использование **x1** в качестве инициализатора для **x3** просто приводит к константе **7**. Однако, лучше не полагаться на особую ин-

теллектуальность компилятора. Кроме того, в больших программах выявление значения переменных во время компиляции либо затруднительно, либо невозможно.

Выразительная мощь константных выражений впечатляет: можно использовать целые, величины с плавающей запятой и перечисления. Можно применять любые операции, не модифицирующие состояние (например +, ?:, и [], но не = или ++). Мы можем использовать **constexpr**-функции (§12.1.6) и литеральные типы (§10.4.3) для предоставления типовой безопасности и выразительной мощи. Будет просто несправедливым даже сравнивать такие возможности с тем, что обычно реализуется посредством макросов (§12.6).

Условная операция **?:** реализует выбор в константных выражениях. Например, мы можем вычислить корень квадратный на этапе компиляции:

```
constexpr int isqrt_helper(int sq, int d, int a)
{
    return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x)
{
    return isqrt_helper(1,3,x)/2 - 1;
}

constexpr int s1 = isqrt(9);        // s1 становится 3
constexpr int s2 = isqrt(1234);
```

Сначала вычисляется условие в операции **?:**, после чего вычисляется выбранная альтернатива. Отринутая альтернатива не вычисляется, и даже может быть неконстантным выражением. Аналогично, операнды для **&&** и **||**, которые не вычисляются, также могут быть неконстантными выражениями. Эта особенность особенно важна для **constexpr**-функций, которые иногда используются как константные выражения, а иногда — не используются в таком качестве.

10.4.1. Символические константы

Простейшее применение константных сущностей (**constexpr** или **const**) состоит в назначении имен константам. Символические константы нужно применять систематически, иначе не избежать так называемых «магических чисел», расположенных непосредственно в коде. Когда литеральные значения разбросаны повсюду в коде, это превращается в головную боль при чтении и изучении кода в целях поддержки и модернизации. Например, если в коде многократно встречается непосредственное числовое значение для размера массива, то в случае необходимости его изменения приходится всюду в коде разыскивать это значение и заменять его на новое, что часто приводит к рассогласованию. Применение символических констант локализует информацию. Обычно числовые константы символизируют наши предположения о программе. Например, число **4** может отражать наше представление о числе байт для целого значения, **128** — число символов, требуемых для буфера ввода, а **6.24** — коэффициент обмена датской кроны на доллар США. Если оставить их в таком виде прямо в коде, то в дальнейшем любому программисту будет трудно догадаться об их смысле. Многие из этих значений потребуется со временем изменить, как в случае портирования программы на другие платформы или в связи с изменениями в требованиях к программе со стороны заказчиков. Представление предположений и требований к программе в виде хорошо прокомментированных символических констант минимизирует проблемы при поддержке программы.

10.4.2. Ключевое слово const

Ключевое слово **const** преимущественно используется в определении интерфейсов (§7.5). Кроме того, это ключевое слово применяется для утверждения о константности величин. Например:

```
const int x = 7;
const string s = "asdf";
const int y = sqrt(x);
```

Константы, инициализированные константными выражениями, могут использоваться в константных выражениях. Объекты, объявляемые с ключевым словом **const**, отличаются от таковых с ключевым словом **constexpr** тем, что первые могут быть инициализированы неконстантными выражениями; в последнем случае их нельзя будет применять как **constexpr**. Пример:

```
constexpr int xx = x;          // OK
constexpr string ss = s;       // ошибка: s не константное выражение
constexpr int yy = y;          // ошибка: sqrt(x) не константное выражение
```

Здесь источники ошибок состоят в том, что **string** не является литеральным типом (§10.4.3) и **sqrt()** не является **constexpr**-функцией (§12.1.6).

Для определения простых констант лучше применять **constexpr** вместо **const**, но ключевое слово **constexpr** введено лишь в C++11, так что в более раннем коде всюду стоит **const**. Во многих случаях альтернативу предоставляют энумераторы (§8.4).

10.4.3. Литеральные типы

Достаточно простые пользовательские типы могут использоваться в константных выражениях. Например:

```
struct Point {
    int x,y,z;
    constexpr Point up(int d) { return {x,y,z+d}; }
    constexpr Point move(int dx, int dy) { return {x+dx,y+dy}; }
    // ...
};
```

Класс с **constexpr**-конструктором называется *литеральным типом* (*literal type*). Степень простоты конструктора в таком случае состоит в том, что он должен иметь пустое тело и все члены должны инициализироваться потенциально константными выражениями. Пример:

```
constexpr Point origo {0,0};
constexpr int z = origo.x;

constexpr Point a[] = {
    origo, Point{1,1}, Point{2,2}, origo.move(3,3)
};
constexpr int x = a[1].x;          // x становится 1

constexpr Point xy{0,sqrt(2)};     // ошибка: sqrt(2) не константное выражение
```

Очевидно, что мы можем определять **constexpr**-функции, принимающие аргументы литеральных типов. Например:

```
constexpr int square(int x)
{
    return x*x;
}
```

```
constexpr int radial_distance(Point p)
{
    return isqrt(square(p.x)+square(p.y)+square(p.z));
}

constexpr Point p1 {10,20,30};          // умолчательный конструктор есть constexpr
constexpr p2 {p1.up(20)};               // Point::up() есть constexpr
constexpr int dist = radial_distance(p2);
```

Я использовал **int** вместо **double** просто потому, что у меня под рукой не было **constexpr**-функции вычисления корня квадратного для чисел с плавающей запятой.

Для членов-функций применение **constexpr** автоматически подразумевает **const**, так что я не обязан писать

```
constexpr Point move(int dx, int dy) const { return {x+dx,y+dy}; }
```

10.4.4. Аргументы-ссылки

Работая с **constexpr**, нужно помнить ключевую идею о том, что все это относится к значениям. Нет никаких объектов, которые меняют значения и нет никаких побочных эффектов: **constexpr** в миниатюре предоставляет язык функционального программирования времени компиляции. Из сказанного вы можете предположить, что **constexpr** не может иметь дело со ссылками, но это верно лишь частично, поскольку **const**-ссылки относятся к значениям и поэтому могут быть применены. Рассмотрим специализацию **complex<T>** для **complex<double>** из стандартной библиотеки:

```
template<> class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    explicit constexpr complex(const complex<long double>&);

    constexpr double real();            // читаем вещественную часть
    void real(double);                  // устанавливаем вещественную часть
    constexpr double imag();            // читаем мнимую часть
    void imag(double);                  // устанавливаем мнимую часть

    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    // ...
};
```

Очевидно, что операции, такие как = и +=, которые модифицируют объект, не могут квалифицироваться как **constexpr**. А операции, не модифицирующие объекты, такие как **real()** и **imag()**, относятся к **constexpr** и могут выполняться на этапе компиляции. Рассмотрим следующее:

```
constexpr complex<float> z1 {1,2};      // отметим: <float> не <double>
constexpr double re = z1.real();
constexpr double im = z1.imag();
constexpr complex<double> z2 {re,im};   // z2 становится копией z1
constexpr complex<double> z3 {z1};      // z3 становится копией z1
```

Здесь копирующий конструктор работает, поскольку компилятор распознает, что ссылка (**const complex<float>&**) относится к константному значению и мы просто используем это значение.

Литеральные типы способствуют программированию для этапа компиляции, богатому типами. Традиционно здесь все ограничивалось использованием целых значений

(и никаких функций). Это приводило к неоправданно сложновоспринимаемому и потенциально богатому на ошибки коду, ибо приходилось кодировать любую информацию целыми числами. Примерами могут служить некоторые варианты шаблонного метапрограммирования (Глава 28).

10.4.5. Адресные константные выражения

Адрес статически выделенного объекта (§6.4.2), такого как глобальная переменная, является константой. Однако, его значение выявляется компоновщиком (линкером), а не компилятором, так что последний не знает числового значения адресной константы. Это ограничивает возможности константных выражений для указателей и ссылок. Пример:

```
constexpr const char* p1 = "asdf";
constexpr const char* p2 = p1;    // OK
constexpr const char* p2 = p1+2; // ошибка: компилятору неизвестно значение p1
constexpr char c = p1[2];         // OK, c=='d'; компилятор знает значение, адресуемое p1
```

10.5. Неявное приведение типов

Интегральные типы и типы чисел с плавающей запятой (§6.2.1) могут свободно смешиваться в присваиваниях и выражениях. Где только возможно, величины преобразуются так, чтобы не было потери информации. К сожалению, ряд портящих информацию преобразований («сужающие» преобразования) также выполняются неявно. Преобразование не теряет информацию, если вы преобразуете величину к некоторому результату, который может быть преобразован в другую сторону с возвратом к первоначальному значению. В противном случае это *сужающее преобразование* (*narrowing conversion*) (§10.5.2.6). В настоящем разделе мы рассмотрим правила преобразований, возникающие при этом проблемы и способы их преодоления.

10.5.1. Продвижения

Неявные преобразования, не изменяющие фактические значения преобразуемых величин, часто называют *продвижениями* (*promotions*). Перед тем, как выполнить арифметическую операцию, выполняется *интегральное продвижение*, которое призвано преобразовать в **int** величины более коротких целых типов. Аналогично выполняется продвижение для чисел с плавающей запятой, целью которого служит приведение от **float** к **double**. Отметим, что нет продвижения к **long** (кроме как из **char16_t**, **char32_t**, **wchar_t** или из простого перечисления, большего чем **int**) или **long double**. Все это отражает основную цель продвижений из языка C: привести операнды к самому естественному с точки зрения арифметических операций представлению.

Выполняются следующие интегральные продвижения:

- **char**, **signed char**, **unsigned char**, **short int**, **unsigned short int** — в **int**, если **int** в состоянии представить все величины перечисленных типов; в противном случае выполняется приведение к **unsigned int**.
- **char16_t**, **char32_t**, **wchar_t** (§6.2.3) или простой тип перечисления (§8.4.2) — в первый из следующих типов, который в состоянии отразить все значения: **int**, **unsigned int**, **long**, **unsigned long** или **unsigned long long**.
- Битовое поле (§8.2.7) преобразуется в **int**, если **int** может отразить все значения битового поля; в противном случае выполняется преобразование к **unsigned int** (если также возможно преобразование). Наконец, если оба варианта не проходят, то интегральное преобразование не производится.
- **bool** преобразуется в **int**; **false** становится **0**, а **true** становится **1**.

Продвижения выполняются как часть обычных арифметических преобразований (§10.5.3).

10.5.2. Преобразования

Фундаментальные типы могут неявно преобразовываться друг в друга невообразимым числом способов (§iso.4). По моему мнению, их даже слишком много. Например:

```
void f(double d)
{
    char c = d;          // осторожно: приведение double к char
}
```

При написании кода старайтесь избегать неопределенного поведения и преобразований, тихо теряющих информацию (сужающих преобразований).

Компилятор может предупредить вас о подозрительных преобразованиях. По счастью, большинство компиляторов так и делают.

Синтаксис {}-инициализации не допускает сужения (§6.3.5). Пример:

```
void f(double d)
{
    char c {d};          // ошибка: приведение double к char
}
```

Если сужащих преобразований невозможно избежать, то попробуйте применить функцию контролируемого преобразования, такую как **narrow_cast<>()** (§11.5).

10.5.2.1. Интегральные преобразования

Целое всегда можно преобразовать к другому целому типу. Простое перечисление можно преобразовать в целый тип (§8.4.2).

Если целевой тип **unsigned**, то результирующее значение суть те биты из источника, которые умещаются в результирующий тип (биты высокого порядка при необходимости могут отбрасываться). Например:

```
unsigned char uc = 1023; // бинарное 1111111111: uc становится 11111111, т.е. 255
```

Если же целевой тип **signed**, то исходное значение не меняется в том случае, если оно умещается в целевом типе; в противном случае результат зависит от реализации:

```
signed char sc = 1023; // определяется реализацией
```

Возможные результаты: **127** и **-1** (§6.2.3).

Булевские значения или же простые перечисления могут неявно преобразовываться к их целочисленным эквивалентам (§6.2.2, §8.4).

10.5.2.2. Преобразования типов с плавающей запятой

Числа с плавающей запятой могут преобразовываться в иной тип с плавающей запятой. Если исходное значение может в точности быть представлено в целевом типе, то изменения значения, естественно, не происходит. Если исходное значение находится между двумя смежными возможными значениями из целевого типа, то результат совпадет с одним из этих значений. Иначе результат неопределен. Пример:

```
float f = FLT_MAX;          // наибольшее значение float
double d = f;               // OK: d == f
```

```
double d2 = DBL_MAX;          // наибольшее значение double
float f2 = d2;                // не определено, если FLT_MAX<DBL_MAX

long double ld = d2;          // OK: ld = d3
long double ld2 = numeric_limits<long double>::max();
double d3 = ld2;              // не определено, если sizeof(long double)>sizeof(double)
```

DBL_MAX и **FLT_MAX** определены в **<climits>**; **numeric_limits** определяется в **<limits>** (§40.2).

10.5.2.3. Преобразования указателей и ссылок

Указатель на объект любого типа может неявно приводиться к типу **void*** (§7.2.1). Указатель (ссылка) на производный класс может неявно приводиться к типу указателя (ссылки) на доступный и однозначный базовый класс (§20.2). Отметим также, что указатель на функцию и указатель на член класса не может неявно конвертироваться в **void***.

Константное выражение (§10.4), вычисление которого приводит к нулю, может неявно приводиться к нулевому указателю любого типа. Аналогично, такое константное выражение может неявно приводиться к указателю, имеющему тип указателя на член класса (§20.6). Например:

```
int* p = (1+2)*(2*(1-1));   // OK, но вычурно
```

Предпочтение отдавайте **nullptr** (§7.2.2).

Тип **T*** может неявно конвертироваться в **const T*** (§7.5). Аналогично, **T&** может неявно конвертироваться в **const T&**.

10.5.2.4. Преобразование указателей на члены классов

Неявное преобразование указателей и ссылок на члены классов выполняется так, как это описывается в §20.6.3.

10.5.2.5. Преобразования для типа bool

Величины указательных типов, интегральных типов и типов с плавающей запятой могут неявно приводиться к **bool** (§6.2.2). Ненулевые величины конвертируются в **true**; нуль конвертируется в **false**. Примеры:

```
void f(int* p, int i)
{
    bool is_not_zero = p;     // true если p!=0
    bool b2 = i;              // true если i!=0
    // ...
}
```

Приведение указателей к типу **bool** приемлемо в условиях, но в иных случаях может сбить с толку:

```
void fi(int);
void fb(bool);

void ff(int* p, int* q)
{
    if (p) do_something(*p); //OK
    if (q!=nullptr) do_something(*q);    // OK, но избыточно
    // ...
    fi(p);      // ошибка: нет конверсии указателя в int
    fb(p);      // OK: конверсия указателя в bool (сюрприз!?)
}
```

В случае с **fb(p)** надеемся на предупреждение от компилятора.

10.5.2.6. Преобразования интегральных типов и типов с плавающей запятой

Когда значение с плавающей запятой конвертируется в целое, дробная часть этой величины отбрасывается. Другими словами, в этом случае происходит усечение значения. Например, **int(1.6)** дает **1**. Результат неопределен, когда усеченное значение не умещается в целевом типе. Например:

```
int i = 2.7;          // i становится 2
char b = 2000.7;      // не определено для 8-bit char: 2000 нельзя представить как 8-bit char
```

Приведение от целых типов к типам с плавающей запятой корректны математически до той степени, до которой это позволяет аппаратура. Потеря точности происходит, если значение интегрального типа не может быть в точности представлено в виде числа с плавающей запятой. Например:

```
int i = float(1234567890);
```

На машине, где типы **int** и **float** занимают 32 бита, значение **i** будет равно **1234567936**.

Ясно, что в общем случае следует избегать неявных преобразований, изменяющих значения. Самые очевидные случаи, вроде приведения значений с плавающей запятой к интегральным типам или **long int** к **char**, компилятор в состоянии распознавать и выдавать предупреждения. Но в общем случае программист должен сам держать ухо востро. Где этого недостаточно, лучше не стесняться внедрять в код явные проверки. Например:

```
char checked_cast(int i)
{
    char c = i;     // предупреждение: не переносимо (§10.5.2.1)
    if (i != c) throw std::runtime_error{"Int-to-char check failed"};
    return c;
}

void my_code(int i)
{
    char c = checked_cast(i);
    // ...
}
```

Более общие методики проверок преобразований типов представлены в §25.2.5.1.

Чтобы усечения значений были независимы от платформы, нужно использовать **numeric_limits** (§40.2).

Усечения значений при инициализации можно избежать, применяя синтаксис {}-инициализаторов (§6.3.5).

10.5.3. Обычные арифметические преобразования

Эти преобразования выполняются над операндами бинарной операции для приведения их к одному типу, который и будет типом результата:

[1] Если один из операндов имеет тип **long double**, то второй приводится к **long double**.
- Иначе, если один из операндов имеет тип **double**, то второй приводится к **double**.
- Иначе, если один из операндов имеет тип **float**, то второй приводится к **float**.
- Иначе, выполняются интегральные продвижения (§10.5.1) на обоих операндах.

[2] В остальных случаях, если операнд имеет тип **unsigned long long**, то второй операнд приводится к типу **unsigned long long**.
- Иначе, если один операнд имеет тип **long long int**, а второй имеет тип **unsigned long int**, тогда если **long long int** в состоянии представить все значения типа

unsigned long int, то **unsigned long int** конвертируется в **long long int**; в противном случае оба операнда конвертируются в **unsigned long long int**. Иначе, если один из операндов типа **unsigned long long**, то второй конвертируется в **unsigned long long**.

- Иначе, если один из операндов типа **long int**, а второй типа **unsigned int**, тогда если **long int** в состоянии представить все значения типа **unsigned int**, то **unsigned int** конвертируется в **long int**; в противном случае оба операнда конвертируются в **unsigned long int**.
- Иначе, если один из операндов типа **long**, то второй конвертируется в **long**.
- Иначе, если один из операндов типа **unsigned**, то второй конвертируется в **unsigned**.
- Иначе, оба операнда имеют тип **int**.

Из этих правил вытекает, что конвертирование беззнакового целого в знаковый вариант большего размера зависит от реализации. Это еще одна причина избегать смешения беззнаковых и знаковых целых.

10.6. Советы

[1] Предпочитайте стандартную библиотеку всем остальным библиотекам и «самодельному коду»; §10.2.8.

[2] Применяйте посимвольный ввод только тогда, когда этого нельзя избежать; §10.2.3.

[3] При вводе всегда помните о возможности ввода некорректных по форме данных; §10.2.3.

[4] Всегда предпочитайте подходящие абстракции (классы, алгоритмы и т.д.) опоре на голые средства языка (операторы, типы **int**, ...); §10.2.8.

[5] Избегайте переусложненных выражений; §10.3.3.

[6] Если сомневаетесь в приоритете операций, применяйте круглые скобки; §10.3.3.

[7] Избегайте выражений с неопределенным порядком вычислений; §10.3.2.

[8] Избегайте сужающих преобразований; §10.5.2.

[9] Определяйте символические константы и применяйте их вместо «магических чисел»; §10.4.1.

[10] Избегайте сужающих преобразований; §10.5.2

11

Операции

Если кто-то скажет
«Хочу язык программирования,
на котором нужно лишь запросить результат»,
дайте ему леденец.
— Алан Перлис

- Отдельные операции

  Логические операции; Побитовые логические операции; Условные выражения; Инкремент и декремент
- Свободная память

  Управление памятью; Массивы; Выделение памяти; Перегрузка операции **new**
- Списки

  Модель реализации; Квалифицированные списки; Неквалифицированные списки
- Лямбда-выражения

  Модель реализации; Альтернативы; Захват; Вызов и возврат; Тип лямбда-выражений
- Явное преобразование типов

  Конструирование; Именованные преобразования; Преобразования в С-стиле; Функциональный стиль
- Советы

11.1. Отдельные операции

В данном раделе рассматривается целая россыпь простых операций: логические операции (**&&**, **||** и **!**), побитовые логические операции (**&**, **|**, **~**, **<<** и **>>**), условная операция (**?:**), а также инкремент и декремент (**++** и **--**). Между этими операциями мало что есть общего, кроме того, что они не вписались в рассмотрение операций где-либо в ином месте книги.

11.1.1. Логические операции

Логические операции **&&** («И»), **||** («ИЛИ») и **!** («НЕ») принимают операнды арифметических и указательных типов, конвертируют их в **bool**, и возвращают результат типа **bool**. Операции **&&** и **||** вычисляют свой второй аргумент только в случае необходимости и их можно использовать для контроля порядка вычислений (§10.3.2). Например:

```
while (p && !whitespace(*p)) ++p;
```

Здесь указатель **p** не разыменовывается в том случае, когда он равен **nullptr**.

11.1.2. Побитовые логические операции

Побитовые логические операции **&** («И»), **|** («ИЛИ»), **^** («ИСКЛЮЧАЮЩЕЕ ИЛИ»), **~** («ДОПОЛНЕНИЕ»), **>>** («ПРАВЫЙ СДВИГ») и **<<** («ЛЕВЫЙ СДВИГ») применяются к объектам интегральных типов — **char**, **short**, **int**, **long**, **long long** и их **unsigned**-вариантам, а также к **bool**, **wchar_t**, **char16_t** и **char32_t**. Простое перечисление (но не **enum class**) неявно приводится к целому и используется в качестве операнда побитовых логических операций. Обычные арифметические преобразования (§10.5.3) определяют тип результата.

Типичное применение побитовых логических операций состоит в формировании понятия малого множества (битового вектора). В этом случае каждый бит беззнакового целого сопоставляется одному элементу множества, так что число бит лимитирует количество элементов. Бинарная операция **&** интерпретируется как операция пересечения, операция **|** — как операция объединения, **^** — как операция симметричной разности и **~** — как дополнение. Перечисление можно использовать для именования элементов такого множества. Вот небольшой пример, заимствованный из реализации **ostream**:

```
enum ios_base::iostate {
    goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

Реализация потокового класса может устанавливать и проверять его состояние следующим образом:

```
state = goodbit;
// ...
if (state&(badbit|failbit)) // некорректное состояние потока
```

Дополнительные круглые скобки необходимы, поскольку операция **&** имеет приоритет над операцией **|** (§10.3).

Функция, достигающая конца ввода, может сообщить об этом следующим образом:

```
state |= eofbit;
```

Операция **|=** добавляет флаг к состоянию. Простое присваивание **state=eofbit** обнулит остальные биты.

Флаги состояния потока можно читать извне реализации. Например, можно сравнить между собой два состояния потока:

```
int old = cin.rdstate();    // rdstate() возвращает состояние
// ... используем cin ...
if (cin.rdstate()^old) {    // что-нибудь изменилось?
    // ...
}
```

Сравнивать состояния потоков — не такая уж и частая задача. Для других подобных типов выявление различий более существенно. Например, можно сравнивать битовый вектор, представляющий набор обработанных прерываний, с вектором прерываний, ожидающих обработки.

Обратите внимание на то, что вся эта возня с битами взята из реализации потоков, а не из пользовательского интерфейса к ним. Конечно, можно было бы реализовать удобный способ манипуляции битами, но из соображений надежности, переносимости и т.д., лучше держать это на низких уровнях системы. Более общие понятия множества реализованы в стандартной библиотеке типами **set** (§31.4.3) и **bitset** (§34.2.2).

Побитовые логические операции используются для извлечения битовых полей из слов. Например, можно извлечь средние 16 бит из 32-битового **int**:

```
constexpr unsigned short middle(int a)
{
    static_assert(sizeof(int)==4,"unexpected int size");
    static_assert(sizeof(short)==2,"unexpected short size");
    return (a>>8)&0xFFFF;
}

int x = 0xFF00FF00;     // полагаем sizeof(int)==4
short y = middle(x);    // y = 0x00FF
```

Применение битовых полей (§8.2.7) является удобным сокращающим приемом для всех этих сдвигов и наложению масок.

Не следует путать побитовые логические операции с просто логическими операциями: **&&**, **||** и **!**. Последние возвращают **true** или **false** и они удобны для проверок условий в операторах **if**, **while** и **for** (§9.4, §9.5). Например, **!0** (НЕ НУЛЬ) имеет значение **true**, которое конвертируется в **1**, в то время как **~0** (ДОПОЛНЕНИЕ НУЛЯ) — это набор битов со значением единица, что в системах с дополнением до двух представляет собой отрицательное значение **-1**.

11.1.3. Условные выражения

Некоторые операторы **if** бывает удобным заменить на *условные выражения*. Например:

```
if (a <= b)
    max = b;
else
    max = a;
```

То же самое, но в более прямолинейной манере, выражается следующим образом:

```
max = (a<=b) ? b : a;
```

Круглые скобки вокруг условия необязательны, но я предпочитаю применять их для большей ясности.

Условные выражения полезны также еще и потому, что их можно использовать в константных выражениях (§10.4).

Пару выражений **e1** и **e2** можно применять в качестве альтернатив в условном выражении **c?e1:e2**, если они имеют один и тот же тип, или же есть тип **T**, к которому каждый из них может быть преобразован неявно. Для арифметических типов обычные арифметические преобразования (§10.5.3) используются при поиске этого общего типа. Для остальных типов или **e1** может неявно приводиться к типу **e2**, или же допустимо обратное. Наконец, одна из ветвей условного выражения может быть **throw**-выражением (§13.5.1). Например:

```
void fct(int* p)
{
    int i = (p) ? *p : std::runtime_error{"unexpected nullptr};
    // ...
}
```

11.1.4. Инкремент и декремент

Операция **++** используется для прямолинейного указания на инкрементирование значения (вместо длинной комбинации из сложения и присваивания). При условии отсутствия побочных эффектов для **lvalue**, имеем **++lvalue** эквивалентным (по результату) **lvalue+=1** или

lvalue=lvalue+1. При этом инкремент порождает однократное вычисление. Декремент обозначается в аналогичной манере с помощью операции --.

Операции ++ и -- существуют в префиксной и постфиксной формах. Значением ++x будет новое (инкрементированное) значение x. Например, y=++x эквивалентно y=(x=x+1). Значением x++ является старое значение x. Например, y=x++ эквивалентно y=(t=x,x=x+1,t), где переменная t имеет тот же тип, что и x.

Как и прибавление целого значения к указателю, или вычитание из указателя этого значения, операции ++ и -- на указателях интерпретируются в терминах элементов массива, на который показывает указатель; p++ заставляет p указывать на следующий элемент (§7.4.1).

Операции ++ и -- особо часто применяются для инкрементирования (декрементирования) переменных цикла. Например, можно следующим образом копировать нуль-терминированный массив в С-стиле:

```
void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

В языке C++, как и в языке C, применяется такой (многими нелюбимый) стиль программирования с опорой на выражения. Рассмотрим следующее:

```
while (*p++ = *q++) ;
```

Это несколько туманно для не-С-программистов, но поскольку это распространенный стиль программирования, то остановимся на нем подробнее. Представим сначала более традиционный стиль кода для копирования массива символов:

```
int length = strlen(q);
for (int i = 0; i<=length; i++)
    p[i] = q[i];
```

Этот вариант избыточен. Длина нуль-терминированного массива определяется чтением строки вплоть до нахождения нуля. Так мы читаем массив дважды: один раз для вычисления длины, а второй раз — для копирования массива. Попробуем иной вариант:

```
int i;
for (i = 0; q[i]!=0 ; i++)
    p[i] = q[i];
p[i] = 0;              // терминальный нуль
```

Переменную i, которая используется для индексирования, можно устранить, поскольку p и q оба указатели:

```
while (*q != 0) {
    *p = *q;
    p++;               // указывает на следующий символ
    q++;               // указывает на следующий символ
}
*p = 0;                // терминальный нуль
```

Поскольку постфиксный инкремент позволяет нам использовать исходное значение и только после этого инкрементировать его, то можем переписать цикл следующим образом:

```
while (*q != 0) {
    *p++ = *q++;
}
*p = 0;    // терминальный нуль
```

Значением выражения *p++ = *q++ является *q. Поэтому наш пример может быть сокращен вот так:

```
while ((*p++ = *q++) != 0) { }
```

Теперь мы не можем обнаружить, что *q равно нулю до того, как мы скопировали его в *p и инкрементировали p. Следовательно, мы можем устранить финальное присваивание терминального нуля. Мы еще более сокращаем наш пример, заметив, что в пустом блоке нет необходимости и что !=0 избыточно, ибо интегральное условие и так всегда сравнивается с нулем. Таким образом мы и приходим к окончательному варианту:

```
while (*p++ = *q++) ;
```

Является ли эта версия менее читаемой, чем предыдущие? Только не для опытных C- и C++-программистов. Является ли она более эффективной, чем предыдущие версии? За исключением самой первой версии с функцией strlen(), это неочевидно; производительность будет одинаковой и даже сгенерированный машинный код может оказаться одинаковым.

Самым эффективным способом копирования нуль-терминированных строковых массивов в типичном случае является применение стандартной C-функции:

```
char* strcpy(char*, const char*);    // из <string.h>
```

В более общих контекстах можно использовать стандартный алгоритм copy (§4.5, §32.5). Всюду, где возможно, применяйте стандартные средства вместо возни с указателями и байтами. Функции из стандартной библиотеки могут быть встраиваемыми (§12.1.3) и даже могут реализовываться с использованием специализированных машинных команд. Поэтому вам стоит не один раз отмерить, прежде чем решить, что самодельный код предпочтительнее средств стандартной библиотеки. И даже если это не совсем так, то преимущество может растаять в случае иной комбинации «ваш самойдельный код + другой компилятор» или превратится в головную боль для программиста, который потом будет поддерживать программу.

11.2. Свободная память

Время жизни именованного объекта определяется его областью видимости (§6.3.4). Но иногда бывает нужным создать объект, который существует независимо от области видимости, в которой он был создан. Например, типичным является необходимость использовать объекты, созданные в функции, после возврата из этой функции. Операция new создает такие объекты, а операция delete используется для их уничтожения. Эти объекты принято называть объектами в *свободной памяти* (или *объектами на куче*, в *динамической памяти*).

Рассмотрим сейчас вопрос о том, как бы мы могли написать компилятор в стиле, в котором мы создавали настольный калькулятор (§10.2). Функции синтаксического анализа могли бы создавать дерево выражений для их использования генератором кода:

```
struct Enode {
    Token_value oper;
    Enode* left;
    Enode* right;
    // ...
};

Enode* expr(bool get)
```

```
{
    Enode* left = term(get);

    for (;;) {
        switch (ts.current().kind) {
        case Kind::plus:
        case Kind::minus:
            left = new Enode {ts.current().kind,left,term(true)};
            break;
        default:
            return left;      // возвращаем узел
        }
    }
}
```

Новый **Enode** в случаях **Kind::plus** и **Kind::minus** создается в свободной памяти и инициализируется значением **{ts.current().kind,left,term(true)}**. Результирующий указатель присваивается **left** и затем возвращается из **expr()**.

Я использую нотацию {}-списков для указания аргументов. Можно было бы вместо этого использовать старый вариант с ()-списками инициализаторов. В то же время, ошибочно использовать операцию = для инициализации объектов, создаваемых операцией **new**:

```
int* p = new int = 7;          // ошибка
```

Если у типа есть умолчательный конструктор, то мы можем вообще опустить инициализаторы, но встроенные типы останутся неинициализированными. Например:

```
auto pc = new complex<double>;      // complex инициализируется в {0,0}
auto pi = new int;                  // int не инициализирован
```

Чтобы быть уверенным в умолчательной инициализации, применяйте {}. Например:

```
auto pc = new complex<double>{};    // complex инициализируется в {0,0}
auto pi = new int{};                // int инициализирован в 0
```

Генератор кода может использовать объекты типа **Enode**, созданные функцией **expr()**, и уничтожать их:

```
void generate(Enode* n)
{
    switch (n->oper) {
    case Kind::plus:
        // используем n
        delete n; // удаляем Enode из свободной памяти
    }
}
```

Объект, созданный операцией **new**, существует до тех пор, пока он не будет уничтожен операцией **delete**. После этого память, которую он занимал, может быть снова использована операциями **new**. Реализации C++ не обязательно предоставляют «сборщик мусора», который следит за объектами без корректных ссылок на них и сам уничтожает их, возвращая напрасно теряемую память в полезный пул для их последующего использования операциями **new**. Так что остается полагаться на то, что объекты, создаваемые операцией **new**, будут явно уничтожены операциями **delete**.

Операцию **delete** можно применять только к указателю, значение которого возвращено операцией **new**, или же к **nullptr**. В последнем случае операция не производит никакого эффекта.

Если уничтожаемый объект имеет тип класса с деструктором (§3.2.1.2, §17.2), то деструктор автоматически вызывается при этом до того, как память из-под объекта возвращается в свободный пул памяти.

11.2.1. Управление памятью

Основные проблемы при работе со свободной паямтью следующие:

- *Утечка объектов*: программист использует **new**, а вызвать операцию **delete** забывает.
- *Преждевременное уничтожение*: удаляется объект, на который есть еще один указатель, который используется после уничтожения объекта.
- *Двойное уничтожение*: двойное уничтожение вызывает деструктор дважды.

Утечка объектов — это серьезно, ибо в результате программа может исчерпать отводимые ей ресурсы. Преждевременное уничтожение также серьезная проблема, ибо использование указателя на уничтоженный объект влечет за собой такие следствия, как возможное чтение неизвестно каких данных, а запись может повредить другой объект. Вот пример очень плохого кода:

```
int* p1 = new int{99};
int* p2 = p1;                  // потенциально проблема
delete p1;                     // теперь p2 не указывает на корректный объект
p1 = nullptr;                  // дает обманчивое чувство надежности
char* p3 = new char{'x'};      // p3 может указывать на память, указуемую через p2
*p2 = 999;                     // это может породить проблему
cout << *p3 << '\n';           // может и не напечатать x
```

Двойное уничтожение — это тоже проблема, ибо менеджеры ресурсов в типичном случае не могут отслеживать, кто именно сейчас владеет ресурсом. Рассмотрим следующее:

```
void sloppy() // очень плохой код
{
    int* p = new int[1000];  // захватываем память
    // ... используем *p ...
    delete[] p;              // освобождаем память

    // ... чуть подождем ...

    delete[] p;              // sloppy() не владеет *p
}
```

К моменту второго уничтожения объекта память, адресуемая через **p**, может быть уже выделена под другие цели. Заменим тип **int** на **string** и увидим, что деструктор типа **string** пытается читать память, которая может быть выделена под что-то новое, а затем пытается освободить прочитанное. В общем случае, двойное уничтожение приводит к неопределенным результатам и поведение кода при этом непредсказуемое и чаще всего катастрофическое.

Причины, по которым люди допускают такие ошибки, вовсе не состоят в злонамеренности или неряшливости; на самом деле нелегко согласованно освобождать память и уничтожать все ранее созданные в свободной памяти объекты, особенно в нужный момент времени и в нужной части большой программы. Часто изучение отдельных частей программы не позволяет выявлять проблему, поскольку это возможно лишь при изучении кода нескольких отдельных частей этой программы.

В качестве альтернативы использования «голых» **new** и **delete** я рекомендую два основных подхода к управлению ресурсами, которые помогают избежать рассмотренных проблем:

Не помещайте объекты в свободную память, если в этом нет необходимости.

Конструируя объекты в свободной памяти, помещайте указатели на них в объекты-менеджеры (иногда называемые дескрипторами), в деструкторах которых произойдет уничтожение объектов. Примерами служат типы **string**, **vector** и все другие контейнеры стандартной библиотеки, **unique_ptr** (§5.2.1, §34.3.1) и **shared_ptr** (§5.2.1, §34.3.2). Старайтесь держать этот объект-менеджер под управлением именованной области видимости. Многие типовые случаи использования свободной памяти могут быть устранены за счет применения семантики перемещения (§3.3, §17.5.2) при возврате из функций объектов большого размера, представляемых менеджерами объектов.

Рассмотренное правило лежит в основе базовой технологии, помогающей избегать утечек памяти, и делающей обработку ошибок с использованием исключений простой и надежной.

Примером использования этой технологии может послужить тип **vector** из стандартной библиотеки:

```
void f(const string& s)
{
    vector<char> v;
    for (auto c : s)
        v.push_back(c);
    // ...
}
```

Тип **vector** хранит элементы в свободной памяти, но всеми выделениями и освобождениями блоков памяти он управляет сам. В данном примере **push_back()** выполняет операции **new** для выделения пространства памяти под элементы и освобождает эту память операциями **delete** тогда, когда она больше не требуется. А пользователи типа **vector** могут и не знать всех этих подробностей реализации; им остается лишь полагаться на качество типа **vector**, не допускающее утечек памяти.

Тип **Token_stream** из проекта настольного калькулятора является еще более простым примером (§10.2.2). Здесь пользователь вызывает **new** и передает полученный указатель типу **Token_stream** для дальнейшей работы с ним:

```
Token_stream ts{new istringstream{some_string}};
```

Мы не обязаны использовать свободную память лишь для того, чтобы возвращать большие объекты из функций. Например:

```
string reverse(const string& s)
{
    string ss;
    for (int i=s.size()-1; 0<=i; --i)
        ss.push_back(s[i]);
    return ss;
}
```

Как и **vector**, тип **string** является дескриптором своих элементов. Так что **ss** при возврате просто перемещается из **reverse()** вместо копирования всех элементов (§3.3.2).

Идея интеллектуальных указателей (т.е. **unique_ptr** и **smart_ptr**) для управления ресурсами является еще одним примером рассматриваемых нами идей (§5.2.1, §34.3.1). Пример:

```
void f(int n)
{
    int* p1 = new int[n];          // потенциальная проблема
    unique_ptr<int[]> p2 {new int[n]};
```

```
    // ...
    if (n%2) throw runtime_error("odd");
    delete[] p1;          // мы можем сюда и не попасть
}
```

В вызове **f(3)** память, адресуемая через **p1**, утекает, а память, адресуемая через **p2**, корректно освобождается неявным образом.

Окончательно, мое правило про **new**/**delete** таково: «никаких голых **new** и **delete**»; нужно **new** загонять в конструкторы и аналогичные по сути операции, **delete** должны принадлежать деструкторам, и вместе они реализуют согласованную стратегию управления ресурсами. Часто также операции **new** передаются в качестве аргументов дескрипторам ресурсов.

На крайний случай, например когда имеется куча старого кода с массой примеров недисциплинированного использования **new**, можно воспользоваться стандартным интерфейсом к сборщику мусора (§34.5).

11.2.2. Массивы

Операцией **new** можно создавать и массивы объектов. Пример:

```
char* save_string(const char* p)
{
    char* s = new char[strlen(p)+1];
    strcpy(s,p);    // копируем из p в s
    return s;
}

int main(int argc, char* argv[])
{
    if (argc < 2) exit(1);
    char* p = save_string(argv[1]);
    // ...
    delete[] p;
}
```

Операция **delete** применяется для удаления единичных объектов: **delete[]** используется для удаления массивов.

Если ничто вас не заставляет напрямую работать с **char***, то можно применить тип **string** из стандартной библиотеки, заметно упрощающий код функции **save_string()**:

```
string save_string(const char* p)
{
    return string{p};
}

int main(int argc, char* argv[])
{
    if (argc < 2) exit(1);
    string s = save_string(argv[1]);
    // ...
}
```

В частности, **new[]** и **delete[]** исчезли.

Чтобы корректно работать со свободной памятью при помощи операций **new**, **delete** и **delete[]**, нужно знать объем выделяемой памяти. Это подразумевает, что объект, размещаемый в свободной памяти стандартной реализацией операции **new**, занимает в памяти чуть больший объем, чем аналогичный статический объект. Как минимум, нужно хранить

размер памяти под объектом. Обычно на это отводится от одного до двух машинных слов. Большинство современных компьютеров используют 8-байтные слова. В итоге, избыточный объем невелик в случае больших объектов, но может вырасти заметно в случае независимого размещения в свободной памяти большого числа маленьких объектов (вроде объектов типа **int** или **Point**).

Заметим, что объекты типа **vector** (§4.4.1, §31.4) могут сами создаваться в свободной памяти и удаляться из нее операциями **new** и **delete**. Например:

```
void f(int n)
{
    vector<int>* p = new vector<int>(n);    // индивидуальный объект
    int* q = new int[n];        // массив
    // ...
    delete p;
    delete[] q;
}
```

Операцию **delete[]** можно применять лишь к указателю на массив, который возвращается операцией **new**, или к **nullptr** (§7.2.2). Применение **delete[]** к **nullptr** не производит никакого эффекта.

Не применяйте **new** для создания локальных объектов. Пример:

```
void f1()
{
    X* p =new X;
    // ... используем *p ...
    delete p;
}
```

Это избыточно, неэффективно и чревато ошибками (§13.3). В частности, возврат оператором **return** или исключение, сгенерированное до выполнения операции **delete**, приведут к утечке памяти. Вместо свободной памяти работайте с локальнымии переменными:

```
void f2()
{
    X x;
    // ... используем x ...
}
```

Локальная переменная **x** неявно уничтожается по выходу из **f2**.

11.2.3. Выделение памяти

Операции со свободной памятью **new**, **delete**, **new[]** и **delete[]** реализуются с помощью функций, представленных в заголовочном файле **<new>**:

```
void* operator new(size_t);    // выделить память под индивидуальный объект
void operator delete(void* p); // если (p), то деаллоцировать память, выдел. operator new()

void* operator new[](size_t);  // выделить память для массива
void operator delete[](void* p); // если (p), деаллоцировать выделенную под массив память
```

Когда операции **new** нужно получить память под создаваемый объект, она обращается к функции **operator new()** для выделения в свободной памяти нужного числа байт. Аналогично, когда речь идет о массиве объектов, то вызывается функция **operator new[]()**.

Стандартные реализации функций **operator new()** и **operator new[]()** никак не инициализируют возвращаемую память.

Функции выделения и освобождения памяти работают с неинициализированной и нетипизированной памятью (часто называемой «сырой памятью») в противоположность работе с типизированными объектами. Как следствие, они принимают аргументы и возвращают значение типа **void***. А затем операции **new** и **delete** переводят слой нетипизированной памяти в типизированные объекты и наоборот.

Что происходит в случае, когда операция **new** не получает запрошенный объем свободной памяти? Поведение по умолчанию заключается в том, что генерируется исключение **bad_alloc** стандартной библиотеки (насчет альтернативы см. §11.2.4.1). Пример:

```
void f()
{
    vector<char*> v;
    try {
        for (;;) {
            char * p = new char[10000];  // захватить некоторую память
            v.push_back(p);              // обеспечиваем ссылаемость новой памяти
            p[0] = 'x';                  // используем новую память
        }
    }
    catch(bad_alloc) {
        cerr << "Memory exhausted!\n";
    }
}
```

Как бы ни было у нас много памяти, все равно в конце концов здесь будет сгенерировано исключение **bad_alloc**. Но важно также иметь в виду, что несмотря на очевидную ограниченность физической памяти компьютера, логическая память для программы может быть больше и, к тому же, на многих операционных системах может включиться так называемая виртуальная память, когда в случае затруднений с физической памятью выполняется обращение к большей дисковой памяти. В итоге, ждать исключения придется очень долго.

Мы сами можем специфицировать поведение операции **new** в случае проблем с выделением памяти; см. §30.4.1.3.

Помимо функций из заголовочного файла **<new>** пользователь может определить функцию **operator new()** для конкретного класса (§19.2.5). Члены класса, как функция **operator new()**, так и другие члены, имеют приоритет перед функциями из **<new>** согласно обычным правилам областей видимости.

11.2.4. Перегрузка операции new

По умолчанию операция **new** создает объект в свободной памяти. А что, если нам нужно динамически создать объект где-то в другом месте? Рассмотрим простой класс:

```
class X {
public:
    X(int);
    // ...
};
```

Можно создавать объекты где угодно, если определить соответствующим образом аллоцирующую функцию (§11.2.3) с дополнительными аргументами, а затем предоставлять дополнительные аргументы для операции **new**:

```
void* operator new(size_t, void* p) { return p; } // операция явного размещения

void* buf = reinterpret_cast<void*>(0xF00F);      // важный адрес
X* p2 = new(buf) X;                               // конструируем X в buf;
                                                  // вызывается: operator new(sizeof(X),buf)
```

Из-за такого способа использования синтаксис **new(buf) X** для предоставления дополнительного аргумента принято называть *синтаксисом размещения*. Отметим, что любая функция **operator new()** принимает размер в качестве первого аргумента и этот размер объекта передается неявным образом (§19.2.5). Выбор функции **operator new()**, который выполняет операция **new**, реализуется на базе обычных правил соответствия аргументов (§12.3); любой вариант **operator new()** принимает **size_t** в качестве первого аргумента.

«Размещающий» вариант функции **operator new()** является простейшим примером аллокатора. Он определяется в стандартном заголовочном файле **<new>**:

```
void* operator new (size_t sz, void* p) noexcept; // поместить объект размера sz в p
void* operator new[](size_t sz, void* p) noexcept; // поместить объект размера sz в p
void operator delete (void* p, void*) noexcept;    // если (p), делаем *p недействительным
void operator delete[](void* p, void*) noexcept;   // если (p), делаем *p недействительным
```

«Размещающий вариант» **delete** ничего особого не делает, разве что сообщает сборщику мусора, что обработанный им указатель больше не может считаться *надежно выведенным* (*safely derived*) (§34.5).

Конструкцию «размещающего **new**» можно использовать для выделения памяти из фиксированной арены:

```
class Arena {
public:
    virtual void* alloc(size_t) =0;
    virtual void free(void*) =0;
    // ...
};

void* operator new(size_t sz, Arena* a)
{
    return a->alloc(sz);
}
```

Объекты произвольных типов можно при необходимости размещать в разных аренах. Пример:

```
extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
    X* p = new(Persistent) X(i);   // X в долговременной памяти
    X* q = new(Shared) X(i);       // X в разделяемой памяти
    // ...
}
```

Размещая объект в области памяти, не находящейся непосредственно под управлением менеджера свободной памяти, не нужно забывать о том, что потребуется дополнительная осторожность при уничтожении объекта. Базовым механизмом будет явный вызов деструктора:

```
void destroy(X* p, Arena* a)
{
    p->~X();        // вызов конструктора
    a->free(p);     // освобождаем память
}
```

Отметим, что следует избегать явного вызова деструктора всюду, кроме как в реализации классов, предназначенных для управления ресурсами. Почти что все дескрипторы ресурсов могут быть написаны с использованием операций **new** и **delete**. Однако будет

нелегко реализовать эффективный общий контейнер в духе стандартного **vector** (§4.4.1, §31.3.3) без явного вызова деструктора. Новичку следует трижды подумать перед тем, как вызвать деструктор явно, либо посоветоваться с более опытными коллегами.

В §13.6.1 рассказывается, как совмещать «размещающее **new**» с обработкой исключений.

Нет никакого специального синтаксиса для размещения массивов. Произвольные типы могут аллоцироваться при помощи «размещающего **new**». В то же время, **operator delete()** можно определить для массивов (§11.2.3).

11.2.4.1. nothrow версии операции new

В программах, в которых нужно избежать исключений (§13.1.5), можно использовать **nothrow**-версии операций **new** и **delete**. Например:

```
void f(int n)
{
    int* p = new(nothrow) int[n]; // аллоцируем n int в свободной памяти
    if (p==nullptr) {// нет достаточной памяти
        // ... обрабатываем ошибку аллоцирования ...
    }
    // ...
    operator delete(nothrow,p);    // деаллоцируем *p
}
```

Имя **nothrow** относится к объекту стандартного типа **nothrow_t**; **nothrow** и **nothrow_t** определены в **<new>**.

Вот объявления соответствующих функций из заголовочного файла **<new>**:

```
void* operator new(siz e_t sz, const nothrow_t&) noexcept; // аллоцируем sz байт;
                                                           // возврат nullptr при неудаче

void operator delete(void* p, const nothrow_t&) noexcept;   // деаллоцируем выделенную память
void* operator new[](siz e_t sz, const nothrow_t&) noexcept; // аллоцируем sz байт;
                                                             // возврат nullptr при неудаче
void operator delete[](void* p, const nothrow_t&) noexcept;  // деаллоцир. выделенную память
```

Перечисленные функции **operator new** возвращают **nullptr**, а не генерируют исключение типа **bad_alloc** (при невозможности выделить запрошенную память).

11.3. Списки

Помимо применения для инициализации именованных переменных (§6.3.5.2) {}-списки можно использовать в качестве выражений почти что во всех (но все-таки не во всех) местах программы. Они могут представать в двух формах:

С квалификацией типом, **T{...}**, что означает «создать объект типа **T** с инициализатором **T{...}**»; §11.3.2.

Без квалификации типом, **{...}**, когда тип должен определяться из контекста использования; §11.3.3.

Пример:

```
struct S { int a, b; };
struct SS { double a, b; };

void f(S);      // f() принимает S
void g(S);
void g(SS);     // g() перегружена
```

```
void h()
{
    f({1,2});        // OK: вызов f(S{1,2})

    g({1,2});        // ошибка: неоднозначность
    g(S{1,2});       // OK: вызов g(S)
    g(SS{1,2});      // OK: вызов g(SS)
}
```

Эти списки, как и в случае их применения для инициализации именованных переменных (§6.3.5), могут иметь ноль, один и более элементов. Так как {}-список применяется для конструирования объекта некоторого типа, то число элементов в нем и их тип должны соответствовать тому, что требуется для конструирования объекта этого типа.

11.3.1. Модель реализации

Вот модель реализации {}-списков в трех частях:

- Если {}-список используется в качестве аргумента конструктора, реализация такая же, как если бы вы применили ()-список. Элементы списка не копируются иначе как в случае аргументов, передаваемых по значению.
- Если {}-список используется для инициализации элементов агрегата (массива или класса без конструктора), каждый элемент списка инициализирует элемент агрегата. Элементы списка не копируются иначе как в случае аргументов конструкторов для элементов агрегата, передаваемых по значению.
- Если {}-список используется для конструирования объекта типа **initializer_list**, то каждый элемент списка используется для инициализации элемента нижележащего массива у **initializer_list**. В типичном случае элементы копируются из **initializer_list** туда, где мы их используем.

Заметьте, что это общая модель для понимания семантики {}-списков; компилятор может выполнить умную оптимизацию в тех рамках, в которых смысл кода не искажается.

Рассмотрим следующее:

```
vector<double> v = {1, 2, 3.14};
```

Тип **vector** из стандартной библиотеки имеет конструктор со списком инициализации (§17.3.4), так что список инициализации **{1,2,3.14}** интерпретируется как временно сконструированный и используется следующим образом:

```
const double temp[] = {double{1}, double{2}, 3.14 } ;
const initializer_list<double> tmp(temp,sizeof(temp)/sizeof(double));
vector<double> v(tmp);
```

То есть, компилятор конструирует массив, содержащий инициализаторы, конвертированные к желаемому типу (здесь **double**). Этот массив передается конструктору типа **vector** со списком инициализации как **initializer_list**. Затем этот конструктор со списком инциализации копирует величины из массива в собственную структуру данных для элементов. Обратите внимание на то, что объект типа **initializer_list** — это небольшой объект (что-то вроде двух слов), так что передача его по значению имеет смысл.

Нижележащий массив неизменен, так что нет способа изменить смысл {}-списка между двумя актами его использования. Рассмотрим следующий код:

```
void f()
{
    initializer_list<int> lst {1,2,3};
    cout << *lst.begin() << '\n';
```

```
    *lst.begin() = 2;      // ошибка: lst неизменяем
    cout << *lst.begin() << '\n';
}
```

В частности, из факта неизменяемости {}-списка следует, что контейнер, получающий из него элементы, должен использовать копирующие операции, а не перемещающие.

Время жизни {}-списка (и его нижележащего массива) определяется областью видимости, в которой он используются (§6.4.2). В случае, когда список используется для инициализации переменной типа **initializer_list<T>**, он живет столько, сколько живет переменная. Когда список используется в выражении (в том числе и в качестве инициализатора переменной некоторого типа, вроде **vector<T>**), он уничтожается в конце полного выражения.

11.3.2. Квалифицированные списки

Основная идея применения списков инициализации в качестве выражений состоит в том, что если вы можете инициализировать переменную **x** в нотации

```
T x {v};
```

то вы можете создавать объект с тем же значением при помощи выражения **T{v}** или **new T{v}**. В случае **new** объект помещается в свободную память и возвращается указатель на него, в то время как в случае **T{v}** создается временный объект в локальной области видимости. Например:

```
struct S { int a, b; };

void f()
{
    S v {7,8};          // прямая инициализация переменной
    v = S{7,8};         // присваивание: используется квалифицированный список
    S* p = new S{7,8}; // конструирование в свободной памяти с квалифиц. списком
}
```

Правила конструирования объекта с использованием квалифицированных списков в те же самые, что и в случае прямой инициализации (§16.2.6).

Можно взглянуть на квалифицированный список инициализации с одним элементом как на приведение одного типа к другому. Пример:

```
template<class T>
T square(T x)
{
    return x*x;
}

void f(int i)
{
    double d = square(double{i});
    complex<double> z = square(complex<double>{i});
}
```

Эта идея рассматривается также в §11.5.1.

11.3.3. Неквалифицированные списки

Неквалифицированные списки используются там, где ожидаемый тип непротиворечивым образом известен. В качестве выражения может использоваться только в качестве:

- Аргумента функции
- Возвращаемого значения
- Правостороннего операнда операций присваивания (=, +=, *=, и т.д.)
- Индекса

Пример:

```
int f(double d, Matrix& m)
{
    int v {7};          // инициализатор (прямая инициализация)
    int v2 = {7};       // инициализатор (инициализация копированием)
    int v3 = m[{2,3}];  // полагаем, что m берет пары значений как индексы

    v ={8};             // правый операнд присваивания
    v += {88};          // правый операнд присваивания
    {v} = 9;            // ошибка: не левый операнд присваивания
    v = 7+{10};         // ошибка: не операнд не операции присваивания
    f({10.0});          // аргумент функции
    return {11};        // возвращаемое значение
}
```

Причина, по которой неквалифицированный список запрещен в левой части операции присваивания, состоит в том, что грамматика C++ допускает знак { в этой позиции для составных операторов (блоков), и в результате читаемость кода для человека проблематична, а для компиляторов такое противоречивое распознавание будет на грани трюка. Нельзя сказать, что это вообще неразрешимо, но было просто решено не расширять C++ в этом направлении.

Когда неквалифиированые {}-списки применяются для инициализации именованного объекта без применения = (как выше для v), они реализуют прямую инициализацию (§16.2.6). В остальных случаях, выполняется копирующая инициализация (§16.2.6). В частности, избыточный в остальных случаях знак = ограничивает набор инициализаций, которые можно выполнить с данным {}-списком.

Стандартный тип initializer_list<T> используется для работы с {}-списками переменной длины (§12.2.3). Распространенным случаем является их использование в качестве списков инициализации для пользовательских контейнеров (§3.2.1.3), но можно их использовать и напрямую; например:

```
int high_value(initializer_list<int> val)
{
    int high = numeric_traits<int>lowest();
    if (val.size()==0) return high;

    for (auto x : val)
        if (x>high) high = x;

    return high;
}

int v1 = high_value({1,2,3,4,5,6,7});
int v2 = high_value({-1,2,v1,4,-9,20,v1});
```

{}-списки представляют собой простейший способ работы с гомогенными списками переменной длины. Однако, нужно помнить, что нулевые элементы — это отдельный случай. Тогда этот случай должен обрабатываться конструктором по умолчанию (§17.3.3).

Тип {}-списка может быть выведен только в том случае, когда все элементы одного типа. Например:

```
auto x0 = {};       // ошибка (нет типа элементов)
auto x1 = {1};      // initializer_list<int>
```

```
auto x2 = {1,2};    // initializer_list<int>
auto x3 = {1,2,3};  // initializer_list<int>
auto x4 = {1,2.0};  // ошибка: негомогенный список
```

К сожалению, мы не можем вывести тип неквалифицированного списка для простого аргумента шаблона. Пример:

```
template<typename T>
void f(T);

f({});          // ошибка: не известен тип инициализатора
f({1});         // ошибка: неквалифицир. список не соответствует "plain T"
f({1,2});       // ошибка: неквалифицир. список не соответ. "plain T"
f({1,2,3});     // ошибка: неквалифицир. список не соответ. "plain T"
```

Я сказал «к несчастью», потому что это ограничение языка, а не фундаментальное правило. В приницпе, было бы возможно вывести тип для этих {}-списков в виде **initializer_list<int>**, как это делается для **auto**-инициализаторов.

Аналогично, мы не выводим тип элементов контейнера, представленного как шаблон. Например:

```
template<class T>
void f2(const vector<T>&);

f2({1,2,3});            // ошибка: невозможно вывести T
f2({"Kona","Sidney"});  // ошибка: невозможно вывести T
```

Это тоже вызывает сожаление, но это можно понять с технической точки зрения на язык C++: нигде в этих вызовах не сказано **vector**. Для вывода **T** компилятор сначала должен понять, что пользователь на самом деле хочет **vector**, а затем ему нужно заглянуть в определение **vector**, чтобы решить, есть ли у него конструктор, который принимает **{1,2,3}**. В общем случае это потребует конкретизации **vector** (§26.2). С этим можно было бы справиться, но время компиляции было бы слишком большим, да и потенциальные неоднозначности в случае перегруженных версий **f2()** также вызывают опасения. Для вызова **f2()** следует действовать более определенно:

```
f2(vector<int>{1,2,3});             // OK
f2(vector<string>{"Kona","Sidney"}); // OK
```

11.4. Лямбда-выражения

Лямбда-выражения (иногда говорят о *лямбда-функциях* или даже просто о *лямбдах*) — это упрощенная нотация для определения и использования анонимных функциональных объектов. Вместо того, чтобы определять именованный класс с функцией **operator()**, затем создавать объект этого класса с тем, чтобы наконец осуществить нужный вам вызов, можно все это сильно сократить. Особенно это полезно, когда нужно передать операцию в качестве аргумента алгоритма. В контексте графического пользовательского интерфейса современных операционных систем такие оперции (функции) принято называть *функциями обратного вызова* (*callbacks*). В настоящем разделе мы сфокусируемся на технической строне дела; примеры и методики использования лямбд можно найти в других местах книги (§3.4.3, §32.4, §33.5.2).

Лямбда-выражение состоит из последовательности следующих частей:

- Потенциально пустого *списка захвата* (*capture list*), специфицирующего, какие имена из контекста определения могут использоваться в теле лямбды и копируются

ли они или же передаются по ссылке. Список захвата ограничивается знаками [] (§11.4.3).

- Опционального списка параметров, перечисляющего нужные для лямбда-выражения аргументы. Список параметров ограничивается знаками () (§11.4.4).
- Опционального спецификатора **mutable**, указывающего на то, что тело лямбды может модифицировать ее состояние (например, изменять значения копий переменных, захваченных по значению) (§11.4.3.4).
- Опционального спецификатора **noexcept**.
- Опционального декларатора типа возврата в форме ->type (§11.4.4).
- Тела, задающего код для выполнения. Тело ограничивается знаками {} (§11.4.3).

Подробности передачи аргументов, возврата результатов и специфицирование тела такие же как у функций и представлены в Главе 12. Понятие захвата локальных переменных для функций не определено. Это позволяет лямбде действовать как локальная функция (а обычная функция не может).

11.4.1. Модель реализации

Лямбда-выражения могут быть реализованы множеством способов, и существуют эффективные методы их оптимизации. Я, однако, считаю полезным разобраться с семантикой лямбд, рассматривая их как способ сократить определение и использование функциональных объектов. Рассмотрим несложный пример:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    for_each(begin(v),end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; }
    );
}
```

Чтобы разобраться с тем, что это значит, определим эквивалентный функциональный объект:

```
class Modulo_print {
    ostream& os;        // члены для хранения захваченного списка
    int m;
public:
    Modulo_print(ostream& s, int mm) :os(s), m(mm) {}    // захват
    void operator()(int x) const
        { if (x%m==0) os << x << '\n'; }
};
```

Список захвата, **[&os,m]**, превращается в два члена-данных и в инициализирующий их конструктор. Знак **&** перед **os** означает, что следует хранить ссылку, а отсутствие этого знака перед **m** означает, что мы будем хранить копию. Указанное применение знака **&** совпадает с таковым в объявлении аргументов функций.

Тело лямбды превращается в тело метода **operator()()**. Так как лямбда не возвращает значения, **operator()()** объявляется с типом возврата **void**. Кроме того, **operator()()** объявлен с **const**, поскольку в теле лямбды захваченные переменные не модифицируются. Если бы в теле лямбды ее состояние модифицировалось, то лямбда объявлялась бы как **mutable** (§11.4.3.4). В таком случае соответствующий метод **operator()()** объявлялся бы без **const**.

Объект класса, генерируемый из лямбды, принято называть объектом-замыканием (или просто *замыканием* — *closure*).

Наконец, перепишем функцию **print_modulo()** следующим образом:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    for_each(begin(v),end(v),Modulo_print{os,m});
}
```

В случае, когда лямбда захватывает все локальные переменные по ссылке (используется список захвата в виде **[&]**), замыкание может быть оптимизировано так, что будет содержать лишь указатель на охватывающий кадр стека.

11.4.2. Альтернативы

Последняя версия **print_modulo()** вполне привлекательна, и, вообще, именование нетривиальных операций является неплохой идеей. Отдельно определяемый класс оставляет много места для комментариев, больше, чем лямбда, погруженная в список аргументов.

Однако, многие лямбды весьма коротки и используется в программе лишь единожды. В таких случаях эквивалентом служит локальный класс, определяемый непосредственно перед его единичным использованием. Пример:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    class Modulo_print {
        ostream& os; // члены для хранения захваченного списка
        int m;
    public:
        Modulo_print (ostream& s, int mm) :os(s), m(mm) {} // захват
        void operator()(int x) const
            { if (x%m==0) os << x << '\n'; }
    };

    for_each(begin(v),end(v),Modulo_print{os,m});
}
```

Из этого примера наглядно видно, что лямбда выигрывает за явным преимуществом. Если нам нужно лишь имя, то можно поименовать и лямбду:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    auto Modulo_print = [&os,m] (int x) { if (x%m==0) os << x << '\n'; };

    for_each(begin(v),end(v),Modulo_print);
}
```

Именуя лямбду, мы в большей степени сосредотачиваем на ней наше внимание и поэтому, волей-неволей, рассматриваем дизайн операции более тщательно. Это также немного улучшает запись кода и допускает рекурсию (§11.4.5).

Еще одной альтернативой будет применение цикла **for** вместо лямбд с **for-each()**:

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    for (auto x : v)
        if (x%m==0) os << x << '\n';
}
```

Многие, наверное, сочтут этот код более ясным, чем все варианты лямбд. Однако, **for_each()** это весьма специальный алгоритм, а **vector<int>** — специфический контейнер. Поэтому обобщим **print_modulo()** для случая произвольного контейнера:

```
template<class C>
void print_modulo(const C& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    for (auto x : v)
        if (x%m==0) os << x << '\n';
}
```

Эта версия прекрасно работает для **map**. В C++ оператор range-**for** предназначен для траверса последовательностей от самого начала до их конца. STL-контейнеры легко допускают такие траверсы. Например, применение оператора **for** для траверса контейнера **map** реализует траверс вида *depth-first* (сначала *проход в глубину*). Если нам нужен вариант траверса *breadth-first* (сначала *проход в ширину*), то нужно использовать соответствующий алгоритм:

```
template<class C>
void print_modulo(const C& v, ostream& os, int m)
    // вывести v[i] в os если v[i]%m==0
{
    breadth_first(begin(v),end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; }
    );
}
```

Таким образом, лямбда может быть использована как «тело» обобщенной конструкции цикл/траверс, представленной алгоритмом. Если здесь вместо **breadth_first** использовать **for_each**, то будет реализован вариант траверса *depth-first*.

Производительность кода с лямбдой в качестве аргумента алгоритма траверса примерно идентична таковой для эквивалентного цикла. В общем, вывод такой: выбор между «алгоритм плюс лямбда» или «**for** с телом цикла» может основываться на стилевых соображениях, а также на возможности расширения и удобства поддержки.

11.4.3. Захват

Основным способом использования лямбд является формирование кода, который передается в качестве аргумента. Лямбды позволяют делать это «на лету», без необходимости именования функций (или функциональных объектов). Некоторым лямбдам не нужно обращаться к своему локальному окружению; такие лямбды определяются с пустыми квадратными скобками []. Например:

```
void algo(vector<int>& v)
{
    sort(v.begin(),v.end());    // сортируем значения
    // ...
    sort(v.begin(),v.end(),[](int x, int y) { return abs(x)<abs(y); }); // сорт. абсолют. знач-я
    // ...
}
```

Если же имеет место обращение к локальным именам, то об этом нужно сообщить специально, иначе будет ошибка компиляции:

```
void f(vector<int>& v)
{
    bool sensitive = true;
```

```
    // ...
    sort(v.begin(),v.end(),
      [](int x, int y) { return sensitive ? x<y : abs(x)<abs(y); } // ошибка: недоступен sensitive
    );
}
```

Я здесь использовал пустой вводный элемент определения лямбды (lambda introducer) []. В этом случае нет возможности ссылаться на имена из контекста вызова. Вводный элемент определения лямбды может иметь несколько форм:

- []: пустой список захвата. Это означает, что в теле лямбды невозможно использовать локальные имена из окружающего контекста. В таких лямбда-выражениях информация черпается из аргументов или из нелокальных переменных.
- [&]: неявный захват по ссылке. Все локальные имена можно использовать. Все локальные переменные доступны по ссылке.
- [=]: неявный захват по значению. Все локальные имена можно использовать. Все локальные имена доступны как копии локальных переменных, взятые в точке вызова лямбды.
- [*список-захвата*]: явный захват; *список-захвата* — это список имен локальных переменных, подлежащих захвату (т.е. хранению в объекте) по ссылке или по значению. Имена с предшествующими знаками **&** захватываются по ссылке. Остальные переменные захватываются по значению. В списке захвата может присутствовать **this** и ... после имен.
- [&, *capture-list*]: неявный захват по ссылке всех переменных, имена которых не входят в представленный список захвата. В списке захвата может присутствовать **this**. Имена, входящие в список, не могут снабжаться знаком **&**. Имена из списка подлежат захвату по значению.
- [=, *capture-list*]: неявный захват по значению всех переменных, имена которых не входят в представленный список захвата. В списке захвата не может присутствовать **this**. Имена, входящие в список, обязаны снабжаться знаком **&**. Имена из списка подлежат захвату по ссылке.

Заметьте, что локальные имена с предшествующим **&** всегда захватываются по ссылке, а имена без такого знака — всегда по значению. Только захват по ссылке позволяет модифицировать переменные в контексте вызова.

Все представленные варианты для *capture-list* позволяют в точности обозначить, какие имена из контекста вызова используются и как именно. Пример:

```
void f(vector<int>& v)
{
    bool sensitive = true;
    // ...
    sort(v.begin(),v.end()
        [sensitive](int x, int y) { return sensitive ? x<y : abs(x)<abs(y); }
    );
}
```

Упомянув **sensitive** в списке захвата, мы делаем это имя доступным из лямбды. Не указав ничего дополнительно, мы гарантируем захват **sensitive** по значению; как и в общем случае передачи аргументов, передача по значению реализуется по умолчанию. Если бы мы хотели захватить **sensitive** по ссылке, нам пришлось бы указать знак **&** перед **sensitive** в списке захвата: **[&sensitive]**.

В своей основе выбор захвата по ссылке или по значению в точности такой же, как и при передаче аргументов функциям (§12.2). Мы захватываем переменную по ссылке, если нужно модифицировать ее значение или же имеет место слишком большой размер переменной. Но для лямбд нужно принять во внимание еще одно соображение о том, что лямбда может прожить дольше того, кто ее вызывает (§11.4.3.1). При передаче лямбды в другой поток, захват по значению ([=]) является в общем случае предпочтительным: доступ к стеку вызова другого потока по ссылке или по указателю может либо ухудшить производительность, либо вообще привести к краху, а таковой доступ в случае терминированного потока порождает трудноуловимые ошибки.

Если вам нужно захватить шаблонный аргумент с переменным числом параметров (§28.6), то следует использовать .... Например:

```
template<typename... Var>
void algo(int s, Var... v)
{
    auto helper = [&s,&v...] { return s*(h1(v...)+h2(v...)); }
    // ...
}
```

Конечно, не стоит слишком полагаться на интеллект механизма захвата. Очень часто имеется возможность выбора между захватом и передачей аргументов; имейте в виду, что у захвата есть проблемы с типизацией и высокая вероятность получить неочевидное поведение.

11.4.3.1. Лямбды и время жизни

Лямбда может пережить того, кто ее вызывает. Это происходит тогда, когда мы передаем лямбду другому потоку или когда вызыватель сохраняет лямбду где-нибудь для последующего использования. Пример:

```
void setup(Menu& m)
{
    // ...
    Point p1, p2, p3;
    // вычисляем позиции p1, p2, и p3
    m.add("draw triangle",[&]{ m.draw(p1,p2,p3); });    // возможна катастрофа
    // ...
}
```

Полагая, что **add()** добавляет пару (имя, действие) к текущему меню и что операция **draw()** имеет смысл, мы остаемся с тикающей бомбой: **setup()** завершается, а затем — спустя, возможно, минуты — пользователь щелкает мышью по пункту меню **draw triangle** и лямбда пытается обратиться к давно уже уничтоженным локальным переменным. А лямбда, которая пытается писать в переменную, захваченную по ссылке, еще хуже в такой ситуации.

Если лямбда в состоянии пережить вызвавший ее объект, то мы должны гарантировать, что вся локальная информация (если таковая имеется) надежно скопирована в объект-замыкание и что значения возвращаются чрез стандартный механизм возврата (оператор **return**) (§12.1.4) или посредством подходящих аргументов. Для примера с **setup()** все это легко реализуется:

```
m.add("draw triangle",[=]{ m.draw(p1,p2,p3); });
```

Можно думать о списке захвата как о списке инициализации для объекта-замыкания и **[=]** и **[&]** как о сокращающих записях (§11.4.1).

11.4.3.2. Имена из пространств имен

Нет нужды захватывать имена из пространств имен (включая глобальные имена), поскольку они и так доступны (при условии, что они находятся в области видимости). Пример:

```
template<typename U, typename V>
ostream& operator<<(ostream& os, const pair<U,V>& p)
{
    return os << '{' << p.first << ',' << p.second << '}';
}

void print_all(const map<string,int>& m, const string& label)
{
    cout << label << ":\n{\n";
    for_each(m.begin(),m.end(),
        [](const pair<string,int>& p) { cout << p << '\n'; }
    );
    cout << "}\n";
}
```

Здесь нет необходимости захватывать **cout** или операцию вывода для **pair**.

11.4.3.3. Лямбды и this

Как можно обратиться к членам классового объекта из лямбды, которая является членом класса? Можно включить члены класса в набор имен, потенциально захватываемых добавлением **this** в список захвата. Это используется, когда мы хотим использовать лямбду в реализации члена-функции. Например, у нас мог бы быть класс для построения запросов и извлечения результатов:

```
class Request {
    function<map<string,string>(const map<string,string>&)> oper;   // операция
    map<string,string> values;                                      // аргументы
    map<string,string> results;                                     // цели
public:
    Request(const string& s);                                       // парсим и храним запрос

    void execute()
    {
        [this]() { results=oper(values); }    // применяем oper к values и получаем results
    }
};
```

Члены классов всегда захватываются по ссылке. То есть, **[this]** предполагает, что члены достигаются через **this**, а не копируются в лямбду. К сожалению, **[this]** и **[=]** несовместимы. Это предотвращает неосторожное применение лямбды, при котором возможна гонка за данными в мультипотоковых программах (§42.4.6).

11.4.3.4. Лямбды с ключевым словом mutable

Обычно мы не собираемся модифицировать состояние объекта-функции (замыкания), так что по умолчанию мы и не можем этого делать. То есть, **operator()()** в генерируемом функциональном объекте (§11.4.1) является **const** членом-функцией. В маловероятном случае, когда мы хотим менять состояние (в противоположность модификации состояния некоторых переменных, захваченных по ссылке; §11.4.3), мы можем объявить лямбду с ключевым словом **mutable**. Пример:

```
void algo(vector<int>& v)
{
    int count = v.size();
    std::generate(v.begin(),v.end(),
        [count]()mutable{ return --count; }
    );
}
```

Здесь мы уменьшаем копию размера **v**, хранящуюся в замыкании.

11.4.4. Вызов и возврат

Правила передачи аргументов лямбдам такие же, как и для функций (§12.2), и такова же ситуация с возвратом результатов (§12.1.4). Фактически, за исключением правил для захвата (§11.4.3) остальные правила для лямбд заимствованы из аналогичных правил для функций и классов. Однако, две особенности стоит отметить:

Если лямбда не принимает аргументов, то список аргументов можно опустить. Так, минимальная лямбда есть **[]{}**.

Тип возврата из лямбды может выводиться из ее тела. К сожалению, это не верно в отношении функций.

Если у лямбды отсутствует оператор возврата **return**, тип возврата лямбды есть **void**. Если тело лямбды состоит из единственного оператора **return**, тип возврата лямбды совпадает с типом выражения из оператора **return**. В остальных случаях мы должны явно предоставить тип возврата. Например:

```
void g(double y)
{
    [&]{ f(y); }                                          // тип возврата void
    auto z1 = [=](int x){ return x+y; }                   // тип возврата double
    auto z2 = [=,y]{ if (y) return 1; else return 2; }    // ошибка: тело слишком сложное
                                                          // для вывода типа возврата
    auto z3 =[y]() { return 1 : 2; }                      // тип возврата int
    auto z4 = [=,y]()->int { if (y) return 1; else return 2; }  // ОК: явный тип возврата
}
```

Если используется суффиксное определение типа возврата, то нельзя опускать список аргументов (даже если он пустой).

11.4.5. Тип лямбда-выражений

Чтобы допускать оптимизированные версии лямбда-выражений, их тип в явном виде не определяется. Тем не менее, считается, что они являются функциональными объектами (§11.4.1). Такой тип, называемый *типом замыканий*, уникален для лямбд, так что никакие две лямбды не имеют один и тот же тип. Если бы две лямбды могли иметь одинаковый тип, то тогда механизм конкретизации шаблонов мог бы дать сбой. Лямбды относятся к локальным классам с конструктором и с **const**-методом **operator()()**. Кроме использования лямбд в качестве аргументов, мы можем использовать их для инициализации переменной, объявленной с ключевым словом **auto** или **std::function<R(AL)>**, где **R** это тип возврата, а **AL** — это список типов аргументов (§33.5.3).

Например, можно было бы попытаться написать лямбду для реверсирования символов в С-строке:

```
auto rev = [&rev](char* b, char* e)
        { if (1<e-b) { swap(*b, *--e); rev(++b,e); } }; // ошибка
```

К сожалению, такое решение не проходит, поскольку я не могу использовать **auto**-переменную до того, как выведен ее тип. Вместо этого, я могу ввести имя, а затем использовать его:

```
void f(string& s1, string& s2)
{
    function<void(char* b, char* e)> rev =
        [&](char* b, char* e) { if (1<e-b) { swap(*b, *--e); rev(++b,e); } };

    rev(&s1[0],&s1[0]+s1.size());
    rev(&s2[0],&s2[0]+s2.size());
}
```

В этом варианте тип **rev** специфицирован до того, как **rev** используется.

Если нам просто нужно поименовать лямбду, вместо рекурсивного обращения к ней, **auto** поможет все упростить:

```
void g(vector<string>& vs1, vector<string>& vs2)
{
    auto rev = [&](char* b, char* e) { while (1<e-b) swap(*b++,*--e); };

    rev(&s1[0],&s1[0]+s1.size());
    rev(&s2[0],&s2[0]+s2.size());
}
```

Ничего не захватывающая лямбда может быть присвоена указателю на функцию соответствующего типа. Например:

```
double (*p1)(double) = [](double a) { return sqrt(a); };
double (*p2)(double) = [&](double a) { return sqrt(a); };  // ошибка: лямбда захватывает
double (*p3)(int) = [](int a) { return sqrt(a); };        // ошибка: несоответ. типов аргументов
```

11.5. Явное преобразование типов

Иногда нам нужно привести значение одного типа к значению другого типа. Много (даже слишком много) таких преобразований выполняется неявно по правилам языка (§2.2.2, §10.5). Например:

```
double d = 1234567890; // целое в вещественное
int i = d;             // вещественное в целое
```

В других случаях мы должны явно выполнять приведение типов.

По историческим и логическим причинам C++ предлагает ряд операций явного приведения типов разной степени удобства и безопасности:

- Конструирование, используя {}-нотацию, которая обеспечивает безопасное по типам конструирование нового значения (§11.5.1)
- Именованные преобразования разной степени внешней непривлекательности:
  - **const_cast** для предоставления права на запись в то, что объявлено с ключевым словом **const** (§7.5)
  - **static_cast** для выполнения хорошо определенных преобразований (§11.5.2)
  - **reinterpret_cast** для изменения смысла битовых последовательностей (§11.5.2)
  - **dynamic_cast** для навигации по классовым иерархиям с динамической проверкой (§22.2.1)
- C-стиль приведений (§11.5.3)
- Функциональный стиль (§11.5.4)

Я упорядочил эти типы преобразований согласно моим предпочтениям, а также по безопасности их результатов.

За исключением {}-конструирования, мне они не очень нравятся, но **dynamic_cast** хоть выполняет динамическую проверку. Для преобразований между двумя скалярными арифметическими типами я предпочитаю использовать собственную функцию **narrow_cast** там, где возможно сужение значения:

```
template<class Target, class Source>
Target narrow_cast(Source v)
{
    auto r = static_cast<Target>(v);    // конверсия значения в целевой тип
    if (static_cast<Source>(r)!=v)
        throw runtime_error("narrow_cast<>() failed");
    return r;
}
```

Если я могу преобразовать некоторое значение к целевому типу, затем конвертировать результат назад к исходному типу и получить оригинальное значение, то я счастлив. Это просто обобщение правил языка, применяемых к величинам при {}-инициализации (§6.3.5.2). Пример:

```
void test(double d, int i, char* p)
{
    auto c1 = narrow_cast<char>(64);
    auto c2 = narrow_cast<char>(-64);          // генерирует, если char беззнаковый
    auto c3 = narrow_cast<char>(264);          // генерирует, если char 8-bit знаковый

    auto d1 = narrow_cast<double>(1/3.0F);     // OK
    auto f1 = narrow_cast<float>(1/3.0);       // возможно генерирует

    auto c4 = narrow_cast<char>(i);            // может генерировать
    auto f2 = narrow_cast<float>(d);           // может генерировать

    auto p1 = narrow_cast<char*>(i);           // ошибка времени выполнения
    auto i1 = narrow_cast<int>(p);             // ошибка времени выполнения

    auto d2 = narrow_cast<double>(i);      // может генерировать (возможно не будет)
    auto i2 = narrow_cast<int>(d);             // может генерировать
}
```

В зависимости от того, как вы работаете с числами с плавающей запятой, может быть разумным применить проверку диапазона значений при выполнений соответствующих преобразований, а не применять операцию !=. Это легко сделать при помощи специализаций (§25.3.4.1) или свойств типов (§35.4.1).

11.5.1. Конструирование

Конструирование значения типа **T** из величины типа **e** записывается согласно нотации **T{e}** (§iso.8.5.4). Пример:

```
auto d1 = double{2};          // d1==2.0
double d2 {double{2}/4};      // d1==0.5
```

Часть положительных особенностей нотации **T{v}** заключается в том, что при этом выполняются только хорошо специфицированные преобразования. Например:

```
void f(int);
void f(double);
```

```
void g(int i, double d)
{
    f(i);                          // вызов f(int)
    f(double{i});                  // ошибка: {} не конвертирует int в double

    f(d);                          // вызов f(double)
    f(int{d});                     // ошибка: {} нет усечения
    f(static_cast<int>(d));        // вызов f(int) с усеченным значением

    f(round(d));                   // вызов f(double) с округленным значением
    f(static_cast<int>(lround(d))); // вызов f(int) с округленным значением
                                   // если d велико для int, то все равно усечение
}
```

Я не считаю усечение значений с плавающей запятой (например, от **7.9** к **7**) «хорошим поведением», так что явное приведение типов для такого случая весьма кстати. Если округление желательно, то следует использовать функцию стандартной библиотеки **round()**; она выполняет стандартное округление, например от **7.9** к **8** и от **7.4** к **7**.

Иногда кажется удивительным, что **{}**-конструирование не допускает конверсию из **int** в **double**, но следует учесть, что в случае, когда размер **int** совпадает с размером **double**, то некоторые из таких преобразований приведут к потере информации. Рассмотрим следующее:

```
static_assert(sizeof(int)==sizeof(double),"unexpected sizes");

int x = numeric_limits<int>::max();      // наибольшее возможное целое
double d = x;
int y = x;
```

Мы здесь не получим **x==y**. В то же время, мы по-прежнему можем инициализировать **double** целым литералом, который может быть представим точно. Например:

```
double d { 1234 }; // отлично
```

Явная квалификация желаемым типом не допускает плохих преобразований. Пример:

```
void g2(char* p)
{
    int x = int{p};       // ошибка: нет конверсии char* в int
    using Pint = int*;
    int* p2 = Pint{p};    // ошибка: нет конверсии char* в int
    // ...
}
```

Для **T{v}** «разумное поведение» определяется как «не допускающее сужений» (§10.5) преобразование из **v** к **T**, или же имеющее соответствующий конструктор для **T** (§17.3).

Нотация в стиле конструктора **T{}** применяется для отражения умолчательного значения типа **T**. Например:

```
template<class T> void f(const T&);

void g3()
{
    f(int{});                  // значение int по умолчанию
    f(complex<double>{});      // значение complex по умолчанию
    // ...
}
```

Для встроенных типов умолчательное значение в случае явного применения нотации конструктора есть **0**, преобразованный к нужному типу (§6.3.5). Таким образом, **int{}** —

это еще один способ написать **0**. Для пользовательского типа **T** нотация **T{}** трактуется согласно умолчательному конструктору этого типа (§3.2.1.1, §17.6), если он есть, иначе посредством умолчательного конструирования, **MT{}**, каждого члена.

Явно сконструированные неименованные объекты являются временными и (не будучи связанными ссылками) их время жизни лимитируется полным выражением, в котором они используются (§6.4.2). В этом они отличаются от неименованных объектов, создаваемых операцией **new** (§11.2).

11.5.2. Именованные преобразования

Некоторые виды преобразований типов не столь просты и не гарантируют проверку типов; они не представляют собой простое конструирование значений из заданного набора исходных значений (аргументов). Например:

```
IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00); // устр-во по адресу 0Xff00
```

Компилятор не может знать, является ли число **0Xff00** действительным адресом. Из-за этого корректность преобразования находится целиком в руках программиста. Явное преобразование типов иногда бывает неизбежным. В то же время, на практике его применяют слишком часто, что нередко приводит к ошибкам.

Еще одним типичным примером необходимости в явном преобразовании типов является работа с «сырой памятью», то есть с памятью, которая предназначена для хранения объектов, тип которых компилятору неизвестен. Например, аллокатор памяти (такой как, например, **operator new()**; §11.2.3) может возвращать указатель типа **void*** на вновь выделенный блок памяти:

```
void* my_allocator(size_t);

void f()
{
    int* p = static_cast<int*>(my_allocator(100)); // аллокация для int
    // ...
}
```

Компилятор не знает тип объекта, адресуемого указателем типа **void***.

Фундаментальной идеей именованных преобразований типов является попытка сделать их как можно более заметными в коде программы, а также для явного отражения намерений программиста:

- **static_cast** конвертирует связанные типы, вроде как от одного типа указателя к другому типу указателя в рамках одной иерархии классов, от интегрального типа к перечислению, от типа чисел с плавающей запятой к интегральному типу. Он также выполняет преобразования, определенные конструкторами (§16.2.6, §18.3.3, §iso.5.2.9) и операциями преобразования (§18.4).
- **reinterpret_cast** обрабатывает преобразования между несвязанными типами, вроде как от целого к типу указателя или же от указателя к другому типу указателя, никак не связанному с первым из них (§iso.5.2.10).
- **const_cast** выполняет преобразования между типами, отличающимися друг от друга квалификаторами **const** или **volatile** (§iso.5.2.11).
- **dynamic_cast** реализует динамически контролируемое преобразование указателей и ссылок в классовой иерархии (§22.2.1, §iso.5.2.7).

Специализация среди перечисленных именованных преобразований типов помогает компилятору выполнить некоторую проверку типов, а также помогает программисту быстро

находить наиболее опасные пребразования, представленные с помощью **reinterpret_cast**. Никаких гарантий для **reinterpret_cast** не предоставляется, а его действие заключается в трактовке той же самой последовательности бит как нового типа. Если целевой тип содержит по крайней мере не меньше бит, чем исходный тип, то мы можем применить **reinterpret_cast** в обратную сторону (в сторону исходного типа) и использовать результат. Гарантируется, что результат операции **reinterpret_cast** может использоваться, если он конвертируется в обратную сторону к исходному типу. Отметим, что **reinterpret_cast** это тот тип преобразования, который применяется к указателям на функцию (§12.5). Рассмотрим следующее:

```
char x = 'a';
int* p1 = &x;                          // ошибка: нет неявной конверсии char* в int*
int* p2 = static_cast<int*>(&x);       // ошибка: нет неявной конверсии char* в int*
int* p3 = reinterpret_cast<int*>(&x);  // OK: на свою голову

struct B { /* ... */ };
struct D : B { /* ... */ };            // см. §3.2.2 и §20.5.2

B* pb = new D;                         // OK: неявная конверсия из D* в B*
D* pd = pb;                            // ошибка: нет неявной конверсии из B* в D*
D* pd = static_cast<D*>(pb);           // OK
```

Преобразования между классовыми указателями и классовыми ссылками обсуждаются в §22.2.

Если вы собираетесь использовать явные приведения типов, то найдите немного времени обдумать, действительно ли вам это необходимо. В C++ отсутствует необходимость в явном приведении типов в большинстве случаев, в отличие от языка C (§1.3.3) и от ранних версий C++ (§1.3.2, §44.2.3). В большинстве программ можно избежать явного приведения типов; в остальных случаях явное приведение типов следует локализовать в немногочисленных процедурах.

11.5.3. Преобразования в С-стиле

От языка C язык C++ наследовал нотацию **(T)e**, в которой может быть выражено любое преобразование как комбинация **static_cast**, **reinterpret_cast** и **const_cast** для получения значения типа **T** из выражения **e** (§44.2.3). К сожалению, этот тип преобразований допускает преобразование от типа указателя на класс к типу указателя на его закрытый базовый класс. Никогда не делайте такого и надейтесь на предупреждение от компилятора в случае, если вы по ошибке все-таки сделаете это. Преобразования в С-стиле более опасны, чем именованные преобразования языка C++, ибо их и выявить в коде программы труднее, и они не отражают истинных намерений программиста. В этой нотации, **(T)e**, возможны и переносимые преобразования между связанными типами, и непереносимые преобразования между несвязанными типами, и устранение ограничений на указатели с модификатором **const**. Невозможно отличить эти случаи, не зная истинных типов **T** и **e**.

11.5.4. Функциональный стиль

Конструирование значения типа **T** из величины **e** можно выразить, используя нотацию в функциональном стиле **T(e)**. Пример:

```
void f(double d)
{
    int i = int(d);              // усечение d
    complex z = complex(d);      // делаем complex из d
    // ...
}
```

К сожалению, для встроенных типов **T(e)** эквивалентно **(T)e** (§11.5.3). Отсюда следует, что для многих встроенных типов преобразование **T(e)** также небезопасно.

```
void f(double d, char* p)
{
    int a = int(d);      // усечение
    int b = int(p);      // не переносимое
    // ...
}
```

Даже явное преобразование из более длинных типов в более короткие (например, **long** в **char**) может привести к непереносимому коду, зависящему от реализации.

Отдавайте предпочтение **T{v}**-преобразованиям для штатного конструирования значений и именованные преобразования (такие как **static_cast** и др.) в иных случаях.

11.6. Советы

[1] Предпочитайте префиксную операции **++**, а не суффиксную; §11.1.4.

[2] Используйте дескрипторы ресурсов, чтобы избегать утечек, преждевременного уничтожения объектов или двойного уничтожения; §11.2.1.

[3] Не размещайте объекты в свободной памяти, если это возможно; предпочитайте переменные областей видимости; §11.2.1.

[4] Избегайте «голых» **new** и **delete**; §11.2.1.

[5] В работе с ресурсами применяйте проверенные технологии; §11.2.1.

[6] Отдавайте предпочтение именованным объектам-функциям перед лямбдами, если требуется снабдить операцию комментариями; §11.4.2.

[7] Отдавайте предпочтение именованным объектам-функциям перед лямбдами, если операцию предполагается использовать многократно; §11.4.2.

[8] Сохраняйте лямбды короткими; §11.4.2.

[9] Будьте осторожны с захватом по ссылке из-за проблем с поддержкой и возможных ошибок; §11.4.3.1.

[10] Позвольте компилятору вывести тип возврата лямбды; §11.4.4

[11] Используйте **T{e}** нотацию для конструирования; §11.5.1.

[12] Избегайте явных приведений типов; §11.5.

[13] Когда явное приведение типа неизбежно, применяйте именованные преобразования; §11.5.

[14] Рассмотрите возможность использования преобразования, такого как **narrow_cast<>()**, выполняющего проверку на этапе выполнения; §11.5.

12

Функции

Смерть фанатикам.
— Парадокс

- Объявления функций

  Почему функции?; Составные части объявления функции; Определение функции; Возврат значений; **inline**-функции; **constexpr**-функции; **[[noreturn]]**-функции; Локальные переменные
- Передача аргументов

  Ссылочные аргументы; Аргументы-массивы; Аргументы-списки; Неопределенное количество аргументов; Аргументы по умолчанию
- Перегрузка функций

  Автоматическое разрешение перегрузки; Перегрузка и тип возврата; Перегрузка и область видимости; Разрешение перегрузки при множественных аргументах; Ручное разрешение перегрузки
- Предусловия и постусловия
- Указатели на функции
- Макросы

  Условная компиляция; Предопределенные макросы; Прагмы
- Советы

12.1. Объявления функций

Основным способом что-то сделать в программе на языке C++ является вызов функции. Определяя функцию, вы формулируете, как именно должна быть выполнена та или иная операция. Функцию нельзя вызвать, если она не была заранее объявлена.

Объявление функции предоставляет в распоряжение программиста имя функции, тип возврата (если он есть) и число и тип ее аргументов, которые должны передаваться функции при ее вызове. Например:

```
Elem* next_elem();      // нет аргументов; возврат Elem*
void exit(int);         // int аргумент; нет возврата
double sqrt(double);    // double аргумент; возврат double
```

Семантика передачи аргументов идентична семантике инициализации (§16.2.6). Проверяется тип аргументов и при необходимости выполняется неявное приведение типов аргументов. Пример:

```
double s2 = sqrt(2);          // вызов sqrt() с аргументом double{2}
double s3 = sqrt("three");    // ошибка: sqrt() требует аргумента double
```

Не стоит недооценивать важность проверки типов и их неявное преобразование.

В объявление функции можно внедрять имена аргументов. Это помогает программисту разобраться в сути дела, но в чистом объявлении (не являющимся определением) компилятор имена аргументов просто игнорирует. Ключевое слово **void** в качестве типа возврата функции означает, что функция не возвращает ничего (§6.2.7).

Тип функции формируется из типа возврата и типов аргументов. Для функций-членов класса (§2.3.2, §16.2) имя класса также относится к типу функции. Например:

```
double f(int i, const Info&);   // тип: double(int,const Info&)
char& String::operator[](int);  // тип: char& String::(int)
```

12.1.1. Почему функции?

Существует долгая традиция написания очень больших функций — в сотни строк кода. Однажды я встретился с функций, написанной вручную, которая содержала буквально 32768 строк кода. Авторы подобных функций недооценивают одну из самых важных особенностей функций — их применение в целях декомпозиции сложных задач на небольшие, четко определенные и понятные куски кода (в виде классов и функций), имеющие значащие имена. Важно сохранять код функции понятным, ибо это важный шаг на пути реализации успешной поддержки кода в будущем. Итак, первым шагом в решении вычислительной задачи нужно сделать шаг по ее разбиению на осмысленные именованные куски кода в виде функций и классов. Такие функции в дальнейшем составят словарь данной задачи, так же как типы (встроенные и пользовательские) составляют словарь данных. Для хорошего старта хорошую службу сослужат алгоритмы стандартной библиотеки (вроде **find**, **sort** и **iota**; Глава 32). Затем можно будет из этих функций создавать более крупные для решения общих или специализированных задач.

Количество ошибок в разрабатываемом коде сильно коррелирует с его размером и сложностью. С этими проблемами помогают справиться многочисленные короткие функции. Применяя функцию для решения специфической задачи, мы избавляем себя от необходимости вставлять этот код куда-то в середину ранее написанного кода; кроме того, написание отдельной функции заставляет нас осмысленно именовать эту часть кода и документировать ее зависимости. Дополнительно, вызывая функцию и получая возврат от нее, мы часто освобождаемся от необходимости плодить лишние операторы **goto** и **continue** (§9.6, §9.5.5). Если только эти операторы не формируют регулярную структуру, то в остальных случаях они являются причиной неизбежных ошибок.

Простым советом относительно размера функции является рекомендация ограничивать размер ее кода так, чтобы можно было просто охватить его взглядом на экране компьютера. Ошибки часто прокрадываются в те щели, которые образуются между обозреваемыми за один раз частями функции. В итоге многие программисты на подсознательном уровне любят ограничивать функции сорока строками кода. А я бы их еще больше ограничивал — где-нибудь семью строками кода.

В большинстве случаев цена вызова функции малосущественна. А там, где эта цена имеет значение (например, в случае часто вызываемых функций доступа вроде индексирования вектора), там можно воспользоваться встраиванием функций (§12.1.5). Полезно использовать функции в качестве структурирующего инструмента.

12.1.2. Составные части объявления функции

Помимо имени, набора аргументов и возвращаемого типа объявление функции может содержать множество спецификаторов и модификаторов. Всего имеется:

- Имя функции (обязательно)
- Список аргументов, возможно пустой () (обязательно)
- Тип возврата, возможно **void**, который может быть префиксным или суффиксным (с использованием **auto**) (обязательно)
- **inline**, что означает пожелание вместо вызова функции реализовать встраивание ее тела (§12.1.5)
- **constexpr**, что означает намерение вычислять функцию во время компиляции при условии, что аргументы являются константными выражениями (§12.1.6)
- **noexcept**, означающее, что функция не может генерировать исключения (§13.5.1.1)
- Спецификатор компоновки, например **static** (§15.2)
- **[[noreturn]]**, означающее, что функция не соблюдает стандартный механизм возврата (§12.1.4)

Кроме перечисленного функции-члены классов могут иметь еще дополнительно ключевые слова:

- **virtual** — означает возможность замещения в производном классе (§20.3.2)
- **override** — означает необходимость замещения виртуальной функции базового класса (§20.3.4.1)
- **final** — означает невозможность замещения в производном классе (§20.3.4.2)
- **static** — означает, что функция не ассоциируется с отдельным объектом класса (§16.2.12)
- **const** — означает, что она не модифицирует состояние объекта (§3.2.1.1, §16.2.9.1)

В случае, если вам хочется вызвать головную боль у тех, кто будет читать ваш код, может написать что-либо вроде этого:

```
struct S {
    [[noreturn]] virtual inline auto f(const unsigned long int *const) -> void const noexcept;
};
```

12.1.3. Определение функции

Любая функция, которую вызывают в программе, должна быть где-либо определена (причем однократно; §15.2.3). Определение функции — это такое объявление, в котором присутствует тело функции. Пример:

```
void swap(int*, int*);          // объявление

void swap(int* p, int* q)       // определение
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

Определение функции и все ее объявления должны относиться к одному и тому же типу. К сожалению, из-за совместимости с языком C, на самом высоком уровне ключевое слово **const** у аргументов игнорируется. Вот два объявления одной функции:

```
void f(int);          // тип void(int)
void f(const int);    // тип void(int)
```

Эта функция может иметь вот такое определение:

```
void f(int x) { /*здесь можно модифицировать x */ }
```

А можно определить **f()** и так:

```
void f(const int x) { /*здесь нельзя модифицировать x */ }
```

В обоих случаях, аргумент, который **f()** может или не может модифицировать, является копией того, что предоставляет вызывающий функцию, так что нет острой проблемы в том, что функция неожиданно изменит что-то в контексте вызывающего кода.

Имена аргументов функции не относятся к типу функции и не должны быть идентичными в разных ее объявлениях. Пример:

```
int& max(int& a, int& b, int& c);          // возврат ссылки на большее из a, b, и c

int& max(int& x1, int& x2, int& x3)
{
    return (x1>x2)? ((x1>x3)?x1:x3) : ((x2>x3)?x2:x3);
}
```

Именование аргументов в объявлениях (не являющихся определением) опционально и служит некоторой формой документирования. Наоборот, можно подчеркнуть то, что аргумент не используется в теле определения функции, просто опустив его имя. Например:

```
void search(table* t, const char* key, const char*)
{
    // третий аргумент не используется
}
```

Часто неименованные аргументы возникают в случае упрощения кода функций или же в случае планирующихся модификаций в будущем. В обоих случаях неименованные аргументы гарантируют, что вызывающий код не будет их использовать.

Кроме функций, есть еще объекты, которые могут быть вызваны; они подчиняются большинству правил, относящихся к функциям, например правилам передачи аргументов (§12.2):

- *Конструкторы* (§2.3.2, §16.2.5) в техническом отношении не являются функциями; в частности, у них нельзя указывать тип возврата (даже **void**), они могут инициализировать базовые части объектов и их члены (§17.4), а также нет возможности взять их адрес.
- *Деструкторы* (§3.2.1.2, §17.2) не могут перегружаться и нельзя взять их адрес.
- *Объекты-функции* (§3.4.3, §19.2.2) не есть функции (это объекты) и они не могут перегружаться, но их члены **operator()** суть функции.
- *Лямбда-выражения* (§3.4.3, §11.4) по сути являются сокращениями для функциональных объектов.

12.1.4. Возврат значений

Любое объявление функции содержит спецификацию типа возврата функции (кроме конструкторов и функций приведения типа). Традиционно в C и C++ тип возврата стоит в самом начале объявления функции (перед именем функции). Однако, можно написать объявление функции так, чтобы тип возврата указывался после списка аргументов. Например, два последующих объявления эквивалентны:

```
string to_string(int a);          // префиксный тип возврата
auto to_string(int a) -> string;  // суффиксный тип возврата
```

То есть префикс **auto** указывает на то, что тип возврата появится после списка аргументов. В таком синтаксисе типу возврата предшествует знак **->**.

Серьезным применением суффиксной формы возврата является его использование в объявлениях шаблонных функций, в которых тип возврата зависит от аргументов. Пример:

```
template<class T, class U>
auto product(const vector<T>& x, const vector<U>& y) -> decltype(x*y);
```

В то же время, суффиксный синтаксис написания типа возврата может применяться во всех случаях. Имеется очевидная корреляция между суффиксным синтаксисом возврата для функций и синтаксисом лямбда-выражений (§3.4.3, §11.4); жаль, что эти конструкции не являются идентичными.

Если у функции возврата нет, то в качестве типа возврата указывается **void**.

Если функция объявлена не с **void**, то она обязана возвращать значение (специальный случай - функция **main()**; см. §2.2.1). Если указана с **void** — то не может возвращать ничего. Пример:

```
int f1() { }              // ошибка: нет возврата
void f2() { }             // OK

int f3() { return 1; }    // OK
void f4() { return 1; }   // ошибка: возврат в void функции

int f5() { return; }      // ошибка: отсутствие возвращаемого значения
void f6() { return; }     // OK
```

Возвращаемое значение определяется оператором **return**. Например:

```
int fac(int n)
{
    return (n>1) ? n*fac(n-1) : 1;
}
```

Функция, которая вызывает саму себя, называется *рекурсивной*.

В функции может быть более одного оператора **return**:

```
int fac2(int n)
{
    if (n > 1)
        return n*fac2(n-1);
    return 1;
}
```

Как и семантика передачи аргументов, семантика возврата функцией значения идентична семантике инициализации копированием (§16.2.6). Оператор **return** инициализирует переменную типа, обозначенного в качестве типа возврата. Тип возвращаемого значения проверяется относительно объявленного типа возврата и все стандартные и определенные пользователем преобразования выполняются. Пример:

```
double f() { return 1; }  // 1 неявно конвертируется в double{1}
```

Каждый раз, когда функция вызывается, создаются новые копии аргументов и создаются локальные переменные. Отводимая под это память освобождается после возврата из функции, так что не стоит возвращать указатель на нестатическую локальную переменную. Содержимое такой памяти предсказать невозможно:

```
int* fp()
{
    int local = 1;
    // ...
    return &local; // плохо
}
```

Эквивалентные ошибки возникают при использовании ссылок:

```
int& fr()
{
    int local = 1;
    // ...
    return local;    // плохо
}
```

По счастью, компилятор может предупредить о возврате ссылок на локальные переменные.

Значений типа **void** не существует. Однако, можно использовать вызов **void**-функции в качестве возвращаемого значения **void**-функции. Пример:

```
void g(int* p);

void h(int* p)
{
    // ...
    return g(p);    // OK: эквивалентно "g(p); return;"
}
```

Эта форма возврата полезна для того, чтобы избежать особых случаев при написании шаблонных функций, где тип возврата является параметром шаблона.

Оператор **return** является одним из пяти способов покинуть функцию:

- Выполнение оператора **return**.
- Выход за конец кода тела функции. Разумеется, это допускается лишь в функциях, не возвращающих значений (то есть, в **void**-функциях), и в функции **main()**, где такой выход означает успешное завершение (§12.1.4).
- Генерирование исключения, не перехватываемого локально (§13.5).
- Завершение из-за генерации исключения, не перехватываемого локально, в функции со спецификацией **noexcept** (§13.5.1.1).
- Прямым или косвенным вызовом системной функции без возврата (например, **exit()**; §15.4).

Функция без нормального возврата (т.е. не через **return** или исчерпанием кода тела функции) может быть маркирована как **[[noreturn]]** (§12.1.7).

12.1.5. inline-функции

Функцию можно определить со спецификатором **inline**. Пример:

```
inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

Спецификатор **inline** служит подсказкой компилятору, что тот должен попытаться сгенерировать код для прямой вставки вместо обычного вызова функции. Например, «умный» компилятор мог бы вставить здесь просто число **720** вместо вызова **fac(6)**. Качество «интеллекта» компилятора может разниться, так что один компилятор сгенерирует число **720**, другой сгенерирует выражение **6*fac(5)**, в то время как найдется и такой, который оставит вызов **fac(6)**. Если вам нужна гарантия, что значение будет вычислено на этапе компиляции, объявите функцию с ключевым словом **constexpr** и обеспечьте гарантию, что все, с чем придется столкнуться при вычислении, относится к **constexpr** (§12.1.6).

Чтобы встраивание стало возможным без совсем уж нереально умного компилятора, определение функции (а не объявление) должно быть все время в области видимости (§15.2). Спецификатор **inline** не оказывает влияния на семантику функции. В частности, **inline**-функции по-прежнему имеют уникальный адрес, и то же верно в отношении статических переменных (§12.1.8) из этих функций.

В случае определения **inline**-функции в разных единицах трансляции (типично из-за частого определения этих функций в заголовочных файлах; §15.2.2) все определения из разных единиц трансляции должны быть идентичными (§15.2.3).

12.1.6. constexpr-функции

В общем случае функцию нельзя вычислить на этапе компиляции, так что она не может включаться в константные выражения (§2.2.3, §10.4). Уточняя функцию ключевым словом **constexpr**, мы намекаем на то, что хотим ее использовать в константных выражениях, при условии передачи ей аргументов в виде константных выражений. Пример:

```
constexpr int fac(int n)
{
    return (n>1) ? n*fac(n-1) : 1;
}

constexpr int f9 = fac(9);      // вычисляется во время компиляции
```

Применение ключевого слова **constexpr** двоякое. В определении функции оно означает «применимо в константных выражениях в случае аргументов в виде константных выражений». Если же оно применяется в определении объекта, то уже значит «вычислить инициализатор во время компиляции». Например:

```
void f(int n)
{
  int f5 = fac(5);              // может вычисляться во время компиляции
  int fn = fac(n);              // вычисляется во время выполнения (n это переменная)

  constexpr int f6 = fac(6);    // должно вычисляться во время компиляции
  constexpr int fnn = fac(n);   // ошибка: нет гарантии вычис-я во время компил. (n – перемен.)
  char a[fac(4)];               // OK: границы массива константы и fac() есть constexpr
  char a2[fac(n)];              // ошибка: границы массива должны быть констант. и n перемен.

  // ...
}
```

Возможность вычисления на этапе компиляции предполагает достаточную простоту функции: **constexpr**-функция должна состоять из единственного оператора **return**; никаких циклов и никаких локальных переменных. Также не должно быть никаких побочных эффектов. Пример:

```
int glob;

constexpr void bad1(int a)    // ошибка: constexpr ф-я не может быть void
{
    glob = a;                 // ошибка: побочный эффект в constexpr функции
}

constexpr int bad2(int a)
{
    if (a>=0) return a; else return -a;  // ошибка: if-оператор в constexpr функции
}
constexpr int bad3(int a)
```

```
{
    sum = 0;                                    // ошибка: локаль. переменная в constexpr ф-ии
    for (int i=0; i<a; +=i) sum +=fac(i);       // ошибка: цикл в constexpr функции
    return sum;
}
```

Правила для **constexpr**-конструкторов также отличаются (§10.4.3); в них разрешены только простые инициализации членов.

Для **constexpr**-функций допускаются рекурсии и условные выражения. В то же время может оказаться, что отладка станет неоправданно сложной и время компиляции сильно возрастет, если только вы не ограничитесь применением этих функций для выполнения относительно несложных задач (для чего они реально и предназначены).

С использованием литеральных типов (§10.4.3) **constexpr**-функции могут определяться и для пользовательских типов.

Как и в случае встраиваемых функций, **constexpr**-функции должны полчиняться правилу одного определения, так что их определения в разных единицах трансляции должны быть идентичными (§15.2.3).

12.1.6.1. constexpr-функции и ссылки

constexpr-функции не могут иметь побочных эффектов, поэтому изменение нелокальных объектов недопустимо. Но ссылаться на нелокальные объекты **constexpr**-функции могут до тех пор, пока на пытаются прописывать их значения.

```
constexpr int ftbl[] { 1, 2, 3, 5, 8, 13 };

constexpr int fib(int n)
{
    return (n<sizeof(ftbl)/sizeof(*ftbl)) ? ftbl[n] : fib(n);
}
```

constexpr-функции могут принимать ссылочные аргументы. Разумеется, они не могут писать через эти ссылки, но **const**-ссылки здесь также полезны, как и в иных местах. Например, в стандартной библиотеке можно найти такой вот код (§40.4):

```
template<> class complex<float> {
public:
// ...
    explicit constexpr complex(const complex<double>&);
    // ...
};
```

Мы можем теперь написать следующее:

```
constexpr complex<float> z {2.0};
```

Временная переменная, логически предназначенная для хранения аргумента типа **const**-ссылки, просто становится величиной, внутренней для компилятора.

Для **constexpr**-функций допускается возврат ссылок или указателей. Например:

```
constexpr const int* addr(const int& r) { return &r; } // OK
```

Правда, это уводит нас в сторону от основной области применения **constexpr**-функций в вычислениях константных выражений. Кроме того, бывает крайне сложно удостовериться, что возврат таких функций является константным выражением. Рассмотрим следующее:

```
static const int x = 5;
constexpr const int* p1 = addr(x);      // OK
constexpr int xx = *p1;                 // OK

static int y;
constexpr const int* p2 = addr(y);      // OK
constexpr int yy = *y;                  // ошибка: попытка читать переменную

constexpr const int* tp = addr(5);      // ошибка: адрес временного объекта
```

12.1.6.2. Вычисления с условиями

Ветви условных выражений, не подпадающие под вычисления в **constexpr**-функциях, могут полагаться на вычисления времени выполнения. Пример:

```
constexpr int check(int i)
{
    return (low<=i && i<high) ? i : throw out_of_range();
}

constexpr int low = 0;
constexpr int high = 99;

// ...
constexpr int val = check(f(x,y,z));
```

Можно представлять **low** и **high** конфигурационными параметрами, известными на этапе компляции, но не ранее (на этапе разработки), и что **f(x,y,z)** вычисляет некоторое зависящее от реализации значение.

12.1.7. [[noreturn]]-функции

Конструкция [[...]] называется *атрибутом*, который можно помещать почти-что всюду. В общем случае, атрибут задает некоторое зависящее от реализации свойство, которое относится к следующей за атрибутом синтаксической сущности. Также атрибут можно разместить перед объявлением. Существует всего два стандартных атрибута (§iso.7.6), и **[[noreturn]]** один из них. Другой атрибут **[[carries_dependency]]** (§41.3).

Помещая **[[noreturn]]** в начало объявления функции, мы отмечаем, что от функции не нужно ждать возврата. Например:

```
[[noreturn]] void exit(int); // exit не возвратит ничего
```

Явное указание на то, что функция ничего не возвращает, помогает как в понимании кода, так и в его генерации. Что происходит, когда **[[noreturn]]**-функция все-таки вовзращает значение, в синтаксисе языка C++ не определено.

12.1.8. Локальные переменные

Все имена, которые определяются внутри функции, принято называть *локальными*. Локальная переменная или константа инициализируются тогда, когда поток исполнения доходит до точки их определения. Кроме как в случае квалификации **static**, каждый вызов функции порождает собственную копию локальной переменной. Но в случае **static** локальная переменная представляет собой один статически выделенный объект (§6.4.2), который используется во всех вызовах функции. Он инициализируется единственный раз, когда поток исполнения первый раз проходит точку его определения. Пример:

```
void f(int a)
{
    while (a--) {
        static int n = 0;    // инициализируется однократно
        int x = 0;           // инициализируется 'a' раз в каждом вызове f()

        cout << "n == " << n++ << ", x == " << x++ << '\n';
    }
}

int main()
{
    f(3);
}
```

Этот код порождает следующий вывод:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

Статические локальные переменные позволяют функции сохранять информацию между вызовами без необходимости обращаться к глобальным переменным, которые доступны иным функциям и могут быть повреждены последними (см. также §16.2.12).

Инициализация статических локальных переменных не приводит к гонке за данными (§5.3.1), кроме как в случае, когда функция обращается к ним рекурсивно или же возникает блокировка (§iso.6.7). То есть, реализация C++ должна защитить инициализацию локальной статической переменной с помощью некой конструкции, свободной от блокировки (вроде **call_once**; §42.3.3). Эффект от рекурсивной инициализации локальной статической переменной не определен. Пример:

```
int fn(int n)
{
    static int n1 = n;          // OK static
    static int n2 = fn(n-1)+1;  // неопределенно
    return n;
}
```

Статические локальные переменные помогают избежать свойственных нелокальным переменным зависимостям от порядка их инициализации (§15.4.1).

Не существует локальных функций; если вы чувствуете нужду в подобных функциях, используйте функциональный объект или лямбда-выражение (§3.4.3, §11.4).

Областью видимости метки является функция, независимо от того, в сколь вложенной области видимости находится она сама.

12.2. Передача аргументов

Когда функция вызывается (используя ()-синтаксис, который известен как *операция вызова*), выделяется память под ее *формальные аргументы* (часто называемые *параметрами*) и каждый формальный аргумент инициализируется соответствующим фактическим аргументом. Семантика передачи параметров идентична семантике инициализации (инициализации копированием, если быть точным; §16.2.6). Тип фактических аргументов соотносится с типом формальных параметров и все стандартные и пользовательские преобразования типов выполняются. Если формальный параметр не является ссылкой, то функции передается копия фактического аргумента. Например:

```
int* find(int* first, int* last, int v)  // ищем x в [first:last)
{
    while (first!=last && *first!=v)
        ++first;
    return first;
}

void g(int* p, int* q)
{
    int* pp = find(p,q,'x');
    // ...
}
```

Здесь аргумент **p** не модифицируется функцией **find()**, так как указатель передается по значению.

Существуют правила передачи массивов (§12.2.2), средства для передачи аргументов без проверки (§12.2.4) и средства для передачи аргументов по умолчанию (§12.2.5). Использование списков инициализации исследуется в §12.2.3, а способы передачи аргументов шаблонным функциям рассматриваются в §23.5.2 и §28.6.2.

12.2.1. Ссылочные аргументы

Рассмотрим следующее:

```
void f(int val, int& ref)
{
    ++val;
    ++ref;
}
```

При вызове **f()** инкрементируется копия первого фактического аргумента, в то время как **++ref** инкрементирует фактический второй аргумент. Теперь рассмотрим вот такой код:

```
void g()
{
    int i = 1;
    int j = 1;
    f(i,j);
}
```

Вызов **f(i,j)** инкрементирует **j**, но не **i**. Первый аргумент, **i**, передается *по значению*; второй аргумент, **j**, передается *по ссылке*. Как об этом уже говорилось в §7.7, функции, которые модифицируют аргументы-ссылки, непросты в восприятии и понимании сторонними программистами, так что лучше их избегать. В то же время, передача больших аргументов по ссылке является заметно более эффективной. В таких случаях ссылку лучше всего объявлять с модификатором **const**, что точно отражает наши намерения добиться эффективности и не позволяет при этом ненароком модифицировать значение соответствующего объекта:

```
void f(const Large& arg)
{
    // значение "arg" не может быть изменено
    // (кроме как явной конверсией типа; §11.5)
}
```

Отсутствие **const** перед аргументом-ссылкой служит фактически отражением намерения модифицировать соответствующую переменную;

```
void g(Large& arg); // предположим, что g() модифицирует arg
```

Аналогично, объявляя аргумент-указатель с модификатором **const**, мы точно сообщаем читателю кода о нашем намерении защитить от изменения объект, на который настроен указатель, в теле функции. Например:

```
int strlen(const char*);                    // число символов в строке в С-стиле
char* strcpy(char* to, const char* from);   // копируем С-строку
int strcmp(const char*, const char*);       // сравниваем С-строки
```

Чем больше размер программы, тем более важны в ней **const**-аргументы (слишком сложно уследить за всем непосредственно глазами без «железной» помощи со стороны компилятора).

Заметьте, что семантика передачи аргументов отличается от семантики присваивания. Это важно в случае **const**-аргументов, ссылочных аргументов и аргументов некоторых пользовательских типов.

Согласно правилам инициализации ссылок литерал, константа и аргумент, требующий преобразования типа, могут передаваться как **const T&**, но не как **T&**. Разрешение преобразований типа для **const T&**-аргументов гарантирует, что таким аргументам могут назначаться те же самые значения, что и **T**-аргументам, передающимся по значению с помощью временных объектов при необходимости. Например:

```
float fsqrt(const float&); // sqrt принимает ссылочный аргумент (стиль Фортрана)

void g(double d)
{
    float r = fsqrt(2.0f);      // передача ссылки на времен. перем-ю значением 2.0f
    r = fsqrt(r);               // передача ссылки на r
    r = fsqrt(d);               // перед. ссылки на врем. пер-ю со значением static_cast<float>(d)
}
```

Запрет преобразований для ссылочных неконстантных аргументов (§7.7) позволяет избегать глупых ошибок, возникающих в связи с введением временных объектов. Пример:

```
void update(float& i);

void g(double d, float r)
{
    update(2.0f);   // ошибка: const-аргумент
    update(r);      // передача ссылки на r
    update(d);      // ошибка: требуется конверсия типа
}
```

Если бы такие вызовы были разрешены, **update()** по-тихому модифицировал бы временные объекты, которые немедленно были бы удалены. Как правило, это бы стало сюрпризом для программиста.

Если выражаться совсем точно, то передача по ссылке есть передача по lvalue-ссылке. А функция может принимать и rvalue-ссылки. Как об этом рассказано в §7.7, rvalue может связываться с rvalue-ссылкой (но не с lvalue-ссылкой), а lvalue может связываться с lvalue-ссылкой (но не с rvalue-ссылкой). Пример:

```
void f(vector<int>&);           // (не-const) lvalue-ссылка в качестве аргумента
void f(const vector<int>&);     // const lvalue-ссылка в качестве аргумента
void f(vector<int>&&);          // rvalue-ссылка в качестве аргумента

void g(vector<int>& vi, const vector<int>& cvi)
{
    f(vi);                      // вызов f(vector<int>&)
    f(vci);                     // вызов f(const vector<int>&)
    f(vector<int>{1,2,3,4});    // вызов f(vector<int>&&);
}
```

Мы должны полагать, что функция модифицирует rvalue-аргумент, оставляя его годным лишь для уничтожения или для повторного присваивания (§17.5). Самым очевидным примером использования rvalue-ссылок являются перемещающие конструкторы и перемещающие присваивания (§3.3.2, §17.5.2). Я полагаю, что кто-нибудь найдет полезным применение аргументов вида **const**-rvalue ссылок, но я не встречал реальных полезных примеров для такого случая.

Обратите внимание, что для аргумента шаблона **T**, правила вывода типа аргумента шаблона придают **T&&** существенно иной смысл, чем **X&&** для типа **X** (§23.5.2.1). Для аргументов шаблонов rvalue-ссылки чаще всего используются для реализации технологии «правильного продвижения» (§23.5.2.1, §28.6.3).

Как же выбирать между разными способами передачи аргументов? Мои эмпирические правила таковы:

1. Передача по значению для малых объектов.
2. Передеча через **const**-ссылку больших объектов, которых не нужно модифицировать в функции.
3. Возвращать результат посредством оператора **return**, а не модифицированием объектов через аргументы.
4. Использовать rvalue-ссылки для реализации перемещения (§3.3.2, §17.5.2) и продвижения (§23.5.2.1).
5. Передавать указатель в случае, когда альтернатива «нет объекта» реальна (представима посредством **nullptr**)
6. Передавать по неконстантной ссылке лишь в случаях, когда это на самом деле требуется.

В последнем пункте мы подчеркиваем тот факт, что «честная» передача указателя с целью модификации объекта (§7.7.1, §7.7.4) вместо передачи по ссылке является более наглядной и проще для понимания сторонними программистами (например, при поддержке кода).

12.2.2. Аргументы-массивы

В случае, когда функции передается массив в качестве аргумента, на самом деле ей передается указатель на его первый элемент. Например:

```
int strlen(const char*);

void f()
{
    char v[] = "Annemarie";
    int i = strlen(v);
    int j = strlen("Nicholas");
}
```

Итак, аргумент типа **T[]** конвертируется в **T*** при передачи аргумента. Отсюда следует, что присваивание элементу массива в функции отразится на элементе исходного массива. То есть, в отличие от аргументов иных типов, аргументы-массивы не передаются по значению. Вместо этого передается указатель (сам указатель, естественно, передается по значению).

Параметр-массив имеет тип указателя. Например:

```
void odd(int* p);
void odd(int a[]);
void odd(int buf[1020]);
```

Все три объявления эквивалентны и объявляют одну и ту же функцию. Как всегда, имена аргументов не влияют на тип функции (§12.1.3). Правила и приемы передачи многомерных массивов рассматриваются в §7.4.3.

Размер массива неизвестен принимающей функции. В этом заключается источник многочисленных ошибок; правда, существует множество способов решить эту проблему. Например, строки в С-стиле (массивы символов) являются нуль-терминированными, так что их длину можно вычислить (например, за счет потенциально дорогостоящего вызова **strlen()**; §43.4). В остальных случаях можно передавать размер массивы дополнительным аргументом. Пример:

```
void compute1(int* vec_ptr, int vec_size);     // один способ
```

Это, конечно, работает. Но лучше всего передавать ссылку на некоторый контейнер, такой как **vector** (§4.4.1, §31.4), **array** (§34.2.1) или **map** (§4.4.3, §31.4.3).

Если же вам непременно требуется передавать массив, а не контейнер или указатель на первый элемент, вы можете объявить параметр с типом ссылки на массив. Например:

```
void f(int(&r)[4]);

void g()
{
    int a1[] = {1,2,3,4};
    int a2[] = {1,2};

    f(a1);      // OK
    f(a2);      // ошибка: неправильное число элементов
}
```

Заметьте, что число элементов входит в тип «ссылка-на-массив». Это делает такой тип намного менее гибким, чем указатели и контейнеры (такие, как **vector**). Основной областью применения ссылок на массивы являются шаблоны, в которых число элементов затем выводится. Например:

```
template<class T, int N> void f(T(&r)[N])
{
    // ...
}

int a1[10];
double a2[100];

void g()
{
    f(a1);      // T есть int; N есть 10
    f(a2);      // T есть double; N есть 100
}
```

Здесь порождается столь много определений функций, сколько имеется вызовов **f()** с различными типами массивов.

Многомерные массивы довольно запутаны (см. §7.3), но часто вместо них можно использовать массивы указателей. Пример:

```
const char* day[] = {
    "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

Как всегда, **vector** и подобные типы являются отличной альтернативой встроенным низкоуровневым массивам и указателям.

12.2.3. Аргументы-списки

Можно в качестве аргументов применять {}-списки для параметров типа:

1. **std::initializer_list<T>**, когда элементы списка могут неявно конвертироваться в тип **T**
2. Типа, который может инициализироваться величинами, представленными в списке
3. Типа ссылки на массив элементов типа **T**, когда элементы списка могут неявно конвертироваться в тип **T**

С технической точки зрения случай [2] покрывает все варианты, но я считаю легче представлять ситуацию по-отдельности в виде трех случаев. Рассмотрим следующее:

```
template<class T>
void f1(initializer_list<T>);

struct S {
    int a;
    string s;
};
void f2(S);

template<class T, int N>
void f3(T (&r)[N]);

void f4(int);

void g()
{
    f1({1,2,3,4});          // T есть int, a size() от initializer_list дает 4
    f2({1,"MKS"});          // f2(S{1,"MKS"})
    f3({1,2,3,4});          // T есть int и N есть 4
    f4({1});                // f4(int{1});
}
```

В случае неоднозначности параметр **initializer_list** имеет приоритет. Например:

```
template<class T>
void f(initializer_list<T>);

struct S {
    int a;
    string s;
};
void f(S);

template<class T, int N>
void f(T (&r)[N]);

void f(int);

void g()
{
    f({1,2,3,4});           // T есть int и size() от initializer_list дает 4
    f({1,"MKS"});           // вызов f(S)
    f({1});                 // T есть int и size() от initializer_list дает 1
}
```

Причина, по которой функция с аргументом **initializer_list** имеет приоритет, заключается в том, что было бы крайне удивительно, если бы разные функции выбирались на

основе разного числа элементов в списке. Невозможно устранить все случаи неоднозначностей при разрешении перегрузки (например, см. §4.4, §17.3.4.1), но придание приоритета **initializer_list** для аргументов типа {}-списков представляется разумным решением, минимизирующим проблемы.

Если в области видимости присутствует функция с **initializer_list**-аргументом, но аргумент-список не подходит для разрешения, то выбирается иная функция. Примером служит вызов **f({1,"MKS"})**.

Заметим, что эти правила применимы только к аргументам **std::initializer_list**. Нет никаких специальных правил для аргументов **std::initializer_list<T>&** или для других типов, случайно именуемых как **initializer_list** в иных областях видимости.

12.2.4. Неопределенное количество аргументов

Для некоторых функций невозможно специфицировать число и тип всех аргументов, которые могут иметь место при ее вызове. Для реализации такого сценария у нас есть три варианта:

1. Использовать шаблон с переменным числом параметров (§28.6): это позволяет работать с произвольным числом произвольных типов в надежной манере посредством написания небольшой метапрограммы, которая интерпретирует список аргументов, чтобы распознать их смысл и предпринять надлежащие действия.
2. Использовать **initializer_list** для типа аргумента (§12.2.3). Это позволяет работать с произвольным числом аргументов единственного типа в надежной манере. Во многих контекстах такие однородные списки являются типовым и важным практическим случаем.
3. Терминировать список аргументов троеточием (...), что означает «и еще некоторое число аргументов». Это позволяет работать с произвольным числом (почти) произвольных типов посредством использования некоторого макроса из заголовочного файла **<cstdarg>**. Такое решение не является надежным (безопасным) по типам и может породить проблемы в случае изощренных пользовательских типов. Однако ж, этот механизм широко используется с самых ранних дней языка C.

Первые два механизма описываются в разных местах, а здесь я сосредоточюсь на третьем механизме. Пример:

```
int printf(const char* ...);
```

Здесь объявляется, что вызов стандартной функции **printf()** (§43.3) должен содержать по крайней мере один аргумент, C-строку, и может содержать также дополнительные аргументы. Например:

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n",2,3,5);
```

Такие функции должны полагаться на информацию, недоступную компилятору при интерпретировании списка аргументов. Для функции **printf()** первый аргумент представляет собой форматирующую строку, содержащую последовательность специальных символов, позволяющую этой функции обработать остальные аргументы корректным образом; **%s** означает «ожидается аргумент типа **char***», а **%d** означает «ожидается аргумент типа **int**». Компилятор не может дать никаких гарантий того, что в реальном вызове аргументы будут ожидаемых типов (и будут ли вообще). Пример:

```
#include <cstdio>

int main()
{
    std::printf("My name is %s %s\n",2);
}
```

Этот код не является корректным, но большинство компиляторов не поймают тут ошибок. В лучшем случае вы получите странный вывод от этого кода (можете попробовать).

Ясно, что если аргумент не объявляется, то надлежащей проверки типов и стандартного преобразования выполнить невозможно. В этом случае **char** или **short** передаются как **int**, а **float** передается как **double**. Программист может и не ожидать такого.

В хорошо спроектированных программах подобного рода функций должно быть как можно меньше. Лучше обратить взор на перегруженные функции, использование аргументов по умолчанию, на **initilizer_list**, на шаблоны с переменным числом параметров — ради надлежащей проверки аргументов в большинстве случаев.

Чаще всего использование троеточия встречается тогда, когда возникает необходимость работать с функциями С-библиотеки, которые разрабатывались задолго до возникновения языка С++:

```
int fprintf(FILE*, const char* ...);   // из <cstdio>
int execl(const char* ...);            // из UNIX хедера
```

Стандартный набор макросов для работы с функциями с неопределенным числом аргументов представлен в заголовочном файле **<cstdarg>**. Попробуйте написать функцию для работы с ошибками, которая принимает один целочисленный аргумент, указывающий на степень серьезности ошибки, и произвольное число строк. Список строковых аргументов должен заканчиваться нулевым указателем:

```
extern void error(int ...);
extern char* itoa(int, char[]);        // int в alpha

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:
        error(0,argv[0],nullptr);
        break;
    case 2:
        error(0,argv[0],argv[1],nullptr);
        break;
    default:
        char buffer[8];
        error(1,argv[0],"with",itoa(argc-1,buffer),"arguments",nullptr);
    }
    // ...
}
```

Функция **itoa()** возвращает строку в С-стиле, соответствующую ее аргументу типа **int**. Эта функция часто встречается в С-коде, но она не входит в стандарт языка С.

Я всегда передаю **argv[0]**, поскольку это имя программы.

Также следует отметить, что передача **0** вместо нулевого указателя не является переносимым решением, так как на некоторых платформах они имеют разное представление (§6.2.8). Это еще одна тонкость и проблема для программиста, если он останавливается на варианте кода с использованием троеточия.

Функцию **error()** можно определить вот так:

```
#include <cstdarg>

void error(int severity ...)      // за "severity" следует нуль-терминированный список char*
{
    va_list ap;
    va_start(ap,severity);        // инициация аргументов

    for (;;) {
        char* p = va_arg(ap,char*);
        if (p == nullptr) break;
        cerr << p << ' ';
    }

    va_end(ap);                   // очистка аргументов

    cerr << '\n';
    if (severity) exit(severity);
}
```

Сначала мы определяем переменную типа **va_list**, которую инициализируем вызовом **va_start()**. В качестве аргументов макрос **va_start** берет имя переменной и имя последнего формального аргумента. Макрос **va_arg()** используется, чтобы выбирать неименованные аргументы по-порядку. В каждом вызове программист должен сообщать тип; **va_arg()** полагает, что передается фактический аргумент этого типа. Перед возвратом из функции, которая воспользовалась вызовом **va_start()**, нужно вызвать **va_end()**, так как возможна модификация стека вызовов и без **va_end()** возврат из функции может пройти нештатно.

Можно определить функцию **error()** и по-другому, применив **initializer_list** из стандартной библиотеки:

```
void error(int severity, initializer_list<string> err)
{
    for (auto& s : err)
        cerr << s << ' ';
    cerr << '\n';
    if (severity) exit(severity);
}
```

Тогда ее нужно вызывать с нотацией списков, например вот так:

```
switch (argc) {
case 1:
    error(0,{argv[0]});
    break;
case 2:
    error(0,{argv[0],argv[1]});
    break;
default:
    error(1,{argv[0],"with",to_string(argc-1),"arguments"});
}
```

Функция **to_string()**, преобразующая **int** в **string**, представлена в стандартной библиотеке (§36.3.5).

Если бы я не пытался мимикрировать под С-стиль, то я бы вообще передавал контейнер в качестве единственного аргумента:

```
void error(int severity, const vector<string>& err)      // почти как ранее
{
    for (auto& s : err)
        cerr << s << ' ';
    cerr << '\n';
    if (severity) exit(severity);
}

vector<string> arguments(int argc, char* argv[])      // упаковка аргументов
{
    vector<string> res;
    for (int i = 0; i!=argc; ++i)
        res.push_back(argv[i]);
    return res
}

int main(int argc, char* argv[])
{
    auto args = arguments(argc,argv);
    error((args.size()<2)?0:1,args);
    // ...
}
```

Вспомогательная функция, **arguments()**, тривиальна, а функции **main()** и **error()** довольно просты. Интерфейс между функциями **main()** и **error()** теперь весьма общий, что поможет в дальнейшем при модификациях. Применение **vector<string>** позволяет резко сократить число потенциально возможных ошибок.

12.2.5. Аргументы по умолчанию

Функции общего применения часто требуют большего числа аргументов, чем требуется в простых частных случаях. В частности, функции, конструирующие объекты (§16.2.5), часто предоставляют разные опциональные возможности ради гибкости. Рассмотрим класс **complex** из §3.2.1.1:

```
class complex {
    double re, im;
public:
    complex(double r, double i) :re{r}, im{i} {}    // конструируем complex из двух скаляров
    complex(double r) :re{r}, im{0} {}              // конст. complex из одного скаляра
    complex() :re{0}, im{0} {}
// complex по умолчанию: {0,0}
    // ...
};
```

Действия, выполняемые конструкторами, тривиальны, но есть что-то странное в наличии трех из них для выполнения по сути одинаковой работы. Также для многих классов типична ситуация, когда конструкторы выполняют довольно много работы, и повтор кода в них неизбежен. Можно справиться с повторяемостью кода посредством выделения одного конструктора в качестве основного и перенаправлением на него из других конструкторов (§17.4.3):

```
complex(double r, double i) :re{r}, im{i} {}    // конструируем complex из двух скаляров
complex(double r) :complex{2,0} {}               // констр. complex из одного скаляра
complex() :complex{0,0} {}                       // complex по умолчанию: {0,0}
```

Если нам потребуется добавить что-нибудь для отладки, для прослеживания траектории выполнения кода или для сбора статистики, то все это можно добавить в единственное место. Но можно выполнить еще большее сокращение кода:

```
complex(double r ={}, double i ={}) :re{r}, im{i} {}    // конструир. complex из двух скаляров
```

Наглядно видно, что если пользователь предоставит менее двух аргументов, то будут использованы значения по умолчанию.

Тип умолчательного аргумента фиксируется во время обработки объявления функции, а его значение — в момент вызова. Например:

```
class X {
public:
    static int def_arg;
    void f(int =def_arg);
    // ...
};

int X::def_arg = 7;

void g(X& a)
{
    a.f();          // может быть f(7)
    a.def_arg = 9;
    a.f();          // f(9)
}
```

Умолчательные аргументы, которые могут менять значение, порождают тонкие контекстные зависимости, и их лучше избегать.

Умолчательные значения аргументов можно назначать лишь аргументам, расположенным в конце списка аргументов. Пример:

```
int f(int, int =0, char* =nullptr);    // OK
int g(int =0, int =0, char*);          // ошибка
int h(int =0, int, char* =nullptr);    // ошибка
```

Заметьте, что пустое место между * и = имеет смысл, так как *= является операцией присваивания (§10.3):

```
int nasty(char*=nullptr);          // синтаксическая ошибка
```

Умолчательный аргумент нельзя повторно специфицировать или изменять в последующих объявлениях в одной и той же области видимости. Например:

```
void f(int x = 7);
void f(int = 7);                   // ошибка: нельзя повторять умолчат. аргумент
void f(int = 8);                   // ошибка: разные умолчат. аргументы

void g()
{
    void f(int x = 9);             // OK: это объявление скрывает внешнее
    // ...
}
```

Объявление имени во вложенной области видимости скрывает таковое в объемлющей области видимости, что чревато ошибками.

12.3. Перегрузка функций

В большинстве случаев идея присваивать разным функциям разные имена абсолютно здравая. В то же время, если функция выполняет одинаковые по смыслу действия над

разными типами, то имеет смысл назначать им одинаковое имя. Использование одного и того же имени для операций над разными типами называют *перегрузкой* (*overloading*). Это изначально используется в C++, так как, например, сложение для разных типов обозначается одинаково знаком + (и для целых типов, и для типов с плавающей запятой, и для комбинаций этих типов). Эта идея распространяется и на функции, которые определяет пользователь. Например:

```
void print(int);              // печатаем int
void print(const char*);      // печатаем C-строку
```

С точки зрения компилятора единственно общее, что есть у функций с одним и тем же именем, это собственно имя. Так что совпадение имен аналогичных функций есть просто удобство нотации. Особо важно это в случае стандартизированных имен, таких как **sqrt**, **print** и **open**. Это важно и для семантически эквивалентных операций, таких как **+**, ***** и **<<**, и для конструкторов (§16.2.5, §17.1), а также в обобщенном программировании (§4.5, Глава 32).

Шаблоны предоставляют систематический способ определения наборов перегруженных функций (§23.5).

12.3.1. Автоматическое разрешение перегрузки

Когда происходит вызов функции **fct**, компилятор должен решить, какую из функций с именем **fct** нужно вызвать. Это делается сравнением типов фактических аргументов с типами формальных параметров всех функций, имеющих имя **fct** в данной области видимости. Выбор должен остановиться на функции, имеющей наилучшее соответствие аргументов, а в случае неудачи выбора — выдать сообщение об ошибке. Пример:

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print(long)
    print(1.0);     // print(double)
    print(1);       // ошибка, двусмысленность: print(long(1)) или print(double(1))?
}
```

В качестве приближения к нашим представлениям о том, что есть разумное совпадение, последовательность критериев применяется по-очереди:

1. Точное совпадение; то есть, совпадение без или же только с тривиальными преобразованиями (например, имя массива в указатель, имя функции в указатель на функцию, или **T** в **const T**)
2. Совпадение на продвижениях; то есть, с интегральными продвижениями (**bool** в **int**, **char** в **int**, **short** в **int** и в их **unsigned**-варианты; §10.5.1) и **float** в **double**
3. Совпадение на стандартных преобразованиях (например, **int** в **double**, **double** в **int**, **double** в **long double**, **Derived*** в **Base*** (§20.2), **T*** в **void*** (§7.2.1), **int** в **unsigned int** (§10.5))
4. Совпадение на пользовательских преобразованиях (например, **double** в **complex<double>**; §18.4)
5. Совпадение с использованием троеточия ... в объявлении функции (§12.2.4)

Если обнаруживаются два совпадения на одном и том же уровне иерархии совпадений, то такой вызов классифицируется как неоднозначный и отвергается. Правила разрешения перегрузки столь изощренные потому, что они таковы в C и C++ для встроенных числовых типов (§10.5). Например:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
    print(c);       // точное соответствие: вызывается print(char)
    print(i);       // точное соответствие: вызывается print(int)
    print(s);       // интегральное продвижение: вызывается print(int)

    print(f);       // продвижение от float к double: print(double)
    print('a');     // точное соответствие: вызывается print(char)
    print(49);      // точное соответствие: вызывается print(int)
    print(0);       // точное соответствие: вызывается print(int)
    print("a");     // точное соответствие: вызывается print(const char*)
    print(nullptr); // продвижение от nullptr_t к const char*: вызов print(cost char*)
}
```

Вызов **print(0)** приводит к выбору **print(int)**, поскольку **0** относится к типу **int**. Вызов **print('a')** сводится к **print(char)**, поскольку **'a'** это **char** (§6.2.3.2). Причина, по которой делается разница между продвижениями и преобразованиями, заключается в том, что мы отдаем приоритет безопасным продвижениям, таким как **char** в **int**, а не безопасным преобразованиям вроде **int** в **char**. См. также §12.3.5.

Разрешение перегрузки не зависит от порядка, в котором в коде программы появляются объявления функций.

Шаблоны функций обрабатываются правилами разрешения неоднозначностей применением к специализации, основанной на наборе аргументов (§23.5.3). Существуют разные правила разрешения перегрузки для случая использования {}-списков (списки инициализации имеют приоритет; §12.2.3, §17.3.4.1) и для шаблонных rvalue-ссылочных аргументов (§23.5.2.1).

Правила перегрузки довольно сложные и время от времени программист может оказаться озадаченным реальным выбором. Так ради чего все эти проблемы? Но все-таки иногда нам действительно нужно выполнять совпадающие по смыслу операции над разными типами. Без перегрузки мы были бы вынуждены использовать разные имена:

```
void print_int(int);
void print_char(char);
void print_string(const char*);      // C-строка

void g(int i, char c, const char* p, double d)
{
    print_int(i);           // OK
    print_char(c);          // OK
    print_string(p);        // OK

    print_int(c);           // OK? вызов print_int(int(c)), печатается число
    print_char(i);          // OK? вызов print_char(char(i)), сужение
    print_string(i);        // ошибка
    print_int(d);           // OK? вызов print_int(int(d)), сужение
}
```

По сравнению с перегруженным вариантом **print()** здесь нам нужно помнить все имена и использовать их корректно. Это может быть утомительным, мешать реализовываать обобщенное программирование (§4.5) и заставляет программиста обращать внимание на низ-

коуровневые проблемы с типами. В отсутствие перегрузки все равно нужно выполнять все стандартные преобразования над аргументами, что тоже может породить ошибки. Например, в примере два вызова зависят от «сужения» типов (§2.2.2, §10.5). А применение перегрузки повышает шанс на то, что неподходящие аргументы будут отвергнуты компилятором.

12.3.2. Перегрузка и тип возврата

Тип возврата не рассматривается в процессе разрешения перегрузки. Причина такого решения состоит в том, чтобы правила можно было применять к индивидуальным операциям (§18.2.1, §18.2.5) и чтобы вызов функции не зависил от контекста. Рассмотрим следующее:

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
    float fl = sqrt(da);      // вызов sqrt(double)
    double d = sqrt(da);      // вызов sqrt(double)
    fl = sqrt(fla);           // вызов sqrt(float)
    d = sqrt(fla);            // вызов sqrt(float)
}
```

Если бы тип возврата принимался бы во внимание, тогда не было бы возможности изолированно изучать вызовы **sqrt()**.

12.3.3. Перегрузка и область видимости

Перегрузка имеет место по отношению к некоторому набору объектов. По умолчанию это относится к функциям единственной области видимости; функции, объявленные иначе (не в одной области видимости) не составляют перегруженного набора. Например:

```
void f(int);

void g()
{
    void f(double);
    f(1);           // вызов f(double)
}
```

Ясно, что **f(int)** был бы наилучшим выбором для **f(1)**, но в текущей области видимости имеем лишь **f(double)**. В таких случаях можно добавить локальное объявление для реализации приемлемого поведения. Как всегда, намеренное сокрытие может быть полезным приемом, а ненамеренное сокрытие — источником «сюрпризов».

Базовый и производный классы формируют разные области видимости, так что перегрузки между членами таких классов по умолчанию не возникает. Например:

```
struct Base {
    void f(int);
};

struct Derived : Base {
    void f(double);
};

void g(Derived& d)
{
    d.f(1);     // вызов Derived::f(double);
}
```

Когда есть нужда в перегрузке через классовые области видимости (§20.3.5) или через границы именованных пространств имен (§14.4.5), можно использовать **using**-объявления или **using**-директивы (§14.2.2). Поиск зависимости от аргументов (§14.2.4) может также приводить к перегрузке через границы пространств имен.

12.3.4. Разрешение перегрузки при множественных аргументах

Можно использовать правила разрешения перегрузки для выбора наиболее подходящей функции в случае, когда эффективность и точность вычислений существенно варьируется для разных типов. Например:

```
int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, complex);

void k(complex z)
{
    int i = pow(2,2);              // вызов pow(int,int)
    double d = pow(2.0,2.0);       // вызов pow(double,double)
    complex z2 = pow(2,z);         // вызов pow(double,complex)
    complex z3 = pow(z,2);         // вызов pow(complex,int)
    complex z4 = pow(z,z);         // вызов pow(complex,complex)
}
```

В процессе выбора из набора перегруженных функций с двумя аргументами отыскиваются наилучшие варианты по каждому аргументу согласно правилам из §12.3. Окончательно выбирается функция, наиболее подходящая по одному аргументу и лучшая или не худшая чем другие по второму аргументу. Если такой функции не находится, то вызов функции отвергается. Пример:

```
void g()
{
    double d = pow(2.0,2);         // ошибка: pow(int(2.0),2) или pow(2.0,double(2))?
}
```

Здесь вызов двусмысленен, поскольку **2.0** является наилучшим выбором первого аргумента для **pow(double,double)**, а **2** — наилучший выбор второго аргумента для **pow(int,int)**.

12.3.5. Ручное разрешение перегрузки

Объявляя слишком мало или слишком много перегруженных функций, мы повышаем вероятность неопределенности. Например:

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
    f1(i);      // двусмыслен.: f1(char) или f1(long)?
    f2(0);      // двусмыслен.: f2(char*) или f2(int*)?
}
```

Всюду, где это возможно, рассматривайте набор перегруженных функций в целом и решите, имеет ли он смысл согласно семантике функции. Часто проблема решается

добавлением нового варианта перегрузки, что устраняет неоднозначность. Например, добавляя

```
inline void f1(int n) { f1(long(n)); }
```

мы устраняем неоднозначность, аналогичную **f1(i)**, в пользу более длинного типа **long int**.

Можно, конечно, ввести в код и явное приведение типа для какого-либо конкретного вызова функции. Пример:

```
f2(static_cast<int*>(0));
```

Тем не менее, такое решение - это чистый паллиатив. Вскоре может потребоваться так поступить еще с одним вызовом.

Многие новички в C++ раздражаются, получив сообщение о неоднозначности. Опытные же программисты ценят эту помощь со стороны компилятора, указывающую им на просчеты в дизайне программы.

12.4. Предусловия и постусловия

Любая функция имеет свои ожидания относительно фактических аргументов, с которыми ее будут вызывать. Часть этих ожиданий формулируется в виде типов формальных аргументов, а остальные сводятся к величине фактических аргументов и к соотношению между ними. Компилятор и компоновщик могут гарантировать аргументы правильных типов, но только программист знает, какие величины аргументов приемлемы. Мы назовем логические критерии, которые должны выполняться в момент вызова функции, *предусловиями*, а логические критерии, которые должны выполняться в момент возврата из функции, *постусловиями*. Например:

```
nt area(int len, int wid)
/*
     вычислен. площади прямоугольника

     Предусловие: len и wid положительны

     Постусловие: возврат положителен

     Постусловие: возврат это площадь прямоуг. Со сторонами len и wid
*/
{
     return len*wid;
}
```

Здесь формулировки предусловия и постусловия длиннее, чем код тела функции. Это может показаться избыточным, но эта информация важна как для того програмиста, который реализует функцию, так и для тех, кто будет впоследствии читать этот код и поддерживать его. К примеру, мы узнаем, что значения **0** и **-12** не считаются корректными аргументами. Кроме того, мы узнаем, что можно передать пару огромных значений, не нарушая предусловия, но если **len*wid** вызовет переполнение, то предусловие не выполнится.

Что нам следует делать по поводу вызова **area(numeric_limits<int>::max(),2)**?

1. Тот, кто вызывает, должен избежать такой ситуации? Да, но что если этого не произойдет?
2. Может быть реализатор функции должен избежать проблемы? Если так, то как он должен обработать ошибку?

Существует несколько ответов на эти вопросы. На стороне вызова легко совершить ошибку и нарушить предусловие. А реализатору функции проверять предусловия доволь-

но дорого. Надеясь на корректный вызов, все же стоит как-то подстраховываться. Некоторые из предусловий и постусловий проверить довольно просто (например, длина **len** положительна). Другие же являются семантическими по сути и их трудно проверить напрямую. Например, как проверить, что возвращаемое значение действительно есть площадь прямоугольника: вычисление могло сорваться из-за переполнения или иных проблем с точностью вычислений. Не производить же повторные вычисления с повышенной точностью. Все это может обойтись слишком дорого.

Но тщательное прописывание предусловий и постусловий реально помогает вскрывать тонкие проблемы. Это отличный инструмент проектирования, провоцирующий хорошую документацию кода. Механизмы для документирования и формулирования условий обсуждаются в §13.4.

Если функция зависит лишь от аргументов, то предусловия формулируется лишь на аргументах. Но нужно быть особо осторожным в отношении функций, зависящих от нелокальных величин (функции-члены зависят от состояния объекта). Нужно рассматривать любое нелокальное значение как скрытый аргумент функции. Аналогично, постусловие функции без побочных эффектов просто постулирует, что возвращаемое значение вычислено корректно, но если функция пишет в нелокальные объекты, то это обстоятельство должно тщательно рассматриваться и документироваться.

При написании функции есть несколько альтернативных стратегий, включая:

1. Убеждение, что все входные значения корректны (в итоге нет нужды в предусловии).
2. Предположение, что вызывающий код выполнит предусловие (нет ошибок в вызывающем коде).
3. Проверка выполнения предусловия и генерирование исключения в противном случае.
4. Проверка выполнения предусловия и завершение программы в противном случае.

Если не выполняется постусловие, то либо было не выполнено предусловие, или же имеется ошибка в коде. В §13.4 представлены альтернативные стратегии проверок.

12.5. Указатель на функцию

Как и любые данные, код, который генерируется для тела функции, должен где-то располагаться в памяти, то есть он имеет адрес. Таким образом, мы можем настроить указатель на функцию так же, как это мы делаем длоя объектов. Но по ряду причин, отчасти из-за особенностей машинных архитектур, отчасти по системным соображениям, с помощью указателя на функцию нельзя модифицировать код. Все что можно сделать с функцией — это вызвать ее или взять ее адрес. Указатель, который получает свое значение путем взятия адреса функции, может затем использоваться для ее вызова. Например:

```
void error(string s) { /* ... */ }

void (*efct)(string); // указ-ль на ф-ю со строковым аргументом и без возврата

void f()
{
    efct = &error; // efct указует на error
    efct("error");  // вызов error через efct
}
```

Компилятор легко распознает, что **efct** это указатель на функцию и вызывает функцию, адресуемую этим указателем. Таким образом, разыменование указателя на функцию

с помощью * является опциональным. Аналогично, использование знака & для взятия адреса функции, также опционально:

```
void (*f1)(string) = &error;    // OK: то же, что и = error
void (*f2)(string) = error;     // OK: то же, что и = &error

void g()
{
    f1("Vasa");                 // OK: то же, что и (*f1)("Vasa")
    (*f1)("Mary Rose");         // OK: как f1("Mary Rose")
}
```

Объявления указателей на функции содержат типы аргументов, как и объявления собственно функций. Для присваиваний указателей на функции полное совпадение типов аргументов обязательно. Например:

```
void (*pf)(string);         // указ-ль на void(string)
void f1(string);            // void(string)
int f2(string);             // int(string)
void f3(int*);              // void(int*)

void f()
{
    pf = &f1;               // OK
    pf = &f2;               // ошибка: неправиль. тип возврата
    pf = &f3;               // ошибка: неправиль. тип аргумента

    pf("Hera");             // OK
    pf(1);                  // ошибка: неправиль. тип аргумента

    int i = pf("Zeus");     // ошибка: void присваивается int
}
```

Правила передачи аргументов в случае вызова функции через указатель на функцию те же самые, что и для функций.

Можно конвертировать один тип указателя на функцию в другой тип указателя на функцию:

```
using P1 = int(*)(int*);
using P2 = void(*)(void);

void f(P1 pf)
{
    P2 pf2 = reinterpret_cast<P2>(pf)
    pf2();                              // вероятна серьезная проблема
    P1 pf1 = reinterpret_cast<P1>(pf2); // обратная конверсия pf2
    int x = 7;
    int y = pf1(&x);                    // OK
    // ...
}
```

Для выполнения преобразований между разными типами указателей на функции мы должны применять самую безобразную на вид операцию явного приведения типа — **reinterpret_cast**. Она самая заметная на вид, так как операция приведения между типами разных указателей на функции очень опасна и результат непредсказуем (и системно зависим). Например, в представленном примере вызываемая функция может к примеру писать что-либо в объект, указуемый ее аргументом, но вызов **pf2()** вообще не имеет аргументов!

Указатели на функции позволяют параметризовать алгоритмы. Так как в языке С нет функциональных объектов (§3.4.3) или лямбда-выражений (§11.4), указатели на функции широко используются в качестве аргументов при вызове функций в С-стиле. Например, мы можем таким образом предоставить операцию сравнения, в которой нуждается функция сортировки:

```
using CFT = int(const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
     Сортируем "n" эл-ов of вектора "base" в возрастающем
     порядке используя ф-ю сравнения, указуемую посредством "cmp".
     Элементы имеют размер "sz"..

     Shell sort (Knuth, Vol3, pg84)
*/
{
     for (int gap=n/2; 0<gap; gap/=2)
          for (int i=gap; i!=n; i++)
               for (int j=i-gap; 0<=j; j-=gap) {
                    char* b = static_cast<char*>(base);      // необходима конверсия
                    char* pj = b+j*sz;                       // &base[j]
                    char* pjg = b+(j+gap) *sz;               // &base[j+gap]
                    if (cmp(pjg,pj)<0) {                     // обмен base[j] и base[j+gap]:
                         for (int k=0; k!=sz; k++) {
                              char temp = pj[k];
                              pj[k] = pjg[k];
                              pjg[k] = temp;
                         }
                    }
               }
}
```

Процедура **ssort()** не знает типа объектов, которые она сортирует, только число элементов (длина массива), размер каждого элемента, а также функцию, которую нужно вызывать для выполнения сравнения. Тип для **ssort()** выбран таким же, как и у стандартной С-функции **qsort()**. В реальных программах используется **qsort()**, алгоритм **sort** из стандартной библиотеки языка C++ (§32.6) или какая-либо специализированная процедура. Показанный стиль кода типичен для языка С, но это не самый элегантный способ демонстрации алгоритма в C++ (см. §23.5, §25.3.4.1).

Такую сортирующую функцию можно применить для сортировки таблицы следующим образом:

```
struct User {
     const "char* name;
     const char* id;
     int dept;
};

vector<User> heads = {
     "Ritchie D.M.",      "dmr",    11271,
     "Sethi R.",          "ravi",   11272,
     "Szymanski T.G.",    "tgs",    11273,
     "Schryer N.L.",      "nls",    11274,
     "Schryer N.L.",      "nls",    11275,
     "Kernighan B.W.",    "bwk",    11276
};
```

```
void print_id(vector<User>& v)
{
    for (auto& x : v)
        cout << x.name << '\t' << x.id << '\t' << x.dept << '\n';
}
```

Чтобы сортировка стала возможной, мы должны предоставить подходящую функцию сравнения. Сравнивающая функция должна возвращать отрицательное значение, если первый аргумент меньше второго ее аргумента, нуль в случае равенства и положительное значение в противном случае:

```
int cmp1(const void* p, const void* q)      // сравнение строк name
{
    return strcmp(static_cast<const User*>(p)->name,static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q)      // сравнение чисел dept
{
    return static_cast<const User*>(p)->dept - static_cast<const User*>(q)->dept;
}
```

Не существует неявных приведений типов в случае присваиваний указателей на функции или их инициализации. Это означает, что вам не удастся избежать операций явного приведения типа:

```
int cmp3(const User* p, const User* q)      // сравниваем id
{
    return strcmp(p->id,q->id);
}
```

Причина состоит в том, что принимая **cmp3** в качестве аргумента для **ssort()**, мы нарушаем гарантию, что **cmp3** будет вызвана с аргументами, тип которых есть **const User*** (см. также §15.2.6).

Эта программа сортирует и выводит результат:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    ssort(heads,6,sizeof(User),cmp1);
    print_id(heads);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    ssort(heads,6,sizeof(User),cmp2);
    print_id(heads);
}
```

Для сравнения, эквивалентно можно было бы написать и так:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    sort(heads.begin(), head.end(),
        [](const User& x, const User& y) { return x.name<y.name; }
    );
    print_id(heads);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    sort(heads.begin(), head.end(),
        [](const User& x, const User& y) { return x.dept<y.dept; }
```

```
        );
        print_id(heads);
}
```

Для вспомогательных функций не требуется указывать никаких размеров. Если явное использование **begin()** и **end()** раздражает, можно устранить их применением версии **sort()**, которая принимает контейнер (§14.4.5):

```
sort(heads,[](const User& x, const User& y) { return x.name<y.name; });
```

Вы можете взять адрес перегруженной функции, инициализируя указатель на функцию или путем присваивания этому указателю. Вот пример:

```
void f(int);
int f(char);

void (*pf1)(int) = &f;      // void f(int)
int (*pf2)(char) = &f;      // int f(char)
void (*pf3)(char) = &f;     // ошибка: нет void f(char)
```

Возможно также взятие адреса функции-члена (§20.6), но указатель на функцию-член существенно отличается от указателей на обычные функции.

Указатель на **noexcept**-функцию может быть объявлен с ключевым словом **noexcept**. Пример:

```
void f(int) noexcept;
void g(int);

void (*p1)(int) = f;              // OK: но выбрасываем ценную информацию
void (*p2)(int) noexcept = f;     // OK: сохраняем noexcept информацию
void (*p3)(int) noexcept = g;     // ошибка: не знаем,что g не генерирует исключения
```

Указатель на функцию должен соответствовать типу компоновки функции (§15.2.6). Ни спецификация компоновки, ни ключевое слово **noexcept**, не допускаются в синонимах типов:

```
using Pc = extern "C" void(int);    // ошибка: специфик. компоновки в синониме
using Pn = void(int) noexcept;      // ошибка: noexcept в синониме
```

12.6. Макросы

Макросы очень важны в языке C, но в языке C++ их роль заметно ниже. Первое правило относительно макросов: не нужно их использовать вообще, пока вы не будете вынуждены сделать это (по разным причинам). Практически любой макрос демонстрирует изъян либо в языке программирования, либо в программе, либо в программисте. Поскольку макросы модифицируют текст программы до того, как с ним начнет знакомиться компилятор, то макросы сильно вредят большинству инструментов поддержки. Если вам приходится применить макрос, то предварительно прочитайте детальное описание работы препроцессора вашей реализации C++ с тем, чтобы ненароком не слишком «наумничать». Ну и по традиции используйте одни заглавные буквы при именовании макроса. Синтаксис макросов представлен в §iso.16.3.

Я рекомендую использовать макросы лишь для условной компиляции (§12.6.1) и для формирования «стражей включения заголовочных файлов» («include guards») (§15.3.3).

Простой макрос определяется следующим образом:

```
#define NAME rest of line
```

Всюду, где **NAME** обнаруживается как лексема, ее заменяют на **rest of line**. Например:

```
named = NAME
```

будет заменено на

```
named = rest of line
```

Макросы могут принимать аргументы. Например:

```
#define MAC(x,y) argument1: x argument2: y
```

Используя **MAC**, нужно предоставлять два аргумента. Они заменят **x** и **y**, когда **MAC** будет раскрыт. Например:

```
expanded = MAC(foo bar, yuk yuk)
```

будет раскрыт до следующего результата

```
expanded = argument1: foo bar argument2: yuk yuk
```

Имена макросов нельзя перегружать и препроцессор не в состоянии разобраться с рекурсивными обращениями:

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c)      /* беда?: переопределен., не перегрузка */

#define FAC(n) (n>1)?n*FAC(n-1):1             /* беда: рекурсивное macro */
```

Работа с макросами заключается в манипулировании текстовыми строками и в этом мало что есть от синтаксиса C++ и вообще ничего нет от системы типов C++ или правил области видимости. Компилятор имеет дело только с раскрытым вариантом макроса и только в этом варианте может обнаружить ошибки. Это является источником туманных сообщений об ошибках.

Вот некоторые макросы, которые кому-то могут показаться полезными:

```
#define CASE break;case
#define FOREVER for(;;)
```

А вот макросы, в которых мало нужды:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

А вот опасные макросы:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

Чтобы понять их опасность, попробуйте раскрыть следующее:

```
int xx = 0;     // глобальный счетчик

void f(int xx)
{
    int y = SQUARE(xx+2);  // y=xx+2*xx+2; that is, y=xx+(2*xx)+2
    INCR_xx;               // инкрементируем аргумент xx (не глобальн. xx)
}
```

Если вы должны использовать макросы, применяйте операцию разрешения области видимости ::, когда ссылаетесь на глобальные имена (§6.3.4), и окружайте аргументы в макросах круглыми скобками всюду, где только можно. Например:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

Это позволяет обойти простейшие проблемы, но не проблемы с побочными эффектами. Пример:

```
int x = 1;
int y = 10;
int z = MIN(x++,y++);     // x становится 3; y становится 11
```

Если вы пишете достаточно сложный макрос, который стоит прокомментировать, используйте комментарии в стиле /**/, поскольку старые препроцессоры языка C (могут присутствовать в некоторых реализациях C++) ничего не знают о комментариях в стиле //. Например:

```
#define M2(a) something(a) /* умный комментарий */
```

С помощью макросов вы можете сконструировать свой собственный язык программирования, но он вряд ли будет хорошо воспринят (и даже понят) другими программистами. Кроме того, препроцессор является очень простодушным механизмом, так что если вам захочется изобрести что-либо весьма заковыристое, вряд ли препроцессор справится с вашими задумками. Имейте в виду, что **auto**, **constexpr**, **const**, **decltype**, **enum**, **inline**, лямбда-выражения, пространства имен, шаблоны — все эти средства языка C++ стоит предпочесть многим традиционным способам использования конструкций препроцессора. Например:

```
const int answer = 42;

template<class T>
inline const T& min(const T& a, const T& b)
{
    return (a<b)?a:b;
}
```

Часто при написании макросов возникает необходимость сформировать новое имя из уже существующих; новая строка здесь может быть создана из двух строк с помощью операции ##. Пример:

```
#define NAME2(a,b) a##b

int NAME2(hack,cah)();
```

Из этого получается

```
int hackcah();
```

Одиночный знак # перед параметром в строке замещения означает строку, содержащую параметр макроса. Например:

```
#define printx(x) cout << #x " = " << x << '\n';
int a = 7;

string str = "asdf";

void f()
{
    printx(a);      // cout << "a" << " = " << a << '\n';
    printx(str);    // cout << "str" << " = " << str << '\n';
}
```

Если написать **#x " = "**, а не **#x << " = "**, то получим менее ясный код, но не ошибку. Смежные строковые литералы конкатенируются (§7.3.2).

Директива

```
#undef X
```

гарантирует, что никакого макроса с именем **X** больше нет, независимо от факта его присутствия до введения этой директивы. Это дает некоторую гарантию от нежелательных макросов.

Список аргументов макроса (список замещения) может быть пустым:

```
#define EMPTY() std::cout<<"empty\n"
EMPTY();                    // печатаем
"empty\n"
EMPTY;                      // ошибка: список замещения для макро отсутствует
```

Я долго бился над примером пустого списка аргументов макроса, чтобы не возникало ошибок или еще каких неприятностей.

В конце концов, макрос может иметь переменное число параметров. Пример:

```
#define err_print(...) fprintf(stderr,"error: %s %d\n", __VA_ARGS__)
err_print("The answer",54);
```

Многоточие означает, что **__VA_ARGS__** представляет собой аргументы, передаваемые в виде строки, так что вывод будет следующий:

```
error: The answer 54
```

12.6.1. Условная компиляция

Одного примера применения макросов трудно избежать. Директива

```
#ifdef IDENTIFIER
```

не делает ничего, если **IDENTIFIER** определен, а если не определен, то директива приводит к тому, что ввод игнорируется до момента в коде, где появится директива **#endif**. Пример:

```
int f(int a
#ifdef arg_two
,int b
#endif
);
```

Если макрос **arg_two** не был определен, вывод этого кода будет таким:

```
int f(int a
);
```

Этот пример сбивает с толку инструменты, которые ожидают от программиста «честной игры».

Большинство случаев применения **#ifdef** не такие причудливые и тогда от них беды нет. См. также §15.3.3.

Имена для макросов, контролирующих директиву **#ifdef**, следует выбирать внимательно, чтобы они не пересекались с обычными идентификаторами. Например:

```
struct Call_info {
    Node* arg_one;
    Node* arg_two;
    // ...
};
```

Этот невинно выглядящий код может оказаться под ударом, если кто-либо напишет

```
#define arg_two x
```

К сожалению, общеупотребительные заголовочные файлы содержат множество опасных и ненужных макросов.

12.6.2. Предопределенные макросы

Некоторое количество макросов предопределены компилятором (§iso.16.8, §iso.8.4.1):

- **__cplusplus**: определен для компиляции C++ (но не C). Его значение **201103L** в программе на C++; в предыдущих стандартах C++ это значение было меньше.
- **__DATE__**: дата в формате «yyyy:mm:dd».
- **__TIME__**: время в формате «hh:mm:ss».
- **__FILE__**: имя текущего исходного файла.
- **__LINE__**: номер строки кода в текущем исходном файле.
- **__FUNC__**: строка в C-стиле, зависящая от реализации, обозначающая текущую функцию.
- **__STDC_HOSTED__**: **1**, если реализация полная (§6.1.1); в противном случае **0**.

Кроме того, несколько макросов определяются реализациями условно:

- **__STDC__**: определяется для C-компиляции (но не C++)
- **__STDC_MB_MIGHT_NEQ_WC__**: **1**, если в кодировке для **wchar_t** члены базового набора символов (§6.1) могли бы иметь значения, отличающиеся от значений простого символьного литерала
- **__STDCPP_STRICT_POINTER_SAFETY__**: **1**, если реализация гарантирует строгую безопасность указателей (§34.5); иначе не определено.
- **__STDCPP_THREADS__**: **1**, если программа может иметь более одного потока исполнения; иначе не определено.

Например:

```
cout << __FUNC__ << "() in file " << __FILE__ << " on line " << __LINE__ << "\n";
```

Большинство реализаций C++ разрешают пользователю определять произвольные макросы в командной строке или в каких-либо иных формах окружения компиляции. Например, макрос **NDEBUG** определен, если только компиляция не выполнялась в режиме отладки, и он используется макросом **assert()** (§13.4).

12.6.3. Прагмы

Реализации часто предоставляют средства, которые выходят за рамки или не определены стандартом. Естественно, в этом случае стандарт ничего не говорит об их функционировании. Но все же есть стандарт на строку лексем, которой предшествует препроцессорная директива **#pragma**. Пример:

```
#pragma foo bar 666 foobar
```

Если можно, не применяйте это средство.

12.7. Советы

[1] «Упаковывайте» значимые операции в тщательно поименованные функции; §12.1.

[2] Следует поручать функции одну логически единую операцию; §12.1.

[3] Сохраняйте код функции коротким; §12.1.
[4] Не возвращайте из функции указатели или ссылки на локальные переменные; §12.1.4.
[5] Если функцию можно вычислить на этапе компиляции, объявляйте ее с ключевым словом **constexpr**; §12.1.6.
[6] Если функция не может иметь возврата, помечайте ее как **[[noreturn]]**; §12.1.7.
[7] Небольшие объекты передавайте по значению; §12.2.1.
[8] Большие объекты, не подлежащие модификации, передавайте константными ссылками; §12.2.1.
[9] Предпочитайте возвращать значение посредством оператора **return**, а не модифицированием объекта посредством аргумента; §12.2.1.
[10] Используйте rvalue-ссылки для реализации перемещения и продвижения; §12.2.1.
[11] Передавайте указатель в случае, когда допускается ситуация «нет объекта» (передается как **nullptr**); §12.2.1.
[12] Передавайте аргументы по неконстантной ссылке только, если это действительно требуется; §12.2.1.
[13] Применяйте **const** последовательно и систематически; §12.2.1.
[14] Можно предполагать, что **char*** или **const char*** аргументы указывают на строку в С-стиле; §12.2.2.
[15] Избегайте передавать массивы как указатели; §12.2.2.
[16] Передавайте однородный список переменной длины как **initializer_list<T>** (или как иной контейнер); §12.2.3.
[17] Избегайте неопределенного числа аргументов (...); §12.2.4.
[18] Используйте перегрузку, когда функции выполняют концептуально одинаковую работу над разными типами; §12.3.
[19] При перегрузке с целыми типами предоставляйте дополнительные функции, чтобы избежать неоднозначностей; §12.3.5.
[20] Формулируйте предусловия и постусловия для ваших функций; §12.4.
[21] Отдавайте предпочтение функциональным объектам (включая лямбды) и виртуальным функциям перед указателями на функции; §11.5.
[22] Старайтесь избегать макросов; §12.6.
[23] Если приходится использовать макросы, используйте нарочито некрасивые имена с обилием заглавных букв; §12.6.

13

Обработка исключений

Не прерывайте меня,
когда я сам прерываю.
— Уинстон Черчиль

- Обработка ошибок
  Исключения; Традиционная обработка ошибок; Альтернативный взгляд на исключения; Когда нельзя использовать исключения; Иерархическая обработка ошибок; Исключения и эффективность
- Гарантии исключений
- Управление ресурсами
  Завершающие конструкции
- Инварианты
- Генерация и перехват исключений
  Генерация исключений; Перехват исключений; Исключения и потоки
- Реализация класса **vector**
  Простой **vector**; Явное представление памяти; Присваивание; Изменение размера
- Советы

13.1. Обработка ошибок

В настоящем разделе мы рассмотрим обработку ошибок посредством исключений. Для эффективной обработки ошибок соответствующие механизмы языка следует использовать систематически на базе четкой стратегии. Как следствие, эта глава концентрируется на идее гарантий безопасности на основе исключений, которая является центральной и для способов восстановления работоспособности программ после возникновения ошибок времени выполнения и для технологии работы с ресурсами на основе конструкторов и деструкторов. Обе эти темы включают в себя идею инвариантов, так что рассматриваются и механизмы внедрения так называемых *утверждений* (*assertions*).

Средства языка и методы их использования рассматриваются в данной главе исключительно по отношению к обработке ошибок в коде программы, проявляющих себя на этапе выполнения; обработку асинхронных событий считаем отдельной темой.

При рассмотрении ошибок мы фокусируемся на ошибках, которые нельзя обработать локально (внутри небольшой функции), так что требуется четко разделить все действия по разным частям программы. Эти части, к тому же, нередко и разрабатываются порознь. Поэтому часто я ссылаюсь на части программы, которые должны вызываться в ответ на ошибку, как на «библиотеки». Любая библиотека представляет собой обычный код, но

разработчик библиотеки может не иметь представления о том, в составе каких иных частей программы его код будет работать:

- Автор библиотеки может обнаружить ошибку во время выполнения кода, но может не иметь даже общей идеи о том, что с этим нужно делать.
- Пользователь библиотеки может знать, как нужно поступать в случае той или иной ошибки, но не может заранее локализовать их (если бы мог, то не оставлял бы это дело библиотеке).

Обсуждение исключений фокусируется на проблемах, с которыми приходится иметь дело в программах, длительно исполняющихся на компьютере, на системах с особыми требованиями к надежности исполнения и на библиотеках. Разные виды программ предъявляют разные требования, и объем усилий и заботы с нашей стороны должен отражать эти различия. Например, я бы не стал применять каждый из рассмотренных в данной главе приемов к какой-нибудь небольшой программе, написанной для самостоятельного применения. Но некоторые из рассмотренных технологий упрощают результирующий код в любом случае, так что их неплохо использовать в стандартной практике.

13.1.1. Исключения

Понятие *исключения* призвано помочь доставить информацию об ошибке из точки, где эта ошибка произошла, в точку программы, где она может быть обработана. Часть программы, которая не в состоянии справиться с проблемой, *генерирует исключение*, надеясь на то, что какая-то другая часть кода, прямо или косвенно обратившаяся к нашей части, сможет с проблемой справиться. Функция, которая намеревается так или иначе откликаться на исключения, формулирует свое намерение при помощи перехвата (catching) соответствующих исключений (§2.4.3.1):

- Вызывающая часть программы указывает вид ошибочных ситуаций, который она хочет обрабатывать, специфицируя соответствующие исключения в **catch**-ветви своего **try**-блока.
- Вызванная часть программы, которая не может закончить предназначенную для нее работу, сообщает об этом, генерируя исключение с помощью **throw**-выражения.

Рассмотрим упрощенный пример:

```
void taskmaster()
{
    try {
        auto result = do_task();
        // используем результат
    }
    catch (Some_error) {
        // не можем выполнить задачу
    }
}

int do_task()
{
    // ...
    if (/* можем выполнить задачу */)
        return result;
    else
        throw Some_error{};
}
```

Функция **taskmaster()** просит **do_task()** выполнить работу. Если **do_task()** может справиться с задачей и вернуть корректный результат, то все чудесно. В противном случае,

do_task() обязана сообщить о проблемах, генерируя исключение. Функция **taskmaster()** готова обработать ошибку типа **Some_error**, но генерироваться могут исключения и других типов. Например, **do_task()** может вызывать иные функции для выполнения каких-либо вспомогательных задач, и какие-либо из этих функций также могут генерировать исключения в случае невозможности выполнить возложенную на них задачу. Исключение типа, отличного от **Some_error**, означает невозможность функцией **taskmaster()** выполнить свою задачу, и это исключение должно обрабатываться где-то в коде, который вызывает саму функцию **taskmaster()**.

Функция, в которой возникла ошибочная ситуация, не может просто вернуть некоторое значение, означающее, что возникла ошибка. Если программу нужно по-прежнему продолжать выполнять (а не закончить, предварительно осуществив вывод сообщения об ошибке), то возвращающая управление функция обязана оставить программу в корректном состоянии и не допустить утечки ресурсов. Механизм обработки исключений интегрирован с механизмом конструкторов/деструкторов и механизмом параллелизма так, чтобы помогать в достижении указанного требования (§5.2). Механизм обработки исключений:

- Является альтернативой традиционным методам там, где последние неэффективны, неэлегантны или сами подвержены ошибкам.
- Является полным и законченным; он может обрабатывать все ошибки, возникающие в обычном коде.
- Позволяет программисту явным образом отделить код обработки ошибок от обычного кода, что делает программы более читаемыми.
- Поддерживает регулярный стиль обработки ошибок, упрощая интеграцию и взаимодействие между изолированно разработанными фрагментами программы.

Исключение — это сгенерированный объект, представляющий собой один экземпляр ошибки. Оно может быть любого типа, который может копироваться, но настоятельно рекомендуется использовать пользовательские типы, специально разработанные с этой целью. Мы таким образом минимизируем столкновение двух (и более) библиотек, для которых число, скажем **17**, может представлять собой разные ошибки, приводя нашу систему работы с ошибками к полному хаосу.

Исключение перехватывается кодом, который явным образом выразил свой интерес в обработке конкретного типа исключений. Простейшим способом определения исключения является определение класса, специально предназначенного для конкретного типа ошибок. Генерация исключения производится конструкцией с ключевым словом **throw**. Например:

```
struct Range_error {};

void f(int n)
{
    if (n<0 || max<n) throw Range_error {};
    // ...
}
```

Если это кажется утомительным, то на помощь может прийти стандартная библиотека, содержащая небольшую иерархию классов исключений (§13.5.2).

Исключение несет информацию об ошибке, которую оно представляет. Тип исключения соответствует типу ошибки, а информация, сопровождающая исключение, соответствует конкретному экземпляру ошибки. К примеру, исключения из стандартной библиотеки содержат строковое значение, которое можно использовать для передачи такой информации, как место генерации ошибки (§13.5.2).

13.1.2. Традиционная обработка ошибок

Рассмотрим альтернативы исключениям для случая, когда в функции обнаруживается ошибка, обработать которую локально нет возможности (например, ошибка выхода за границы корректного диапазона индексов) и поэтому о ней нужно сообщить коду, который вызвал функцию. Все традиционные способы работы имеют собственные проблемы и никакие из них не являются общеупотребительными:

- *Закончить выполнение программы*. Это довольно радикальный подход. Пример:

```
if (something_wrong) exit(1);
```

Для большинства ошибок мы можем и должны сделать большее. Например, в типичном случае мы должны хотя бы вывести приличное сообщение об ошибке перед тем, как завершить программу. В частности, библиотека, которая ничего не знает о стратегии работы программы, не должна просто вызывать стандартные завершающие функции **exit()** или **abort()**. Такие библиотеки не могут использоваться в составе программ, не допускающих аварийного завершения.

- *Возврат «ошибочного значения»*. Это в общем случае недостижимо потому, что может не существовать разумного «ошибочного значения». Например:

```
int get_int(); // взять следующее целое
```

Для этой функции ввода все целые значения являются корректным возвратом, так что нельзя найти целое, индицирующее ошибку. Минимально, нам нужно переделать **get_int()** под возврат пары значений. Даже в случаях, когда это допустимо, решение не может считаться оптимальным из-за своей неэффективности — ведь приходится каждый раз проверять возврат (когда и ошибок-то нет). Можно таким образом легко удвоить объем программы (§13.1.7). А вызывающий код может забыть проверить возврат на наличие ошибки. Как следствие, этот подход редко применяется для проверки наличия ошибок разных типов. Например, **printf()** (§43.3) возвращает отрицательное значение в случае, когда нет возможности осуществить вывод или же возникает проблема с кодировками, но программисты редко когда проверяют этот возврат. Наконец, некоторые операции вообще не имеют возврата; конструкторы служат очевидным примером.

- *Вернуть штатное значение и оставить программу в некорректном состоянии*. Тут неприятность в том, что вызвавший код может и не заметить, что программа вошла в некорректное состояние. Например, многие стандартные функции языка C устанавливают нелокальную переменную **errno**, которая индицирует ошибку (§43.4, §40.3):

```
double d = sqrt(-1.0);
```

Здесь величина **d** не имеет смысла, а **errno** устанавливается в **-1**, чтобы указать на то, что **-1.0** не является приемлемым аргументом для функции вычисления корня квадратного из значения с плавающей запятой. На практике программы не всегда устанавливают и проверяют **errno** или другие нелокальные величины и бывает трудно отличить их значения, устанавливаемыми разными вызванными функциями. Кроме того, установка нелокальных переменных сталкивается с дополнительными сложностями в случае параллелизма исполнения.

- *Вызов функции обработки ошибок*. Например:

```
if (something_wrong) something_handler(); // здесь возможно продолжение
```

Здесь проблема просто камуфлируется, ибо сразу же возникает вопрос о том, как работает функция обработки ошибок. Если только функция обработки ошибок не решает полностью проблему, то она в свою очередь либо терминирует программу, либо возвращает какое-либо значение для индицирования ошибки, либо устанавливает нелокальную переменную, либо генерирует исключение. И, наконец, если функция обработки ошибки может так уладить ситуацию, что об этом нечего знать вызывающей функции самого верхнего уровня, то тогда нам вообще нет никакой причины задумываться о проблеме.

На практике часто встречаются программы, в которых применяется некоторая бессистемная комбинация перечисленных подходов.

13.1.3. Попытка кое-как дотянуть до конца

В конце концов, предельная реакция на необработанную ошибку (неперехваченное исключение) — терминирование программы. Можно, конечно, при всем при этом надеяться на лучшее. Но программы тогда получаются весьма «хрупкими» в том смысле, что нужны дополнительные усилия с тем, чтобы программа выполнялась хоть минимально приемлемо. Правда, при этом возможно повторение ошибок на более поздних этапах разработки или вообще у невинных пользователей, которым программа будет «сдана». Если терминирование программы неприемлемо, то нужно перехватывать все исключения (§13.5.2.2). Ведь программа терминируется только тогда, когда программист позволяет это. В типичном случае, безусловное терминирование предпочтительно лишь в случае, когда традиционное неполное восстановление приводит к катастрофическим последствиям. Там же, где терминирование приемлемо, этого можно добиться неперехватываемым исключением, поскольку в этом случае вызывается функция **terminate()** (§13.5.2.5). Спецификатор **noexcept** (§13.5.1.1) подчеркивает это намерение.

Иногда программисты пытаются хотя бы частично замаскировать неприятные аспекты методики «затягивать время и не решать проблему» выводом сообщений об ошибках, применением диалоговых окон с вопросами к пользователю программы о том, что бы он хотел в такой ситуации сделать и т.д. Такие подходы приемлемы на стадии отладки программы, когда ее пользователем является программист, знакомый с внутренней структурой программы. В руках же конечных пользователей библиотека, спрашивающая что делать, вряд ли имеет смысл. Хорошие библиотеки так не поступают. Если обработчик исключения считает, что нужно информировать пользователя, то он должен сформировать грамотно составленное сообщение и поместить его в тот или иной журнальный файл. Исключения формируют путь, по которому код, обнаруживший ошибку, передает информацию о ней наверх тому коду, который знаком с контекстом работы программы и может, по крайней мере, сформировать информативное сообщение об ошибке.

Но стоить помнить и том, что механизм обработки исключений хоть и составляет разумную альтернативу традиционным способам, все равно остается нетривиальным по сравнению со средствами языка, оперирующими локально. Исключения открывают всю сложность проблемы обработки динамических ошибок, но сами они не являются источником этих сложностей. Не нужно винить гонца за плохие новости.

13.1.4. Альтернативный взгляд на исключения

«Исключения» относятся к тем терминам, на которые у разных людей имеется разные взгляды. Механизм языка C++ для обработки исключений разработан для того, чтобы обрабатывать ошибки, с которыми невозможно справиться локально (в точке возникно-

вения). Особенно такой механизм важен в случае, когда отдельные части программы разрабатываются независимо отдельными группами программистов. Термин «исключение» является обманчивым. Действительно, разве можно считать событие, которое происходит в программе почти всегда, исключением? И далее, разве можно называть событие в программе ошибкой, если заранее планируется деятельность по обработке такого события. Но в данном контексте исключение не есть нечто крайне редкое или катастрофическое.

13.1.4.1. Асинхронные события

Механизм обработки исключений относится лишь к синхронным событиям, таким как нарушение диапазона индексов массива или ошибки ввода/вывода. Асинхронные события, такие как прерывания от клавиатуры или проблемы с питанием, не являются исключениями и не подпадают под действие этого механизма. Асинхронные события нуждаются в механизме обработки, принципиально отличающимся от механизма исключений. Во многих системах имеются такие механизмы, например сигналы, направленные на отражение асинхронности. Поскольку такие механизмы являются системно-зависимыми, мы не будем их рассматривать в настоящей книге.

13.1.4.2. Исключения, которые не являются ошибками

Следует трактовать исключения следующим образом: «часть системы, которая не может выполнить то, что ей поручено» (§13.1.1, §13.2).

Генерация исключений в нормальной программе должна встречаться реже, чем вызов функций, иначе структура программы станет туманной. В то же время, в большинстве программ должны генерироваться и перехватываться хотя бы основные исключения, чтобы программа выполнялась предсказуемо и без неожиданностей.

Если исключение ожидается и перехватывается, так что никаких неприятных последствий для работы программы не остается, то можно ли это трактовать как ошибку? Только как представление программиста об этом. Альтернативно можно думать о механизме обработки исключений как еще об одной управляющей структуре, как об альтернативном способе вернуть некоторое значение. Рассмотрим пример, в котором выполняется поиск в бинарном дереве:

```
void fnd(Tree* p, const string& s)
{
    if (s == p->str) throw p;        // находим s
    if (p->left) fnd(p->left, s);
    if (p->right) fnd(p->right, s);
}

Tree* find(Tree* p, const string& s)
{
    try {
        fnd(p, s);
    }
    catch (Tree* q) {    // q->str==s
        return q;
    }
    return 0;
}
```

Такое решение не лишено некоторого очарования, но лучше этого избегать во избежание проблем с пониманием и по причине неэффективности. Лучше придерживаться парадигмы «обработка исключений есть обработка ошибок». Когда следуют этой пара-

дигме, код четко разделяется на две категории: обычный код и код обработки исключения. Это помогает лучше понять код. Такое четкое разделение также способствует повышению эффективности кода.

И, тем не менее, обработка ошибок является непростой темой. Все, что помогает при этом следовать ясной модели, не должно недооцениваться.

13.1.5. Когда нельзя использовать исключения

Использование исключений является прямым и систематическим способом обработки ошибок в программах на C++. Но с неохотой нужно признать, что существуют программы, в которых по практическим и историческим причинам мы не можем использовать исключения. Вот примеры:

- Критический по времени выполнения компонент системы реального времени должен гарантировать завершение за фиксированный отрезок времени. В отсутствие инструментов измерения времени, за которое проходит распространение кода от точки генерации исключения до точки, в которой оно перехватывается, следует обратиться к альтернативным механизмам работы с ошибками.
- Большая старая программа, в которой управление ресурсами выполняется на основе хаотического вызова голых **new** и **delete** вместо использования дескрипторов ресурсов (таких как **string** или **vector**; §4.2, §4.4).

В таких случаях мы вынужденно оказываемся на территории традиционных приемов (до эпохи исключений) обработки ошибок. Поскольку эти программы создавались в различных исторических контекстах и со множеством практических ограничений, невозможно дать общие рекомендации «как со всем этим бороться». Все же, я бы выделил две популярные методики:

- Наделять каждый класс с конструктором соответствующим методом **invalid()**, возвращающим код ошибки (**error_code**). В случае успеха традиционное значение для **error_code** равно **0**. Если конструктору не удается отработать беспроблемно, вызов **invalid()** должен гарантировать отсутствие утечек, а значение **error_code** будет ненулевым. Предложенный вариант решает проблему возврата кода ошибки из конструктора (после работы конструктора) — пользователь систематически вызывает **invalid()** после конструирования любого объекта, и в случае ошибки получает возможность обработки ошибки по коду возврата. Пример:

```
void f(int n)
{
    my_vector<int> x(n);
    if (x.invalid()) {
        // ... обрабатываем ошибку ...
    }
    // ...
}
```

- Для имитирования работы функции, которая либо возвращает значение, либо генерирует исключение, определяют функцию, возвращающую **pair<Value, Error_code>** (§5.4.3). Пользователь может систематически проверять **Error_code** после каждого вызова функции, и в случае проблемы обрабатывать ошибку. Пример:

```
void g(int n)
{
    auto v = make_vector(n); // возвращаем pair
    if (v.second) {
        // ... обрабатываем ошибку ...
```

```
        }
        auto val = v.first;
        // ...
    }
```

Различные вариации этой схемы широко встречаются на практике, но все равно они несколько более неуклюжи, чем применение исключений.

13.1.6. Иерархическая обработка ошибок

Главная цель механизма исключений — информировать одну часть программы о том, что другая ее часть не в состоянии выполнить свою работу, так как возникли исключительные обстоятельства. Исходными предположениями для этого механизма служат соображения о том, что разные части программы разрабатываются независимо и разными группами программистов, и что какая-то часть программы знает, что нужно делать после того, как возникла исключительная ситуация.

Чтобы применять обработчики исключений в программе, нужна цельная стратегия. Это означает, что разные части программы должны заранее согласовываться в том отношении, что какие-то части, сталкиваясь с невозможностью выполнить работу, генерируют исключения, для которых в иных частях предназначены соответствующие обработчики. То есть, стратегия не является локальной, и поэтому лучше всего ее продумать на ранних стадиях разработки программы. Желательно, чтобы стратегия была простой и наглядной, иначе ее трудно будет продвигать систематически на всех этапах разработки программы.

Удачные системы, толерантные к потенциальным ошибкам, должны быть многоуровневыми. Каждый уровень должен справляться с тем набором ошибок, к которому он приспособлен, а остальные ошибки оставлять на обработку верхним частям программы. Механизм исключений языка C++ поддерживает такой взгляд на проблему. Стандартная функция **terminate()** дополняет этот механизм, обеспечивая завершение в тех случаях, когда разработанный в программе механизм работы с ошибками не охватывает всех возможностей. Аналогично, **noexcept** разрешает отказаться от обработки ошибок, если положительные цели недостижимы.

Не каждая функция в состоянии выполнить все свои предусловия. Причины этого варьируются от программы к программе и от программиста к программисту. Но для больших программ эти причины таковы:

1. Объем работы, гарантирующий полную проверку всех необходимых для успеха работы условий (предусловий) слишком велик, так что систематически выполнять проверки нежелательно.
2. Дополнительная нагрузка на память и увеличение времени работы неприемлемы.
3. Функции, написанные на других языках программирования, могут не выполнять проверку предусловий.
4. Чисто локальные проверки на надежность ведут к сложностям, перегружающим совокупную систему и часто понижающим общую надежность.

Разделение программы на четко разграниченные части, которые либо успешно выполняются, либо не могут этого сделать при определенном исходе — это правильно, существенно, достижимо и эффективно. Все следует проектировать в этом стиле: основные библиотеки, подсистемы и ключевые интерфейсные функции. Вообще-то, в большинстве случаев можно буквально каждую функцию спроектировать в этом ключе.

Правда, чаще всего у нас нет такой возможности, чтобы проектировать и создавать все подсистемы с нуля. Поэтому при проектировании системы нам приходится учитывать те ее части, которые реализуют более старые методики обработки ошибок. И мы долж-

ны внимательнейшим образом отследить все их способы работы с ресурсами и выяснить, в каком состоянии они оставляют систему после возникновения ошибки. Важно попытаться так облагородить поведение старых частей системы, чтобы они имитировали общую стратегию обработки ошибок.

В ряде случаев нам приходится конвертировать сообщения об ошибках от одного стиля к другому. Например, мы можем после вызова функции из С-библиотеки проверить errno и сгенерировать исключение или, наоборот, перехватить исключение и установить значение для **errno** перед тем, как возвратиться к С-частям программы из ее С++-частей:

```
void callC()    // вызов С функции из С++; конвертируем errno в исключение
{
    errno = 0;
    c_function();
    if (errno) {
        // ... локальная очистка, если необходимо и возможно ...
        throw C_blewit(errno);
    }
}

extern "C" void call_from_C() noexcept // вызов С++ функции из С; конвертир. исключение в errno
{
    try {
        c_plus_plus_function();
    }
    catch (...) {
        // ... локальная очистка, если необходимо и возможно ...
        errno = E_CPLPLFCTBLEWIT;
    }
}
```

В этих случаях важно систематически выполнять все преобразования между стилями. Но, к сожалению, необходимость в них возникает в программах, представляющих собой мешанину разностилевого кода, в которой отсуствует единая стратегия и поэтому внесение систематичности не является простой задачей.

Обработку ошибок следует стремиться делать (насколько возможно) иерархической. Если функция сталкивается во время выполнения с ошибкой, она не должна просить ближайшую функцию (которая ее вызвала) разобраться с проблемой. Часто это приводит к циклическим зависимостям в программе.

13.1.7. Исключения и эффективность

Обработка исключений может быть реализована так, что никаких дополнительных накладных расходов в связи с генерацией исключений не будет. Генерация исключений в сравнении с вызовом функции не столь и накладна. И нужно помнить, что альтернативные техники работы с ошибками тоже не бесплатны; нередки случаи, когда традиционные приемы обработки ошибок сводятся буквально к половине объема кода всей программы.

Рассмотрим простую функцию **f()**, которая, как кажется, ничего общего не имеет с обработкой исключений:

```
void f()
{
    string buf;
    cin>>buf;
    // ...
    g(1);
    h(buf);
}
```

Но функции **g()** или **h()** могут генерировать исключения, так что **f()** должна содержать код, гарантирующий, что **buf** удаляется корректно в случае исключения.

А если **g()** не генерирует исключения, то она должна сообщать об ошибке каким-либо иным способом. Поэтому код, который обрабатывает ошибки традиционным способом вместо исключений должен отличаться от простого кода, представленного выше:

```
bool g(int);
bool h(const char*);
char* read_long_string();

bool f()
{
    char* s = read_long_string();
    // ...
    if (g(1)) {
        if (h(s)) {
            free(s);
            return true;
        }
        else {
            free(s);
            return false;
        }
    }
    else {
        free(s);
        return false;
    }
}
```

Можно упростить код при помощи локального буфера для **s**, так как это исключит вызов **free()**, но тогда потребуется проверка диапазона. Так что одни сложности просто меняют другие, а не исчезают.

Часто люди не слишком систематично работают с ошибками и часто это не страшно. Но в тех случаях, когда действительно нужна серьезная работа с ошибками, всю эту кухню лучше поручить компьютеру, то есть механизму исключений.

Спецификатор **noexcept** (§13.5.1.1) может помочь улучшить генерируемый код. Рассмотрим следующее:

```
void g(int) noexcept;
void h(const string&) noexcept;
```

В этом случае код, генерируемый для **f()**, может быть улучшен.

Традиционные С-функции не генерируют исключений, так что все их можно снабжать спецификатором **noexcept**. Реализаторы стандартной библиотеки языка С знают, что за исключением минимума функций (вроде **atexit()** и **qsort()**) остальные не генерируют исключений и могут использовать это для оптимизиции кода функций.

Перед написанием С-функции со спецификатором **noexcept** подумайте минуту над тем, не может ли она генерировать исключения. Например, она может быть конвертирована в вариант с операцией языка C++ **new**, которая может генерировать исключение **bad_alloc**, или она может вызвать функцию из стандартной библиотеки C++, где генерируются исключения.

И, как всегда, дискуссиии об эффективности бессмысленны в отсутствие измерений.

13.2. Гарантии исключений

Для восстановления программы после ошибки, то есть после перехвата исключения и продолжения исполнения, нам нужно знать, что можно допустить в отношении состояния программы до и после предпринятых восстановительных действий. Только в этом случае можно организовать осмысленное восстановление после ошибки. Поэтому мы называем функцию безопасной по поводу исключений, если она оставляет программу в корректном состоянии, когда завершается по причине исключения. Естественно, нужно точно специфицировать, что понимается под корректным состоянием программы. С практической точки зрения нам нужно разбить понятие безопасности в связи с исключениями на несколько специфических гарантий.

Когда мы рассматривали объекты, мы ввели понятие классового инварианта (§2.4.3.2, §17.2.1). Мы полагаем, что инвариант устанавливается конструктором и поддерживается всеми функциями, имеющими доступ к представлению объекта вплоть до момента его уничтожения. В этом контексте мы полагаем корректность состояния объекта в моменты после работы конструктора и до срабатывания деструктора. Для данных, которые невозможно рассматривать как объект, мы должны рассуждать аналогичным образом. Например, если два представителя нелокальных данных предполагают конкретное соотношение между ними, мы можем трактовать это как инвариант и наши восстановительные действия должны сохранять его. Например:

```
namespace Points {      // (vx[i], vy[i]) точки для всех i
    vector<int> vx;
    vector<int> vy;
};
```

Здесь полагается, что равенство **vx.size()==vy.size()** всегда верно. Однако, это было постулировано лишь в комментарии, а компиляторы читать комментарии и понимать их не могут. Такие неявные инварианты трудно обнаружить и трудно поддержать.

Перед генерированием исключения функция должна перевести все сконструированные объекты в корректное состояние. Такое корректное состояние может не устроить вызвавшую функцию. Например, **string** может оказаться пустой или контейнер может оказаться неотсортированным. Поэтому для более полного восстановления обработчик ошибки может порождать значения, которые более соответствуют приложению чем те, что имеют место непосредственно перед предложением **catch**.

В стандартной библиотеке C++ имеется концептуальный набор компонентов для создания безопасного в связи с исключениями кода. Библиотека имеет следующие гарантии для каждой библиотечной операции:

- *Базовые гарантии* для всех операций: поддерживаются базовые инварианты всех объектов и никакие ресурсы, такие как память, не утекают. В частности, для всех встроенных или библиотечных типов вы можете уничтожить объект или присвоить ему значение после любой библиотечной операции (§ iso.17.6.3.1).
- *Строгие гарантии* для ключевых операций: дополнительно к базовым гарантиям гарантируется, что либо операция успешно завершается, либо она не имеет эффекта. Это справедливо для ключевых операций, таких как **push_back()**, **insert()** для одного элемента в списке и для **uninitialized_copy()**.
- *Гарантии отсутствия исключений* для некоторых операций: дополнительно к базовым гарантиям для некоторых операций гарантируется, что они не генерируют исключений. Эта гарантия распространяется на небольшое число простых операций вроде **swap()** для двух контейнеров или **pop_back()**.

Обе гарантии — базовая и строгая — имеют место при условии, что

- пользовательские операции (присваивания и функции **swap()**) не оставляют элементы контейнера в некорректном состоянии,
- пользовательские операции не приводят к утечке ресурсов, и
- деструкторы не генерируют исключений (§ iso.17.6.5.12).

Нарушение требований стандартной библиотеки, таких как запрет на выход из деструктора посредством исключения, эквивалентно нарушению таких правил языка, как разыменование по нулевому указателю. Последствия в обоих случаях эквивалентны и часто ужасны.

Как базовые гарантии, так и строгие гарантии требуют отсутствия утечек ресурсов. Это важно в случаях, когда система не может позволить себе утечек ресурсов. Операция, генерирующая исключение, должна не только гарантировать сохранение своих операндов в корректном состоянии, но и должна гарантировать, что все захваченные ею ресурсы уже освобождены. В частности, до момента генерации исключения захваченная память должна быть освобождена или же ею владеет объект, который гарантирует ее последующее освобождение. Пример:

```
void f(int i)
{
    int* p = new int[10];
    // ...
    if (i<0) {
        delete[] p;      // уничтожить до throw или утечки
        throw Bad();
    }
    // ...
}
```

Не нужно сводить ресурсы лишь к памяти. Все, что может быть захвачено одной частью системы у другой, все это относится к ресурсам. Это и файлы, блокировки, сетевые соединения и потоки. Функция должна все их освободить или же передать под управление какому-либо дескриптору ресурсов перед тем, как сгенерировать исключение.

Правила языка C++ относительно частичного конструирования и уничтожения дают гарантию, что исключения, генерируемые во время конструирования подобъектов или членов класса будут корректно обработаны без специального внимания со стороны кода стандартной библиотеки (§17.2.3).

В самом общем случае мы должны полагать, что если функция может генерировать исключение, она его сгенерирует. Поэтому нужно структурировать код всегда так, чтобы не потеряться в крысиных норах их управляющих структур и структур данных. При анализе кода по поводу потенциальных ошибок следует добиваться того, чтобы код был четко структурированным и стилизованным идеально; §13.6 содержит реалистичный пример такого кода.

13.3. Управление ресурсами

Когда функция захватывает ресурс, то есть открывает файл, выделяет блок в свободной памяти, захватывает мьютекс (семафор) и т.д., очень важно для качественного функционирования целевого кода, чтобы все ресурсы освобождались надлежащим образом. Часто это означает, что функция, приобретающая ресурс, должна освободить его перед возвратом из функции. Например:

```
void use_file(const char* fn) // наивный код
{
    FILE* f = fopen(fn,"r");"

    // ... используем f ...

    fclose(f);
}
```

Это выглядит приемлемым до тех пор, пока вы не осознаете, что после вызова **fopen()** и перед вызовом **fclose()** может произойти исключение и выход из функции **use_file()**. Такая проблема возникает и в языках, не поддерживающих обработку исключений. Например, функция **longjmp()** из стандартной библиотеки языка С также порождает эту проблему. Наконец, обычный оператор **return** может оставить дескриптор **f** без закрытия.

Первая попытка улучшить поведение функции **use_file()** может быть такой:

```
void use_file(const char* fn) // неуклюжий код
{
    FILE* f = fopen(fn,"r");
    try {
        // ... используем f ...
    }
    catch (...) {    // перехват всех исключений
        fclose(f);
        throw;
    }
    fclose(f);
}
```

Этот код работу с файлом заключает в **try**-блок, перехватывает любые исключения, закрывает файл и повторно генерирует исключение. Но такое решение слишком накладное и может стать довольно непростым в случае, когда захватывается множество ресурсов. Существует и более элегантное решение. Общая форма проблемы, с которой мы сталкиваемся, выглядит так:

```
void acquire()
{
    // захват ресурса 1
    // ...
    // захват ресурса n

    // ... используем ресурсы ...

    // освобождаем ресурс n
    // ...
    // освобождаем ресурс 1
}
```

Заметим, что в типичном случае важно, чтобы ресурсы освобождались в порядке, обратном тому, в котором они захватываются. Такая схема сильно напоминает поведение локальных объектов, создаваемых конструкторами и уничтожаемых деструкторами. Поэтому мы можем справиться с проблемой, используя объекты классов с конструкторами и деструкторами. Например, мы можем определить класс **File_ptr**, действующий аналогично указателю **FILE***:

```
class File_ptr {
    FILE* p;
public:
```

```
    File_ptr(const char* n, const char* a) // открыть файл n
        :p{fopen(n, a)}
    {
        if (p==nullptr) throw runtime_error{"File_ptr:Can't open file"};
    }

    File_ptr(const string& n, const char* a) // открыть файл n
        :File_ptr{n.c_str(), a}
    {}

    explicit File_ptr(FILE*pp)      // полагаем, что владеем pp
        :p{pp}
    {
        if (p==nullptr) throw runtime_error("File_ptr:nullptr"};
    }

    // ... подходящие операции копирования и перемещения ...

    ~File_ptr() {fclose(p);}

    operator FILE*() {return p;}
};
```

Можно конструировать **File_ptr** на основе **FILE*** или с помощью аргументов, требуемых функцией **fopen()**. В обоих случаях **File_ptr** будет уничтожен по выходу из области видимости и деструктор закроет файл. **File_ptr** генерирует исключение в случае, когда не удается открыть файл, а иначе каждой операции с файлом пришлось бы проводить тестирование на **nullptr**. В итоге наша функция ужимается до минимума:

```
void use_file(const char* fn)
{
    File_ptr f(fn,"r");
    // ... используем f ...
}
```

Деструктор будет вызван автоматически независимо от способа выхода из функции — нормального или по исключению. В результате механизм обработки исключений позволяет очистить основной алгоритм работы от кода обработки ошибок. Результирующий код проще и менее подвержен программистским ошибкам, чем традиционные варианты.

Эта технология работы с ресурсами через локальные объекты обычно называется «Захват ресурса есть инициализация» (§5.2). Она опирается на свойства конструкторов и деструкторов и их взаимодействие с обработкой исключений.

Часто возражают, что описанная техника дескрипторных классов занудная и ее вполне можно заменить на надлежащие **catch()** с правильными действиями по восстановлению. Но проблема тут в том, что вам нужно отследить все места в программе (которых может быть десятки и более), где нужно вставлять эти **catch()**, а код для дескрипторных классов пишется лишь один раз.

Объекты не считаются сконструированными до того момента, как завершит свою работу конструктор. Только тогда стек раскручивается (§13.5.1) для вызова деструктора объекта. Объект, составленный из подобъектов, конструируется до той степени, до которой сконструированы его подобъекты. Массив конструируется до той степени, до которой конструируются его элементы (и лишь полностью сконструированные элементы уничтожаются в процессе раскрутки стека).

Конструкторы стараются гарантировать полное и корректное создание объектов. Когда этого достичь не удается, хорошо написанный конструктор восстанавливает (насколько это возможно) состояние системы до момента начала его работы. В идеале хорошо спроектированный конструктор не должен оставлять объект недоконструированным. Этого можно добиться, применяя дескрипторную технику к членам класса.

Рассмотрим класс **X**, конструктор которого должен захватить два ресурса: файл **x** и мьютекс **y** (§5.3.4). Процесс захвата ресурсов может сорваться и будет сгенерировано исключение. Конструктор класса **X** не должен завершаться, захватив, например, файл, но не мьютекс (или наоборот). Более того, этого желательно добиваться, не переваливая на программиста дополнительную ношу. Мы будем использовать два класса, **File_ptr** и **std::unique_lock** (§5.3.4), для того, чтобы с их помощью представлять ресурсы. Захват ресурса будет индицироваться инициализацией локального объекта, представляющего ресурс:

```
class Locked_file_handle {
    File_ptr p;
    unique_lock<mutex> lck;
public:
    X(const char* file, mutex& m)
        :p{file,"rw"},    // захват "file"
        lck{m}            // захват "m"
    {}
    // ...
};
```

Здесь, как и всегда в отношении локальных объектов, реализация позаботится о всей необходимой «кухне». Программисту не будет нужды в это вмешиваться. Если, например, возникнет исключение тогда, когда **p** будет сконструирован, а **lck** нет, то только деструктор для **p** будет вызван для работы (а для **lck** не будет вызван).

Из этого следует, что там, где такая простая модель захвата ресурсов имеет место, там автор текста конструктора не должен писать явный код обработки исключения.

Наиболее общим типом ресурса является память, и **string**, **vector** и другие контейнеры стандартной библиотеки используют дескрипторную технику для неявного манипулирования ресурсами (захвата и освобождения). По сравнению с древней техникой с применением **new** и **delete** это экономит много усилий и помогает избежать ошибок со стороны программиста.

Когда вместо локального объекта нужно использовать указатели, рассмотрите возможность использования стандартных типов **unique_ptr** и **shared_ptr** (§5.2.1, §34.3) во избежание утечек.

13.3.1. Завершающие конструкции

Может так оказаться, что строгая дисциплина представления ресурсов локальными объектами (класс которых имеет деструктор) может надоесть. С этой целью программисты изобрели завершающие конструкции для написания практически произвольного кода, выполняющего нужную очистку после исключения. Сначала при этом определяется класс, деструктор которого выполняет некое (произвольное) действие.

```
template<typename F>
struct Final_action {
    Final_action(F f):clean{f} {}
    ~Final_action() {clean();}
    F clean;
};
```

Это завершающее действие передается в качестве аргумента деструктору. Затем мы определяем функцию для удобного вывода типа действия:

```
template<class F>
Final_action<F> finally(F f)
{
    return Final_action<F>(f);
}
```

Напишем тест для **finally()**:

```
void test()
    // работа с недисциплинированным захватом ресурса
    // показываем, что возможны произвольные действия
{
    int* p = new int{7};                    // стоит использовать unique_ptr (§ 5.2)
    int* buf = (int*)malloc(100*sizeof(int)); // выделение памяти в C-стиле

    auto act1 = finally([&]    {delete p;
        free(buf);                          // освобождение памяти в C-стиле
        cout<< "Goodby, Cruel world!\n";
            }
        );

    int var = 0;
    cout << "var = " << var << '\n';

    // вложенный блок:
    {
        var = 1;
        auto act2 = finally([&]{cout<< "finally!\n"; var=7;});
        cout << "var = " << var << '\n';
    } // здесь вызывается act2

    cout << "var = " << var << '\n';
} // здесь вызывается act1
```

Получаем такой результат:

```
var = 0
var = 1
finally!
var = 7
Goodby, Cruel world!
```

Кроме того, аллоцированная память, адресуемая указателем **p**, и буфер **buf** освобождаются при помощи **delete** и **free**, соответственно.

Связь между **finally()** и ресурсами, которыми она манипулирует, остается целиком в ведении программиста и она не такая явная, как в случае дескрипторов ресурсов, но все же применение **finally()** лучше, чем хаотический разброс очищающего кода по всему блоку.

По сути, **finally()** делает для блока то же, что инкрементирующая часть делает для оператора **for** (§9.5.2): она специфицирует финальные действия в начале блока, где их легче увидеть и где они должны быть с точки зрения логических спецификаций. Она говорит о том, что должно быть выполнено по выходу из блока, спасая программиста от размещения очищающего кода во всех местах, из которых возможен выход потока управления из области видимости.

13.4. Инварианты

Когда предусловие функции (§12.4) не выполняется, она не может корректно выполнить свою работу. Аналогично, когда конструктор не может установить инвариант класса (§2.4.3.2, §17.2.1), объект не пригоден к использованию. В таких случаях я обычно генерирую исключение. Но бывают случаи, когда генерирование исключения не выход (§13.1.5), и есть люди, у которых иной взгляд на то, что нужно делать в случае невыполнения предусловия (и аналогичных условий):

- *Не делайте этого*: обязанность того, кто вызывает код, выполнять предусловия, и если он этого не делает, то пусть все плохое произойдет — тогда постепенно все такие ошибки будут из кода вычищены в процессе модернизации, отладки и тестирования.
- *Терминируйте* (завершайте) *программу*: нарушение предусловия является серьезным просчетом в дизайне программы и она не должна продолжать работать при наличии таких ошибок. Можно надеяться, что вся система не рухнет из-за отказа одного из компонентов, и в последствии система будет исправлена за счет улучшенного дизайна, отладки и тестирования.

Когда люди выбирают одно из таких решений? Первый подход часто вытекает из требований к производительности: повторяющиеся проверки предусловий ведут к тому, что то и дело выполняются проверки логически ненужных условий (если вызывающий код предоставляет корректные данные, то миллионы проверок зря его потом проверяют). Это может приводить к серьезному понижению производительности. Лучше уж перетерпеть несколько зависаний системы в процессе нескольких первых тестов, но не терять производительность.

Второй подход желателен в тех случаях, когда нельзя допустить длительного выполнения мер по исправлению ошибок с предусловиями. Например, может потребоваться слишком сильное усложнение проекта и его реализации. С другой стороны, завершение программы может быть приемлемым, например, если нетрудно перезапустить программу с другими входными данными и вернуться в работу. К такому классу программ относятся мелкие программы, которые мы пишем для собственного потребления.

В реальности во многих системах используется комбинированный вариант из исключений и двух описанных выше альтернатив; различаются же они в том, насколько жестко они относятся к выполнению предусловий и допустимо ли для них восстановление. Часто исключения генерируются без какой-либо надежды на восстановление. Я, например, часто использую исключения для того, чтобы получить приемлемое сообщение об ошибке в журнальном файле перед тем, как повторно попробую запустить процесс (пример: **catch(...)** в функции **main()**).

Применяются разные способы для реализации проверок желаемых условий и инвариантов. Нейтральным по отношению к логике программы способом является использований *утверждений* (*assertion*). Это просто некоторое условие, которое должно иметь значение **true**. В зависимости от конкретной системы различаются результаты проверок утверждений:

- Нужно различать утверждения времени компиляции (проверяются компилятором) и утверждениями времени исполнения (проверяются во время выполнения программы).
- Для утверждений времени исполнения имеется выбор результата проверки между генерацией исключения, завершением или игнорированием.
- Не нужно генерировать дополнительный код, пока логическое условие есть **true**. Например, не нужно проверять утверждения времени исполнения, пока логическое

условие есть **true**. Обычно, такое логическое условие есть нечто вроде флага отладки, уровня проверки и т.п.

- Утверждения не следует делать слишком сложными.

Стандарт предлагает два простых механизма:

- В файле **<cassert>** стандартная библиотека предоставляет макрос **assert(A)**, который проверяет свое условие, **A**, во время выполнения, если только не определена макроконстанта **NDEBUG** (нет отладки) (§12.6.2). Компилятор вставляет вывод сообщения, содержащего невыполненное утверждение, имя исходного файла и номер строки кода, после чего завершает выполнение программы.
- В языке имеется **static_assert(A, message)**, проверяющее свое утверждение, **A**, во время компиляции (§2.4.3.3). Если утверждение не выполняется, компилятор выводит сообщение **message** и приостанавливает компиляцию.

Там, где недостаточно **assert()** и **static_assert()**, можно самим написать проверяющий код. Пример:

```
void f(int n)
    // n должно быть в пределах [1:max)
{
    if (2<debug_level && (n<=0 || max<n)
        throw Assert_error("range problem");
    // ...
}
```

Однако, использование такого кода может затуманить вопрос о том, что собственно мы проверяем:

- Вычисляем условие, при котором выполняется тестирование? (Да, это часть **2<debug_level**).
- Проверяем условие, которое может выполняться в некоторых вызовах и не выполняться в других? (Нет, поскольку мы генерируем исключение — если только кто-то не пытается использовать исключения просто как еще один механизм возврата; §13.1.4.2).
- Проверяем предусловие, которое никогда не должно нарушаться? (Да, исключение это просто наш выбор ответа).

Еще хуже то, что проверка предусловия (или инварианта) может так расползтись по всему коду программы, что будет трудно на них сосредоточиться и легко допустить ошибку. Нам бы пригодился какой-либо признанный механизм для проверки утверждений. Далее мы представим возможный механизм для выражения разных вариаций утверждений и разных вариантов реакции на неуспех их проверок. Сначала я определю механизм, позволяющий решить, нужно ли тестировать и что делать в случае провала:

```
namespace Assert {
    enum class Mode {throw_, terminate_, ignore_};
    constexpr Mode current_mode = CURRENT_MODE;
    constexpr int current_level = CURRENT_LEVEL;
    constexpr int default_level = 1;

    constexpr bool level(int n)
        {return n<=current_level;}

    struct Error:runtime_error {
        Error(const string& p):runtime_error(p) {}
    };

    // ...
}
```

Идея состоит в том, чтобы выполнять проверку, если уровень утверждения ниже или равен **current_level**. Если проверка утверждения дает отрицательный результат, используется **current_mode** для выбора из трех альтернатив. Значения **current_level** и **current_mode** являются константами. Представим, что **CURRENT_MODE** и **CURRENT_LEVEL** задаются в среде построения программы, возможно как опции компилятора.

Программист будет использовать **Assert::dynamic()** для формирования утверждений:

```
namespace Assert {
    // ...

    string compose(const char* file, int line, const string& message)
        // составляем message из имени файла и номера строки
    {
        ostringstream os ("(");
        os << file << "," << line << "):" << message;
        return os.str();
    }

    template<bool condition =level(default_level), class Except = Error>
    void dynamic(bool assertion, const string& message ="Assert::dynamic failed")
    {
        if (assertion)
            return;
        if (current_mode == Assert_mode::throw_)
            throw Except{message};
        if (current_mode == Assert_mode::terminate_)
            std::terminate();
    }

    template<>
    void dynamic<false, Error>(bool, const string&)   // ничего не делаем
    {
    }

    void dynamic(bool b, const string& s)             // действие по умолчанию
    {
        dynamic<true, Error>(b, s);
    }

    void dynamic(bool b)                              // умолчательное сообщение
    {
        dynamic<true, Error>(b);
    }
}
```

Я выбрал имя **Assert::dynamic** (что значит «отработать во время выполнения») по контрасту со **static_assert** (означающее «отработать во время компиляции»; §2.4.3.3).

Этот **Assert** не является частью стандарта и представлен здесь в качестве иллюстрации проблемы и методов реализации. Вариативность требований к реализации механизма утверждений столь велика, что единственный вариант не сможет удовлетворить все запросы.

Мы можем использовать **Assert::dynamic** следующим образом:

```
void f(int n)
    // n должно быть в границах [1:max)
{
    Assert::dynamic<Assert::level(2), Assert::Error>(
        (n<=0 || max<n), Assert::compose(__FILE__,__LINE__,"range problem");
    // ...
}
```

Макросы __FILE__ и __LINE__ раскрываются в точке их появления в исходном коде (§12.6.2).

Если мы хотим использовать умолчательный уровень утверждений, нам не нужно ссылаться на него явным образом:

```
void f(int n)
    // n должно быть в границах [1:max)
{
    Assert::dynamic((n<=0 || max<n), Assert::compose(__FILE__,__LINE__,"range
problem");
    // ...
}
```

Я не думаю, что стоит волноваться по поводу объема кода, затрачиваемого на утверждения, но с применением пространств имен (§14.2.3) и умолчательного сообщения можно добиться минимума:

```
void f(int n)
    // n должно быть в границах [1:max)
{
    dynamic(n<=0||max<n);
    // ...
}
```

Можно управлять проверками и реакцией на них с помощью опций сборки или опциями в программном коде. Таким образом можно иметь отладочную версию системы, в которой проверки ведутся интенсивно, и окончательную пользовательскую версию, в которой их может и не быть.

Лично я склонен все же оставлять небольшое число проверок в финальной версии программы. Например, с **Assert** утверждения, промаркированные нулевым уровнем, будут проверяться всегда. На самом деле, никогда нельзя добиться в больших программах, даже посредством интенсивного тестирования и длительного использования, чтобы в них не осталось абсолютно никакой ошибки. И даже если предположить теоретически такую ситуацию, то всегда остается возможность отказа аппаратуры.

Только разработчик финальной сборки цельной программы имеет возможность решать, какие потенциальные ошибки могут быть приемлемыми. А вот разработчик библиотеки никогда не имеет роскоши завершить выполнение кода в безусловном порядке (молча). Это можно интерпретировать таким образом, что при разработке библиотеки крайне существенно сообщать об ошибках, предпочтительно посредством исключений.

Как обычно, деструкторы не должны генерировать исключения. Так что и думать забудьте о генерации исключения типа **Assert** в деструкторе.

13.5. Генерация и перехват исключений

В этом разделе мы взглянем на исключения с точки зрения языка программирования.

13.5.1. Генерация исключений

Можно генерировать исключения любых типов, которые допускают копирование или перемещение. Например:

```
class No_copy {
    No_copy(const No_copy&) = delete; // запрет копирования (§ 17.6.4)
};
```

```
class My_error {
    // ...
};

void f(int n)
{
    switch (n) {
    case 0:throw My_error{};      // OK
    case 1:throw No_copy{};       // ошибка: нет копированию No_copy
    case 2:throw My_error;        // ошибка: My_error это тип, а не объект
    }
}
```

Перехваченный объект исключения (§13.5.2) в принципе является копией сгенерированного объекта (хотя оптимизатору кода разрешается минимизировать копирование); так что оператор **throw x;** инициализирует временный объект (того же типа, что и тип объекта **x**) на основе **x**. Этот временный объект может затем несколько раз копироваться перед тем, как будет перехвачен: исключение передается назад по цепочке вызовов до тех пор, пока не встретится подходящий обработчик. Тип исключения используется для выбора подходящего обработчика в **catch**-предложения некоторого **try**-блока. Данные, содержащиеся в объекте исключения — если таковые имеются — обычно используются для формирования сообщения или в процессе восстановления. Процесс передачи исключения по стеку в направлении вызывающих функций называется *раскруткой стека* (*stack unwinding*). По выходу из любой области видимости деструкторы отрабатывают так, чтобы все полностью сконструированные объекты надлежащим образом уничтожались. Пример:

```
void f()
{
    string name {"Byron"};
    try {
        string s = "in";
        g();
    }
    catch (My_error) {
        // ...
    }
}

void g()
{
    string s = "excess";
    {
        string s = "or";
        h();
    }
}

void h()
{
    string s = "not";
    throw My_error{};
    string s2 = "at all";
}
```

После генерации исключения в **h()** все строки типа **string**, которые были сконструированы, будут уничтожены в обратном порядке: **"not"**, **"or"**, **"excess"**, **"in"**, но не **"at all"**, которую поток управления не достигает, и не **"Byron"**, которая не была затронута.

Поскольку исключения потенциально копируются несколько раз до того, как будут перехвачены, мы обычно не помещаем в них много данных. Распространены исключения, которые содержат всего несколько слов. Семантика распространения исключений аналогична семантике инициализации, так что не очень накладно генерировать исключения типов (вроде **string**) с семанитикой перемещения. Общеупотребительны исключения без данных вообще; имени типа достаточно для сообщения об ошибке. Например:

```
struct Some_error {};

void fct()
{
    // ...
    if (something_wrong)
        throw Some_error{};
}
```

Имеется небольшая иерархия исключений стандартной библиотеки (§13.5.2), которые можно использовать или непосредственно, или в качестве базовых классов. Пример:

```
struct My_error2:std::runtime_error {
    const char* what() const noexcept {return "My_error2";}
};
```

Стандартные исключения вроде **runtime_error** и **out_of_range** принимают строковые значения в качестве параметров конструкторов и имеют виртуальную функцию **what()** для работы с этими строками. Пример:

```
void g(int n)   // генерирует некое исключение
{
    if (n)
        throw std::runtime_error{"I give up!"};
    else
        throw My_error2{};
}

void f(int n)   // смотрим исключение из g()
{
    try {
        void g(n);
    }
    catch (std::exception& e) {
        cerr << e.what() << '\n';
    }
}
```

13.5.1.1. noexcept-функции

Некоторые функции не генерируют исключений, а другие и не должны этого делать. Чтобы отразить этот факт, мы можем объявить функцию с **noexcept**. Например:

```
double compute(double) noexcept; // не может генерировать исключение
```

Теперь из функции **compute()** никаких исключений не будет. И программисту не нужно теперь беспокоиться о том, чтобы вставлять **try**-блоки и оптимизатору кода не нужно беспокоиться о путях перехода в связи с обработкой исключений.

Однако, компиляторы и компоновщики (линкеры) могут не в полной мере учитывать наличие **noexcept**. А что если программист обманывает, когда ставит **noexcept**, а функция

случайно (или неслучайно) генерирует исключение, которое не перехватывается до выхода из функции? Рассмотрим следующее:

```
double compute(double x) noexcept;
{
    string s = "Courtney and Anya";
    vector<double> tmp(10);
    // ...
}
```

Конструктор класса **vector** может не суметь выделить память под десять значений типа **double** и сгенерировать исключение **std::bad_alloc**. В этом случае программа терминируется. Сделает это она в безусловном порядке, вызвав функцию **std::terminate()** (§30.4.1.3), которая никаких деструкторов из цепочки вызова функций не активирует. Вызываются ли деструкторы из иных областей видимости, лежащих между точкой генерации исключения и нашей **noexcept**-функцией (например, для **s** из функции **compute()**) — это зависит от реализации. Правда, программа все равно уже практически перестает работать, так что нет сильной зависимости от состояния объектов.

Можно сказать, что проставляя **noexcept**, мы говорим о том, что наш код не предназначен для работы с исключениями.

13.5.1.2. Операция noexcept

Имеется возможность объявить функцию, которая принимает спецификатор **noexcept** в зависимости от условия. Пример:

```
template<typename T>
void my_fct(T& x) noexcept(Is_pod<T>());
```

Выражение **noexcept(Is_pod<T>())** означает, что функция **my_fct()** не может генерировать исключений, если **Is_pod<T>()** дает **true**, и может генерировать исключения, если это выражение имеет значение **false**. Я мог бы так объявить функцию, если **my_fct()** копирует свой аргумент, ибо я знаю, что при копировании POD-объектов (см. §8.2.6) исключения не генерируются, в то время как для других типов (например, **string** или **vector**) могут генерироваться.

Предикат в **noexcept()**-спецификации должен быть константным выражением. Простой вариант **noexcept** означает **noexcept(true)**.

Стандартная библиотека предоставляет много разных типов предикатов, полезных для выражения условий, при которых функция может генерировать исключения (§35.4).

Что если предикат, который мы хотим использовать, не так легко выразить с помощью одних лишь предикатов типа? Например, что если критической операцией, которая может или не может генерировать исключения, является вызов **f(x)**? Операция **noexcept()** берет в качестве операнда выражение и возвращает **true**, если компилятор «знает», что оно не может генерировать исключение, и возвращает **false** в противном случае. Например:

```
template<typename T>
void call_f(vector<T>& v) noexcept(noexcept(f(v[0]))
{
    for (auto x:v)
        f(x);
}
```

Двойное упоминание **noexcept** выглядит несколько странно, но **noexcept** не является самой часто используемой операцией.

Операнд у **noexcept()** не вычисляется, так что в данном примере мы не получим ошибку времени выполнения, если мы передадим **call_f()** пустой вектор.

Операция **noexcept(expr)** не совершает никаких подвигов, определяя, может ли **expr** генерировать исключения; она просто смотрит на все операции в выражении **expr** и если для всех из них спецификации **noexcept** дают **true**, она возвращает **true**. Эта операция не влезает в детали определений операций, используемых в выражении **expr**.

Условные **noexcept**-спецификации и **except()** операция крайне важны для операций стандартной библиотеки, применяемых к контейнерам. Например (§ iso.20.2.2):

```
template<class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

13.5.1.3. Спецификации исключений

В старом C++-коде можно встретить *спецификации исключений*. Например:

```
void f(int) throw(Bad, Worse);      // может генерировать лишь Bad или Worse исключения
void g(int) throw();                // не может генерировать исключения
```

Спецификация **throw()** эквивалентна **noexcept** (§13.5.1.1). То есть, если исключение генерируется, то программа завершается.

Иные спецификации, например **throw(Bad, Worse)** имеют такой смысл: если функция (здесь **f()**) генерирует исключения, не перечисленные в списке (или открыто не наследуются от перечисленных исключений), то вызывается *обработчик неожиданных исключений* (*unexpected handler*). По умолчанию результатом неожиданного исключения станет терминирование программы (§30.4.1.3). Непустые **throw**-спецификации (то есть со списком) использовать на практике весьма затруднительно, что предполагает серьезные накладные расходы во время выполнения для определения того, правильное ли исключение возникло. В общем, это средство не давало хороших результатов и теперь оно помечено как устаревшее (deprecated). Не используйте его.

Если есть необходимость в динамическом определении типа исключения, применяйте **try**-блоки.

13.5.2. Перехват исключений

Рассмотрим следующее:

```
void f()
{
    try {
        throw E{};
    }
    catch(H) {
        // не может генерировать исключения
    }
}
```

Обработчик вызывается, если:

1. **H** того же типа, что и **E**
2. **H** однозначный базовый тип для **E**
3. **H** и **E** являются указателями и пп. 1 или 2 справедливы для типов, на которые они указывают
4. **H** это ссылка и пп. 1 или 2 справедливы для типа, на который ссылается **H**

Кроме того, можно добавить **const** к типу, используемому для захвата исключений в той же манере, в которой мы можем добавить этот модификатор к параметру функции.

Это не меняет множество исключений, которые мы перехватываем, но ограничивает нас в возможности их модифицировать.

Исключение копируется после того, как оно сгенерировано (§13.5). Реализации могут применять разные стратегии для хранения и передачи исключений. Гарантируется, однако, что имеется достаточно памяти для операции **new**, чтобы она могла сгенерировать исключение **bad_alloc** (§11.2.3).

Отметим возможность перехвата исключений по ссылке. Типы исключений часто определяются как часть иерархий классов для того, чтобы отразить взаимосвязи между типами ошибок, которые они отражают. Например, см. §13.5.2.3 и §30.4.1.1. Прием организации классов исключений в иерархии хорошо знаком программистам, предпочитающим перехватывать исключения по ссылке.

Фигурные скобки как в **try**-части, так и в **catch**-предложении **try**-блоков, формируют свои области видимости. Следовательно, если имя нужно использовать в обеих частях **try**-блока или вне его, то имя нужно объявлять вне **try**-блока. Пример:

```
void g()
{
    int x1;

    try {
        int x2 = x1;
        // ...
    }
    catch (Error) {
        ++x1;     // OK
        ++x2;     // ошибка: x2 не в области видимости
        int x3 = 7;
        // ...
    }
    catch(...) {
        ++x3;     // ошибка: x3 не в области видимости
        // ...
    }

    ++x1;     // OK
    ++x2;     // ошибка: x2 не в области видимости
    ++x3;     // ошибка: x3 не в области видимости
}
```

Предложение **catch(...)** для перехвата «чего угодно» объясняется в §13.5.2.2.

13.5.2.1. Повторная генерация исключения

Перехватив исключение, бывает нужным снова его сгенерировать, чтобы отправить выше по стеку вызовов в тех случаях, когда локально обработать ошибку в полной мере не удается. Так что далее ее кто-то сможет исправить дополнительно (распределенная система исправления ошибки). Например:

```
void h()
{
    try {
        // ... код, который может генерировать исключения ...
    }
    catch (std::exception& err) {
        if (can_handle_it_completely) {
            // ... обработка ...
            return;
        }
```

```
        else {
            // ... делаем что можем в этом месте ...
            throw; // повторно генерируем
        }
    }
}
```

Повторная генерация выражается ключевым словом **throw** без операнда. Повторную генерацию можно выполнить в **catch**-предложении или в функции, которая вызывается из **catch**-предложения. Если операция повторной генерации будет выполнена тогда, когда нет никакого исключения, то программа завершится при помощи **std::terminate()** (§13.5.2.5). Компилятор может предупредить о некоторых из подобных случаев.

Повторно сгнерированное исключение — это оригинально сгенерированное исключение, а не часть его, идентифицируемая как **exception**. Например, пусть сгенерировано исключение **out_of_range**, тогда функция **h()** перехватит его как простое **exception**, но посредством **throw** будет повторно сгенерировано именно исключение **out_of_range**. А вот если бы я написал **throw err;** вместо простого **throw**, то было бы сгенерировано срезанное исключение (§17.5.4.1) и тот, кто вызвал **h()**, не сумеет перехватить его как **out_of_range**.

13.5.2.2. Перехват всех исключений

В файле **<stdexcept>** определена небольшая иерархия классов исключений стандартной библиотеки с общим базовым классом **exception** (§30.4.1.1). Пример:

```
void m()
{
    try {
        // ... делаем что-нибудь ...
    }
    catch (std::exception& err) { // обработ. исключений стандарт. биб-ки
        // ...очистка ...
        throw;
    }
}
```

Здесь перехватываются все исключения стандартной библиотеки. Однако, исключения стандартной библиотеки являются лишь некоторым множеством типов исключений. Следовательно, вы не можете перехватить все исключения, специфицируя их как **std::exception**. Если кто-либо сгенерирует исключение типа **int** (что неразумно) или исключение из какой-либо частной иерархии, оно не будет перехвачено обработчиком, предназначенным для **std::exception&**.

Тем не менее, нам часто нужно иметь дело с любыми исключениями. Например, если полагается, что **m()** должна оставлять некоторые указатели в исходном состоянии, тогда мы можем написать в обработчике присваивание им приемлемых значений. Как в функциях троеточие означает «любые аргументы» (§12.2.4), так и в **catch(...)** оно означает «перехватить любое исключение». Например:

```
void m()
{
    try {
        // ... что-то ...
    }
    catch (...) { // обработка всех исключений
        // ... очистка ...
        throw;
    }
}
```

13.5.2.3. Множественные обработчики

У **try**-блока может быть несколько **catch**-предложений (обработчиков). Поскольку исключения производных типов могут перехватываться обработчиками разных типов исключений, то порядок, в котором записываются обработчики в рамках **try**-блока, существенен. Обработчики просматриваются по-порядку. Например:

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // ... обработка любых iostream ошибок (§ 30.4.1.1) ...
    }
    catch (std::exception& e) {
        // ...обработка любых ошибок стандарт. биб-ки (§ 30.4.1.1) ...
    }
    catch (...) {
        // ... обработка остальных ошибок (§ 13.5.2.2) ...
    }
}
```

Компилятору известна иерархия классов и поэтому он может побеспокоиться относительно некоторых логических ошибок. Например:

```
void g()
{
    try {
        // ...
    }
    catch (...) {
        // ... обработка любых ошибок (§ 13.5.2.2) ...
    }
    catch (std::exception& e) {
        // ...обработ. ошибок стандарт. биб-ки (§ 30.4.1.1) ...
    }
    catch (std::bad_cast) {
        // ... обработка неудач dynamic_cast (§ 22.2.1) ...
    }
}
```

Здесь **exception** никогда не рассматривается. Даже если мы уберем обработчик, перехватывающий все исключения, **bad_cast** не будет рассматриваться, поскольку он является производным от **exception**.

13.5.2.4. Функции из try-блоков

Тело функции может быть **try**-блоком. Пример:

```
int main()
try
{
    // ... делаем что-то ...
}
catch (...) {
    // ... обработка исключений ...
}
```

Для большинства функций выигрыш в таком решении, когда ее тело состоит из **try**-блока, состоит лишь в ясности нотации. Однако, такое решение позволяет нам иметь дело

с исключениями, генерируемыми инициализаторами членов или базового класса в конструкторах (§17.4). По умолчанию исключение в инициализаторах членов или в базовом классе передается тому, кто вызывает конструкторы класса членов. Но конструктор сам может перехватить эти исключения, если все его тело вместе со списком инициализации членов будет заключено в **try**-блок. Пример:

```
class X {
    vector<int> vi;
    vector<string> vs;

    // ...
public:
    X(int, int);
    // ...
};

X::X(int sz1, int sz2)
try
    :vi(sz1),   // создаем vi с sz1 целых
    vs(sz2),    // создаем vs с sz2 строк
{
    // ...
}
catch (std::exception& err) { // здесь перехватываются исключения из vi и vs
    // ...
}
```

Итак, мы можем перехватывать исключения, генерируемые конструкторами членов. Аналогично, мы можем перехватить исключения в деструкторе, сгенерированные деструкторами членов (хотя деструктор не должен генерировать исключения). Однако, мы не можем «починить» объект и осуществить нормальный возврат, как будто исключения не было: исключение из конструктора члена сигнализирует о том, что сам член может не находиться в корректном состоянии. Также остальные объекты членов класса могут не быть сконструированными, или для них были вызваны деструкторы в процессе раскрутки стека.

Самое большее, что мы можем сделать в конструкторах/деструкторах с телом, состоящим из **try**-блока, это сгенерировать исключение в соответствующем **catch**-предложении (§ iso.15.3).

Всех этих ограничений нет для **try**-блоков в обычных функциях.

13.5.2.5. Завершение

Существуют случаи, когда от обработки исключений следует отказаться в пользу менее тонких приемов обработки ошибок. Здесь можно руководствоваться следующими правилами:

- Не генерируйет исключения во время обработки исключения.
- Не генерируйте исключения, которое не может быть перехвачено.

Реализации механизма исключений в таких случаях выполняют завершение программы.

Если вы умудритесь создать два исключения, которые активны в один и тот же момент времени, то система не будет знать, какое из них обрабатывать: новое исключение или то, что оно уже пытается обработать. Заметим, что исключение считается обработанным сразу, как только происходит вход в **catch**-преложение. Повторное генерирование исключения (§13.5.2.1) или генерация нового исключения внутри **catch**-предложения считается актом

появления нового исключения в момент, когда исходное исключение уже обработано. Вы можете генерировать исключение в деструкторе (даже во время раскрутки стека) до тех пор, пока вы можете перехватить его до того, как оно покинет деструктор.

Специфические правила вызова **terminate()** (§ iso.15.5.1) таковы, что вызов происходит:

- Когда нет походящего обработчика для сгенерированного исключения
- Когда **noexcept**-функция пытается осуществить выход посредством **throw**
- Когда деструктор, вызванный в процессе раскрутки стека, пытается осуществить выход посредством **throw**
- Когда код, вызванный в процессе распространения исключения (например, копирующий конструктор), пытается осуществить выход посредством **throw**
- Когда пытаются повторно сгенерировать исключение (**throw;**) тогда, когда нет текущего исключения в процессе его обработки
- Когда деструктор для статически созданного объекта или объекта, локального для потока, пытается осуществить выход посредством **throw**
- Когда инициализатор для статически созданного объекта или объекта, локального для потока, пытается осуществить выход посредством **throw**
- Когда функция, вызванная посредством **atexit()**, пытается осуществить выход посредством **throw**

В таких случаях вызывается **std::terminate()**.

Под выражением «пытается осуществить выход посредством **throw**» я понимаю то, что где-нибудь генерируется исключение и оно тут же не перехватывается, так что система пытается продвинуть исключение от функции к коду, который ее вызвал.

По умолчанию **terminate()** вызывает **abort()** (§15.4.3). Для большинства пользователей такой выбор разумен — особенно в процессе отладки. Если же такое поведение неприемлемо, то вы сами можете вызвать **std::set_terminate()** из файла **<exception>**, чтобы задать свою терминирующую функцию-обработчик:

```
using terminate_handler = void(*)();      // из <exception>

[[noreturn]] void my_handler()      // обработчик завершения не имеет возврата
{
    // обработка завершения своим способом
}

void dangerous()    // очень опасно!
{
    terminate_handler old = set_terminate(my_handler);
    // ...
    set_terminate(old);  // восстанавливаем старый обработчик завершения
}
```

Возвращаемое значение — это предыдущая функция, переданная **set_terminate()**.

К примеру, терминирующая функция может использоваться для закрытия процесса или для реинициализации системы. Главная идея состоит в том, чтобы назначить функцию **terminate()** в качестве радикальной меры, к которой прибегают тогда, когда стратегия восстановления после ошибок провалилась и самое время перейти на другой уровень решений по поводу того, что делать дальше. Как только произошел вход в обработчик завершения, ничего конкретного по поводу данных в программе сказать нельзя; их нужно считать испорченными. Даже попытка вывести сообщение в **cerr** может окончиться катастрофой. Также отметим, что такие функции, как **dangerous()** не являются безопасными

в плане исключений. Даже простой **return** из этой функции (кроме **throw**) до вызова **set_terminate(old)** оставит **my_handler** на месте, где предполагается, что его там нет.

Обработчик завершения не может выполнять возврат к коду, который его вызвал — в этом случае **terminate()** вызовет **abort()**.

Функция **abort()** означает ненормальный выход из программы. Функция **exit()** осуществляет выход из программы с вовзращаемым значением, которое отражает факт нормального или ненормального выхода (§15.4.3).

Вызов деструкторов (или невызов) при терминировании программы является системно зависимым. В некоторых системах невызов деструкторов имеет важное значение для процесса отладки. В иных системах бывает практически невозможно с архитектурной точки зрения не вызвать деструкторы.

Если нужно гарантировать полную очистку, вы можете добавить обработчик для всех исключений (§13.5.2.2) в функцию **main()** дополнительно к обработчикам исключений, о которых вы реально заботитесь. Например:

```
int main()
try {
    // ...
}
catch (const My_error& err) {
    // ... обработка собственных ошибок ...
}
catch (const std::range_error&)
{
    cerr << "range error:Not again!\n";
}
catch (const std::bad_alloc&)
{
    cerr << "new ran out of memory\n";
}
catch (...) {
    // ...
}
```

Здесь будут перехвачены все исключения, кроме тех, что генерируются в процессе конструирования и деструкции переменных из пространств имен и локальных переменных потока (§13.5.3); не существует способа перехватить перечисленные исключения. Это еще один аргумент против глобальных переменных.

Когда исключение перехватывается, точная точка его генерации в общем случае неизвестна. Это говорит о некоторой потере информации по сравнению с той, которой располагает отладчик. В некоторых средах разработки, для некоторых программ может быть предпочтительным не перехватывать исключения, если нет продуманного проекта восстановления программы после него.

Смотри **Assert** (§13.4) в качестве примера того, как можно закодировать положение оператора **throw** в соответствующем объекте исключения.

13.5.3. Исключения и потоки

Если исключение не перехватывается в потоке **thread** (§5.3.1, §42.2), то вызывается **std::terminate()** (§13.5.2.5). Так что если мы не хотим, чтобы ошибка в одном потоке прекратила работу всей программы, то мы должны перехватывать все ошибки, с которых можем восстановиться, и сообщить об этом другим частям программы, заинтересованным в результатах работы потока. Для этого подходит конструкция **catch(...)** для перехвата всех исключений.

Мы можем передать исключение, сгенерированное в одном потоке, обработчику из другого потока, используя стандартную функцию **current_exception()** (§30.4.1.2). Например:

```
try {
    // ... работа ...
}
catch(...) {
    prom.set_exception(current_exception());
}
```

Эта базовая технология используется **package_task** для обработки исключений в пользовательском коде (§5.3.5.2).

13.6. Реализация класса vector

Стандартный **vector** дает прекрасные примеры приемов написания кода, безопасного по исключениям: его реализация исллюстрирует проблемы, которые возникают во многих контекстах, и решения проблем, которые широко применимы на практике.

Очевидно, что реализация **vector** базируется на многих особенностях языка, предназначенных для поддержки реализации и использования классов. Если вы еще не чувствуете себя накоротке с классами и шаблонами, то вам стоит отложить рассмотрение этого примера до того момента, когда вы уже ознакомитесь с Главой 16, Главой 25 и Главой 26.

Базовыми средствами написания безопасного по исключениям кода являются:

- **try**-блок (§13.5).
- Поддержка технологии «Захват ресурса есть инициализация» (§13.3).

А вот базовые принципы, которыми следует руководствоваться:

- Никогда не освобождайте элемент информации до того, как готова ему замена.
- Всегда оставляйте объекты в корректных состояниях при генерации (или повторной генерации) исключений.

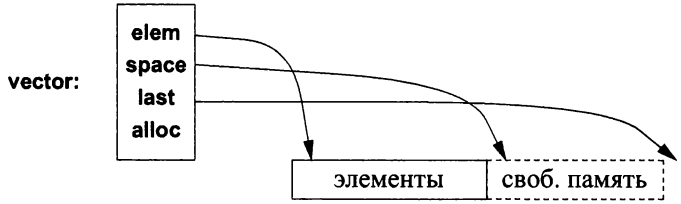
Практическая сложность в применении этих принципов состоит в том, что невинно выглядящие операции, вроде **<**, **=** и **sort()** могут генерировать исключения, поэтому нужен опыт, чтобы понимать, где следует «копать».

Когда вы пишете библиотеку, в идеале нужно стремиться обеспечить строгую гарантию безопасности исключений (§13.2), но всегда нужно обеспечивать базовую гарантию. В конкретных программах можно уделить и меньшее внимание безопасности исключений. Например, если я пишу простую программу анализа данных для собственных нужд, я скорее предпочту, чтобы программа закрылась в случае ошибки, чем допущу исчерпание памяти.

Приемы, обеспечивающие базовые гарантии исключений, такие как формулирование и проверка инвариантов (§13.4), аналогичны простым приемам, помогающим получать компактные и корректные программы. Оказывается, что накладные расходы для обеспечения базовых гарантий безопасности исключений (§13.2) — и даже строгих гарантий — могут быть минимальными или несущественными.

13.6.1. Простой vector

Типичная реализация **vector** (§4.4.1, §31.4) будет заключаться в дескрипторе, который содержит указатели на первый элемент, на элемент, следующий за последним и на оставшееся свободное пространство, выделенное под вектор (§31.2.1) (или эквивалентная информация из указателя и смещения):



Кроме того, дескриптор содержит аллокатор (здесь **alloc**), с помощью которого вектор запрашивает память под элементы. Аллокатор по умолчанию (§34.4.1) использует **new** и **delete** для приобретения и освобождения памяти.

Представим объявление для **vector**, упрощенное с целью демонстрации того, что нужно для иллюстрации вопросов безопасности исключений и устранения опасности утечек ресурсов:

```
template<class T, class A = allocator<T>>
class vector {
private:
    T* elem;  // начало выделяемой памяти
    T* space; // конец последовательности элементов, начало области памяти для расширений
    T* last;  // конец выделенной памяти
    A alloc;  // аллокатор
public:
    using size_type = unsigned int;          // тип для размера

    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);                 // копирующий конструктор
    vector& operator=(const vector& a);      // копирующее присваивание

    vector(vector&& a);                      // перемещающий конструктор
    vector& operator=(vector&& a);           // перемещающее присваивание

    ~vector();

    size_type size() const {return space-elem;}
    size_type capacity() const {return last-elem;}
    void reserve(size_type n);                    // увеличиваем емкость до n

    void resize(size_type n, const T& = {});      // увеличиваем размер до n
    void push_back(const T&);                     // добавляем элемент в конец

    // ...
};
```

Сначала рассмотрим наивную реализацию конструктора, который инициализирует **vector** n-элементами со значением **val**:

```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a) // наивная реализация
    :alloc{a}                    // копия аллокатора
{
    elem = alloc.allocate(n); // память для элементов (§ 34.4)
    space = last = elem+n;
    for (T* p = elem; p!=last; ++p)
        a.construct(p, val); // конструируем копию val в *p (§ 34.4)
}
```

Здесь есть два потенциальных источника исключений:

1. Функция **allocate()** может сгенерирорвать исключение в случае нехватки свободной памяти.
2. Копирующий конструктор типа **T** может сгенерировать исключение, если он не может скопировать **val**.

А что можно сказать про аллокатор? Можно представить, что он тоже может генерировать исключения, но стандарт выдвигает требование, чтобы он этого не делал (§ iso.17.6.3.5). В любом случае, я написал код так, чтобы он не зависел от этого факта.

В случае обоих видов **throw** никакой **vector** не будет сконструирован, так что деструктор для **vector** не будет вызываться (§13.3).

Когда **allocate()** не в состоянии выполнить свою работу, происходит выход по **throw** до того, как захватываются какие-либо ресурсы, так что все в порядке.

Когда копирующий конструктор типа **T** терпит неудачу, мы уже захватили некоторую память, которую нужно освободить во избежание утечек. Еще хуже то, что копирующий конструктор может сгенерировать исключение после того, как некоторое количество элементов уже сконструированы, но не все элементы. Такие объекты типа **T** могут иметь захваченные ресурсы, которые «утекут» из системы.

Для преодоления проблемы мы можем отслеживать те элементы, которые были сконструированы и уничтожить их в случае ошибки:

```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a) // проработанная реализация
    :alloc{a}                          // копир. аллокатор
{
    elem = alloc.allocate(n);          // память под элементы

    iterator p;

    try {
    iterator end = elem+n;
    for (p=elem; p!=end; ++p)
        alloc.construct(p, val);       // конструируем элемент (§ 34.4)
    last = space = p;
    }
    catch (...) {
        for (iterator q = elem; q!=p; ++q)
            alloc.destroy(q);          // уничтожаем сконструированные элементы
        alloc.deallocate(elem, n);     // освобождаем память
        throw;                         // повторная генерация
    }
}
```

Заметьте, что объявление **p** находится вне **try**-блока; в противном случае мы не смогли бы иметь к нему доступ как из **try**-блока, так и из **catch**-предложения.

У нас дополнительные накладные расходы происходят из наличия **try**-блока, но в хороших C++-реализациях эти дополнительные расходя пренебрежимо малы. А для реализации, где это не так, можно добавить проверку **if(n)** перед **try**-блоком для явной обработки распространенного случая пустого вектора.

Большая часть реализации конструктора повторяет реализацию **std::uninitialized_fill()**:

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p=beg; p!=end; ++p)
            ::new(static_cast<void*>(&*p)) T(x);     // конструир. копию x в *p (§ 11.2.4)
```

```
        }
        catch (...) {
            for (For q = beg; q!=p; ++q)
                (&*q)->~T();      // конструир. копию x в *p (§ 11.2.4)
            throw;                // повторная генерация (§ 13.5.2.1)
        }
    }
```

Мудреная конструкция **&*p** заботится об итераторах, не являющихся указателями. В этом случае нам нужно взять адрес элемента, полученного операцией разыменования указателя. Вместе с **::new**, приведение к **void*** гарантирует, что операция размещения стандартной библиотеки (§17.2.4) используется для активизации конструктора, а не определяемой пользователем **operator new()** для типа **T**. Вызов **alloc.destroy()** скрывает явный вызов деструктора (вроде **(&*q)->~T()**). Этот код работает на довольно низком уровне, где написание действительно обобщенного кода затруднительно.

По-счастью, нам не нужно изобретать и реализовывать **uninitialized_fill()**, поскольку в стандартной библиотеке это уже есть (§32.5.6). Часто бывает важным иметь операции инициализации, которые выполняются либо полностью успешно, инициализируя каждый элемент, либо терпят неудачу, не сконструировава ни одного элемента. Поэтому стандартная библиотека предоставляет **uninitialized_fill()**, **uninitialized_fill_n()** и **uninitialized_copy()** (§32.5.6), которые предоставляют сильную гарантию (§13.2).

Алгоритм **uninitialized_fill()** беззащитен против исключений, генерируемых деструкторами элементов и операциями итераторов (§32.5.6). Иначе его было бы реализовать слишком затратно и, вероятно, невозможно.

Алгоритм **uninitialized_fill()** можно применять ко многим видам последовательностей. Он принимает прямой итератор (§33.1.2) и не гарантирует уничтожение элементов в порядке, обратном порядку конструирования.

С помощью **uninitialized_fillll()** можно упростить наш конструктор:

```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a)      // все еще сумбурно
    :alloc(a)                                  // копир. аллокатор
{
    elem = alloc.allocate(n);                  // память под элементы
    try {
        uninitialized_fill(elem, elem+n, val);      // копируем элементы
        space = last = elem+n;
    }
    catch (...) {
        alloc.deallocate(elem, n);             // освобождаем память
        throw; // повторная генерация
    }
}
```

Это уже сильное улучшение по сравнению с первой версией конструктора, но в следующем разделе мы пойдем еще дальше.

Конструктор повторно генерирует пойманное исключение. Идея заключается в том, чтобы **vector** был прозрачным по отношению к исключениям так, чтобы пользователь мог определить, что стало причиной проблемы. Все контейнеры стандартной библиотеки обладают этим свойством. Прозрачность по исключением является наилучшей стратегией для шаблонов и иных «тонких» слоев программного обеспечения. Это контрастирует с основными частями системы («модулями»), которые обычно берут на себя ответственность за все генерируемые исключения. Для этого все исключения группируются в иерархии (§13.5.2) и используется **catch(...)** (§13.5.2.2).

13.6.2. Явное представление памяти

Опыт показывает, что написание корректного кода, безопасного по исключениям с применением **try**-блоков, задача более сложная, чем это себе представляет большинство программистов. Задачу можно упростить, применив технику «Захват ресурса есть инициализация» (§13.3). В таких случаях ключевым ресурсом для **vector** является память под элементы вектора. Посредством вспомогательного класса, представляющего собой модель памяти, которую использует вектор, мы можем упростить код и уменьшить шансы забыть о ее освобождении:

```
template<class T, class A = allocator<T> >
struct vector_base {      // структура памяти под vector
    A alloc;   // аллокатор
    T* elem;   // старт зоны аллокации
    T* space; // конец последовательности эл-ов, начало выделенной области для расширений
    T* last;   // конец выделенной памяти

    vector_base(const A& a, typename A::size_type n)
        :alloc{a}, elem{alloc.allocate(n)}, space{elem+n}, last{elem+n} {}
    ~vector_base() {alloc.deallocate(elem, last-elem);}

    vector_base(const vector_base&) = delete;          // нет копирующим операциям
    vector_base& operator=(const vector_base&) = delete;

    vector_base(vector_base&&);                        // перемещающие операции
    vector_base& operator=(vector_base&&);
};
```

До тех пор, пока **elem** и **last** находятся в корректном состоянии, **vector_base** можно уничтожить. Класс **vector_base** имеет дело с памятью для типа **T**, а не с объектами типа **T**. Как следствие, пользователь класса **vector_base** должен конструировать объекты явным образом в выделенной памяти и позднее уничтожать их перед тем, как будет уничтожаться **vector_base**.

Класс **vector_base** сконструирован исключительно для того, чтобы быть частью реализации **vector**. Всегда трудно предугадать, где и как класс будет использоваться, так что я позаботился о том, чтобы **vector_base** нельзя было копировать и чтобы при перемещении **vector_base** надлежащим образом переходило владение памятью, выделенной под элементы:

```
template<class T, class A>
vector_base<T, A>::vector_base(vector_base&& a)
    :alloc{a.alloc},
    elem{a.elem},
    space{a.space},
    last{a.space}
{
    a.elem = a.space = a.last = nullptr; // теперь не владеем никакой памятью
}

 template<class T, class A>
vector_base<T, A>::& vector_base<T, A>::operator=(vector_base&& a)
{
    swap(*this, a);
    return *this;
}
```

В этом коде определение перемещающего присваивания использует **swap()** для перемещения владения памятью, выделенной под элементы. Здесь нет объектов типа **T**, под-

лежащих уничтожению: **vector_base** имеет дело с памятью и оставляет для **vector** заботы об объектах типа **T**.

Имея **vector_base**, можно определить **vector** следующим образом:

```
template<class T, class A = allocator<T> >
class vector {
    vector_base<T, A> vb;                    // данные здесь
    void destroy_elements();
public:
    using size_type = unsigned int;

    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);                 // копирующий конструктор
    vector& operator=(const vector& a);      // копирующее присваивание

    vector(vector&& a);                      // перемещающий конструктор
    vector& operator=(vector&& a);           // перемещающее присваивание

    ~vector() {destroy_elements();}

    size_type size() const {return vb.space-vb.elem;}
    size_type capacity() const {return vb.last-vb.elem;}

    void reserve(size_type);                 // увеличиваем емкость

    void resize(size_type, T = {});          // изменяем число элементов
    void clear() {resize(0);}                // опустошаем вектор
    void push_back(const T&);                // добавляем элемент в конец

    // ...
};

template<class T, class A>
void vector<T, A>::destroy_elements()
{
    for (T* p = vb.elem; p!=vb.space; ++p)
        p->~T(); // уничтожаем элемент (§ 17.2.4)
    vb.space=vb.elem;
}
```

Деструктор у **vector** явно вызывает деструктор типа **T** для каждого элемента. Из этого следует, что если деструктор элементов генерирует исключение, то уничтожение для **vector** потерпит крах. Это может быть катастрофой, если это происходит во время раскрутки стека, и тогда вызывается **terminate()** (§13.5.2.5). Не существует хороших путей защиты против исключений, генерируемых деструкторами, и поэтому стандартная библиотека не дает никаких гарантий, если генерируются исключения в деструкторах элементов (§13.2).

Теперь мы можем определить конструктор предельно просто:

```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a)
    :vb{a, n} // аллоцируем память под n элементов
{
    uninitialized_fill(vb.elem, vb.elem+n, val); // получаем n копий val
}
```

Упрощение, которое мы достигли в определении конструктора, переносится на все операции, имеющие дело с инициализацией или с выделением памяти. Например, копирующий конструктор отличается лишь заменой **uninitialized_fill()** на **uninitialized_copy()**:

```
template<class T, class A>
vector<T, A>::vector(const vector<T, A>& a)
    :vb{a.alloc, a.size()}
{
    uninitialized_copy(a.begin(), a.end(), vb.elem);
}
```

Стиль, в котором мы написали конструктор, опирается на фундаментальное правило языка: когда исключение генерируется в конструкторе, подобъекты (включая базовые части), которые уже были сконструированы, будут надлежащим образом уничтожены (§13.3). Алгоритм **uninitialized_fill()** и его «родственники» (§13.6.1) предоставляют эквивалентную гарантию для частично сконструированных последовательностей.

Перемещающие операции столь же просты (если не проще):

```
template<class T, class A>
vector<T, A>::vector(vector&& a)    // перемещающий конструктор
    :vb{move(a.vb)}                 // перемещаем владение
{
}
```

Перемещающий конструктор у **vector_base** устанавливает представление аргумента в значение «пусто».

В случае перемещающего присваивания мы должны позаботиться о старом значении целевого объекта:

```
template<class T, class A>
vector<T, A>::& vector<T, A>::operator=(vector&& a)   // перемещ. присваивание
{
    clear();            // уничтожаем элементы
    swap(*this, a);     // перемещаем владение
}
```

Вызов **clear()** является, строго говоря, избыточным, поскольку можно предположить, что rvalue **a** будет уничтожено сразу после присваивания. Однако, я не знаю, как ранее программисты играли с **std::move()**.

13.6.3. Присваивание

Как обычно, присваивание отличается от конструирования лишь тем, что нужно заботиться о старом значении. Сначала рассмотрим прямолинейную реализацию:

```
template<class T, class A>
vector<T, A>& vector<T, A>::operator=(const vector& a) // строгая гарантия (§ 13.2)
{
    vector_base<T, A> b(alloc, a.size());                  // получить память
    uninitialized_copy(a.begin(), a.end(), b.elem);        // копировать элементы
    destroy_elements();                                    // уничтожить старые элементы
    swap(vb, b);                                           // переместить владение
    return *this;                                          // неявно уничтожить старое значение
}
```

Это присваивание предоставляет строгую гарантию, но оно повторяет много кода из конструктора и деструктора. Можно избежать повторов:

```
template<class T, class A>
vector<T, A>& vector<T, A>::operator=(const vector& a) // строгая гарантия (§ 13.2)
{
    vector temp {a};          // копир. аллокатор
    std::swap(*this, temp);   // свопировать представления
    return *this;
}
```

Старые элементы уничтожаются деструктором у **temp** и память под старые элементы освобождается деструктором **vector_base**.

Причина, по которой стандартная функция **swap()** (§35.5.2) работает для **vector_base**, состоит в том, что мы определили перемещающие операции для **vector_base** так, чтобы можно было использовать **swap()**.

Эффективность этих двух версий примерно одинаковая. По сути, они являются просто двумя разными способами определения одних и тех же операций. Однако, вторая версия короче и не дублирует код из функций класса **vector**, что понижает вероятность ошибок программиста и проще в поддержке.

Заметьте, что я не рассмотрел отдельно самоприсваивание, вроде **v=v**. Наше присваивание работает так, что сперва конструируется копия, а потом происходит обмен (swap) представлений. Это очевидно работает и для самоприсваиваний.

У нас в коде отсутствуют два потенциальных пути для оптимизации:

1. Если емкость вектора, которому производится присваивание, больше емкости присваиваемого вектора, нет нужды в аллоцировании новой памяти.
2. Присваивание элементов может быть более эффективным, чем уничтожение элементов с последующим конструированием.

Реализуя эти улучшения, получаем следующее:

```
template<class T, class A>
vector<T, A>& vector<T, A>::operator=(const vector& a) // оптимиз-я: базовая гарантия (§ 13.2)
{
    if (capacity() < a.size()) {        // аллоцировать новое представление для вектора:
        vector temp {a};                // копир. аллокатор
        swap(*this, temp);              // свопировать представления
        return *this;                   // неявно уничтожить старое значение
    }

    if (this == &a) return *this;       // оптимизируем самоприсваивание

    size_type sz = size();
    size_type asz = a.size();
    vb.alloc = a.vb.alloc;              // копир. аллокатор
    if (asz<=sz) {
        copy(a.begin(), a.begin()+asz, vb.elem);
        for (T* p = vb.elem+asz; p!=vb.space; ++p)  // уничтожаем допол-е эл-ты (§ 16.2.6)
            p->~T();
    }
    else {
        copy(a.begin(), a.begin()+sz, vb.elem);
        uninitialized_copy(a.begin()+sz, a.end(), vb.space);      // конструир. доп. эл-ты
    }
    vb.space = vb.elem+asz;
    return *this;
}
```

Конечно, такие улучшения не бесплатны. Очевидно, что сложность кода возросла. Здесь приходится отдельно учитывать самоприсваивание.

Алгоритм **copy()** (§32.5.1) не предоставляет строгой гарантии по исключениям. То есть, если **T::operator=()** генерирует исключение во время работы **copy()**, то **vector**, которому производится присваивание, не обязан быть копией вектора, который присваивается, и не обязан оставаться неизменным. К примеру, первые пять элементов могут быть целевыми копиями, а остальные элементы могут оказаться нетронутыми. Также возможна ситуация, когда элемент, во время присваивания которому было сгенерировано исключение, оста-

нется со значением, не совпадающим ни с исходным, ни с точной копией. Тем не менее, если **T::operator=()** поддерживает операнды в корректном состоянии вплоть до момента генерации исключения (как и должен), то **vector** останется в корректном состоянии, даже если и не в том, в котором мы ожидаем.

Для типа **vector** из стандартной библиотеки реализуется базовая гарантия безопасности по исключениям (с преимуществами в плане производительности). Если вам нужна строгая гарантия неизменности исходного вектора в случае возникновения исключения, вам придется предоставить вашу собственную реализацию операции присваивания. Например:

```
template<class T, class A>
void safe_assign(vector<T, A>& a, const vector<T, A>& b) // простое a = b
{
    vector<T, A> temp{b};   // копируем эл-ты b во временную переменную y
    swap(a, temp);
}
```

В качестве альтернативы можно было бы использовать вызов по значению (§12.2):

```
template<class T, class A>
void safe_assign(vector<T, A>& a, vector<T, A> b) // простое a = b (b передается о значению)
{
    swap(a, b);
}
```

Я так и не смог решить, является ли последняя версия просто красивой, или же слишком умной для реального применения.

13.6.4. Изменение размера

Очень полезной чертой типа **vector** является возможность изменения его размера в соответствии с практическими потребностями. Наиболее распространенными функциями, изменяющими размер вектора, являются **v.push_back(x)**, добавляющая **x** в конец **v**, и **v.resize(s)**, где **s** это новое число элементов в **v**.

13.6.4.1. reserve()

Ключом к простой реализации перечисленных функций является функция **reserve()**, которая добавляет свободную память в конец вектора, что позволяет ему продолжать расти. Другими словами, **reserve()** увеличивает емкость (**capacity()**) вектора. Если новая память под вектор больше прежней, то функция **reserve()** должна выделить новую память и переместить элементы вектора в эту новую память. Можно попробовать следующий трюк из неоптимизированной версии присваивания (§13.6.3):

```
template<class T, class A>
void vector<T, A>::reserve(size_type newalloc)    // первая попытка
{
    if (newalloc<=capacity()) return;             // никогда не уменьшаем аллокацию
    vector<T, A> v(capacity());                   // создаем vector с новой емкостью
    copy(elem, elem+size(), v.begin())            // копируем элементы
    swap(*this, v);                               // внедряем новое значение
} // неявно освободить старое значение
```

К тому же здесь мы имеем роскошь строгой гарантии. К сожалению, не все типы располагают умолчательными значениями. Кроме того, двойной проход по элементам немного неуклюж. Так что давайте оптимизируем:

```
template<class T, class A>
void vector<T, A>::reserve(size_type newalloc)
{
    if (newalloc<=capacity()) return;                    // не уменьшается
    vector_base<T, A> b {vb.alloc, newalloc};            // новый объем памяти
    uninitialized_move(elem, elem+size(), b.elem);       // перемещаем эл-ты
    swap(vb, b);                                         // новая база
} // неявно освободить старый объем
```

Последняя проблема состоит в том, что стандратная библиотека не предоставляет **uninitialized_move()**, так что нам самим придется это написать:

```
template<typename In, typename Out>
Out uninitialized_move(In b, In e, Out oo)
{
    for (; b!=e; ++b,++oo) {
        new(static_cast<void*>(&*oo)) T{move(*b)};   // конструирование перемещением
        b->~T();                                     // уничтожаем
    }
    return b;
}
```

В самом общем случае, не существует способа восстановления исходного состояния после неудачного перемещения, так что я и пытаться не буду. Наш вариант **uninitialized_move()** предлагает только базовую гарантию. Но он простой и для большинства случаев весьма быстр. Аналогично, **reserve()** из стандартной библиотеки также предлагает лишь базовую гарантию.

Если **reserve()** перемещает элементы, то никакие итераторы в этом векторе больше не действительны (§31.3.3).

Помните, что перемещающие операции не должны генерировать исключения. В редких случаях, когда очевидные варианты реализации перемещений могут генерировать исключения, мы не в состоянии с ними справиться. Исключения из перемещающих операций редки, неожиданны и вредят нормальному рассуждению о коде. Если возможно, то избегайте их. Тут может помочь **move_if_noexcept()** из стандартной библиотеки (§35.5.1).

Явный вызов **move()** обязателен, поскольку компилятор не знает, что **elem[i]** вот-вот будет уничтожен.

13.6.4.2. resize()

Метод **resize()** класса **vector** изменяет число элементов в векторе. При наличии **reserve()** реализация **resize()** очень проста. Если число элементов увеличивается, нужно будет конструирорвать новые элементы. А в обратную сторону, когда число элементов уменьшается, мы должны уничтожать излишние элементы:

```
template<class T, class A>
void vector<T, A>::resize(size_type newsize, const T& val)
{
  reserve(newsize);
  if (size()<newsize)
    uninitialized_fill(elem+size(), elem+newsize, val); // новые элементы: [size():newsize)
  else
    destroy(elem.size(), elem+newsize);        // уничтожаем доп. эл-ты: [newsize:size())
  vb.space = vb.last = vb.elem+newsize;
}
```

Стандартной функции **destroy()** нет, но ее легко написать:

```
template<typename In>
void destroy(In b, In e)
{
    for (; b!=e; ++b)    // уничтожаем [b:e)
        b->~T();
}
```

13.6.4.3. push_back()

С точки зрения безопасности по исключениям **push_back()** аналогична присваиванию, так что мы должны заботиться о том, чтобы вектор остался неизменным в случае невозможности добавить новый элемент:

```
template< class T, class A>
void vector<T, A>::push_back(const T& x)
{
    if (capacity()==size())                    // больше нет свобод памяти; relocate:
        reserve(sz?2*sz:8);                    // расти или стартовать с 8
    vb.alloc.construct(&vb.elem[size()], val); // добавить val (значение) в конец
    ++vb.space;                                // инкрементировать размер
}
```

Естественно, что копирующий конструктор, используемый для инициализации ***space**, можен генерировать исключение. Если это произойдет, значение вектора останется неизменным с неинкрементированным **space**. Однако **reserve()** мог уже переаллоцировать существующие элементы.

Определение **push_back()** содержит два «магических числа» (**2** и **8**). Промышленного уровня код не стал бы так делать, но он все равно должен иметь значение, задающее размер начального аллоцирования (здесь это **8**), а также скорость роста (здесь **2** означает, что каждый раз речь идет об удваивания во избежание переполнения). Коэффициент **2** больше математически оптимального значения **1.618** (минимизация используемой в среднем памяти), но он хорош для большего быстродействия на системах с не слишком маленькой памятью.

13.6.4.4. Финальные размышления

Обратите внимание на отсутствие **try**-блоков в реализации **vector** (за исключением одного скрытого внутри **uninitialized_copy()**). Изменения в состоянии достигнуты посредством тщательного упорядочения операций с тем, чтобы в случае исключения вектор оставался неизменным или, по крайней мере, в корректном состоянии.

Подход в достижении безопасности исключений за счет упорядочения операций или за счет техники «Захват ресурса есть инициализация» (§13.3) кажется более элегантным и более эффективным по сравнению с явной обработкой ошибок с использованием **try**-блоков. Основным правилом в упорядочении кода является идея о том, чтобы не уничтожать информацию до того, как будет сконструирована замещающая информация, и чтобы присваивать без возможности исключений.

Исключения являются источником сюрпризов относительно неожиданных скачков (перемещений) потока управления. В образцах кода с простым потоком управления, как в случае **reserve()**, **safe_assign()** и **push_back()**, возможностей для сюрпризов немного. Можно просто взглянуть на все строки кода и спросить, если ли возможность для исключения в той или иной из них? Но в больших частях кода с непростым потоком управления, например в случае условных операторов и вложенных циклов, это сделать непросто. Добавление **try**-блоков еще больше усложняет управление потоком выполнения и может по-

родить дополнительные ошибки и неясности (§13.3). Поэтому я всегда утверждаю, что простой по стилю код легче понимать, легче исправлять и легче получать на его основе эффективный машинный код.

Наш пример с **vector** служит иллюстрацией проблем, которые привносят исключения, и приемов борьбы с этими проблемами. Стандарт не требует в точности следовать показанным приемам, но стандарт требует гарантий безопасности исключений, как это продемонстрировано в нашем примере.

13.7. Советы

[1] Разрабатывайте стратегию обработки ошибок на ранних стадиях проекта; §13.1.

[2] Генерируйте исключение с целью индицирования невозможности выполнить поставленную задачу; §13.1.1.

[3] Используйте исключения для обработки ошибок; §13.1.4.2.

[4] В качестве исключений применяйте специально с этой целью разработанные пользовательские типы; §13.1.1.

[5] Если по каким-то соображениям вы не можете использовать исключения, старайтесь эмулировать их; §13.1.5.

[6] Применяйте иерархическую систему обработки ошибок; §13.1.6.

[7] Старайтесь делать отдельные части системы обработки ошибок простыми; §13.1.6.

[8] Не пытайтесь ловить все исключения во всех функциях; §13.1.6.

[9] Всегда предоставляйте базовую гарантию; §13.2, §13.6.

[10] Предоставляйте строгую гарантию, если только нет явных причин не делать этого; §13.2, §13.6.

[11] Пусть конструктор устанавливает инвариант и генерирует исключение в случае невозможности это выполнить; §13.2.

[12] Освобождайте локальные ресурсы перед тем, как сгенерировать исключение; §13.2.

[13] Удостоверьтесь, что все ресурсы, захваченные в конструкторе, освобождаются, когда в нем генерируется исключение; §13.3.

[14] Не используйте исключения в случаях, когда пострадают локальные структуры управления; §13.1.4.

[15] Для управления ресурсами используйте технику «Захват ресурса есть инициализация»; §13.3.

[16] Минимизируйте число **try**-блоков; §13.3.

[17] Не во всех программах нужна безопасность исключений; §13.1.

[18] Для поддержки инвариантов используйте обработчики исключений и технику «захват ресурса есть инициализация»; §13.5.2.2.

[19] Отдавайте предпочтение дескрипторам ресурсов перед **finally**; §13.3.1.

[20] Пытайтесь организовать вашу стратегию обработки ошибок вокруг инвариантов; §13.4.

[21] Предпочитайте проверять во время компиляции все то, что можно проверить на этапе компиляции (применяя **static_assert**); §13.4.

[22] Старайтесь проектировать вашу стратегию обработки ошибок на разных уровнях проверок; §13.4.

[23] Если ваша функция не может генерировать исключений, маркируйте ее ключевым словом **noexcept**; §13.5.1.1.

[24] Не применяйте спецификации исключений; §13.5.1.3.
[25] Перехватывайте исключения из иерархии по ссылке; §13.5.2.
[26] Не предполагайте, что любое исключение является производным от **exception**; §13.5.2.2.
[27] Заставляйте функцию **main()** перехватывать исключения и выдавать о них сообщения; §13.5.2.2, §13.5.2.4.
[28] Не уничтожайте информацию до того, как ей будет готова замена; §13.6.
[29] Перед тем, как сгенерировать исключение в присваивании, оставляйте операнды в корректном состоянии; §13.2.
[30] Не позволяйте исключениям выскальзывать за пределы деструкторов; §13.2.
[31] Отделяйте код обработки исключений от обычного кода; §13.1.1, §13.1.4.2.
[32] Не забывайте об утечках памяти, выделенных с помощью **new**, и не освобожденных в случае исключений; §13.3.
[33] Полагайте, что любое исключение, которое может быть сгенерировано в функции, будет сгенерировано; §13.2.
[34] Библиотека не должна останавливать программу в безусловной манере. Вместо этого сгенерируйте исключение и пусть вызвавший код принимает решение; §13.4.
[35] Библиотеке не следует выдавать диагностические сообщения, ориентированные на конечного пользователя. Вместо этого сгенерируйте исключение и пусть вызвавший код принимает решение; §13.1.3.

14

Пространства имен

Сейчас 787 год!
Нашей эры?
— Монти Пайтон

- Проблемы композиции
- Пространства имен

  Явная квалификация; **using**-объявления; **using**-директивы; Поиск на основе аргументов; Пространства имен открыты
- Модуляризация и интерфейсы

  Пространства имен как модули; Реализация; Интерфейсы и реализации
- Композиция на основе пространств имен

  Удобства против надежности; Синонимы для пространств имен; Композиция пространств имен; Композиция и выбор; Пространства имен и перегрузка; Версии; Вложенные пространства имен; Безымянные пространства имен; Заголовочные файлы языка C
- Советы

14.1. Проблемы композиции

Все реальные программы состоят из некоторого количества отдельных частей. Функции (§2.2.1, Глава 12) и классы (§3.2, Глава 16) обеспечивают такое разделение на части, при котором «библиотеки», исходные файлы и единицы трансляции (§2.4, Глава 15) служат грубыми строительными блоками. Однако, в идеале хотелось бы иметь такую модульность, которая позволяла бы держать порознь разные логические составляющие, доступ к которым выполнялся бы строго по четко определенному интерфейсу. В языке C++ нет какого-то одного средства, реализующего понятие модуля. Вместо этого модульность реализуется комбинацией таких конструкций языка, как функции, классы и пространства имен, а также при помощи организации исходных файлов.

Данная глава и последующая за ней имеют дело со схематической структурой программы и ее физической реализацией в виде исходных файлов. Таким образом, эти две главы сосредоточены на программировании «в большом», а не на элегантной реализации типов, алгоритмов и структур данных.

Рассмотрим некоторые проблемы, когда программисты проектируют без оглядки на модульность. Например, графическая библиотека может предоставлять разные виды фигур (**Shape**) и функций для работы с ними:

```
// Graph_lib:

class Shape {/* ... */};
class Line : public Shape {/* ... */};
class Poly_line : public Shape {/* ... */};          // связанная последовательность линий
class Text : public Shape {/* ... */};               // текстовая метка

Shape operator+(const Shape&, const Shape&);  // соединение

Graph_reader open(const char*);                      // открыть файл с Shapes
```

Потом кто-то начинает работу с библиотекой, предоставляющей средства манипуляции текстами:

```
// Text_lib:

class Glyph {/* ... */};
class Word {/* ... */};     // последовательность из Glyphs
class Line {/* ... */};     // последовательность из Words
class Text {/* ... */};     // последовательность из Lines

File* open(const char*); // открыть текстовый файл

Word operator+(const Line&, const Line&); // конкатенировать
```

Сейчас мы не будем вдаваться в детали работы с графикой и текстом, а рассмотрим лишь проблемы совместного использования библиотек **Graph_lib** и **Text_lib** в одной и той же программе.

Допустим, что средства библиотеки **Graph_lib** определяются в заголовочном файле (§2.4.1) **Graph_lib.h**, а средства библиотеки **Text_lib** — в файле **Text_lib.h**. Теперь я могу наивно включить оба этих заголовочных файла в свою программу:

```
#include "Graph_lib.h"
#include "Text_lib.h"
// ...
```

В результате получаю кучу сообщений об ошибках: **Line**, **Text** и **open()** определены дважды, так что компилятор не может справиться с неоднозначностью.

Существуют разные приемы, помогающие справиться с такой неоднозначностью, заключающейся в столкновении имен. Например, можно поместить все средства библиотеки в несколько классов с необычными именами или систематически применять какой-либо отличительный префикс имен (вроде **gl_shape** и **gl_line**). Такие приемы можно назвать обходными, но они в ряде случаев работают, хотя и не являются общим решением и часто неудобны в практическом использовании. Например, имена имеют тенденцию удлиняться сверх меры, а их обилие и разнообразие препятствуют обобщенному программированию (§3.4).

14.2. Пространства имен

Понятие пространства имен введено для того, чтобы в явной манере отражать некое единство программных средств, принадлежащих, например, одной библиотеке. Все члены пространства имен разделяют одну и ту же область видимости и могут непосредственно ссылаться друг на друга, в то время как доступ к ним извне требует явной квалификации. В частности, можно избежать столкновения имен, помещая группы объявлений (интерфейсы библиотек) в разные пространства имен. Можно, например, средствам графической библиотеки дать общее имя **Graph_lib**:

```
namespace Graph_lib {
    class Shape {/* ... */};
    class Line : public Shape {/* ... */};
    class Poly_line : public Shape {/* ... */};          // связанная последовательность линий
    class Text : public Shape {/* ... */};               // текстовая метка

    Shape operator+(const Shape&, const Shape&);  // соединение

    Graph_reader open(const char*);                      // открыть файл с Shapes
}
```

Аналогично, очевидным именем для текстовой библиотеки будет **Text_lib**:

```
namespace Text_lib {
    class Glyph {/* ... */};
    class Word {/* ... */};     // последовательность из Glyphs
    class Line {/* ... */};     // последовательность из Words
    class Text {/* ... */};     // последовательность из Lines

    File* open(const char*);                        // открыть текстовый файл

    Word operator+(const Line&, const Line&);  // конкатенируем
}
```

До тех пор, пока нам удается подбирать разные подходящие имена для различных пространств имен, например **Graph_lib** и **Text_lib** (§14.4.2), одновременная компиляция двух наборов объявлений происходит без столкновения имен.

Пространства имен должны отражать некоторую логическую структуру: объявления в пространствах имен должны в совокупности составлять нечто, что объединяет их в глазах пользователей и отражает единообразный набор проектных решений. Они должны представлять логически единое целое, например «графическую библиотеку» или «библиотеку для работы с текстом», то есть действовать в той же манере, в какой мы относимся к членам класса. На самом деле, мы ссылаемся на сущности, объявленные в пространстве имен, как к членам пространства имен.

Пространство имен формирует именованную область видимости. Вы можете ссылаться на члены ранее объявленного пространства имен, но не можете это делать извне без дополнительных усилий. Например:

```
class Glyph {/* ... */};
class Line {/* ... */};

namespace Text_lib {

    class Glyph {/* ... */};
    class Word {/* ... */};     // последовательность из Glyphs
    class Line {/* ... */};     // последовательность из Words
    class Text {/* ... */};     // последовательность из Lines

    File* open(const char*);                        // открыть текстовый файл

    Word operator+(const Line&, const Line&);  // конкатенируем
}
Glyph glyph(Line& ln, int i);                       // ln[i]
```

Здесь **Word** и **Line** в объявлении **Text_lib::operator+()** относятся к **Text_lib::Word** и **Text_lib::Line**. На такой выбор локальных имен никак не влияет присутствие глобального имени **Line**. И наоборот, **Glyph** и **Line** в объявлении глобальной функции **glyph()** ссылаются на глобальные **::Glyph** и **::Line**. На этот выбор никак не влияет наличие **Glyph** и **Line** в пространстве имен **Text_lib**.

Для ссылки на члены пространства имен мы можем использовать полностью квалифицированные имена. Например, если бы нам нужно было, чтобы **glyph()** использовала объекты из **Text_lib**, мы бы написали

```
Text_lib::Glyph glyph(Text_lib::Line& ln, int i); // ln[i]
```

Другими способами ссылки извне на имена из пространства имен являются **using**-объявления (§14.2.2), **using**-директивы (§14.2.3) и выбор имен на основе аргументов (§14.2.4).

14.2.1. Явная квалификация

Можно объявить член пространства имен и позже обращаться к нему в нотации *имя-пространства-имен* :: *имя-члена*.

Члены пространства имен объявляются следующим образом:

```
namespace namespace-name {
    // объявления и определения
}
```

Пример:

```
namespace Parser {
    double expr(bool);      // объявления
    double term(bool);
    double prim(bool);
}

double val = Parser::expr(); // применение

double Parser::expr(bool b)  // определение
{
    // ...
}
```

Нельзя объявить член пространства имен вне определения пространства имен, используя квалифицирующую нотацию (§ iso.7.3.1.2). Идея состоит в том, чтобы минимизировать ошибки из-за проблем с орфографией и для того, чтобы легко можно было находить имена членов непосредственно в объявлении пространства имен. Например:

```
void Parser::logical(bool);     // ошибка: logical() нет в Parser
double Parser::trem(bool);      // ошибка: trem() нет в Parser (описка)
double Parser::prim(int);       // ошибка: Parser::prim() bool аргумент (не тот тип)
```

Пространство имен составляет область видимости. Обычные правила для областей видимости работают и для пространств имен. В итоге имеем то, что пространства имен формируют фундаментальную и относительно простую концепцию. Чем больше размер программы, тем большую роль начинают играть пространства имен для логического разделения частей программы. Глобальное пространство — это тоже пространство имен, на которое можно сослаться с помощью ::, например:

```
int f();    // глобальная ф-я

int g()
{
    int f;      // локальная переменная (скрывает глобаль. ф-ю)
    f();        // ошибка: нельзя вызвать int
    ::f();      // OK: вызов глобальной ф-ии
}
```

Классы являются пространствами имен (§16.2).

14.2.2. using-объявления

Когда имя часто используется в программе вне пространства имен, становится утомительно ссылаться на него при помощи явной квалификации. Рассмотрим следующее:

```
#include<string>
#include<vector>
#include<sstream>

std::vector<std::string> split(const std::string& s)
    // расщепить s на подстроки по пробельным символам
{
    std::vector<std::string> res;
    std::istringstream iss(s);
    for (std::string buf; iss>>buf;)
        res.push_back(buf);
    return res;
}
```

То и дело повторять квалификатор **std** весьма утомительно:мы повторили **std::string** четыре раза в столь маленькой программе. Можно избежать таких назойливых повторений с помощью **using**-объявлений, чтобы сказать про текущий кусок кода, что в нем «**string** означает **std::string**»:

```
using std::string; // используем "string" в смысле "std::string"

std::vector<string> split(const string& s)
    // расщепить s на подстроки по пробельным символам
{
    std::vector<string> res;
    std::istringstream iss(s);
    for (string buf; iss>>buf;)
        res.push_back(buf);
    return res;
}
```

Фактически **using**-объявление вносит синоним в область видимости. Неплохо сохранять локальные синонимы как можно локальнее, чтобы избежать недоразумений.

Когда имя является перегруженным, **using**-объявление аппелирует ко всем перегруженным версиям. Например:

```
namespace N {
    void f(int);
    void f(string);
};

void g()
{
    using N::f;
    f(789);          // N::f(int)
    f("Bruce");      // N::f(string)
}
```

Насчет применения **using**-объявлений в классовых иерархиях смотри §20.3.5.

14.2.3. using-директивы

В функции **split()** (§14.2.2) у нас остались три примера использования **std::**после внедрения **using**-объявления, которое вводит синоним для **std::string**. Очень часто мы хотим

применять имена из пространства имен вообще без квалификации. Этого можно достичь посредством **using**-объявления для каждого имени из пространства имен, но это откровенно утомительно. В этом случае удобна **using**-директива, которая разрешает использовать имена из пространства имен в нашей области видимости вообще без квалификации. Например:

```
using namespace std; // делаем любое имя из std доступным

vector<string> split(const string& s)
    // расщепить s на подстроки по пробельным символам
{
    vector<string> res;
    istringstream iss(s);
    for (string buf; iss>>buf;)
        res.push_back(buf);
    return res;
}
```

Применение **using**-директивы для имен из популярных библиотек является распространенной практикой, упрощающей код. Эту технику я применяю на протяжении всей книги для доступа к именам из стандартной библиотеки. Все средства стандартной библиотеки определены в пространстве имен **std**.

Внутри функций **using**-директивы можно спокойно использовать с целью упрощения записи кода, но нужно быть осторожными с применением глобальных **using**-директив, поскольку их избыточное применение непременно приведет к столкновению имен, избежать которых и призваны пространства имен. Пример:

```
namespace Graph_lib {
    class Shape {/* ... */};
    class Line : Shape {/* ... */};
    class Poly_line : Shape {/* ... */}; // связанная последовательность линий
    class Text : Shape {/* ... */};      // текстовая метка

    Shape operator+(const Shape&, const Shape&);  // сборка

    Graph_reader open(const char*);               // открыть файл из Shapes
}

namespace Text_lib {
    class Glyph {/* ... */};
    class Word {/* ... */};    // последовательность из Glyphs
    class Line {/* ... */};    // последовательность из Words
    class Text {/* ... */};    // последовательность из Lines

    File* open(const char*); // открыть текстовый файл

    Word operator+(const Line&, const Line&);  // конкатенация
}

using namespace Graph_lib;
using namespace Text_lib;

Glyph gl;                  // Text_lib::Glyph
vector<Shape*> vs;         // Graph_lib::Shape
```

Пока что все хорошо. В частности, можно использовать имена, которые не сталкиваются между собой, такие как **Glyph** и **Shape**. Но в случае пересекающихся имен мы снова «сталкиваемся со столкновением», как будто мы и не используем пространства имен. Например:

```
Text txt;                          // ошибка: неоднозначность
File* fp = open("my_precious_data"); // ошибка: неоднозначность
```

Ясно, что с глобальными **using**-директивами нужно быть осторожными. В частности, не помещайте **using**-директивы глобального пространства имен в заголовочные файлы за исключением уникальных обстоятельств, поскольку вы никогда наперед не знаете, куда именно этот заголовочный файл будет включен.

14.2.4. Поиск на основе аргументов

Чаще всего функция, принимающая аргумент пользовательского типа **X**, не определяется в том же самом пространстве имен, что и **X**. Но если не удается найти эту функцию в контексте ее использования, то тогда обращаются к пространству имен ее аргументов. Пример:

```
namespace Chrono {
    class Date {/* ... */};

    bool operator==(const Date&, const std::string&);

    std::string format(const Date&);   // представление для string
    // ...
}

void f(Chrono::Date d, int i)
{
    std::string s = format(d);         // Chrono::format()
    std::string t = format(i);         // ошибка: нет format() в области видимости
}
```

Это правило (поиск на основе аргументов) спасает программиста от избыточной печати на клавиатуре, при этом не допуская загрязнения пространств имен, как это делает **using**-директива (§14.2.3). Это особенно полезно в случае аргументов (операндов) у операций (§18.2.5) и у шаблонов (§26.3.5), где явная квалификация была бы неуклюжей.

Отметим, что пространство имен само должно находиться в области видимости и функция должна быть объявлена до того, как она может быть использована.

Естественно, что у функции могут быть аргументы из более чем одного пространства имен. Пример:

```
void f(Chrono::Date d, std::string s)
{
    if (d == s) {
        // ...
    }
    else if (d == "August 4, 1914") {
        // ...
    }
}
```

В таком случае мы ищем функцию в области видимости ее вызова и в пространствах имен каждого из ее аргументов (включая класс каждого аргумента и их базовые классы), после чего выполняем обычное разрешение перегрузки (§12.3) для всех найденных функций. В частности, для **d==s** мы ищем **operator==** в области видимости, окружающей вызов **f()**, в пространстве имен **std** (где операция == определена для типа **string**) и в пространстве имен **Chrono**. Имеется **std::operator==()**, но эта операция не принимает аргумент типа **Date**, так что мы выбираем **Chrono::operator()**. См. также §18.2.5.

Когда член класса вызывает именованную функцию, другие члены класса и его базового класса имеют преимущество над функциями, которые могут находиться по правилу на основе аргументов (операции следует иному правилу; §18.2.1, §18.2.5). Пример:

```
namespace N {
    struct S {int i};
    void f(S);
    void g(S);
    void h(int);
}

struct Base {
    void f(N::S);
};

struct D : Base {
    void mf();

    void g(N::S x)
    {
        f(x);       // вызов Base::f()
        mf(x);      // вызов D::mf()
        h(1);       // ошибка: нет h(int)
    }
};
```

В стандарте языка правила для поиска на основе аргументов формулируются в терминах ассоциированных пространств имен (§ iso.3.4.2):

- Если аргумент является членом класса, ассоциированное пространство имен — это сам класс (включая базовые классы) и содержащие его пространства имен.
- Если аргумент является членом пространства имен, ассоциированные пространства имен — это объемлющие пространства имен.
- Если аргумент имеет встроенный тип, то никаких ассоциированных пространств имен нет.

Поиск на основе аргументов полезен, может помочь программисту уменьшить объем печати на клавиатуре, но может и привести к сюрпризам. Например, такой поиск объявления функции **f()** не дает преимущества для функций из пространства имен, в которых имеет место вызов **f()** (но это имеет место для функций в классе, в котором вызывается **f()**):

```
namespace N {
    template<class T>
        void f(T, int); // N::f()
    class X {};
}

namespace N2 {
    N::X x;

    void f(N::X, unsigned);

    void g()
    {
        f(x,1); // вызов N::f(X, int)
    }
}
```

Может показаться очевидным выбор **N2::f()**, но это не так. Применяются правила разрешения перегрузок и выбирается **N::f()** — это наилучшее совпадение для **f(x,1)**, поскольку **1** имеет тип **int**, а не **unsigned**. И наоборот, имеются примеры, в которых выбирается функция из пространства имен вызова, но программист может ожидать выбора функции из известного пространства имен (например, из пространства **std** стандартной библиотеки). Все это довольно запутано. См. дополнительно §26.3.6.

14.2.5. Пространства имен открыты

Пространства имен открыты; то есть, вы можете добавлять в них новые имена из разных объявлений. Например:

```
namespace A {
    int f();    // теперь в A есть член f()
}

namespace A {
    int g();    // теперь в A есть два члена, f() и g()
}
```

То есть нет необходимости размещать члены пространства имен смежным образом в одном файле. Это может быть полезным при конвертации старых программ в новые варианты с использованием пространств имен. К примеру, рассмотрим заголовочный файл, который был написан без использования пространств имен:

```
// my header:

    void mf();      // моя функция function
    void yf();      // твоя функция function
    int mg();       // моя функция
// ...
```

Здесь мы разместили объявления без всяких мыслей о модульности. Это можно переписать без переупорядочения объявлений:

```
// my header:

    namespace Mine {
        void mf();      // моя функция
        // ...
    }

    void yf();          // твоя функция (пока что не введена в пространство имен)

    namespace Mine {
        int mg();       // моя функция
        // ...
}
```

Когда я пишу новый код, я предпочитаю небольшие пространства имен (см. §14.4), а не пытаюсь вместить большие куски кода в единственное пространство имен. Это, однако, непрактично в случае конвертации больших кусков кода на их вариант с использованием пространств имен.

Другой причиной размещать члены пространства имен в разных объявлениях является желание разделить члены на интерфейсные и на те, что поддерживают реализацию: в §14.3 приводится соответствующий пример.

Синоним пространства имен (§14.4.2) не может использоваться для повторного входа в пространство имен.

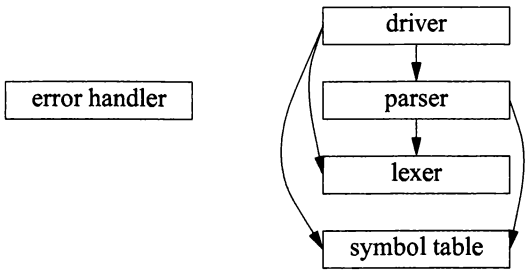
14.3. Модуляризация и интерфейсы

Любая реалистичная программа состоит из некоторого числа различных частей. Например, даже простейшая программа «Hello, World!» включает две части: пользовательский код, запрашивающий печать строки **"Hello, World!"**, и систему ввода/вывода, которая выполняет печать.

Рассмотрим проект настольного калькулятора из §10.2. Его можно считать состоящим из пяти частей:

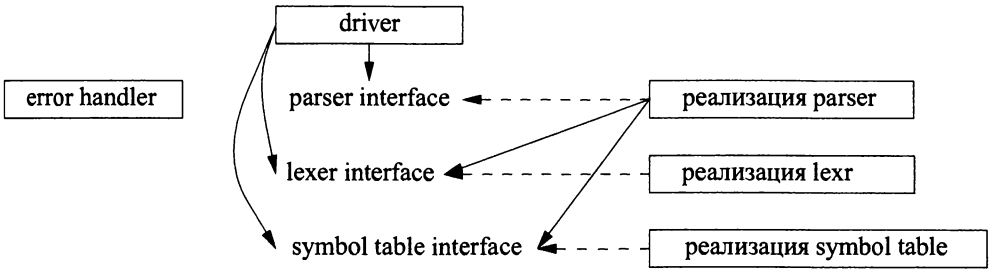
1. Парсер (parser), выполняющий синтаксический анализ: **expr()**, **term()** и **prim()**
2. Сборщик лексем (lexer) из символов: **Kind**, **Token**, **Token_stream** и **ts**
3. Символьная таблица (symbol table), содержащая пары (строка, значение): **table**
4. Основная (управляющая) программа (driver): **main()** и **calculate()**
5. Обработчик ошибок (error handler): **error()** и **number_of_errors**

Можно привести графическую иллюстрацию:



На этом рисунке стрелка означает «использует». Для упрощения графического образа я опустил тот факт, что все части полагаются на обработчик ошибок.

Когда один модуль использует другой модуль, он не обязан знать об используемом модуле абсолютно все. Более того, в идеале все лишние детали модуля должны быть скрыты от его пользователей. Поэтому мы и проводим четкую границу между интерфейсом модуля и его реализацией. Например, парсер непосредственно полагается лишь на интерфейс (interface) сборщика лексем, а не на весь код последнего. Это можно отразить графически следующим образом:



Здесь пунктирные линии означают «реализует». Я рассматриваю эту структуру как план программы и нам, как программистам, остается лишь честно его реализовать. В таком случае код должен получиться ясным, эффективным, простым в понимании, удобным в поддержке и т.д., поскольку он будет напрямую отражать наше проектное решение.

Последующие подразделы показывают, как логическая структура программы настольного калькулятора может быть отражена в коде, а в §15.3 внимание уделено физической организации программы в виде набора файлов с исходными текстами. Калькулятор является небольшой программой, и поэтому в реальной жизни я не стал бы для нее так сильно упирать на пространства имен и раздельную компиляцию (§2.4.1, §15.1), как я это делаю в книге. Максимальное проявление структуры программы-калькулятора служит иллюстрацией для приемов, которые на самом деле важны и полезны в больших программах. В больших программах каждый модуль, представленный отдельным пространством имен, может содержать сотни функций, классов, шаблонов и т.д.

Обработка ошибок затуманивает структуру программы. Разбивая проект программы на модули и (обратно) собирая программу из модулей, нужно стремиться минимизировать зависимости между модулями, вызванные обработкой ошибок. Язык C++ предоставляет исключения как средство разделения обнаружения ошибок и их обработки (§2.4.3.1, Глава 13).

Существуют разные понятия модульности, отличные от представленных в данной главе и в следующей. Например, мы могли бы использовать параллельно выполняющиеся и взаимодействующие между собой задачи (§5.3, Глава 41) или процессы для представления важных аспектов модульности. Аналогично, разделение на адресные пространства и обмен информацией между ними тоже являются важными темами, которые мы здесь не затрагиваем. Я считаю все эти разные понятия модульности перпендикулярными друг другу и относительно независимыми. Серьезной проблемой является обеспечение надежной и эффективной коммуникации через границы модулей.

14.3.1. Пространства имен как модули

Пространства имен служат механизмом, отражающим логическую группировку. То есть, если ряд объявлений совокупно отвечают одному и тому же критерию, они могут быть помещены в одно пространство имен с целью явным образом отразить этот факт. Поэтому мы и воспользуемся пространствами имен для отражения структуры нашей иллюстративной программы-калькулятора. К примеру, объявления для парсера настольного калькулятора (§10.2.1) можно помесить в пространство имен **Parser**:

```
namespace Parser {
    double expr(bool);
    double prim(bool get) {/* ... */}
    double term(bool get) {/* ... */}
    double expr(bool get) {/* ... */}
}
```

Функция **expr()** сначала должна быть объявлена и позже определена, чтобы разорвать циклическую зависимость, описанную в §10.2.1.

Часть настольного калькулятора, которая выполняет ввод, также можно поместить в ее собственное пространство имен:

```
namespace Lexer {
    enum class Kind : char {/* ... */};
    class Token {/* ... */};
    class Token_stream {/* ... */};

    Token_stream ts;
}
```

Символьная таблица предельно проста:

```
namespace Table {
    map<string, double> table;
}
```

Главная программа не может быть помещена целиком в пространство имен, так как правила языка требуют, чтобы функция **main()** была глобальной:

```
namespace Driver {
    void calculate() {/* ... */}
}

int main() {/* ... */}
```

Обработчик ошибок также тривиален:

```
namespace Error {
    int no_of_errors;
    double error(const string& s) {/* ... */}
}
```

Применение пространств имен раскрывает то, что конкретно парсер и сборщик лексем предоставляют пользователю. Если бы я включил и исходный код для функций, то только затуманил бы картину. Действительно, если бы для реалистичных программ я включил бы тела функций в их объявления, то пришлось бы перелистывать содержание множества экранов просто для того, чтобы ознакомиться со списком предлагаемых сервисов (то есть с интерфейсом).

За дополнительной помощью можно обратиться к какому-либо автоматизирующему средству, извлекающему из модуля чистый интерфейс, очищая его от деталей реализации. Вряд ли это самое лучшее решение, поскольку чистое определение интерфейса служит еще и фундаментальным способом проектирования, и к тому же модуль может предоставлять разные интерфейсы разным пользователям; и, наконец, интерфейс может разрабатываться задолго до реализации.

Вот версия **Parser** с интерфейсом, отделенным от реализации:

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
}

double Parser::prim(bool get) {/* ... */}
double Parser::term(bool get) {/* ... */}
double Parser::expr(bool get) {/* ... */}
```

Заметьте, что результатом разделения интерфейса и реализации стал тот факт, что теперь для каждой функции есть ровно одно объявление и одно определение. Пользователи увидят только интерфейс, содержащий объявления. А реализация будет расположена где-то в другом месте, куда пользователю нет нужды обращаться.

В идеале, каждая сущность в программе должна принадлежать какой-нибудь отличительной части («модулю»). Поэтому каждое объявление в нетривиальной программе в идеале должно быть в некотором пространстве имен, поименованном так, чтобы отражать его логическую роль в программе. Исключением служит лишь функция **main()**, которая должна быть глобальной, чтобы компилятор распознал ее (§2.2.1, §15.4).

14.3.2. Реализация

Как будет выглядеть код, будучи модуляризованным? Это зависит от того, как мы хотим обращаться к коду из других пространств имен. К именам из собственного про-

странства имен мы обращаемся без каких-либо изменений. А для имен из других пространств имен мы можем выбирать между явной квалификацией, **using**-объявлениями и **using**-директивами.

Функция **Parser::prim()** наглядно демонстрирует пример использования пространств имен в коде реализации, поскольку она обращается к каждому из других пространств имен (за исключением **Driver**). Если мы применим явную квалификацию, то получим следующее:

```
double Parser::prim(bool get)        // работаем с примитивами
{
  if (get) Lexer::ts.get();

  switch (Lexer::ts.current().kind) {
  case Lexer::Kind::number: // константа с плавающей запятой
  { double v = Lexer::ts.current().number_value;
    Lexer::ts.get();
    return v;
  }
  case Lexer::Kind::name:
  { double& v = Table::table[Lexer::ts.current().string_value];
    if (Lexer::ts.get().kind == Lexer::Kind::assign) v = expr(true); // '=' видим: присваивание
    return v;
  }
  case Lexer::Kind::minus:    // унарный минус
    return -prim(true);
  case Lexer::Kind::lp:
  { double e = expr(true);
    if (Lexer::ts.current().kind!= Lexer::Kind::rp) return Error::error(" ')' expected");
    Lexer::ts.get();        // съедаем ')'
    returne;
  }
  default:
    return Error::error("primary expected");
  }
}
```

Я насчитал 14 повторов **Lexer::** и считаю, что явное отражение модульности улучшает читаемость кода. Я не использовал **Parser::**, поскольку это избыточно внутри пространства имен **Parser**.

Если мы применим **using**-объявления, то получим следующее:

```
using Lexer::ts;     // восемь появлений "Lexer::"
using Lexer::Kind;   // шесть появлений "Lexer::"
using Error::error;  // два появления "Error::"
using Table::table;  // одно появление "Table::"

double prim(bool get) // работаем с примитивами
{
    if (get) ts.get();

    switch (ts.current().kind) {
    case Kind::number:      // константа с плавающей запятой
    {   double v = ts.current().number_value;
        ts.get();
        return v;
    }
    case Kind::name:
    {   double& v = table[ts.current().string_value];
        if (ts.get().kind == Kind::assign) v = expr(true); // '=' видим: присваивание
        return v;
    }
```

```
        case Kind::minus:  // унарный минус
            return -prim(true);
        case Kind::lp:
        {     double e = expr(true);
            if (ts.current().kind!= Kind::rp) return error("')' expected");
            ts.get();   // съедаем ')
            returne;
        }
        default:
            return error("primary expected");
        }
}
```

Мое мнение таково, что применение **using**-объявления для **Lexer**::стоило того. Если мы применим **using**-директивы, то будет следующий результат:

```
using namespace Lexer;      // два появления "Lexer::"
using namespace Error;      // два появления "Error::"
using namespace Table;      // одно появление "Table::"

double prim(bool get)       // работаем с примитивами
{
    // все как ранее
}
```

А **using**-объявления для **Error** и **Table** больших усилий не стоят и утверждение, что они затуманивают источники, из которых вытекают полностью квалифицированные имена, спорно.

Итак, выбор между явной квалификацией, **using**-объявлениями и **using**-директивами зависит от конкретных обстоятельств. Примерные правила таковы:

1. Если квалификация часто встречается для некоторых имен, используйте **using**-директиву для этого пространства имен.
2. Если квалификация часто встречается для конкретного имени из пространства имен, используйте **using**-объявление для этого имени.
3. Если квалификация для имени редка, то применяйте явную квалификацию, отражающую тот факт, откуда происходит имя.
4. Не применяйте явную квалификацию для имен из того же пространства имен, что и у пользователя.

14.3.3. Интерфейсы и реализации

Ясно, что пространство имен, которое мы использовали для **Parser**, не является идеальным интерфейсом для предъявления пользователям. Он скорее представляет собой набор объявлений, который требуется для того, чтобы было удобно (и понятно) программисту для реализации индивидуальных функций. А вот идеальный интерфейс **Parser** для пользователя должен быть существенно более простым и немногословным:

```
namespace Parser {// интерфейс пользователя
    double expr(bool);
}
```

Итак, мы видим, что пространство имен **Parser** применялось для двух целей:

1. Общий контекст для функций, реализующих парсер
2. Внешний интерфейс парсера для пользователей

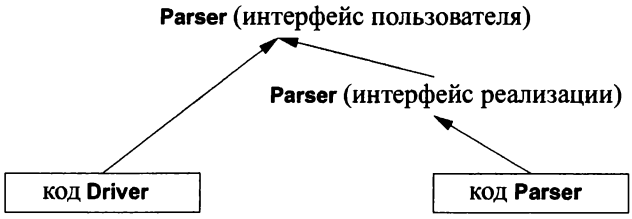
Так, управляющий код, то есть функция **main()**, должен иметь дело только с пользовательским интерфейсом.

Функции, реализующие парсер, должны видеть интерфейс, который мы сочли наиболее подходящим для определения общего контекста этих функций, то есть:

```
namespace Parser {      // интерфейс реализации
    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;      // используем все средства lexer
    using Error::error;
    using Table::table;
}
```

Или графически:



Здесь стрелки означают «полагаются на интерфейс».

Мы могли бы дать интерфейсам для пользователя и для реализации разные имена, но (поскольку интерфейсы открытые; §14.2.5) мы не обязаны так делать. Это не приводит к недоразумениям, поскольку физическая раскадка программы (см. §15.3.2) предоставляет разные имена файлов. Но если бы мы решили дать иное имя для интерфейса реализации, структура проекта для пользователей выглядела бы точно так же:

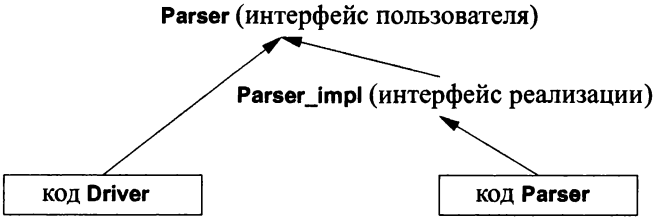
```
namespace Parser { // интерфейс пользователя
    double expr(bool);
}

namespace Parser_impl {     // интерфейс реализации
    using namespace Parser;

    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;      // использовать все средства Lexer
    using Error::error;
    using Table::table;
}
```

Или графически:



Для больших программ я предпочитаю использовать префикс **_impl**.

Интерфейс для реализации более обширный, чем интерфейс для пользователей. И для реалистичных программ интерфейс реализации менялся бы чаще, чем интерфейс для пользователей. Очень важно, чтобы пользователи модуля (здесь это **Driver**, использующий **Parser**) были изолированы от таких изменений.

14.4. Композиция на основе пространств имен

В больших программах используются множество пространств имен. В данном разделе рассматриваются технические вопросы композии кода из пространств имен.

14.4.1. Удобства против надежности

using-объявление добавляет имя в локальную область видимости. А **using**-директива этого не делает; она просто предоставляет имена из области видимости, в которой они были объявлены. Например:

```
namespace X {
    int i, j, k;
}

int k;

void f1()
{
    int i = 0;
    using namespace X;      // делаем имена из X доступными
    i++;                    // локальное i
    j++;                    // X::j
    k++;                    // ошибка: k из X или глобальное k?
    ::k++;                  // глобальное k
    X::k++;                 // k из X
}

void f2()
{
    int i = 0;
    using X::i;     // ошибка: i дважды объявлена в f2()
    using X::j;
    using X::k;     // скрывает глобальную k

    i++;
    j++;        // X::j
    k++;        // X::k
}
```

Локальные имена (объявленные с помощью обычных объявлений или посредством **using**-объявлений) скрывают нелокальные объявления тех же самых имен, так что любое перегруженное объявление имени обнаруживается в точке объявления.

Обратите внимание на ошибку, связанную с неопределенностью для **k++** в **f1()**. Глобальные имена не имеют преимущества перед именами из пространств имен, внесенными в глобальную область видимости. Это обеспечивает защиту против непроизвольных столкновений имен и, что тоже важно, не предоставляет никакой выгоды из загрязнения глобального пространствав имен.

Когда имена из библиотек, объявляющих большое количество имен, делаются доступными посредством **using**-директив, неиспользуемые имена, сталкивающиеся с други-

ми именами, не рассматриваются как ошибки (с практической точки зрения это очень удобно).

14.4.2. Синонимы для пространств имен

Если пользователь присваивает своим пространствам имен короткие имена, возрастает вероятность столкновения таких имен между собой:

```
namespace A {      // короткое имя - с последущим конфликтом
    // ...
}

A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

В то же время, длинные имена для пространств имен не слишком удобно использовать с практической точки зрения:

```
namespace American_Telephone_and_Telegraph { // слишком длинное
    // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

Эта дилемма разрешается посредством предоставления короткого синонима для длинного имени пространства имен:

```
// для укорочения используйте синонимы для пространств имен:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

Синонимы пространств имен позволяют ссылаться на «библиотеку» и иметь единственное объявление о том, что это за библиотека. Например:

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Sibelius";
```

Это значительно упрощает задачу замены одной версии библиотеки на другую. Применяя **Lib** вместо прямолинейного **Foundation_library_v2r11**, вы можете перейти на версию «v3r02» заменой инициализации синонима **Lib** и перекомпиляцией.

14.4.3. Композиция пространств имен

Часто нам нужно составить интерфейс из нескольких существующих интерфейсов. Например:

```
namespace His_string {
    class String {/* ... */};
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
```

```
        void fill(char);
        // ...
}

namespace Her_vector {
        template<class T>
                class Vector {/* ... */};
        // ...
}

namespace My_lib {
        using namespace His_string;
        using namespace Her_vector;
        void my_fct(String&);
}
```

Теперь мы можем написать программу в терминах **My_lib**:

```
void f()
{
        My_lib::String s = "Byron"; // находим My_lib::His_string::String
        // ...
}

using namespace My_lib;

void g(Vector<String>& vs)
{
        // ...
        my_fct(vs[5]);
        // ...
}
```

Если явно квалифицированное имя (вроде **My_lib::String**) не имеет объявления в упомянутом пространстве имен, то компилятор ищет его в пространстве имен из **using**-директив (таких как **His_string**).

И только в случае необходимости определения сущности нам нужно знать реальное пространство имен:

```
void My_lib::fill(char c) // ошибка: нет fill(), объявленной в My_lib
{
        // ...
}

void His_string::fill(char c) // OK: fill() объявлена в His_string
{
        // ...
}

void My_lib::my_fct(String& v) // OK: String есть My_lib::String, означая His_string::String
{
        // ...
}
```

В идеале пространство имен должно:

1. Быть логически согласованным набором сущностей,
2. Не давать пользователю доступа к иным сущностям и
3. Не налагать слишком большой именующей нагрузки на пользователей.

Совместно с механизмом директив **#include** (§15.2.2) представленная здесь техника композиции осуществляет серьезную поддержку перечисленному.

14.4.4. Композиция и выбор

Комбинация композиции (с помощью **using**-директив) и выбора (посредством **using**-объявлений) дает достаточную гибкость для большинства реальных задач. С помощью этих механизмов мы обеспечиваем доступ к набору средств таким образом, что удается разрешить и столкновение имен и неоднозначности, вытекающие из композиции. Пример:

```
namespace His_lib {
    class String {/* ... */};
    template<class T>
        class Vector {/* ... */};
    // ...
}

namespace Her_lib {
    template<class T>
        class Vector {/* ... */};
    class String {/* ... */};
    // ...
}

namespace My_lib {
    using namespace His_lib;    // что угодно из His_lib
    using namespace Her_lib;    // что угодно из Her_lib

    using His_lib::String;      // разрешаем потенциальное столкновение в пользу His_lib
    using Her_lib::Vector;      // разрешаем потенциальное столкновение в пользу Her_lib

    template<class T>
        class List {/* ... */};     // дополнительный материал
    // ...
}
```

Имена, явно объявленные в пространстве имен, имеют преимущество перед именами, внесенными туда с помощью **using**-директив (см. также §14.4.1). Следовательно, пользователи **My_lib** в столкновении имен **String** и **Vector** увидят выбор в пользу **His_lib::String** и **Her_lib::Vector**. Также **My_lib::List** будет использоваться по умолчанию, независимо от того, предоставляется ли **List** в **His_lib** или **Her_lib**.

Обычно я предпочитаю не менять имя, когда внедряю его в новое пространство имен. Поэтому не возникает необходимости помнить два имени для одной и той же сущности. Но иногда нужно дать новое имя. Например:

```
namespace Lib2 {
    using namespace His_lib;    // что угодно из His_lib
    using namespace Her_lib;    // что угодно из Her_lib

    using His_lib::String;      // разрешаем потенциальное столкновение в пользу His_lib
    using Her_lib::Vector;      // разрешаем потенциальное столкновение в пользу Her_lib

    using Her_string = Her_lib::String;             // переименование
    template<class T>
        using His_vec = His_lib::Vector<T>;         // переименование

    template<class T>
        class List {/* ... */};                     // доп. материал
    // ...
}
```

Для переименования не существует общего механизма языка, но для типов и шаблонов мы можем ввести синонимы с помощью **using** (§3.4.5, §6.5).

14.4.5. Пространства имен и перегрузка

Перегрузка функций (§12.3) работает поверх пространств имен. Это помогает нам мигрировать со старых библиотек, применяя пространства имен с минимальными изменениями исходного кода. Например:

```
// старый A.h:
    void f(int);
    // ...

// старый B.h:
    void f(char);
    // ...

// старый user.c:
    #include "A.h"
    #include "B.h"

    void g()
    {
        f('a'); // вызов f() из B.h
    }
```

Эту программу можно апгрейдировать до версии, использующей пространства имен, не прибегая к изменению реального кода:

```
// новый A.h:

    namespace A {
        void f(int);
        // ...
    }

// новый B.h:

    namespace B {
        void f(char);
        // ...
    }

// новый user.c:

    #include "A.h"
    #include "B.h"

    using namespace A;
    using namespace B;

    void g()
    {
        f('a'); // вызов f() из B.h
    }
```

Если бы мы захотели совсем не изменять файл **user.c**, то мы бы поместили **using**-директивы в заголовочные файлы. Но обычно лучше избегать **using**-директив в заголовочных файлах, поскольку это сильно увеличивает вероятность столкновения имен.

Правило перегрузки предоставляет механизм для расширения библиотек. Например, многие часто удивляются, почему они должны явно ссылаться на последователь-

ность при манипулировании контейнером, используя алгоритм стандартной библиотеки. Например:

```
sort(v.begin(), v.end());
```

Почему бы не написать

```
sort(v);
```

Причина кроется в необходимости обобщения (§32.2), но манипулирование контейнером чаще встречается в настоящее время. Можно все это согласовать:

```
#include<algorithm>

namespace Estd {
    using namespace std;
    template<class C>
        void sort(C& c) {std::sort(c.begin(), c.end());}
    template<class C, class P>
        void sort(C& c, P p) {std::sort(c.begin(), c.end(), p);}
}
```

Здесь **Estd** (extended **std** — расширенное **std**) предоставляет контейнерную версию для **sort()**. А для реализации все равно используется **std::sort()** из **<algorithm>**. Это можно использовать следующим образом:

```
using namespace Estd;

template<class T>
void print(const vector<T>& v)
{
    for (auto& x : v)
        cout << v << ' ';
    cout << '\n';
}

void f()
{
    std::vector<int> v {7, 3, 9, 4, 0, 1};

    sort(v);
    print(v);
    sort(v,[](int x, int y) {return x>y;});
    print(v);
    sort(v.begin(), v.end());
    print(v);
    sort(v.begin(), v.end(),[](int x, int y) {return x>y;});
    print(v);
}
```

Правила поиска имен в пространствах имен и правила перегрузки для шаблонов гарантируют нахождение и вызов корректного варианта для **sort()**, так что получается ожидаемый вывод результатов работы:

```
013479
974310
013479
974310
```

Если мы уберем **using namespace std;** из **Estd**, то наш пример все равно будет работать, поскольку **sort()** из **std** будет выбран по правилу поиска на основе аргументов (§14.2.4).

Однако, мы тогда не выйдем на стандартный **sort()** для наших собственных контейнеров, определенных вне пространства имен **std**.

14.4.6. Версии

Самым жестким тестом для всех видов интерфейсов является необходимость справляться с последовательностью новых версий. Рассмотрим такой широко известный интерфейс, как стандартный заголовок ISO C++. Через какое-то время выходит новая версия, например C++11 версия для заголовочного файла стандарта C++98. Могут быть добавлены новые функции, переименованы классы, устранены какие-либо частные расширения (которых, вообще-то, там и быть не должно), модифицированы типы и шаблоны. Описанные нами средства, основанные на пространствах имен, могут решить указанные проблемы за минимальными исключениями. Правда, когда эти минимальные исключения тянут на миллионы строк кода, проблема все равно не маленькая. Как следствие, нужен способ явно выбрать из двух версий, который гарантирует, что пользователь будет видеть строго одну версию. Этот способ называется *встраиваемым пространством имен* (*inline namespace*):

```
namespace Popular {

    inline namespace V3_2 {// V3_2 предоставляет умолчательный смысл для Popular
        double f(double);
        int f(int);
        template<class T>
            class C {/* ... */};
    }

    namespace V3_0 {
        // ...
    }
    namespace V2_4_2 {
    double f(double);
    template<class T>
        class C {/* ... */};
    }
}
```

Здесь **Popular** содержит три подпространства имен, каждое из которых определяет свою версию. Ключевое слово **inline** специфицирует **V3_2** как умолчательное значение для **Popular**. Так что можно писать:

```
using namespace Popular;

void f()
{
    f(1);           // Popular::V3_2::f(int)
    V3_0::f(1);     // Popular::V3_0::f(double)
    V2_4_2::f(1);   // Popular::V2_4_2::f(double)
}

template<class T>
Popular::C<T*> {/* ... */};
```

Решение на основе **inline namespace** требует модификации кода заголовочных файлов. Можно минимизировать объем работ по модификации кода с помощью трюков с **#include**. Например:

```
// файл V3_common:
    // ... масса объявлений ...
```

```
// файл V3_2:

    namespace V3_2 {// V3_2 предоставляет умолчательный смысл для Popular
        double f(double);
        int f(int);
        template<class T>
            class C {/* ... */};
        #include "V3_common"
    }

// файл V3_0.h:

    namespace V3_0 {
        #include "V3_common"
    }

// файл Popular.h:

    namespace Popular {
        inline
        #include "V3_2.h"
        #include "V3_0.h"
        #include "V2_4_2.h"
    }
```

Конечно, не стоит применять такие трюкачества, кроме как в случаях, когда без этого никак не обойтись.

В большинстве случаев мы можем поддержать версионность без подобного рода трюков. Я могу привести лишь один пример, когда другими средствами сделать ничего нельзя:это специализация шаблона, явно использующая имя пространства имен (например, **Popular::C<T*>**).

14.4.7. Вложенные пространства имен

Очевидным примером использования вложенных пространств имен является случай, когда нужно обернуть весь набор объявлений и определений отдельным пространством имен:

```
namespace X {
    // ... все мои объявления ...
}
```

Здесь в полном списке объявлений вполне могут встретиться пространства имен; таким образом возникают вложенные пространства имен. Вложенные пространства имен в C++ разрешены, да и вообще торжествует принцип, что вложения допускаются всюду кроме тех конструкций, где существуют конкретные и серьезные причины отказа от вложений. Пример:

```
void h();

namespace X {
    void g();
    // ...
    namespace Y {
        void f();
        void ff();
        // ...
    }
}
```

Применяются обычные правила областей видимости и операций квалификации:

```
void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f();        // ошибка: нет f() в X
    Y::f();     // OK
}

void h()
{
    f();        // ошибка: нет глобальной f()
    Y::f();     // ошибка: нет глобальной Y
    X::f();     // ошибка: нет f() в X
    X::Y::f();  // OK
}
```

По поводу примеров вложенных пространств имен в стандартной библиотеке смотри **chrono** (§35.2) и **rel_ops** (§35.5.3).

14.4.8. Безымянные пространства имен

Иногда нужно просто обернуть объявления пространством имен с целью избежать столкновения имен. То есть, целью является локализация имен, а не представление интерфейса для пользователей. Пример:

```
#include "header.h"
namespace Mine {
    int a;
    void f() {/* ... */}
    int g() {/* ... */}
}
```

Поскольку не нужно, чтобы имя **Mine** было видно вне локального контекста, то объявление его на глобальном уровне порождает лишь неприятности, поскольку засоряет глобальное пространство имен (увеличивает вероятность столкновения имен). В таком случае можно просто опустить имя в объявлении пространства имен:

```
#include "header.h"
namespace {
    int a;
    void f() {/* ... */}
    int g() {/* ... */}
}
```

Ясно, что при этом должен быть какой-то способ обеспечить доступ к именам извне пространства имен. Решением является существование соответствующей **using**-директивы. Предыдущее объявление эквивалентно следующему:

```
namespace $$$ {
    int a;
    void f() {/* ... */}
    int g() {/* ... */}
}
using namespace $$$;
```

где **$$$** есть некоторое имя, уникальное для области видимости, в которой определено пространство имен. В частности, неименованные пространства имен в разных единицах трансляции различаются между собой. Как это и задумано, не существует способа именования члена безымянного пространства имен из другой единицы трансляции.

14.4.9. Заголовочные файлы языка C

Вот пример канонической программы для начинающих на языке C:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
```

Правила языка C++ для пространств имен разработаны с целью, чтобы можно было относительно легко модифицировать программы, написанные ранее (например на C) без использования пространств имен, в более структурированные варианты с применением пространств имен. По сути, программа калькулятора (§10.2) служит здесь примером.

Одним из способов внесения средств ввода/вывода языка C в пространства имен является помещение объявлений из заголовочного файла **stdio.h** в пространство имен **std**:

```
// cstdio:

    namespace std {
        int printf(const char* ...);
        // ...
}
```

Имея файл **<cstdio>**, можно обеспечить обратную совместимость добавлением **using**-директивы:

```
// stdio.h:

    #include<cstdio>
    using namespace std;
```

К сожалению такая **using**-директива делает все имена из пространства имен **std** доступными в глобальном окружении. Например:

```
#include<vector>    // тщательно избегаем загрязнения глобального пространства имен
vector v1;          // ошибка: нет "vector" в глобальной области видимости
#include<stdio.h>   // содержит "using namespace std;"
vector v2;          // упс: теперь работает
```

В результате стандарт требует, чтобы **<stdio.h>** помещал только имена из **<cstdio>** в глобальную область видимости. Этого можно добиться, предоставляя **using**-объявление для каждого объявления из **<cstdio>**:

```
// stdio.h:

    #include<cstdio>
    using std::printf;
    // ...
```

Еще одним преимуществом использования **using**-объявлений состоит в том, что это не дает пользователю определить (намеренно или нет) нестандартную функцию **printf()**

в глобальной области видимости. Большинство имен из пространств имен, отличных от **std**, можно четко отразить явной квалификацией и **using**-объявлениями.

Взаимоотношение между пространствами имен и правилами компоновки (линковки) описывается в §15.2.5.

14.5. Советы

[1] Применяйте пространства имен для отражения логической структуры программы; §14.3.1.

[2] Помещайте любое нелокальное имя (кроме **main()**) в некоторое пространство имен; §14.3.1.

[3] Так проектируйте ваши пространства имен, чтобы ненароком не получить доступ к несвязанным с ними пространствам имен; §14.3.3.

[4] Избегайте слишком коротких имен у пространств имен; §14.4.2.

[5] При необходимости используйте синонимы для сокращения длинных имен у пространств имен; §14.4.2.

[6] Не налагайте на пользователей слишком большой нагрузки в плане синтаксических конструкций доступа к вашим пространствам имен; §14.2.2, §14.2.3.

[7] Используйте разные пространства имен для интерфейсов и реализации; §14.3.3.

[8] Используйте нотацию **ПространствоИмен::член** в определении членов пространства имен; §14.4.

[9] Используйте **inline**-пространства имен для поддержки версий; §14.4.6.

[10] Используйте **using**-директивы с транзитными целями, для фундаментальных библиотек (вроде **std**) или внутри локального контекста; §14.4.9.

[11] Не размещайте **using**-директивы внутри заголовочных файлов; §14.2.3.

15

Исходные файлы и программы

Форма должна следовать за функцией.
— Корбюзье

- Раздельная компиляция
- Компоновка

  Имена, локальные в файлах; Заголовочные файлы; Правило одного определения; Заголовочные файлы стандартной библиотеки; Связывание с отличным от C++ кодом; Компоновка и указатели на функции
- Применение заголовочных файлов

  Единственный заголовочный файл; Организация программы со множеством заголовочных файлов; «Стражи включения»
- Программы

  Инициализация нелокальных переменных; Инициализация и параллелизм; Завершение программы
- Советы

15.1. Раздельная компиляция

Любая реалистичная программа состоит из множества логически различающихся компонентов (например, пространств имен; Глава 14). Для лучшей организации работы с проектом можно представить программу в виде набора исходных файлов, каждый из которых содержит один или несколько логических компонентов. Нашей задачей является разработка физической структуры (набора файлов) для программы, которая представляла бы логические компоненты в согласованной, понятной и гибкой манере. В частности, мы нацеливаемся на четкое разделение интерфейсов (объявлений функций) и реализации (определений функций). Файлы являются традиционными единицами хранения (в рамках файловых систем) и традиционной единицей компиляции. Существуют системы, которые не хранят, не компилируют и не представляют C++-программы в виде набора файлов. Но в нашей книге мы будем придерживаться только варианта с традиционным использованием файлов.

Располагать целую программу в единственном файле в принципе возможно. Например, код для стандартной библиотеки и операционной системы в типичном случае не представляется набором исходных файлов пользовательского уровня. Для реалистичных программ даже пользовательский код не стоит располагать в единственном файле, ибо это непрактично и неудобно. Разделяя исходный код на файлы, мы поддчеркиваем ее логическую структуру, помогаем сторонним программистам разбираться в коде (например, в случае поддержки), да и для компилятора это часто полезно, ибо он может усилить пре-

имущества от такого логического разделения. Когда единицей компиляции является файл, весть этот файл нужно перекомпилировать в случае изменений (даже минимальных) в его содержимом или в чем-то, от чего он зависит. Для программ умеренного и большого размеров потеря времени на перекомпиляцию больших файлов может быть существенной. Это время можно заметно уменьшить, разбивая программу на файлы подходящего размера.

Итак, программист предоставляет файл с исходным кодом компилятору. Тот его препроцессирует; то есть, выполняется макропроцессирование (§12.6) и внесение содержимого заголовочных файлов посредством директив **#include** (§2.4.1, §15.2.2). результатом препроцессирования и станет *единица трансляции*. Это то, над чем будет работать компилятор и это то, что относится к правилам языка C++. В данной книге я разграничиваю исходный файл и единицу трансляции только там, где важно подчернуть разницу между тем, что видит программист и что видит компилятор.

Для поддержки раздельной компиляции программист должен предоставить объявления, специфицирующие информацию о типах, нужную для анализа единицы трансляции в изоляции от остальной части программы. Все объявления в программе, состоящей из многих раздельно компилируемых частей, должны строго согласовываться между собой точно так же, как это должно быть в случае программы, состоящей из единственного исходного файла. Вам в этом вопросе могут помочь многие автоматизирующие средства. В частности, компоновщик может распознать много видов несогласованностей. *Компоновщик* — это программа, которая соединяет воедино раздельно скомпилированные части. Компоновка выполняется до того, как окончательная программа может быть запущена на выполнение. Но в качестве альтернативы возможна динамическая компоновка, когда недостающие части присоединяются к уже запущенной программе.

Организация программы в виде набора раздельно компилируемых файлов должна подчиняться логической структуре программы. При этом нужно придерживаться тех же правил, что и при разделении программы на пространства имен. Правда, соответствия один-в-один может и не быть. Например, бывает полезным для хранения функций из одного пространства имен использовать несколько исходных файлов, а бывает и что несколько определений пространств имен хранятся в единственном исходной файле. Можно также разбросать определение пространства имен по нескольким файлам (§14.3.3).

В настоящей главе мы сначала рассмотрим некоторые технические подробности компоновки, а затем перейдем к разбиению программы настольного калькулятора (§10.2, §14.3.1) на файлы.

15.2. Компоновка

Имена функций, классов, шаблонов, переменных, пространств имен, перечислений и энумераторов должны быть согласованы поверх всех единиц трансляции во всех случаях, кроме как если они явным образом специфицированы в качестве локальных.

Это является заботой программиста, чтобы все объявления во всех единицах трансляции согласовывались между собой. К примеру рассмотрим два файла:

```
// file1.cpp:
    int x = 1;
    int f() {/* что-то делаем */}

// file2.cpp:
    extern int x;
    int f();
    void g() {x = f();}
```

Здесь **x** и **f()**, использующиеся в **g()** в файле **file2.cpp**, определены в **file1.cpp**. Ключевое слово **extern** указывает на то, что объявление **x** в файле **file2.cpp** — это именно объявление, а не определение (§6.3). Если бы **x** инициализировалось, то **extern** было бы проигнорировано, поскольку объявление с инициализацией есть определение. Любой объект должен быть определен в программе ровно один раз. Объявлять его можно многократно, но во всех случаях типы должны совпадать полностью. Пример:

```
// file1.cpp:
    int x = 1;
    int b = 1;
    extern int c;

// file2.cpp:
    int x;      // означает "int x = 0;"
    extern double b;
    extern int c;
```

Здесь имеются три ошибки: переменная **x** определена дважды, **b** объявлена дважды с разными типами, а **c** объявлена дважды, но не определена. Такого рода ошибки (ошибки компоновки) не могут обнаруживаться компилятором, который имеет дело с одним исходным файлом за один вызов. Многие из рассмотренных ошибок обнаруживаются компоновщиком. Например, многие из известных мне компоновщиков обнаруживают двойное определение переменной **x**. Однако, несогласованность в объявлениях **b** не обнаруживается во многих реализациях, а отсутствие определения для **c** обнаруживается лишь в случае попытки использования переменной **c**.

Отметим, что переменная, объявленная в глобальном пространстве или в пространстве имен без явной инициализации, на самом деле инициализщируется по умолчанию (§6.3.5.1). Этого не происходит в случае нестатических локальных переменных или объектов, созданных в свободной памяти (§11.2).

Вне рамок тела класса любая сущность должна быть определена до того, как она будет использоваться (§6.3.4). Например:

```
// file1.cpp:
    int g() {return f()+7;}     // ошибка: f() не объявлена (пока)
    int f() {return x;}         // ошибка: x не объявлена (пока)
    int x;
```

Про имена, которые могут использоваться в единицах трансляции, отличных от тех, в которых они определяются, говорят, что они имеют *внешнее связывание* (*external linkage*). Все имена из предыдущих примеров имеют внешнее связывание. В противном случае, когда на имя можно ссылаться исключительно в той единице трансляции, в которой оно определено, такое имя называют имеющим *внутреннее связывание*. Например:

```
static int x1 = 1;          // внутренняя компоновка: не доступно из иных единиц трансляции
const char x2 = 'a';        // внутренняя компоновка: не доступно из иных единиц трансляции
```

Если ключевое слово **static** используется в глобальной области видимости или в пространстве имен (§14.2.1), то оно означает «не может быть доступным из других исходных файлов». Если требуется, чтобы **x1** могла быть доступна из других исходных файлов, нужно убрать ключевое слово **static**. По умолчанию, ключевое слово **const** подразумевает внутреннее связывание, так что если вы хотите, чтобы **x2** имело внешнее связывание, то нужно предварить его определение ключевым словом **extern**:

```
int x1 = 1;                 // внешняя компоновка: доступно из иных единиц трансляции
extern const char x2 = 'a'; // внешняя компоновка: доступно из иных единиц трансляции
```

Про имена, которые компоновщик «не видит», говорят как о не имеющих связывание.

inline-функция (§12.1.3, §16.2.8) должна определяться идентично во всех единицах трансляции, в которых она используется (§15.2.3). Поэтому следующий пример — это не просто плохой вкус; это не допускается:

```
// file1.cpp:
    inline int f(int i) {return i;}

// file2.cpp:
    inline int f(int i) {return i+1;}
```

К сожалению, подобную ошибку трудно перехватить в реализациях и следующая — хотя с логической точки зрения и корректная — комбинация внешней компоновки и встраивания запрещена, что упрощает жизнь разработчикам компилятора:

```
// file1.cpp:
    extern inline int g(int i);
    int h(int i) {return g(i);}  // ошибка: g() не определена в данной единице трансляции

// file2.cpp:
    extern inline int g(int i) {return i+1;}
    // ...
```

Согласованность определений **inline**-функций можно поддерживать с помощью заголовочных файлов (§15.2.2). Например:

```
// h.h:
    inline int next(int i) {return i+1;}

// file1.cpp:
    #include "h.h"
    int h(int i) {return next(i);}     // прекрасно

// file2.cpp:
    #include "h.h"
    // ...
```

По умолчанию, **const**-объекты (§7.5), **constexpr**-объекты (§10.4), синонимы типов (§6.5) и все, что объявлено со **static** (§6.3.4) в области видимости пространства имен, обладает внутренним связыванием. Следовательно, следующий пример законен:

```
// file1.cpp:
    using T = int;
    const int x = 7;
    constexpr T c2 = x+1;

// file2.cpp:
    using T = double;
    const int x = 8;
    constexpr T c2 = x+9;
```

Ради согласованности помещайте синонимы, объявления с **const**, **constexpr** и **inline** в заголовочные файлы (§15.2.2).

Объектам с **const** можно явно назначить внешнее связывание явным образом:

```
// file1.cpp:
    extern const int a = 77;

// file2.cpp:
    extern const int a;
```

```
void g()
{
    cout << a << '\n';
}
```

Здесь **g()** выведет число **77**.

Техника управления определениями шаблонов рассматривается в §23.7.

15.2.1. Имена, локальные в файлах

В общем случае следует избегать глобальных имен, порождающих проблемы в процессе поддержки программ. В частности, трудно выяснить, где именно они используются, а также они служат источником для гонки за данными в случае многопоточных программ (§41.2.4).

Немного помогает размещение имен в пространстве имен, но такие переменные все равно подвержены гонке за данными.

Если уж вам нужно использовать глобальную переменную, то ограничьте ее использованием единственным исходным файлом. Такое ограничение может быть реализовано двумя способами:

1. Поместить объявление в безымянное пространство имен.
2. Объявить сущность с ключевым словом **static**.

Безымянные пространства имен (§14.4.8) можно использовать, чтобы сделать имена локальными по отношению к единице трансляции. Результат почти аналогичен внутреннему связыванию. Например:

```
// file 1.cpp:
    namespace {
        class X {/* ... */};
        void f();
        int i;
        // ...
    }

// file2.cpp:
    class X {/* ... */};
    void f();
    int i;
    // ...
```

Функция **f()** в файле **file1.cpp**, это не та же самая функция, что и **f()** в файле **file2.cpp**. Располагать именем, локальным в единице трансляции, и использовать такое же имя в иных частях программы с внешним связыванием — это значит пытаться накликать беду.

Ключевое слово **static** означает «использовать внутреннее связывание» (§44.2.3). Это «посылка» нам из самых ранних дней языка C.

15.2.2. Заголовочные файлы

Типы во всех объявлениях одних и тех же объектов, функций, классов и т.д. должны быть согласованными. Следовательно, должен быть согласован исходный код, предоставляемый компилятору, который затем компонуется в исполняемую программу. Пусть и несовершенным, но практически эффективным способом достижения согласованности объявлений в разных единицах трансляции, является размещение объявлений в *заголовочных файлах*, включаемых в текст исходных файлов директивами **#include**.

Механизм директив **#include** обеспечивает текстовую сборку программных фрагментов в единое целое, предоставляемое компилятору для его работы. Рассмотрим следующее:

```
#include "to_be_included"
```

Директива **#include** заменяет строчку, в которой она находится, на содержимое файла **to_be_included**. Содержимым последнего должен быть текст на языке C++, ибо именно с этим и будет работать компилятор с этого языка.

Для включения стандартных заголовочных файлов вокруг имени включаемого файла ставят угловые кавычки < и > вместо двойных кавычек. Пример:

```
#include <iostream>      // из стандартного include-каталога
#include "myheader.h"    // из текущего каталога
```

К сожалению, нечаянные пробельные символы в таких записях не допускаются, ибо они принимаются во внимание внутри угловых скобок:

```
#include < iostream >    // не найдем <iostream>
```

При изменении включаемых (заголовочных) файлов приходится заново компилировать исходный файл, что часто порождает много лишней работы, ибо в заголовочных файлах содержится много интерфейсной информации, не используемой в конкретном контексте. Многие современные реализации C++ содержат те или иные формы решения этой проблемы в виде предкомпиляции заголовочных файлов.

Грубо говоря, заголовочные файлы могут содержать:

Пространства имен	`namespace N {/* ... */}`
inline пространства имен	`inline namespace N {/* ... */}`
Определения типов	`struct Point {int x, y;};`
Объявления шаблонов	`template<class T> class Z;`
Определения шаблонов	`template<class T> class V {/* ... */};`
Объявления функций	`extern int strlen(const char*);`
Определения **inline** ф-ий	`inline char get(char* p) {/* ... */}`
Определения **constexpr** ф-ий	`constexpr int fac(int n) {return (n<2)? 1: fac(n-1);}`
Объявления данных	`extern int a;`
Определения **const**	`const float pi = 3.141593;`
Определения **constexpr**	`constexpr float pi2 = pi*pi;`
Перечисления	`enum class Light {red, yellow, green};`
Объявления имен	`class Matrix;`
Синонимы типов	`using value_type = long;`
Статические утверждения	`static_assert(4<=sizeof(int),"small ints");`
Директивы **#include**	`#include<algorithm>`
Макроопределения	`#define VERSION12.03`
Директивы условной компиляции	`#ifdef __cplusplus`
Комментарии	`/* check for end of file */`

Приведенные приблизительные правила не являются правилами языка C++. Это просто практические рекомендации, как с помощью механизма **#include** лучше выразить физическую структуру программы.

Теперь о противоположном: что никогда не нужно включать в заголовочные файлы:

Определения обычных ф-ий	`char get(char* p) {return *p++;}`
Определения данных	`int a;`
Определения агрегатов	`short tbl[] = {1, 2, 3};`
Безымянные пространства имен	`namespace {/* ... */}`
Директивы **using**	`using namespace Foo;`

Заголовочные файлы с подобным содержимым вызовут ошибку компиляции или (в случае **using**-директив) приведут к недоразумениям. Заголовочные файлы традиционно имеют расширение.**h**, а файлы с определениями функций и данных имеют расширение. **cpp**. Но на практике встречаются и расширения **.c**, **.C**, **.cxx**, **.cc**, **.hh** и **hpp**, соответственно. Ознакомтесь на этот счет с руководством по вашему компилятору.

Причина, по которой определения простых констант включаются в заголовочные файлы, а определения агрегатов не включаются, состоит в том, что реализациям трудно избежать реплицирования агрегатов, представленных в разных единицах трансляции. Кроме того, простые случаи чаще встречаются, так что они важнее в плане генерации качественного кода.

Отметим, что важно не «переумничать», используя **#include**. Мои рекомендации таковы:

- Не включайте директивами **#include** обычные файлы, содержащие определения переменных и не-**inline**-функций.
- Включайте только полные (завершенные) объявления и определения.
- Располагайте директивы **#include** только в глобальном контексте, в блоках спецификации компоновки, а также в определениях пространств имен при конвертации старого кода (§15.2.4).
- Располагайте все директивы **#include** компактно перед остальным кодом ради минимизации нежелательных зависимостей.
- Избегайте слишком мудреных макроопределений.
- Минимизируйте использование в заголовочных файлах имен, нелокальных по отношению к этим файлам.

15.2.3. Правило одного определения

Любой класс, перечисление, шаблон и т.д. должен иметь ровно одно определение в рамках одной программы. С практической точки зрения это означает, что должно иметься единственное, например, определение класса, находящееся в каком-то одном файле. К сожалению, все не так просто. Например, определение класса может составляться посредством макрорасширения, так что определение класса может быть текстуально рассредоточено по двум исходным файлам посредством директив **#include** (§15.2.2). И что еще хуже, файл не является понятием языка C++; существуют реализации, в которых программы не хранятся в исходных файлах.

Как следствие, правило одного определения звучит в стандарте в несколько более сложной манере: два определения класса, шаблона или inline-функции принимаются как экземпляры одного уникального определения тогда и только тогда, если

1. Они встречаются в разных единицах трансляции.
2. Они идентичны лексема-к-лексеме, и
3. Смысл этих лексем одинаковый в обоих единицах трансляции.

Например:

```
// file1.cpp:
    struct S {int a; char b;};
    void f(S*);

// file2.cpp:
    struct S {int a; char b;};
    void f(S* p) {/* ... */}
```

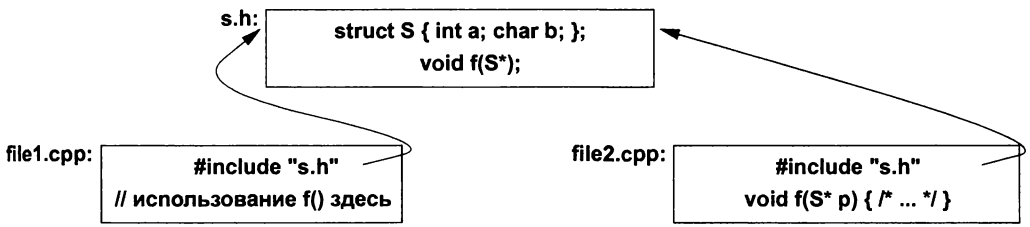
Правило одного определения признает, что это пример корректный и что **S** ссылается на один и тот же класс в обоих исходных файлах. Тем не менее, нет ничего хорошего, чтобы дважды писать определение в показанной манере. Кто-нибудь, кто поддерживает **file2.cpp**, может предположить, что определение **S** в этом файле — это единственное определение **S** во всей программе, и он свободен изменять его как угодно. В результате, будет получена трудноуловимая ошибка. Правило одного определения помогает включать определение класса в разные единицы трансляции из единственного исходного файла. Например:

```
// s.h:
    struct S {int a; char b;};
    void f(S*);

// file1.cpp:
    #include "s.h"
    // использование f() здесь

// file2.cpp:
    #include "s.h"
    void f(S* p) {/* ... */}
```

Или графически:



Приведем три примера нарушения правила одного определения:

```
// file1.cpp:
    struct S1 {int a; char b;};

    struct S1 {int a; char b;};     // ошибка: двойное определение
```

Ошибка состоит в том, что **struct** не может определяться дважды в одной единице трансляции.

```
// file1.cpp:
    struct S2 {int a; char b;};

// file2.cpp:
    struct S2 {int a; char bb;}; // ошибка
```

Здесь ошибка состоит в том, что **S2** именует классы с отличными именами членов.

```
// file1.cpp:
    typedef int X;
    struct S3 {X a; char b;};
```

```
// file2.cpp:
    typedef char X;
    struct S3 {X a; char b;};      // ошибка
```

Здесь два определения для **S3** идентичны лексема-в-лексему, но пример все равно ошибочен, поскольку смысл имени **X** различается в этих двух файлах.

Проверка на несогласованность определений классов в разных единицах трансляции находится вне возможностей большинства реализаций C++. Поэтому объявления, нарушающие правило одного определения, служат источником коварных ошибок. К сожалению, помещение определения в заголовочный файл и его включение в исходные файлы при помощи директив **#include**, не спасает от последней из только что проиллюстрированных ошибок. Локальные синонимы типов и макросы могут изменять смысл объявлений из включаемых заголовочных файлов:

```
// s.h:
    struct S {Point a; char b;};

// file1.cpp:
    #define Point int
    #include "s.h"
    // ...

// file2.cpp:
    class Point {/* ... */};
    #include "s.h"
    // ...
```

Наилучшей защитой от этой формы «хакерства» будет формирование заголовочных файлов в максимально самосогласованной и самодостаточной манере. Например, если бы класс **Point** был объявлен в файле **s.h**, то ошибка была бы обнаружена.

Определения шаблонов могут размещаться в заголовочных файлах, включаемых директивой **#include** в разные единицы трансляции, до тех пор, пока удается придерживаться правила одного определения. Это относится даже к определениям шаблонов функций и к шаблонам классов, содержащих определения функций-членов.

15.2.4. Заголовочные файлы стандартной библиотеки

Средства стандартной библиотеки представлены набором стандартных заголовочных файлов (§4.1.2, §30.2). У имен стандартных заголовочных файлов нет никаких суффиксов. Отсутствие суффикса **.h** не говорит ничего о том, как хранятся эти файлы. Заголовочные файлы вроде **<map>** обычно хранятся в виде текстовых файлов в некотором стандартном каталоге. Однако, реализации в принципе вольны предоставлять стандартную библиотеку в том виде, в каком им будет удобнее. Например, реализация может знать все о стандартной математической библиотеке (§40.3) и трактовать **#include<cmath>** просто как своего рода «рубильник», открывающий программе доступ к стандартным математическим функциям, не читая при этом никакого специального файла.

Для каждого стандартного для языка C заголовочного файла **<X.h>** имеется соответствующий файл **<cX>**, отвечающий стандарту C++. Например, **<cstdio>** предоставляет те же средства, что и **<stdio.h>**. Типичный **stdio.h** выглядит примерно таким образом:

```
#ifdef __cplusplus  // только для C++ компиляторов (§ 15.2.5)
namespace std {     // стандарт. библиотека определена в пространстве имен std (§ 4.1.2)
extern "C" {        // stdio-функции имеют C-компоновку (§ 15.2.5)
```

```
#endif
    /* ... */
    int printf(const char*, ...);
    /* ... */
#ifdef __cplusplus
}
}
// ...
using std::printf;    // делаем printf доступной в глобальном пространстве имен
// ...
#endif
```

Таким образом, реальные объявления одни и те же, а вопросы связывания и пространств имен позволяют разделять этот заголовочный файл для языков C и C++. Макрос **__cplusplus** определен для компилятора C++ (§12.6.2) и применяется для того, чтобы отделять C++-код от того, что предназначено для компилятора языка C.

15.2.5. Связывание с отличным от C++ кодом

Нередки случаи, когда C++-программа имеет части, написанные на других языках программирования (например C или Fortran). И наоборот, фрагменты кода на C++ могут использоваться в составе программ, написанных преимущественно на других языках (Python или Matlab). Кооперация между программными фрагментами, написанными на разных языках, не является простой задачей (что верно и для фрагментов на C++, скомпилированных с помощью разных компиляторов). К примеру, разные языки и разные компиляторы с одного и того же языка могут по разному использовать регистры процессора для хранения аргументов, могут иметь разную раскладку аргументов в стеке, разную картину памяти для встроенных типов вроде строк и целых, могут иметь разный формат имен, передаваемых компоновщику и т.д. Помощь в этом вопросе может предоставить ключевое слово **extern**, означающее, что стандартные функции вроде **strcpy()** должны связываться согласно (зависящим от системы) правилам компоновки языка C:

```
extern "C" char* strcpy(char*, const char*);
```

Это объявление отличается от «простого» объявления

```
extern char* strcpy(char*, const char*);
```

только форматом связывания для вызовов функции **strcpy()**.

Директива **extern "C"** особенно полезна из-за близкого родства языков C и C++. Заметим, что **C** в **extern "C"** означает именно специфическое правило связывания, а не собственно язык. Часто **extern "C"** используется для связывания с процедурами языков Fortran или Assembler, которые по счастливой случайности подчиняются тем же правилам связывания, что работают для языка C.

Директива **extern "C"** специфицирует лишь правило связывания и никак не влияет на семантику вызова функции. В частности, функция, объявленная с **extern "C"** по-прежнему подчиняется правилам проверки типов и правилам преобразования аргументов языка C++, а не более слабым правилам языка C. Например:

```
extern "C" int f();

int g()
{
    return f(1);    // ошибка: аргумент не ожидается
}
```

Добавление **extern "C"** ко многим объявлением может оказаться весьма утомительным. Поэтому имеется механизм коллективной спецификации правил связывания для группы объявлений. Пример:

```
extern "C" {
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...
}
```

Эту конструкция, называемую *блоком компоновки*, можно использовать для полного содержимого заголовочных файлов языка С с целью сделать их приемлемыми для программ на С++. Например:

```
extern "C" {
#include <string.h>
}
```

Этот прием часто применяется для того, чтобы получить заголовочный файл для языка С++ из заголовочного файла языка С. Альтернативно, директивы условной компиляции (§12.6.1) используются для формирования заголовочных файлов, общих для С и С++:

```
#ifdef __cplusplus
extern "C" {
#endif
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...
#ifdef __cplusplus
}
#endif
```

Предопределенный макрос **__cplusplus** (§12.6.2) используется для гарантии того, что конструкции С++ будут вычищены, если файл используется в качестве заголовочного файла языка С.

Любое объявление может стоять в блоке компоновки:

```
extern "C" {        // любое объявление здесь, например:
    int g1;         // определение
    extern int g2; // объявление, не определение
}
```

При этом область видимости и класс хранения (§6.3.4, §6.4.2) для переменной не изменяются, так что **g1** по-прежнему глобальная и она определена, а не просто объявлена. Для того, чтобы лишь объявить переменную, вы должны добавить ключевое слово **extern** непосредственно в объявлении. Пример:

```
extern "C" int g3;        // объявление, не определение
extern "C" {int g4;}      // определение
```

На первый взгляд это выглядит странно. Однако, это лишь простое следствие сохранения смысла при добавлении **"C"** в **extern**-объявлениях и сохранении смысла файла, когда его содержимое оборачивается блоком компоновки (связывания).

Имя с С-компоновкой можно объявлять в пространстве имен. Пространство имен влияет на то, как можно ссылаться на имя в C++-программе, а не на то, как видит его компоновщик. Функция **printf()** из пространства имен **std** служит хорошим примером:

```
#include<cstdio>

void f()
{
    std::printf("Hello, ");      // OK
    printf("world!\n");          // ошибка: нет глобальной printf()
}
```

Даже в случае вызова **std::printf()** это по-прежнему старая С-функция **printf()** (§43.3).

Отметим, что это позволяет нам включать библиотеки с С-компоновкой в пространства имен по нашему выбору, а не засорять глобальное пространство имен.

15.2.6. Компоновка и указатели на функции

При смешении в одной программе фрагментов кода на С и C++ иногда требуется передать указатель на функцию, определенную в одном языке, функции, определенной в другом языке. Если реализации языков согласуются по правилам компоновки и механизмам вызова функций, то такая задача тривиальна. К сожалению, нельзя предполагать такую ситуацию в безусловном порядке, поэтому следует проявлять осторожность, чтобы функция вызывалась так, как ожидается.

Если формат компоновки отражен в объявлениях, то он применяется к типам функций, именам функций и именам переменных, вводимых этими объявлениями. Это порождает некоторые странные и существенные комбинации разных видов компоновки. Например:

```
typedef int (*FT)(const void*, const void*);                    // FT имеет C++ компоновку

extern "C" {
    typedef int (*CFT)(const void*, const void*);               // CFT имеет C компоновку
    void qsort(void* p, size_t n, size_t sz, CFT cmp);          // cmp имеет C компоновку
}

void isort(void* p, size_t n, size_t sz, FT cmp);               // cmp имеет C++ компоновку
void xsort(void* p, size_t n, size_t sz, CFT cmp);              // cmp имеет C компоновку
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp);    // cmp имеет C++ компоновку

int compare(const void*, const void*);                          // compare() имеет C++ компоновку
extern "C" int ccmp(const void*, const void*);                  // ccmp() имеет C компоновку

void f(char* v, int sz)
{
    qsort(v, sz,1,&compare); // ошибка
    qsort(v, sz,1,&ccmp);    // OK

    isort(v, sz,1,&compare); // OK
    isort(v, sz,1,&ccmp);    // ошибка
}
```

В реализациях С и C++, в которых применяются одинаковые правила компоновки, объявления, помеченные здесь у нас как *error*, могут быть приняты как расширения языка. Тем не менее, даже для совместимых реализаций С и C++ **std::function** (§33.5.3) или лямбды с любой формой захвата (§11.4.3) не сумеют преодолеть языкового барьера.

15.3. Применение заголовочных файлов

Для иллюстрации использования заголовочных файлов я покажу несколько альтернативных способов отражения физической структуры программы-калькулятора (§10.2, §14.3.1).

15.3.1. Единственный заголовочный файл

Простейшим решением проблемы секционирования программы в виде набора файлов является решение, при котором определения помещаются в некоторое количество **cpp**-файлов, а объявления типов, функций, классов и т.д. помещаются в единственный заголовочный файл, который остальные файлы включают в себе директивой **#include**. Такую структуру файлов я предпочел бы для простой программы, предназначенной для собственного использования; если потом пришлось бы эту программу расширять и усиливать, то можно было бы тогда и реорганизовать ее файловую структуру.

Для программы-калькулятора можно было бы применить пять **cpp**-файлов — **lexer.cpp**, **parser.cpp**, **table.cpp**, **error.cpp** и **main.cpp**, которые и содержали бы определения функций и данных. А заголовочный файл **dc.h** содержал бы объявления всех имен, которые используются более чем в одном **cpp**-файле:

```
// dc.h:

#include <map>
#include<string>
#include<iostream>

namespace Parser {
    double expr(bool);
    double term(bool);
    double prim(bool);
}

namespace Lexer {
    enum class Kind: char {
        name, number, end,
        plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
    };

    struct Token {
        Kind kind;
        string string_value;
        double number_value;
    };

    class Token_stream {
    public:
        Token(istream& s):ip{&s}, owns{false}, ct{Kind::end} {}
        Token(istream* p):ip{p}, owns{true}, ct{Kind::end} {}

        ~Token() {close();}

        Token get();                    // прочитать и вернуть следующую лексему
        Token& current();               // последняя прочитанная лексема
        void set_input(istream& s) {close(); ip = &s; owns=false;}
        void set_input(istream* p) {close(); ip = p; owns = true;}
    private:
        void close() {if (owns) delete ip;}
```

```
            istream* ip;                    // указатель на входной поток
            bool owns;                      // владеет ли Token_stream потоком istream?
            Token ct {Kind::end};           // current_token
        };

        extern Token_stream ts;
    }
    namespace Table {
        extern map<string, double> table;
    }

    namespace Error {
        extern int no_of_errors;
        double error(const string& s);
    }

    namespace Driver {
        void calculate();
    }
```

Ключевое слово **extern** применяется в отношении каждого объявления переменной, чтобы гарантировать отсутствие множественных определений при включении файла **dc.h** в разные **cpp**-файлы.

Я добавил стандартные заголовочные файлы, которые требуются для объявлений в **dc.h**, но я не добавлял объявлений (вроде **using**-объявлений), которые служат для удобства в индивидуальных **cpp**-файлах.

Опуская реальный код, представим **lexer.cpp**:

```
// lexer.cpp:

#include "dc.h"
#include <cctype>
#include <iostream>     // избыточно: в dc.h

Lexer::Token_stream ts;

Lexer::Token Lexer::Token_stream::get() {/* ... */}
Lexer::Token& Lexer::Token_stream::current() {/* ... */}
```

Я применил явную квалификацию **Lexer::** для определений вместо заключения всех их в

```
namespace Lexer {/* ... */}
```

Так можно избежать нечаянного добавления членов в **Lexer**. С другой стороны, в случае необходимости добавления новых членов в **Lexer**, я мог бы повторно открыть пространство имен (§14.2.5).

Показанный вариант использования заголовочных файлов гарантирует, что каждое объявление попадет в некоторой точке в файл, содержащий определение. Например, при компиляции **lexer.cpp** компилятор встретится со следующим:

```
namespace Lexer {// из dc.h
    // ...
    class Token_stream {
    public:
        Token get();
        // ...
    };
}
```

```
// ...

Lexer::Token Lexer::Token_stream::get() {/* ... */}
```

Это гарантирует, что компилятор обнаружит любую несогласованность в типах, специфицированных для имени. Например, если бы **get()** объявлялась с возвратом **Token**, а определялась с возвратом **int**, то в процессе компиляции была бы выявлена ошибка несовпадения типов. Если отсутствует определение, то это обнаруживает компоновщик. А если отсутствует объявление, то некоторый.**cpp**-файл не будет скомпилирован.

Файл **parser.cpp** будет выглядеть следующим образом:

```
// parser.cpp:

#include "dc.h"

double Parser::prim(bool get) {/* ... */}
double Parser::term(bool get) {/* ... */}
double Parser::expr(bool get) {/* ... */}
```

А вот как выглядит файл **table.cpp**:

```
// table.cpp:

#include "dc.h"

std::map<std::string, double> Table::table;
```

Таблица символов реализована с помощью контейнера **map** стандартной библиотеки.

Теперь приведем содержимое файла **error.cpp**:

```
// error.cpp:

#include "dg.h"
// другие #include или объявления

int Error::no_of_errors;
double Error::error(const string& s) {/* ... */}
```

Наконец, вот содержимое файла **main.cpp**:

```
// main.cpp:

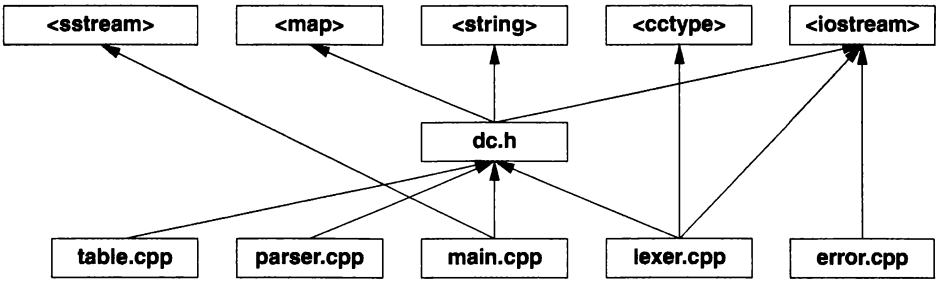
#include "dc.h"
#include <sstream>
#include <iostream>      // избыточно: в dc.h

void Driver::calculate() {/* ... */}

int main(int argc, char* argv[]) {/* ... */}
```

Чтобы быть главной (стартовой) функцией в программе, функция **main()** должна быть глобальной (§2.2.1, §15.4), так что никаких пространств имен здесь быть не должно.

Физическую структуру системы можно отобразить графически следующим образом:



Заголовочные файлы в верхней строке — это все средства стандартной библиотеки. Во многих случаях при анализе структуры программы их можно вообще не рассматривать.

Представленный стиль физического секционирования программы с единственным заголовочным файлом весьма полезен для небольших программ, части которых не будут использоваться порознь. Если используются пространства имен, то логическая струтура программы также отражается в заголовочном файле **dc.h**. В отсутствие пространств имен логическая структура программы плохо отображена, но в какой-то степени тут могут помочь комментарии.

В случае больших программ подход с единственным заголовочным файлом непрактичен. Любые изменения в этом файле требуют перекомпиляции всех остальных исходных файлов программы. Да и логическая струтура программы вряд ли наглядно отображается в таком случае, хотя применение пространств имен и строгое следование единым программным стилям исправляют положение.

15.3.2. Организация программы со множеством заголовочных файлов

Альтернативным способом организации программы служит подход, когда каждый логический модуль имеет свой заголовочный файл, в котором объявляются все средства модуля. В таком случае каждый.**cpp**-файл имеет свой заголовочный файл, для которого он предоставляет определения. Каждый.**cpp**-файл включает посредством директив **#include** свой заголовочный файл и другие заголовочные файлы, специфицирующие то, в чем.**cpp**-файл нуждается от других модулей. Такая физическая организация соответствует логической организации модуля. Интерфейс для пользователей помещается в соответствующий.**h**-файл, а интерфейс для реализаторов помещается в заголовочный файл с префиксом **_impl.h**; определения средств модуля находятся в.**cpp**-файлах. В таком варианте парсер будет размещаться в трех файлах. Интерфейс пользователя для парсера предоставляется файлом **parser.h**:

```
// parser.h:

namespace Parser {      // интерфейс для пользователей
    double expr(bool get);
}
```

Разделяемое окружение для функций **expr()**, **prim()** и **term()**, реализующих парсер, представлено в файле **parser_impl.h**:

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"
```

```
using Error::error;
using namespace Lexer;

namespace Parser {      // интерфейс для реализаторов
    double prim(bool get);
    double term(bool get);
    double expr(bool get);
}
```

Дифференциация интерфейсов для пользователя и для реализации была бы еще более ясной, если бы мы использовали пространство имен **Parser_impl** (§14.3.3).

Мы включаем заголовочный файл пользовательского интерфейса **parser.h**, чтобы у компилятора была возможность проверить согласованность объявлений (§15.3.1).

Функции, реализующие парсер, хранятся в файле **parser.cpp**, где также присутствуют директивы **#include**, включающие заголовочные файлы, в которых функции парсера нуждаются:

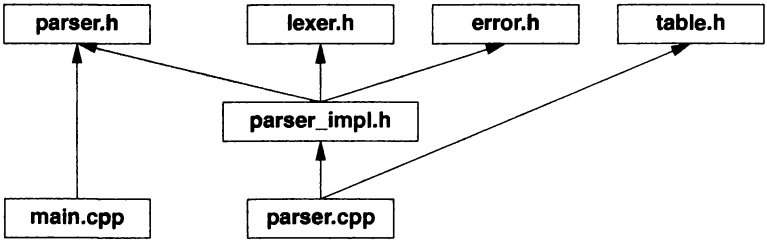
```
// parser.cpp:

#include "parser_impl.h"
#include "table.h"

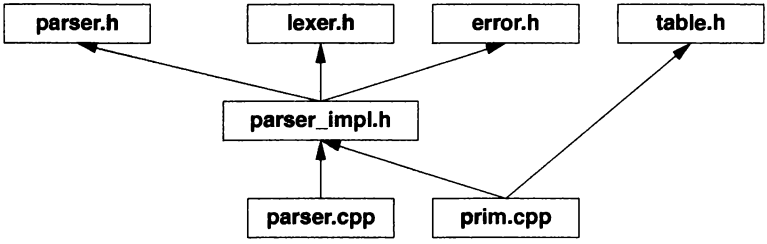
using Table::table;

double Parser::prim(bool get) {/* ... */}
double Parser::term(bool get) {/* ... */}
double Parser::expr(bool get) {/* ... */}
```

Графически система отображается следующим образом:



Как и предполагалось, это близко к логической структуре программы, описанной в §14.3.1. Для упрощения этой структуры можно было бы включить **table.h** в **parser_impl.h**, а не в **parser.cpp**. В то же время **table.h** не нужен для отражения разделяемого контекста функций парсера; он нужен лишь их реализации. Фактически, этот файл используется лишь одной функцией, **prim()**, так что если уж нам на самом деле нужно минимизировать зависимости, мы можем поместить **prim()** в свой отдельный.**cpp**-файл и включить **table.h** только в этот файл:



Столь подробная разработка структуры нужна лишь для больших модулей. В программах реалистичных рамеров часто включаются лишние файлы, если они вдруг требуются тем или иным функциям. Кроме того, часто бывает не один, а множество **_impl.h** файлов, поскольку разные подмножества функций модуля нуждаются в разных разделяемых контекстах.

Обратите внимание на то, что нотация **_impl.h** не является каким-либо стандартом или общеупотребительной практикой; это просто мой личный любимый способ именования.

Почему вообще стоит придавать значение этой схеме со множественными заголовочными файлами? Дело в том, что такая схема помогает масштабированию до уровня модулей, которые в сотни раз больше модулей из нашего примера, и до размеров программ, которые в сотни раз больше нашей учебной программы-калькулятора. Эта схема обеспечивает большую локализацию идей и понятий. При анализе большой программы программисту выгодно сконцентрироваться на относительно небольшом куске кода. Схема со множественными заголовочными файлами позволяет определить, от чего зависит код парсера, и игнорировать все остальное. А в случае единственного заголовочного файла приходится просматривать все объявления во всех модулях.

Большая локализация приводит к тому, что меньше информации требуется для компиляции модуля, что только ускоряет процесс. Причем эффект может быть огромным. Я лично наблюдал снижение времени компиляции в 1000 раз просто из-за более точной структурированности проекта, приведшей к более эффективному использованию заголовочных файлов.

15.3.2.1. Другие модули калькулятора

Остальные модули калькулятора можно организовать так же, как это было сделано с парсером. Правда, эти модули столь малы, что не требуют отдельных **_impl.h**-файлов.

Обработчик ошибок предоставляет свой интерфейс в файле **error.h**:

```
// error.h:

#include<string>

namespace Error {
    int Error::number_of_errors;
    double Error::error(const std::string&);
}
```

Реализация располагается в файле **error.cpp**:

```
// error.cpp:

#include "error.h"

int Error::number_of_errors;
double Error::error(const std::string&) {/* ... */}
```

Сборщик лексем предоставляет довольно большой и хаотичный интерфейс:

```
// lexer.h:

#include<string>
#include<iostream>

namespace Lexer {

    enum class Kind : char {/* ... */};
```

```
    class Token {/* ... */};
    class Token_stream {/* ... */};

    extern Token_stream is;
}
```

Кроме файла **lexer.h** реализация сборщика лексем зависит от **error.h** и от функций из **<cctype>** (классификация символов; §36.2):

```
// lexer.cpp:

#include "lexer.h"
#include "error.h"
#include <iostream>          // избыточно: в lexer.h
#include <cctype>

Lexer::Token_stream is;      // умолчательно "читать из cin"

Lexer::Token Lexer::Token_stream::get() {/* ... */};
Lexer::Token& Lexer::Token_stream::current() {/* ... */};
```

Как обычно, мы включаем интерфейс, предлагаемый модулем — в данном случае это **lexer.h** — в файл с реализацией модуля, чтобы позволить компилятору выполнить проверку согласованности.

Таблица символов по существу является самосогласованной, хотя стандартный файл **<map>** может привнести массу интересного материала ради реализации эффективного шаблона **map**:

```
// table.h:

#include <map>
#include <string>

namespace Table {
    extern std::map<std::string, double> table;
}
```

Из-за предположения, что любой заголовочный файл может быть включен в несколько **.cpp**-файлов, мы должны отделить объявление **table** от ее определения:

```
// table.cpp:

#include "table.h"

std::map<std::string, double> Table::table;
```

Основная (стартовая) программа погружена в файл **main.cpp**:

```
// main.cpp:

#include "parser.h"
#include "lexer.h"       // возможность установить ts
#include "error.h"
#include "table.h"       // предустановленные имена
#include <sstream>       // возможность передать аргументы main() в поток строк

namespace Driver {
    void calculate() {/* ... */}
}

int main(int argc, char* argv[]) {/* ... */}
```

В больших программах я бы разделил **Driver** и стартовую функцию **main()**. Это особо полезно, если предполагается, что код может использоваться и как библиотека. В последнем случае старт вычислений может производиться не из **main()**, а из других функций библиотеки.

15.3.2.2. Использование заголовочных файлов

Количество заголовочных файлов, применяемых в той или иной программе, зависит от множества факторов. Некоторые из этих факторов больше зависят от системного окружения на машине, нежели от собственно языка C++. Например, такой простейший фактор, как зависимость от среды разработки, в которой может, например, быть затруднен просмотр и работа со множественными файлами, и тогда применение многих заголовочных файлов станет неудобным.

Не будет лишним и следующее предупреждение. Работа с десятками собственных заголовочных файлов плюс неизбежные стандартные заголовочные файлы (которых тоже может оказаться не один десяток) вполне по силам программистам и компьютерам. Но если вы переусердствуете с разбиением на мельчайшие логические заголовочные файлы (содержащие по одному важному определению на весь файл), то вы легко можете потонуть в невообразимом месиве из многих сотен файлов даже для не слишком больших проектов. Я считаю это избыточным и ненужным.

Конечно, для больших проектов множественные заголовочные файлы неизбежны. В таких проектах нередки случаи сотен заголовочных файлов (не считая стандартных заголовочных файлов). Серьезная проблема возникает, когда их количество начинает исчисляться тысячами. Вот на таком уровне масштаба становятся полезными все те соображения, которые мы здесь рассматривали. Конечно, могут помочь анализаторы кода, вскрывающие зависимости, но они не могут помочь увеличить производительность компилятора и компоновщика в случае полной мешанины заголовочных файлов.

Выбор между единственным заголовочным файлом и множественными заголовочными файлами не является альтернативой. Это дополняющие друг друга стратегии, которые по мере эволюционирования проекта (в сторону разрастания) должны применяться совместно по отношению к разным частям программы. Важно помнить, что единственный интерфейс не может одинаково удовлетворить всех. Важно различать интерфейс для пользователей и для реализаторов. Даже пользователей желательно разделять на обычных пользователей и «экспертов». Для экспертов демонстрируются все возможности, в то время как для обычных пользователей предоставляется типовой интерфейс, который годится в большинстве случаев. Не следует думать, что понятие обычного пользователя унижает их в каком-то отношении. Я, например, во всех случаях, в которых не собираюсь вникать буквально во все тонкости и возможности, предпочитаю оставаться обычным пользователем (зачем мне усложнять себе жизнь без какой-либо очевидной причины).

15.3.3. «Стражи включения»

Идея в основе применения множественных заголовочных файлов заключается в том, чтобы представить каждый модуль в качестве самосогласованной и самодостаточной единицы программы. С высоты всей программы множество объявлений ради того, чтобы модуль был самодостаточным, является избыточным. В особо больших программах эта избыточность может приводить к ошибкам, возникающим из-за того, что заголовочные файлы, содержащие определения классов или определения **inline**-функций, включаются дважды в одни и те же единицы трансляции (§15.2.3).

У нас в связи с этим есть две возможности. Мы можем

1. Реорганизовать программу ради исключения избыточности, или
2. Найти способ разрешить проблему повторных включений заголовочных файлов.

Первый подход, который привел у нас к финальной версии программы-калькулятора, трудозатратный и непрактичный в случае программ реальных размеров.

Конечно, есть достоинства в приемах по уменьшению зависимостей в программе — это уменьшение времени компиляции и достижение большей логической ясности. Но все равно неплохо бы найти возможность решения проблемы повторных включений заголовочных файлов. Этот способ тогда нужно использовать систематически, поскольку нельзя заранее знать, где и куда будут включаться заголовочные файлы.

Традиционным решением проблемы является внедрение так называемых *стражей включения* в заголовочные файлы. Пример:

```
// error.h:

#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
    // ...
}

#endif // CALC_ERROR_H
```

Содержимое файла между **#ifndef** и **#endif** игнорируется компилятором, если определена константа **CALC_ERROR_H**. Таким образом, когда файл **error.h** первый раз просматривается компилятором, его содержимое читается и макроконстанта **CALC_ERROR_H** получает значение. Если компилятор встретится еще раз во время компиляции с заголовочным файлом **error.h**, его содержимое будет проигнорировано. Это, конечно, некий трюк, но он работает и все стандартные заголовочные файлы имеют в своем составе стражи включения.

Заголовочные файлы могут включаться в произвольных контекстах и нет никакой защиты от столкновения макроимен. Поэтому имеет смысл выбирать для них длинные и достаточно неуклюжие имена.

Как только люди привыкают использовать стражи включения в своих заголовочных файлах, они начинают их включать куда-ни-попадя. Это, конечно, нежелательно, даже для реализаций, располагающих теми или иными механизмами оптимальной обработки заголовочных файлов. Они могут сильно увеличить время компиляции, а также внести массу ненужного в непредназначенные для этого области видимости. Важно: заголовочные файлы следует включать строго по необходимости.

15.4. Программы

Программа есть коллекция раздельно скопилированных единиц, собранных вместе компоновщиком (линкером). Каждая функция, объект, тип и т.д., использованный в этой коллекции, должен иметь уникальное определение (§6.3, §15.2.3). В программе должна иметься ровно одна функция **main()** (§2.2.1). Все основные работы в программе начинаются с вызова функции **main()** и заканчиваются в момент выхода из этой функции. У функции **main()** тип возврата **int**. Следующие две версии функции **main()** поддерживаются всеми реализациями:

```
int main() {/* ... */}
int main(int argc, char* argv[]) {/* ... */}
```

В программе может использоваться лишь одна из этих версий. Дополнительно, конкретные реализации могут предоставлять и большее количество возможных версий функции. Параметры **argc**, **argv** используются для передачи аргументов из программного окружения (§10.2.7).

Возврат функции **main()** передается тому системному коду, который вызвал **main()**. Ненулевой возврат из **main()** символизирует неудачу.

Столь простая история усложняется в случае глобальных переменных (§15.4.1) или при генерации неперехваченных исключений (§13.5.2.5).

15.4.1. Инициализация нелокальных переменных

В принципе, любая переменная, определенная вне определений функций (глобально, в пространстве имен, как **static**-член класса) инициализируется то того, как будет вызвана функция **main()**. Такие нелокальные переменные в единицах трансляции инициализируются в порядке их определения. Если у этих переменных нет явного инициализатора, то они инициализируются умолчательными значениями их типа (§17.3.3). Для встроенных типов и перечислений умолчательным значением служит **0**. Например:

```
double x = 2; // нелокальные переменные
double y;
double sqx = sqrt(x+y);
```

Здесь **x** и **y** инициализируются до **sqx**, так что вызывается **sqrt(2)**.

Не существует заданного порядка инициализации глобальных переменных в разных единицах трансляции. Поэтому будет неправильным рассчитывать на какой-то специфический порядок. Кроме того, невозможно перехватить исключение, сгенерированное в процессе инициализации глобальной переменной (§13.5.2.5). В целом разумно минимизировать использование глобальных переменных, требующих нетривиальной инициализации.

Существует ряд приемов по навязыванию порядка инициализации глобальных переменных в разных единицах трансляции. Но все они непереносимы и не слишком эффективны. Кроме того, динамические библиотеки плохо уживаются с глобальными переменными, требующими нетривиальной инициализации и подверженными разным зависимостям.

Часто хорошей альтернативой глобальной переменной является функция, возвращающая ссылку. Например:

```
int& use_count()
{
    static int uc = 0;
    return uc;
}
```

Теперь вызов **use_count()** действует как глобальная переменная за исключением того факта, что здесь инициализация происходит при ее первом вызове (§7.7). Например:

```
void f()
{
    cout << ++use_count();  // прочитать и инкрементировать
    // ...
}
```

Как и в иных случаях определений с ключевым словом **static**, эта техника не является надежной в многопоточном варианте. Но инициализация статической локальной переменной надежна в случае многих потоков (§42.3.3). В этом случае проходит даже инициализация константным выражением (§10.4), так как выполняется во время компиляции и не

подвержена «гонке за данными» (§42.3.3). Однако, применение операции ++ может приводить к гонке за данными.

Инициализация нелокальных (статически распределяемых) переменных контролируется механизмом, который реализации используют для запуска C++-программ. Эти механизмы гарантируют корректную работу только в том случае, если вызывается функция **main()**.

Отметим, что переменные, инициализируемые константными выражениями (§10.4), не могут иметь зависимости от объектов из других единиц трансляции и не требуют инициализации во время выполнения.

15.4.2. Инициализация и параллелизм

Рассмотрим следующее:

```
int x = 3;
int y = sqrt(++x);
```

Каковы могут быть значения для **x** и **y**? Очевидный ответ это «**3** и **2**!». Почему? Инициализация статически аллоцированного объекта константным выражением выполняется на этапе компоновки, так что **x** становится равным **3**. Однако, инициализатор для **y** не является константным выражением, так что **y** не инициализируется до начала выполнения программы. Но порядок инициализации статически аллоцируемых объектов в одном и той же единице трансляции четко определен: они инициализируются в порядке их определения (§15.4.1). Таким образом, **y** принимает значение **2**.

Есть и проблема, заключающаяся в том, что если имеются несколько потоков выполнения (§5.3.1, §42.2), то каждый из них будет инициализировать на этапе выполнения программы. И нет никакой неявной блокировки во избежание гонки за данными. Тогда вызов **sqrt(x+y)** в одном потоке может произойти до или после того, как другой поток сподобится инкрементировать **x**. Поэтому значение для **y** может быть **sqrt(4)** или **sqrt(5)**.

Во избежание таких проблем нам нужно:

- Минимизировать применение статически аллоцируемых объектов и выполнять для них столь простую инициалиацию, насколько возможно.
- Избегать зависимостей от динамически инициализируемых объектов в других единицах трансляции (§15.4.1).

Кроме того, чтобы избежать гонки за данными при инициализации, можно попробовать использовать следующие приемы по-порядку:

1. Инициализация константными выражениями (отметим, что встроенные типы без явных инициализаторов инициализируются нулями и что стандартные контейнеры и тип **string** инициализируются пустым содержимым на этапе компоновки).
2. Инициализация выражениями, не имеющими побочных эффектов.
3. Инициализация в стандартной однопотоковой стартовой фазе вычислений.
4. Применение каких-либо форм взаимных блокировок (§5.3.4, §42.3).

15.4.3. Завершение программы

Программу можно терминировать (завершить исполнение) разными способами:

1. Возвратом из **main()**
2. Вызовом **exit()**
3. Вызовом **abort()**
4. Генерацией неперехватываемого исключения
5. Нарушением обещания **noexcept**
6. Вызовом **quick_exit()**

Еще имеются многочисленные зависящие от реализации способы спровоцировать крах программы (вроде деления переменной типа **double** на нуль).

Если функция завершает работу вызовом стандартной функции **exit()**, то будут вызваны деструкторы для ранее сконструированных статических объектов (§15.4.1, §16.2.12). А если программа завершается посредством вызова стандартной функции **abort()**, то этого не происходит. Ясно, что из сказанного следует, что **exit()** не выполняет немедленного завершения программы. А вызов **exit()** из деструктора может спровоцировать бесконечную рекурсию. У функции **exit()** следующий тип:

```
void exit(int);
```

Как и возврат функции **main()** (§2.2.1) аргумент функции **exit()** возвращается системе в качестве результата программы. Нуль означает успешное завершение.

При вызове **exit()** деструкторы локальных переменных вызвавшей функции (и тех функций, которые вызвали эту функцию) не сработают. Генерация исключения и его перехват обеспечат надлежащее уничтожение локальных объектов (§13.5.1). Дополнительно, вызов **exit()** не дает возможности вызвавшей функции попробовать справиться с проблемой. Генерация исключения позволяет в его обработчике решить вопрос о том, что делать дальше. К примеру, в функции **main()** можно перехватить любое исключение (§13.5.2.2).

Стандартная функция языка C (и C++ тоже) **atexit()** имеет возможность предоставить код, который выполнится при терминировании программы. Например;

```
void my_cleanup();

void somewhere()
{
    if (atexit(&my_cleanup)==0) {
        // my_cleanup вызывается при обычном терминировании
    }
    else {
        // упс: слишком много функций atexit
    }
}
```

Это сильно напоминает автоматическую активацию деструкторов для глобальных переменных при терминировании программы (§15.4.1, §16.2.12). Функция, которая служит аргументом для **atexit()**, сама не может принимать аргументы и не может иметь возврата, что ограничивает для реализации число возможных вариантов таких функций. Это делает функцию **atexit()** менее полезной, чем это кажется на первый взгляд. По сути, **atexit()** является для языка C заменителем отсутствующих в этом языке деструкторов.

Деструкторы сконструированных статически аллоцируемых объектов (§6.4.2), созданных до вызова **atexit(f)**, будут активизироваться после вызова **f**. А деструкторы подобных объектов, соданных после вызова **atexit(f)**, будут активизироваться до вызова функции **f**.

Функция **quick_exit()** напоминает **exit()** за исключением того, что она не активизирует никаких деструкторов. Регистрация функций, активизируемых функцией **quick_exit()**, выполняется с помощью функции **at_quick_exit()**.

Функции **exit()**, **abort()**, **quick_exit()**, **atexit()** и **at_quick_exit()** объявлены в файле **<cstdlib>**.

15.5. Советы

[1] Применяйте заголовочные файлы для представления интерфейсов и для подчеркивания логической структуры; §15.1, §15.3.2.

[2] Включайте заголовочный файл в исходный файл, в котором реализуются его функции; §15.3.1.

[3] Не допускайте определений глобальных сущностей с одинаковыми именами, но с разным смыслом в разных единицах трансляции; §15.2.
[4] Избегайте определений не-**inline**-функций в заголовочных файлах; §15.2.2.
[5] Применяйте директивы **#include** в глобальном контексте или же в пространствах имен; §15.2.2.
[6] Директивами **#include** включайте только полные (законченные) объявления; §15.2.2.
[7] Используйте «стражи включения»; §15.3.3.
[8] Внедряйте директивы **#include** для заголовочных файлов языка C в пространства имен во избежание глобальных имен; §14.4.9, §15.2.4.
[9] Делайте заголовочные файлы самодостаточными; §15.2.3.
[10] Различайте итерфейсы пользователей и интерфейсы реализаторов; §15.3.2.
[11] Делайте различие между интерфейсом для обычных пользователей и интерфейсом для экспертов; §15.3.2.
[12] Избегайте нелокальных объектов, требующих инициализации времени исполнения в коде, предназначенном для использования в составе не-C++-программы; §15.4.1.

Часть III

Механизмы абстракции

В этой части описаны средства C++ для определения и использования новых типов. Представлены техники, обычно называемые *объектно-ориентированным* и *обобщенным программированием*.

Главы

16. Классы
17. Конструирование, очистка, копирование и перемещение
18. Перегрузка операций
19. Специальные операции
20. Производные классы
21. Классовые иерархии
22. RTTI (идентификация типа на этапе выполнения)
23. Шаблоны
24. Обобщенное программирование
25. Специализации
26. Конкретизация
27. Шаблоны и иерархии
28. Метапрограммирование
29. Проект для работы с матрицами

«... нет дела, коего устройство было бы труднее, ведение опаснее, а успех сомнительнее, нежели замена старых порядков новыми. Кто бы не выступал с подобным начинанием, его ожидает враждебность тех, кому выгодны старые порядки, и холодность тех, кому выгодны новые ...»

— Николо Макиавелли («Государь» §vi)

16

Классы

Эти типы не «абстрактные»;
они столь же реальны, как **int** *и* **float**.
— Дуг МакИлрой

- Введение
- Основы классов

  Функции-члены; Копирование по умолчанию; Контроль доступа; **class** и **struct**; Конструкторы; **explicit**-конструкторы; Внутриклассовая инициализация; Определение функций внутри класса; Изменяемость; Ссылка на себя; Доступ к членам; Статические члены; Типы данных-члены класса
- Конкретные классы

  Функции-члены; Вспомогательные функции; Перегруженные операции; Роль конкретных классов
- Советы

16.1. Введение

Классы C++ служат основным средством создания новых типов, которые могут использоваться столь же удобно, как и встроенные типы. Кроме того, производные классы (§3.2.4, Глава 20) и шаблоны (§3.4, Глава 23) позволяют программисту выразить (иерархические и параметрические) зависимости между классами и использовать эти зависимости при создании программ.

Тип это конкретное представление концепции. Например, встроенный тип **float** с операциями **+,-, *** и т.д. представляет собой аппроксимацию математической концепции действительных (вещественных) чисел. Класс — это тип, определяемый пользователем. Мы разрабатываем классы для того, чтобы отразить концепции, не имеющие прямого аналога среди встроенных типов. Например, можно было бы определить тип **Trunk_line** для программы телефонии, или тип **Explosion** для игровой программы, или тип **list<Paragraph>** для программы обработки текста. Программы с типами, тесно связанными с основными концепциями предметной области, легче поддаются пониманию, модификации, поддержке и т.д. Хорошо подобранный набор пользовательских типов обычно способствует также и уменьшению совокупного объема программы. Он помогает компилятору обнаруживать случаи нештатного обращения с объектами, которые иначе были бы выявлены лишь в результате интенсивных мер отладки.

Важной идеей, лежащей в основе определения новых типов, является стремление отделить массу случайных деталей реализации (например, точной раскладки дан-

ных, формирующих объекты типа) от тех свойств, которые способствуют корректному применению типа. Это достигается четким определением интерфейса, специфичного для типа.

Настоящая глава фокусируется на относительно простых «конкретных» пользовательских типах, которые с логической точки зрения мало чем отличаются от встроенных типов:

§16.2 *Основы классов* рассматривают базовые средства определения классов и их членов.

§16.3 *Конкретные классы* рассматривают дизайн элегантных и эффективных конкретных классов.

Последующие главы вдаются в более глубокие детали и представляют абстрактные классы и классовые иерархии:

Глава 17 *Конструирование*, *Очистка*, *Копирование* и *Перемещение* — рассматриваются средства инициализации объектов класса, как можно копировать и перемещать объекты и как обеспечить «очищающие действия», которые нужно выполнять при уничтожении объектов (когда объект выходит из области видимости).

Глава 18 *Перегрузка операций* — в этой главе рассматривается, как можно определить унарные и бинарные операции (такие как +, * и!) для пользовательских типов и как можно их использовать.

Глава 19 *Специальные операции* — рассматриваются способы определения и использования операций (таких как [], (),—> **new**), которые сильно отличаются от простых арифметических и логических операций. В частности, в этой главе рассматривается строковый класс.

Глава 20 *Производные классы* — в этой главе представлены базовые средства языка, поддерживающие объектно-ориентированное программирование: базовые и производные классы, виртуальные функции и контроль доступа.

Глава 21 *Классовые иерархии* — здесь внимание сфокусировано на том, как можно использовать базовые и производные классы для эффективной организации кода программы. По большей части рассмотрение концентрируется вокруг технических вопросов, включая вопросы множественного наследования.

Глава 22 *Информация о типе на этапе выполнения* — представлены технологии навигации по иерархиям классов. В частности, рассматриваются операции **dynamic_cast** и **static_cast**, а также операция **typeid**.

16.2. Основы классов

Вот краткая сводка информации по классам:

- Класс это тип, определяемый пользователем.
- Класс состоит из набора членов. Наиболее типичные члены — члены-данные и функции-члены.
- Функции-члены могут определять смысл инициализации (создания), копирования, перемещения и очистки ресурсов.
- Доступ к членам осуществляется с помощью операций . (точка) для объектов и -> (стрелка) для указателей.
- Для класса можно определять операции (вроде операций +, ! и []).
- Для своих членов класс формирует пространство имен.

- Интерфейс класса формируют члены с уровнем доступа **public**, а члены с доступом **private** формируют представление.
- **struct** — это **class** со всеми открытыми членами (**public**).

Пример:

```
class X {
private:                        // представление закрытое
    int m;
public:                         // пользовательский интерфейс открытый
    X(int i =0): m{i} {}        // конструктор (инициализирует член данных m)

    int mf(int i)               // функция-член
    {
        int old = m;
        m = i;                  // задаем новое значение
        return old;             // возвращаем старое значение
    }
};

X var {7};                      // переменная типа X, инициализируется числом 7

int user(X var, X* ptr)
{
    int x = var.mf(7);          // доступ посредством . (точка)
    int y = ptr->mf(9);         // доступ посредством -> (стрелка)
    int z = var.m;              // ошибка: доступ к закрытому члену запрещен
}
```

Последующие разделы помогут лучше понять столь краткое введение в базовые средства. Их стиль изложения близок к учебному: постепенное развитие идей, а детали откладываются на более позние стадии.

16.2.1. Функции-члены

Представим концепцию даты с помощью **struct** (§2.3.1, §8.2) в виде типа **Date** с набором функций, предназначенных для манипуляций с переменными этого типа:

```
struct Date {       // представление
    int d, m, y;
};

void init_date(Date& d, int, int, int);     // инициализируем d
void add_year(Date& d, int n);              // добавляем n лет к d
void add_month(Date& d, int n);             // добавляем n месяцев к d
void add_day(Date& d, int n);               // добавляем n дней к d
```

Между типом **Date** и представленными функциями нет явной связи. Такую связь можно отразить определением функций в качестве членов типа:

```
struct Date {
    int d, m, y;

    void init(int dd, int mm, int yy);  // инициализация
    void add_year(int n);               // добавляем n лет
    void add_month(int n);              // добавляем n месяцев
    void add_day(int n);                // добавляем n дней
};
```

Функции, объявленные внутри определения класса (**struct** задает вариант класса; §16.2.4), называются *функциями-членами* и могут вызываться для конкретной переменной

этого типа, используя стандартный синтаксис доступа к членам структур (§8.2). Например:

```
Date my_birthday;

void f()
{
    Date today;

    today.init(16,10,1996);
    my_birthday.init(30,12,1950);

    Date tomorrow = today;
    tomorrow.add_day(1);
    // ...
}
```

Поскольку разные структуры могут иметь члены с одинаковыми именами, мы должны указывать имя структуры, когда определяем функцию-член:

```
void Date::init(int dd, int mm, int yy)
{
    d = dd;
    m = mm;
    y = yy;
}
```

Внутри определения функции-члена имена других членов могут использоваться без явной квалификации: имена ссылаются на члены объекта, для которого функция будет вызвана. Например, когда **Date::init()** будет вызвана для объекта **today**, присваивание **m=mm** выполняется для **today.m**. С другой стороны, когда **Date::init()** вызывается для объекта **my_birthday**, тогда **m=mm** выполняет присваивание для **my_birthday.m**. Функция-член класса «знает» объект, для которого она вызывается. Про **static**-члены см. §16.2.12.

16.2.2. Копирование по умолчанию

По умолчанию объекты можно копировать. В частности, объект класса может быть инициализирован копией другого объекта класса. Например:

```
Date d1 = my_birthday; // инициализация копированием
Date d2 {my_birthday}; // инициализация копированием
```

По умолчанию, копия объекта класса содержит копии всех членов. Если такое умолчательное копирование не подходит для класса **X**, то для этого класса может быть определено более подходящее копирование (§3.3, §17.5).

Аналогично, объект класса может быть скопирован по умолчанию операцией присваивания. Например:

```
void f(Date& d)
{
    d = my_birthday;
}
```

Опять-таки, умолчательное присваивание сводится к почленному присваиванию значений членов (§3.3, §17.5).

16.2.3. Контроль доступа

Объявление типа **Date** в предыдущем разделе предоставляет набор функций для манипуляций с этим типом. К сожалению, нет никаких указаний на то, что именно эти функции следует использовать для непосредственного доступа к представлению **Date**. И что нет никаких других функций для этого. Поэтому лучше определить **class** вместо **struct**:

```
class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy);   // инициализация

    void add_year(int n);                // добавляем n лет
    void add_month(int n);               // добавляем n месяцев
    void add_day(int n);                 // добавляем n дней
};
```

Метка **public** делит тело класса на две части. Первая часть (*private*) может использоваться лишь функциями-членами этого класса. Вторая часть — **public**, составляет открытый публичный интерфейс класса (интерфейс к объектам класса). Фактически **struct** это **class** со всеми открытыми по умолчанию членами (§16.2.4). Функции-члены можно использовать в точности так же, как и раньше:

```
void Date::add_year(int n)
{
    y += n;
}
```

Но вот функции, которые не являются функциями-членами этого класса, уже не могут иметь доступ к частным членам класса. Например:

```
void timewarp(Date& d)
{
    d.y -= 200;    // ошибка: Date::y закрытое
}
```

Теперь возрастает важность функции **init()**, поскольку при закрытых членах-данных нам нужно иметь способ их инициализации. Например:

```
Date dx;
dx.m = 3;          // ошибка: m закрытое
dx.init(25,3,2011); // OK
```

Имеется целый ряд преимуществ в закрытии доступа к полям данных, ограничивая доступ к ним объявленным списком функций. Например, ошибка, заключающаяся в том, что для **Date** устанавливается неправильное значение (например, Декабрь 36, 2016), может быть вызвана лишь ошибкой в коде функции-члена. Что сразу локализует поиск ошибки. Это отражает фундаментальное представление о том, что любое изменение в поведении класса может вызываться лишь изменениями в его членах. В частности, если мы меняем представление класса, то это затрагивает лишь его функции-члены. А пользовательский код зависит лишь от интерфейса класса и не должен переписываться в таком случае. А новый пользователь класса должен ознакомиться лишь с определениями функций-членов, что и будет означать его ознакомление с классом. Менее заметным преимуществом является общее соображение о том, что нарочитое фокусирование на интерфейсе заставляет разработчиков формировать более точный и ясный дизайн, который впоследствии откликнется уменьшением времени и сил на отладку кода.

Защита частных данных класса базируется на ограничении в использовании имен членов класса. Это в принципе можно обойти за счет манипуляций с адресами (§7.4.1) и за счет явного приведения типов (§11.5). Язык C++ защищает данные от непреднамеренной порчи, а не от хакерства. Только аппаратные ограничения могут осуществить защиту от злонамеренной эксплуатации штатных средств языка программирования, но даже это трудно выполнить в рамках реалистичных сценариев.

16.2.4. class и struct

Конструкция

```
class X {...};
```

называется *определением класса*; она определяет тип **X**. По историческим причинам определение класса часто называют объявлением класса. Как и в случае иных объявлений, определение класса можно реплицировать в разных исходных файлах, используя директивы **#include** без нарушения правила одного определения (§15.2.3).

Как мы уже говорили, **struct** определяет класс со всеми членами, которые открыты по умолчанию, то есть

```
struct S {/* ... */};
```

есть просто более короткая запись для

```
class S {public: /* ... */};
```

Эти два определения **S** взаимозаменяемы, хотя разумно придерживаться какого-то одного стиля. Здесь выбор больше зависит от вкуса программиста. Я предпочитаю использовать **struct** в случаях, когда думаю о «простых структурах данных». Когда же я создаю тип с инвариантами, то использую ключевое слово **class**. Конструкторы и функции доступа полезны и в случае **struct**, но они тут не служат гарантами инвариантов (§2.4.3.2, §13.4).

Члены классов по умолчанию закрытые:

```
class Date1 {
    int d, m, y;    // закрытые данные по умолчанию
public:
    Date1(int dd, int mm, int yy);
    void add_year(int n);    // добавить n лет
};
```

Но можно и явно применить спецификатор доступа **private:** чтобы наглядно заявить, что все последующие члены закрытые, точно так же, как **public** объявляет все последующие члены открытыми:

```
struct Date2 {
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);
    void add_year(int n);    // добавить n лет
};
```

За исключением разных имен типы **Date1** и **Date2** эквивалентны.

Нет необходимости объявлять поля данных в классе первыми. Более того, часто полезно объявлять их под конец, чтобы выдвинуть на авансцену функции, составляющие открытый интерфейс класса. Например:

```
class Date3 {
public:
    Date3(int dd, int mm, int yy);
    void add_year(int n);          // добавить n лет
private:
    int d, m, y;
};
```

В реалистичных примерах, где и поля данных, и интерфейс составляются из большого числа членов, я предпочитаю стиль примера с **Date3**.

Спецификаторы доступа могут использоваться в единственном объявлении класса много раз. Пример:

```
class Date4 {
public:
    Date4(int dd, int mm, int yy);
private:
    int d, m, y;
public:
    void add_year(int n);          // добавить n лет
};
```

При ручном программировании множество спецификаторов доступа может внести сумятицу, но при машинном составлении кода они вполне приемлемы.

16.2.5. Конструкторы

Применение функций вроде **init()** для выполнения инициализации объектов класса неэлегантно и, к тому же, может приводить к ошибкам. Поскольку нигде нет жестких рамок, заставляющих программиста в обязательном порядке выполнять инициализацию, он вполне может и забыть об этом. Или выполнить инициализацию дважды (часто с катастрофическими последствиями). Намного лучше обеспечить возможность объявить функцию-член, специально и явно предназначенную для инициализации. Поскольку такие функции конструируют объекты класса, их принято называть *конструкторами*. Конструкторы обязаны иметь имя, совпадающее с именем класса. Пример:

```
class Date {
    int d, m, y;
public:
    Date(int dd, int mm, int yy);  // конструктор
    // ...
};
```

Когда у класса есть конструктор, все объекты этого класса инициализируются вызовом конструктора. Если у конструктора есть аргументы, эти аргументы должны быть ему предоставлены:

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);         // короче для m
Date my_birthday;              // ошибка: отсутствует инициализатор
Date release1_0(10,12);        // ошибка: отсутствует третий аргумент
```

Так как конструктор задает инициализацию для класса, то мы можем применить и {}-инициализаторы:

```
Date today = Date {23,6,1983};
Date xmas {25,12,1990};        // короче для m
Date release1_0 {10,12};       // ошибка: отсутствует третий аргумент
```

Я предпочитаю {}-инициализацию, поскольку она явно отражает то, что делается, помогает избежать некоторых потенциальных ошибок и может использоваться систематически (§2.2.2, §6.3.5). Иногда должна использоваться ()-инициализация (§4.4.1, §17.3.2.1), но это редкий случай.

Предоставляя несколько конструкторов, мы реализуем разные способы инициализации объектов для данного типа. Например:

```
class Date {
    int d, m, y;
public:
    // ...

    Date(int, int, int);        // день, месяц, год
    Date(int, int);             // день, месяц, текущий год
    Date(int);                  // день, текущий месяц и год
    Date();                     // умолчательный Date: сегодня
    Date(const char*);          // дата в строковом представлении
};
```

Конструкторы подчиняются тем же самым правилам перегрузки, что и обыкновенные функции (§12.3). Пока конструкторы существенно отличаются друг от друга набором своих аргументов, компилятору нетрудно выбрать один из них:

```
Date today {4};                 // 4, today.m, today.y
Date july4 {"July 4, 1983"};
Date guy {5,11};                // 5, November, today.y
Date now;                       // сегодня как умолчательная инициализация
Date start {};                  // сегодня как умолчательная инициализация
```

Для типа **Date** выбор набора конструкторов вполне очевиден. В общем случае, обдумывая дизайн класса, не нужно включать в него абсолютно все средства, которые от вас могут просить со стороны. Нужно тщательно все обдумывать и включать только то, что в соответствии с вашим проектом является органичным для типа. Одним из способов уменьшения числа связанных друг с другом функций является применение умолчательных аргументов (§12.2.5). Для типа **Date** можно для каждого аргумента выбрать значение, говорящее «**today** (сегодня)»:

```
class Date {
    int d, m, y;
public:
    Date(int dd =0, int mm =0, int yy =0);
    // ...
};

Date:: Date(int dd, int mm, int yy)
{
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;

    // проверка корректности даты
}
```

Когда выбирается значение аргумента, указывающее на некоторый умолчательный смысл, то это значение нужно выбирать так, чтобы оно лежало вне обычного диапазона значений. Для **day** и **month** такое значение очевидно, а вот для **year** нуль не является совсем уж очевидным выбором. Но к счастью, нет нулевого года в Европейском календаре: 1AD (**year==1**) идет сразу за 1BC (**year==-1**).

Альтернативным образом можно в качестве умолчательных аргументов сразу указать умолчательные значения:

```
class Date {
    int d, m, y;
public:
    Date(int dd =today.d, int mm =today.m, int yy =today.y);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    // проверка корректности даты
}
```

Я выбрал **0** для того, чтобы избежать внедрения конкретных значений непосредственно в интерфейс типа **Date**. В таких случаях за нами остается выбор более подходящих значений при модификации дизайна проекта.

Отметим, что конструкторы сильно упрощают реализацию других функций-членов: им теперь нет необходимости дотошно проверять корректность состояния данных (§16.3.1).

16.2.6. explicit-конструкторы

По умолчанию, конструктор с одним аргументом реализует неявное приведение от типа аргумента к типу класса. Например:

```
complex<double> d {1}; // d=={1,0} (§5.6.2)
```

Такое неявное приведение типа полезно. Примером служат комплексные числа: если опустить мнимую часть, то получается комплексное число, лежащее на реальной оси. Это в точности совпадает с математическим представлением о комплексных числах. Но, в то же время, в иных случаях такие неявные приведения типов могут вносить недоразумения и ошибки. Рассмотрим пример с **Date**:

```
void my_fct(Date d);

void f()
{
    Date d {15};    // x становится {15, today.m, today.y}
    // ...
    my_fct(15);     // неясно
    d = 15;         // неясно
    // ...
}
```

В лучшем случае, это туманное решение. Нет прямой логической связи между числом **15** и датой (**Date**), не зависящей от внутренностей нашего кода.

По счастью, мы можем явным образом отметить, что конструктор не используется для неявного приведения типа. Для этого конструктор объявляется с ключевым словом **explicit**: такой конструктор может использоваться лишь для инициализации и явных приведений типа. Пример:

```
class Date {
    int d, m, y;
public:
    explicit Date(int dd =0, int mm =0, int yy =0);
    // ...
};
```

```
Date d1 {15};          // OK: явно
Date d2 = Date{15};    // OK: явно
Date d3 = {15};        // ошибка: = инициализация не делает неявных конверсий
Date d4 = 15;          // ошибка: = инициализация не делает неявных конверсий

void f()
{
    my_fct(15);        // ошибка: при передаче аргумента нет неявных конверсий
    my_fct({15});      // ошибка: при передаче аргумента нет неявных конверсий
    my_fct(Date{15});  // OK: явно
    // ...
}
```

Вариант инициализации со знаком = трактуется как *инициализация копированием*. По сути, копия инициализатора помещается в инициализируемый объект. Но это копирование может быть оптимизировано вплоть до его устранения, или же может быть применена перемещающая семантика (§3.3.2, §17.5.2) в случае, если инициализатор представляет из себя rvalue (§6.4.1). Если убрать знак =, то получается явная инициализация, известная как *прямая инициализация*.

В общем случае стоит объявлять конструктор с единственным аргументом с ключевым словом **explicit**. Должны быть явные причины против такого решения (пример — тип **complex**). В таких случаях нужно четко документировать ваш выбор.

Если конструктор объявлен с ключевым словом **explicit**, то при его определении вне тела класса не нужно повторять **explicit**:

```
class Date {
    int d, m, y;
public:
    explicit Date(int dd);
    // ...
};

Date::Date(int dd) {/* ... */}           // OK
explicit Date::Date(int dd) {/* ... */}  // ошибка
```

В большинстве случаев **explicit** используется для конструкторов с единственным аргументом. Но это ключевое слово может оказаться полезным и для конструкторов без аргументов или же с большим числом аргументов. Например:

```
struct X {
    explicit X();
    explicit X(int, int);
};

X x1 ={};          // ошибка: неявно
X x2 = {1,2};      // ошибка: неявно

X x3{};            // OK: явно
X x4 {1,2};        // OK: явно

int f(X);

int i1 = f({});    // ошибка: неявно
int i2 = f({1,2}); // ошибка: неявно

int i3 = f(X{});   // OK: явно
int i4 = f(X{1,2}); // OK: явно
```

Различие между прямой инициализацией и инициализацией копированием сохраняется для инициализации списком (§17.3.4.3).

16.2.7. Внутриклассовая инициализация

Когда у нас множество конструкторов, инициализация членов может стать повторяющейся. Например:

```
class Date {
    int d, m, y;
public:
    Date(int, int, int);        // день, месяц, год
    Date(int, int);             // день, месяц, текущий год
    Date(int);                  // день, текущий месяц и год
    Date();                     // умолчательный Date: сегодня
    Date(const char*);          // дата в строковом представлении
    // ...
};
```

Мы можем с этим справиться, вводя умолчательные аргументы для уменьшения количества конструкторов (§16.2.5). В качестве альтернативы можно добавить инициализаторы для членов-данных:

```
class Date {
    int d {today.d};
    int m {today.m};
    int y {today.y};
public:
    Date(int, int, int);        // день, месяц, год
    Date(int, int);             // день, месяц, текущий год
    Date(int);                  // день, текущий месяц и год
    Date();                     // умолчательный Date: сегодня
    Date(const char*);          // дата в строковом представлении
    // ..
```

Теперь для каждого конструктора **d**, **m** и **y** проинициализированы, даже если он сам этого не делает. Пример:

```
Date::Date(int dd)
    : d{dd}
{
    // проверить корректность даты
}
```

Это эквивалентно следующему:

```
Date::Date(int dd)
    : d{dd}, m{today.m}, y{today.y}
{
    // проверить корректность даты
}
```

16.2.8. Определение функций внутри класса

Функция-член, определенная внутри тела класса — а не только объявленная — считается встраиваемой (§12.1.5). Таким образом, внутриклассовое определение функций-членов подходит для небольших, редко модифицируемых, но часто используемых функций. Как и определение класса, которому принадлежат такие функции, они реплицируются

во все единицы трансляции (где они требуются) с помощью директив **#include**; при этом смысл класса и его встраиваемых функций-членов должен быть одинаковым во всех единицах трансляции (§15.2.3).

Любой член класса может ссылаться на другой член класса независимо от того, где тот определен (§6.3.4). Рассмотрим следующее:

```
class Date {
public:
    void add_month(int n) {m+=n;}          // инкрементировать m у Date
    // ...
private:
    int d, m, y;
};
```

Порядок определения функций и членов-данных не важен. Я мог бы эквивалентным образом написать:

```
class Date {
public:
    void add_month(int n) {m+=n;}          // инкрементировать m у Date
    // ...
private:
    int d, m, y;
};

inline void Date:: add_month(int n)        // добавить n месяцев
{
    m+=n;                                  // инкрементировать m у Date
}
```

Последний стиль помогает делать определения классов простыми и удобными для чтения. При этом достигается текстуальное разделение интерфейса класса и реализации.

Очевидно, что я наивно упростил определение для **Date::add_month()**, просто надеясь, что прибавление числа **n** не повредит дату (§16.3.1).

16.2.9. Изменяемость

Мы можем определить именованный объект как константу или как переменную. Другими словами, имя может ссылаться на *неизменяемое* или же *изменяемое* значение. Систематическое применение в языке C++ неизменяемых объектов приводит к более ясному коду, к раннему обнаружению ошибок и даже к более эффективному коду. Особо ценно свойство неизменяемости в многопоточных программах (§5.3, Глава 41).

Для достижения пользы далеко за пределами констант встроенных типов мы должны иметь возможность определять функции, оперирующие с константными объектами пользовательских типов. Для глобальных функций это означает, что они принимают аргументы вида **const T&**. Для функций-членов это означает, что мы должны иметь возможность определять функции, оперирующие с константными объектами.

16.2.9.1. Константные функции-члены

Тип **Date**, с которым мы имели дело до сих пор, определяет функции, назначающие значение для дат. Но, к сожалению, до сих пор не было способов выявить элементы дат. Этот недостаток легко устраняется введением функций для чтения дня, месяца и года:

```
class Date {
    int d, m, y;
```

```
public:
    int day() const {return d;}
    int month() const {return m;}
    int year() const;

    void add_year(int n);      // добавить n лет
    // ...
};
```

Ключевое слово **const** после списка аргументов в объявлении функции означает, что эта функция не модифицирует состояние объекта типа **Date**.

Естественно, что компилятор легко обнаружит попытки нарушить такое обещание. Например:

```
int Date::year() const
{
    return ++y;          // ошибка: попытка изменить значение члена в const-функции
}
```

Когда константная функция-член определяется вне тела класса, повтор ключевого слова **const** обязателен:

```
int Date::year()         // ошибка: const отсутствует в типе функции-члена
{
    return y;
}
```

Другими словами, **const** является частью типа для **Date::day()**, **Date::month()** и **Date::year()**.

Константная функция-член может быть вызвана как для константных объектов, так и для неконстантных, в то время как неконстантная функция-член — только для неконстантных объектов. Например:

```
void f(Date& d, const Date& cd)
{
    int i = d.year();      // OK
    d.add_year(1);         // OK

    int j = cd.year();     // OK
    cd.add_year(1);        // ошибка: нельзя изменить значение у const Date
}
```

16.2.9.2. Физическая и логическая неизменяемость

Иногда функция-член с логической точки зрения константная, но ей все равно требуется изменить значение члена данных. То есть, с пользовательской точки зрения функция воспринимается константной, однако в рамках деталей, которые пользователю не видны, модификация объекта все-таки выполняется. Про такую ситуацию говорят как о логической неизменяемости (константности). Например, класс **Date** мог бы иметь функцию, возвращающую строковое представление. Но конструирование такого представления могло бы быть накладным. Поэтому было бы полезно хранить копию, которую возвращали бы повторные запросы всегда, когда изменения даты не было. Кэширование значений, вроде рассмотренного примера, нередки в случае более сложных конструкций, но мы сейчас рассмотрим вопрос о том, как это реализовать на простом примере класса **Date**:

```
class Date {
public:
    // ...
    string string_rep() const;      // строковое представление
```

```
private:
    bool cache_valid;
    string cache;
    void compute_cache_value(); // fill cache
    // ...
};
```

С точки зрения пользователя **string_rep()** не изменяет состояния **Date**, так что она очевидно константная функция-член. С другой стороны, **cache** и **cache_valid** должны время от времени изменяться, чтобы это имело смысл.

Такие проблемы может решить посредством силового приведения типа, например с помощью **const_cast** (§11.5.2). Однако, есть и более элегантные решения, не грешащие мешаниной с типами.

16.2.9.3. mutable

Можно определить член класса с ключевым словом **mutable**, означающим, что его значение может изменяться даже для объектов, объявленных с **const**:

```
class Date {
public:
    // ...
    string string_rep() const;                  // строковое представление
private:
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value() const;           // заполнить (изменчивый) кэш
    // ...
};
```

Теперь можно определить **string_rep()** в очевидной манере;

```
string Date::string_rep() const
{
    if (!cache_valid) {
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```

Мы можем использовать **string_rep()** как для **const**-объектов, так и для не-**const**-объектов. Пример:

```
void f(Date d, const Date cd)
{
    string s1 = d.string_rep();
    string s2 = cd.string_rep();    // OK!
    // ...
}
```

16.2.9.4. Изменяемость посредством косвенного обращения

Разумно объявлять член с ключевым словом **mutable** тогда, когда речь идет об изменении малой части представления небольших объектов. В более сложных случаях целесообразно помещать изменяющиеся данные в отдельный объект и обращаться к нему косвенным образом. В таком случае наш пример с кэшированием строкового представления обретает следующий вид:

```
struct cache {
    bool valid;
    string rep;
};

class Date {
public:
    // ...
    string string_rep() const;                  // строковое представление
private:
    cache* c;                                   // инициализация в конструкторе
    void compute_cache_value() const;           // заполнить то, на что ссылается кэш
    // ...
};

string Date::string_rep() const
{
    if (!c->valid) {
        compute_cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

Программные техники, поддерживающие кэширование, обобщаются до различных форм «ленивых трансформаций».

Отметим, что **const** не применяется (транзитивно) к объектам, к которым доступ выполняется по указателям или ссылкам. Читатель-человек может трактовать такие объекты как подобъекты, но компилятор не может трактовать такие указатели или ссылки как отличающиеся от общего случая указателей и ссылок. Таким образом, указатель-член класса не имеет семантики, отличной от остальных указателей.

16.2.10. Ссылка на себя

Функции **add_year()**, **add_month()** и **add_day()** (§16.2.3), изменяющие состояние, были определены без возвратов. Для подобного набора связанных функций часто полезно возвращать ссылку на обновленный объект, чтобы эти операции можно было соединять в цепочку. Например, мы бы могли захотеть написать следующее:

```
void f(Date& d)
{
    // ...
    d.add_day(1).add_month(1).add_year(1);
    // ...
}
```

чтобы прибавить день, месяц и год к **d**. Для этого каждая функция должна возвращать ссылку на **Date**:

```
class Date {
    // ...

    Date& add_year(int n);          // добавить n лет
    Date& add_month(int n);         // добавить n месяцев
    Date& add_day(int n);           // добавить n дней
};
```

Любая (нестатическая) функция-член знает объект, для которого она вызвана, и может явно ссылаться на этот объект. Например:

```
Date& Date::add_year(int n)
{
    if (d==29 && m==2 &&!leapyear(y+n)) { // осторожно: Февраль 29
        d = 1;
        m = 3;
    }
    y += n;
    return *this;
}
```

Выражение ***this** ссылается на объект, для которого функция-член вызвана.

В нестатической функции-члене ключевое слово **this** означает указатель на объект, для которого функция вызвана. В неконстантной функции-члене класса **X** тип **this** есть **X***. Однако, **this** рассматривается как rvalue, так что нет возможности взять адрес у **this** или присвоить что-либо **this**. В константной функции-члене класса **X** тип **this** есть **const X*** для предотвращения модификации самого объекта (см. также §7.5).

Большинство случаев использования **this** выполняется неявно. В частности, любая отсылка к нестатическому члену изнутри определения класса полагается на неявное использование **this** для доступа к соответствующему объекту. Например, функция **add_year()** могла бы эквивалентно (но утомительно) делать следующее:

```
Date& Date::add_year(int n)
{
    if (this->d==29 && this->m==2 &&!leapyear(this->y+n)) {
        this->d = 1;
        this->m = 3;
    }
    this->y += n;
    return *this;
}
```

Есть пример явного применения **this** в манипуляции со связанным списком:

```
struct Link {
    Link* pre;
    Link* suc;
    int data;

    Link* insert(int x)        // вставить x перед this
    {
        return pre = new Link{pre, this, x};
    }

    void remove()              // удалить и уничтожить this
    {
        if (pre) pre->suc = suc;
        if (suc) suc->pre = pre;
        delete this;
    }

    // ...
};
```

Явное использование **this** требуется для доступа к членам базового класса из производного класса, являющегося шаблоном (§26.3.7).

16.2.11. Доступ к членам

Для доступа к члену класса **X** используется операция. (точка), применяемая к объекту класса **X**, или используется операция **->** (стрелка) к указателю на объект класса **X**. Например:

```
struct X {
    void f();
    int m;
};

void user(X x, X* px)
{
    m = 1;    // ошибка: m нет в области видимости
    x.m = 1;  // OK
    x->m = 1;     // ошибка: x не указатель
    px->m = 1;    // OK
    px.m = 1;     // ошибка: px указатель
}
```

Очевидно, что здесь имеет место некоторая степень избыточности: компилятор знает, ссылается ли имя на **X** или на **X***, так что единственной операции было бы достаточно. Но программист мог бы и путаться в таком случае, так что с самых первых дней языка C использовались две отдельные операции.

Изнутри класса вообще нет нужды в какой-либо операции. Пример:

```
void X::f()
{
    m = 1;    // OK: "this->m = 1;" (§16.2.10)
}
```

То есть неквалифицированное имя члена работает так, как если бы оно имело префиксом **this->**. Заметьте, что функция-член может использовать имя члена, который объявляется позже:

```
struct X {
    int f() {return m;}   // отлично: возвращаем этот m у X
    int m;
};
```

Если нужно обратиться к члену класса в общем случае, а не к члену конкретного объекта, нужно его квалифицировать именем класса и ::. Например:

```
struct S {
    int m;
    int f();
    static int sm;
};

int X::f() {return m;}      // X's f
int X::sm {7};              // X's статический член sm (§ 16.2.12)
int (S::*) pmf() {&S::f};   // X's член f
```

Последняя конструкция (указатель на член) используется очень редко; см. §20.6. Я упомянул ее здесь исключительно с целью показать общность правила для ::.

16.2.12. Статические члены

Удобство умолчательных значений для **Date** было куплено ценой скрытой проблемы: наш класс **Date** стал зависим от глобальной переменной **today**. Класс **Date** можно использовать лишь в контексте, в котором переменная **today** определена и корректно используется всеми частями кода. То есть, вне такого контекста класс бесполезен. Пользователи такого класса могут встретиться с неприятными сюрпризами, а поддержка такого кода затруднена. Конечно, «всего лишь одна маленькая глобальная переменная» и не наносит огромного вреда, но, в принципе, такой стиль программирования приводит к коду, который часто просто бесполезен всем, кроме его создателя. В общем, такой стиль кода следует избегать.

По счастью, можно получить результаты, которые мы добились применением глобальной переменной, и без глобальной переменной. Член класса, которому не соответствует часть объекта класса, называется статическим (маркируется ключевым словом **static**). Существует единственная копия такого члена, а не одна копия на каждый объект класса, как это имеет место для нестатических членов (§6.4.2). Аналогично, функция, нуждающаяся в доступе к членам класса, но для которой возможен вызов безотносительно объекта класса, называется статической функцией-членом.

Вот наш новый дизайн класса **Date**, сохраняющий семантику умолчательных значений для конструктора, но не зависящий от проблем, связанных с глобальными переменными:

```
class Date {
    int d, m, y;
    static Date default_date;
public:
    Date(int dd =0, int mm =0, int yy =0);
    // ...
    static void set_default(int dd, int mm, int yy); // устанавливаем умолчательный Date(dd, mm, yy)
};
```

Теперь мы можем определить конструктор для **Date**, использующий **default_date** следующим образом:

```
Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : default_date.d;
    m = mm ? mm : default_date.m;
    y = yy ? yy : default_date.y;

    // ... проверка корректности даты...
}
```

Используя **set_default()**, мы можем изменить умолчательную дату, когда это необходимо. На статический член можно ссылаться точно так же, как и на любой другой член. Кроме того, на статический член можно ссылаться без упоминания объекта. Вместо этого его имя квалифицируется именем класса. Пример:

```
void f()
{
    Date::set_default(4,5,1945);    // вызов статического члена Date::set_default()
}
```

Для использования статического члена — функции или члена данных — его нужно где-либо определить (создать). При этом повторного применения ключевого слова **static** не требуется. Например:

```
Date Date::default_date {16,12,1770};          // определение Date::default_date

void Date::set_default(int d, int m, int y)    // определение Date::set_default
{
    default_date = {d, m, y};                  // новое значение default_date
}
```

Теперь умолчательной датой является день рождения Бетховена — пока кто-либо не решит изменить это значение.

Заметим, что **Date{}** служит сокращенным обозначением для значения **Date::default_date**. Пример:

```
Date copy_of_default_date = Date{};

void f(Date);

void g()
{
    f(Date{});
}
```

Следовательно, нам не нужна отдельная функция для чтения умолчательной даты. Более того, там, где целевой тип однозначно **Date**, там простого обозначения **{}** достаточно. Например:

```
void f1(Date);

void f2(Date);
void f2(int);

void g()
{
    f1({});          // OK: эквивалентно f1(Date{})
    f2({});          // ошибка: неоднозначно: f2(int) или f2(Date)?
    f2(Date{});      // OK
}
```

В многопоточном коде статические члены-данные нуждаются в каком-либо способе блокировки или дисциплине доступа, чтобы избежать гонки за данными (§5.3.4, §41.2.4). При переводе старого кода в многопоточные варианты статические члены вызывают проблемы, с которыми приходится основательно бороться.

16.2.13. Типы данных-члены класса

Типы и синонимы типов могут быть членами классов. Например:

```
template<typename T>
class Tree {
    using value_type = T;              // синоним члена
    enum Policy {rb, splay, treeps};   // член-перечисление
    class Node {                       // класс-член
        Node* right;
        Node* left;
        value_type value;
    public:
        void f(Tree*);
    };
    Node* top;
public:
    void g(const T&);
    // ...
};
```

Класс-член (часто называемый *вложенным классом*) может ссылаться на типы и статические члены окружающего класса. На нестатические члены он может ссылаться лишь в том случае, когда он располагает объектом окружающего класса. Не углубляясь в детали бинарных деревьев, я просто в техническом плане приведу несложные примеры.

Вложенный класс имеет доступ к членам окружающего класса, даже к закрытым членам, но он не обладает знанием о текущем объекте окружающего класса. Например:

```
template<typename T>
void Tree::Node::f(Tree* p)
{
    top = right;                    // ошибка: нет объекта типа Tree
    p->top = right;                 // OK
    value_type v = left->value;     // OK: value_type не связан с объектом
}
```

Класс не имеет никаких специальных прав доступа к членам его внутреннего (вложенного) класса. Пример:

```
template<typename T>
void Tree::g(Tree::Node* p)
{
    value_type val = right->value;      // ошибка: нет объекта типа Tree::Node
    value_type v = p->right->value;     // ошибка: Node::right закрытый
    p->f(this);      // OK
}
```

Вложенные классы являются удобным средством, а не средством фундаментальной важности. С другой стороны, члены-синонимы типов важны как базис технологии обобщенного программирования, базирующейся на ассоциированных типах (§28.2.4, §33.1.3). Перечисления-члены класса часто являются альтернативой для **enum class**, когда желательно избегать загрязнения окружающей области видимости именами энумераторов (§8.4.1).

16.3. Конкретные классы

В предыдущем разделе мы обсуждали детали дизайна класса **Date** в контексте изучения средств языка для определения классов. В настоящем разделе я разверну фокус нашего внимания и обсужу именно дизайн простого и эффективного класса **Date**.

Небольшие абстракции, интенсивно применяемые в программировании, весьма распространены в самых разных предметных областях. Примеры: латинские символы, символы китайского языка, целые числа, числа с плавающей запятой, комплексные числа, геометрические точки, координаты, трансформации, пары (точка, смещение), даты, время, диапазоны, связи, ассоциации, узлы, пары (значение, единица), место на диске, место расположения исходного кода, величины валют, линии, прямоугольники, масштабированные числа с фиксированной запятой, числа с дробной частью, строки символов, вектора и массивы. Практически каждая программа использует что-нибудь из этого списка. В отдельных программах перечисленные конкретные типы используются крайне интенсивно. В типичном случае, приложения используют некоторые из перечисленных типов напрямую, но еще больше конкретных типов используется косвенно посредством библиотек.

Некоторые из этих абстракций язык C++ поддерживает в виде встроенных типов. Естественно, что все подобного рода типы не могут априорно поддерживаться языком из-за их количества. Да и предвидеть, какие именно типы потребуются впредь, невозможно. Поэтому в языке C++ реализован механизм, с помощью которого пользователи могут соз-

давать необходимые им типы, в том числе и маленькие конкретные типы. Последние так называются с целью отличать их от абстрактных классов (§20.4) и классов из классовых иерархий (§20.3, §21.2).

Класс называется *конкретным*, если его представление является частью его определения. Этим они отличаются от абстрактных классов (§3.2.2, §20.4), предоставляющих интерфейс ко многим реализациям. Располагая представлением, мы можем:

- Помещать объекты в стек, в статически аллоцированную память и в другие объекты
- Копировать и перемещать объекты (§3.3, §17.5)
- Прямо ссылаться на именованные объекты (в противоположность доступу по указателям и ссылкам)

Это делает конкретные классы простыми и удобными для оптимизации компилятором. Поэтому конкретные классы предпочтительны для компактных и простых концепций, вроде комплексных чисел (§5.6.2), интеллектульных указателей (§5.2.1) и контейнеров (§4.4).

С самых первых дней языка C++ его целью была поддержка определения и эффективного применения таких пользовательских типов. Они являются фундаментом элегантного программирования. В свете сказанного, давайте построим более совершенный тип **Date**:

```
namespace Chrono {

    enum class Month {jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

    class Date {
    public:                                                    // открытый интерфейс:
        class Bad_date {};                                     // класс исключения
        explicit Date(int dd ={}, Month mm ={}, int yy ={}); // {} значит "умолчательно"
    // немодифицирующие Date функции:
        int day() const;
        Month month() const;
        int year() const;

        string string_rep() const;                             // строковое представление
        void char_rep(char s[], in max) const;                 // представление С-строкой

    // модифицирующие Date функции:
        Date& add_year(int n);                                 // добавить n лет
        Date& add_month(int n);                                // добавить n месяцев
        Date& add_day(int n);                                  // добавить n дней
    private:
        bool is_valid();                                  // проверка, что Date является датой
        int d, m, y;                                      // представление
    };

    bool is_date(int d, Month m, int y);                  // true для корректной даты
    bool is_leapyear(int y);                              // true, если y високосный год

    bool operator==(const Date& a, const Date& b);
    bool operator!=(const Date& a, const Date& b);

    const Date& default_date();                           // умолчательная дата

    ostream& operator<<(ostream& os, const Date& d);      // выводим d в os
    istream& operator>>(istream& is, Date& d);            // читаем Date из is в d
}   // Chrono
```

Показанный набор операций довольно типичен для пользовательских типов:

1. Конструктор определяет, как инициализируются объекты типа (§16.2.5).
2. Набор функций, с помощью которых можно исследовать **Date**. Эти функции маркируются **const**, чтобы показать, что они не модифицируют состояние объекта, для которого они вызваны.
3. Набор функций, позволяющий пользователю модифицировать **Date** без того, чтобы в точности знать его представление.
4. Неявно определенные операции, позволяющие свободно копировать объекты типа **Date** (§16.2.2).
5. Класс **Bad_date** используется, чтобы с помощью исключений информировать об ошибках.
6. Набор полезных вспомогательных функций, которые не являются членами и не имеют прямого доступа к представлению **Date**, но они указаны как имеющие отношение к **Date** при помощи включения в пространство имен **Chrono**.

Я определил тип **Month**, чтобы избавиться от необходимости помнить порядок месяц/день, например **{6,7}** (7 Июня в американском стиле) или **{7,6}** (то же в европейском стиле).

Я рассматривал возможность определить типы **Day** и **Year**, чтобы избежать неразберихи с **Date{1995, Month::jul,27}** и **Date{27, Month::jul,1995}**. Однако, эти типы не столь полезны, как **Month**. И кроме того, такие ошибки ловятся во время выполнения — 26 Июля 27 года не являются такими уж употребительными в моей работе. А исторические даты до 1800 года лучше оставить экспертам-историкам.

Чтобы избавить пользователя от необходимости явно задавать год и месяц тогда, когда они вытекают из контекста, я добавил механизм предоставления умолчаний. Заметьте, что для **Month** выражение **{}** дает умолчательное значение **0**, как для целых чисел, хоть это и не является корректным значением для **Month** (§8.4). Однако в данном случае это то, что нужно: ненастоящее значение, сигнализирующее об умолчании. Вообще, проблема умолчательных значений не такая простая. Для некоторых типов выработаны общеупотребительные решения, например **0** для целых; для некоторых типов вообще нет разумных умолчательных значений. Ну, и есть типы, для которых этот вопрос дискуссионен, например это как раз тип **Date**. Я здесь выбрал конкретное умолчательное значение просто для того, чтобы проиллюстрировать проблему.

Я опустил описание техники кэширования (§16.2.9) для столь простого типа, как наш. Если будет необходимость, его можно включить в качестве детали реализации без воздействия на пользовательский интерфейс.

Вот простой пример использования типа **Date**:

```
void f(Date& d)
{
    Date lvb_day {16, Month::dec, d.year()};

    if (d.day()==29 && d.month()==Month::feb) {
        // ...
    }

    if (midnight()) d.add_day(1);

    cout << "day after:" << d+1 << '\n';

    Date dd; // инициализируем умолчательной датой
    cin>>dd;
    if (dd==d) cout << "Hurray!\n";
}
```

Предполагается, что операция + была определена для **Date**. Я это сделал в §16.3.3.

Обратите внимание на явную квалификацию для **dec** и **feb** типом **Date**. Я использовал **enum class** (§8.4.1) специально для того, чтобы иметь возможность применять короткие имена для месяцев.

Почему вообще стоит определять специальный тип для такой простой концепции, как дата? В конце концов, можно было бы определить простую структуру данных:

```
struct Date {
    int day, month, year;
};
```

А потом пусть каждый программист сам думает, что с ней делать. Если поступить таким образом, то программистам пришлось бы манипулировать компонентами дат напрямую, или предоставить некоторый набор функций для этого. В результате, концепция даты расползлась бы по всему коду хаотически. А программу стало бы труднее понимать, документировать и модифицировать по мере ее использования. То есть, реализация концепции слишком простыми средствами возлагает всю работу на программистов, которые должны использовать эту концепцию.

Но даже такая простая концепция как **Date**, на самом деле не столь уж и проста. Например, инкрементирование даты встречается с самыми разными вопросами: как быть с високосными годами, как учесть разную длительность разных месяцев, и многое другое. Далее, представление даты в виде день-месяц-год не всегда удовлетворяет практические потребности. Если здесь нужно внести коррективы в представление (например, представить дату в виде количества дней до или после 1 января 1970 года), то придется перепрограммировать только наш набор функций (при сохранении интерфейса).

Мне пришлось долго думать над выбором умолчательной даты. В принципе, я вообще бы отказался от умолчания и заставил пользователей детально инициализировать тип **Date**. Однако, многие общие интерфейсы обобщенного программирования требуют умолчательных конструкций (§17.3.3). Это заставляет меня все-таки выбрать умолчательное значение даты. В конце концов, я остановился на 1 января 1970 года, поскольку это является стартовой точкой для функций времени из стандартной библиотеки (§35.2, §43.6). В результате отпадает нужда в функции-члене **set_default_date()**.

Чтобы сохранить некоторый потенциал для будущих модификаций, я объявил вспомогательную функцию **default_date()**:

```
const Date& Chrono::default_date();
```

Это ничего не говорит о том, как на самом деле устанавливается дата по умолчанию.

16.3.1. Функции-члены

Естественно, что реализация функций-членов должна где-нибудь присутствовать. Например:

```
Date::Date(int dd, Month mm, int yy)
    :d{dd}, m{mm}, y{yy}
{
    if (y == 0) y = default_date().year();
    if (m == Month{}) m = default_date().month();
    if (d == 0) d = default_date().day();

    if (!is_valid()) throw Bad_date();
}
```

Конструктор проверяет аргумент-дату на корректность. Если предоставленная дата некорректна, например **{30, Month::feb,1994}**, то генерируется исключение (§2.4.3.1, Глава 13), свидетельствующее о том, что так дело не пойдет. Если же дата корректная, то дальше выполняется инициализация. Можно сказать, что инициализация здесь непростая, поскольку приходится оценивать предоставленные данные. Но после того, как объект типа **Date** создан, его можно далее использовать и копировать, ничего не проверяя. Другими словами, конструктор устанавливает инвариант для класса (в данном случае это корректность даты). Другие функции-члены опираются на этот инвариант и поддерживают его. Эта проектная идея способствует надежности кода и значительно упрощает его (§2.4.3.2, §13.4).

Я использую выражение **Month{}** — которое не представляет собой никакого месяца и имеет значение **0**, для того чтобы выразить требование «выбрать месяц по умолчанию». Я мог бы специально для этого определить энумератор в типе **Month**, но я отказался от такого решения, чтобы не создавать иллюзии, что в году 13 месяцев. Отметим, что можно использовать **Month{}**, имеющее значением **0**, так как это значение находится в диапазоне, гарантированном для перечисления **Month** (§8.4).

Я использую синтаксис инициализаторов (§17.4) для инициализации членов. После этого я проверяю на нуль и модифицирую величины при необходимости. Это не дает оптимальной производительности в случае (по счастью редкой) ошибки, но делает структуру кода очевидной. Если бы нужно было сделать акцент на производительности, то тогда пришлось бы реализовать три конструктора, а не один конструктор с умолчательными значениями для аргументов.

Я рассматривал вариант с открытой функцией **is_valid()**, но решил, что при этом пользовательский код станет более сложным, так что предпочел вариант с исключением:

```
void fill(vector<Date>& aa)
{
    while (cin) {
        Date d;
        try {
            cin >> d;
        }
        catch (Date::Bad_date) {
            // ... обработка своей ошибки ...
            continue;
        }
        aa.push_back(d); // см. §4.4.2
    }
}
```

А функцию **is_valid()** я реализовал в качестве вспомогательной:

```
bool Date::is_valid()
{
    return is_date(d, m, y);
}
```

Зачем иметь две функции — **is_valid()** и **is_date()**? В данном простом примере это излишне, но я имел в виду системы, для которых **is_date()** проверяет тройку **(d, m, y)** на предмет представления корректной даты, а **is_valid()** выполняет дополнительную проверку на предмет, можно ли эту дату разумным образом представить. Например, **is_valid()** может отвергать даты до момента, когда начали использовать современные каландарные системы.

Как это свойственно простым конкретным типам, определения функций-членов **Date** варьируются от тривиальных до не слишком сложных. Например:

```
inline int Date::day() const
{
    return d;
}

Date& Date::add_month(int n)
{
    if (n==0) return *this;

    if (n>0) {
        int delta_y = n/12;                    // число целых лет
        int mm = static_cast<int>(m)+n%12;     // число месяцев
        if (12 < mm) {                         // десятичное значение 12
            ++delta_y;
            mm -= 12;
        }

        // ... обработка случая, когда в месяце mm нет дня d ...

        y += delta_y;
        m = static_cast<Month>(mm);
        return *this;
    }

    // ... обработка отрицательного n ...

    return *this;
}
```

Если бы я добавил в функцию **add_month()** все детали, то она могла бы по сложности приблизиться к функциям из реальных (не учебных) проектов. В связи с этим возникает вопрос: наращивание месяца вроде как совсем несложная задача, так почему же код столь сложен? Причина кроется в том, что **d**, **m**, **y**-представление не столь удобно для компьютера, как для человека. Удобным представлением было бы число дней после какой-то фиксированной даты («нулевого дня» — 1 января 1970). Тогда все манипуляции с датами стали бы простыми, но сложность переместилась бы в процедуру визуализации для людей.

Отметим, что инициализация присваиванием и копированием предоставляются по умолчанию (§16.2.2). Кроме того, тип **Date** не нуждается в деструкторе, поскольку этот тип не владеет ресурсами и не требует никаких процедур очистки при выходе из области видимости (§3.2.1.2).

16.3.2. Вспомогательные функции

В типичном случае, для класса предоставляется набор функций, связанных с этим классом, но которым нет нужды быть функциями-членами, поскольку они не требуют доступа к представлению. Например:

```
int diff(Date a, Date b);          // число дней в диапазоне [a,b) или [b,a)

bool is_leapyear(int y);
bool is_date(int d, Month m, int y);

const Date& default_date();
Date next_weekday(Date d);
Date next_saturday(Date d);
```

Если определять такие функции внутри класса, то это усложнило бы интерфейс класса и потребовало бы большей работы по проверке всех функций-членов в сдучае изменения представления класса.

Как эти функции ассоциируются с классом **Date**? В ранних версиях языка C++ (как и в языке C) их объявления просто помещались в тот же самый файл, в котором было объявление класса **Date**. Пользователи, которым нужны эти функции, включали их вместе с заголовочным файлом, содержащим определение интерфейса класса (§15.2.2). Например:

```
#include "Date.h"
```

Дополнительно (или альтернативно) можно класс и его вспомогательные функции заключить в пространство имен (§14.3.1):

```
namespace Chrono {     // средства работы со временем

    class Date {/* ... */};

    int diff(Date a, Date b);
    bool is_leapyear(int y);
    bool is_date(int d, Month m, int y);
    const Date& default_date();
    Date next_weekday(Date d);
    Date next_saturday(Date d);
    // ...
}
```

Естественно, что пространство имен **Chrono** может включать и связанные классы, такие как **Time** и **Stopwatch**, а также их вспомогательные функции.

Естественно, что вспомогательные функции должны быть где-нибудь определены:

```
bool Chrono::is_date(int d, Month m, int y)
{
    int ndays;

    switch (m) {
    case Month::feb:
        ndays = 28+is_leapyear(y);
        break;
    case Month::apr: case Month::jun: case Month::sep: case Month::nov:
        ndays = 30;
        break;
    case Month::jan: case Month::mar: case Month::may: case Month::jul:
    case Month::aug: case Month::oct: case Month::dec:
        ndays = 31;
        break;
    default:
        return false;
    }

    return 1<=d && d<=ndays;
}
```

Я здесь намеренно впадаю в паранойю. **Month** не должен быть вне диапазона от **jan** до **dec**, но из-за неряшливости так может случиться, так что я проверяю.

Наконец, вот функция **default_date()**:

```
const Date& Chrono::default_date()
{
    static Date d {1, Month::jan,1970};
    return d;
}
```

16.3.3. Перегруженные операции

Часто очень полезно определять функции, которые в пользовательском коде предоставят стандартные нотации операций. Например, **operator==()** обеспечит возможность для типа **Date** применять операцию сравнения ==:

```
inline bool operator==(Date a, Date b)  // равно
{
    return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

А вот другие очевидные кандидаты:

```
bool operator!=(Date, Date);          // не равно
bool operator<(Date, Date);           // меньше, чем
bool operator>(Date, Date);           // больше, чем
// ...

Date& operator++(Date& d) {return d.add_day(1);}          // увеличить Date на один день
Date& operator--(Date& d) {return d.add_day(-1);}         // уменьшить Date на один день

Date& operator+=(Date& d, int n) {return d.add_day(n);}   // добавить n дней
Date& operator-=(Date& d, int n) {return d.add_day(-n);}  // вычесть n дней

Date operator+(Date d, int n) {return d+=n;}              // добавить n дней
Date operator-(Date d, int n) {return d+=n;}              // вычесть n дней

ostream& operator<<(ostream&, Date d);                    // вывести d
istream& operator>>(istream&, Date& d);                   // прочитать в d
```

Эти операции определяются в **Chrono** вместе с **Date** во избежание проблем с перегрузками и чтобы получить пользу от поиска на основе аргументов (§14.2.4).

Для типа **Date** определение указанных операций можно рассматривать как средство повышения удобства для пользователей. В то же время для комплексных чисел (§18.3), для векторов (§4.4.1) и функциональных объектов (§3.4.3, §19.2.2) применение стандартных операций практически обязательно, ибо люди так к ним привыкли, что просто не поймут разработчиков в случае их отсутствия. В целом перегрузка операций рассматривается в Главе 18.

Для типа **Date** я хотел предоставить операции **+=** и **-=** вместо **add_day()**. В таком случае я бы следовал общей идиоме (§3.2.1.1).

Отметим, что инициализация присваиванием и копированием предоставляется по умолчанию (§16.3, §17.3.3).

16.3.4. Роль конкретных классов

Я называю простые пользовательские типы, вроде **Date**, конкретными типами для того, чтобы отличать их от абстрактных классов (§3.2.2) и классовых иерархий (§20.4), а также, чтобы подчеркнуть их сходство со встроенными типами вроде **int** и **char**. Они и используются как встроенные типы. Их модель использования и философия их дизайна сильно отличаются от того, что называется объектно-ориентированным программированием (§3.2.4, Глава 21).

Цель конкретных типов состоит в необходимости эффективно выполнить относительно простую работу. Чаще всего пользователю не предоставляются средства модификации поведения конкретных типов. Кроме того, конкретные типы не должны обеспечивать полиморфизм времени выполнения (§3.2.3, §20.3.2).

Если вы не довольны какими-то аспектами конкретного типа, вы просто строите новый тип с желаемым поведением. А если вы хотите воспользоваться кодом конкретного типа, вы можете использовать его реализацию в вашем новом типе точно так же, как вы бы это сделали с типом **int**. Пример:

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    Date_and_time(Date d, Time t);
    Date_and_time(int d, Date::Month m, int y, Time t);
    // ...
};
```

Альтернативный механизм наследования классов рассматривается в Главе 20: его можно использовать для построения новых типов из конкретных классов, просто описывая желаемую разницу. Примером может служить построение класса **Vec** из **vector** (§4.4.1.2). Однако, наследование от конкретного класса следует выполнять с осторожностью и в редких случаях из-за отсутствия виртуальных функций и поддержки информации о типе на этапе выполнения (§17.5.1.4, Глава 22).

В случае достаточно эффективного компилятора конкретные классы вроде **Date** не предусматривают никаких дополнительных накладных расходов ни по памяти, ни по времени выполнения. В частности, не обязательно применение указателей для доступа к объектам, и нет никакой «внутренней кухни», хранящейся в объектах конкретных классов. Размеры объектов конкретных типов известны на этапе компиляции, так что их можно располагать в стеке. Раскладка объектов также известна на этапе компиляции, так что встраивание операций легко достигается. И, наконец, совместимость по раскладке с другими языками, такими как C или Fortran, достигается без дополнительных усилий.

Хороший набор таких типов может послужить надежным основанием для многих приложений. В частности, они помогают проектировать более специальные и менее подверженные ошибкам интерфейсы. Например:

```
Month do_something(Date d);
```

В таком варианте менее вероятны недопонимания и неправильное использование, чем в случае отсутствия конкретных пользовательских классов:

```
int do_something(int d);
```

Отсутствие конкретных типов ведет к неясным программам и к потере времени программистов на их изучение, когда каждый программист напрямую обращается к простым и часто используемым структурам, представленным в виде простых агрегатов из встроенных типов. А если вместо простых и эффективных конкретных типов используется слишком большие и переобобщенные классы, то резко страдает эффективность как по времени, так и по расходу памяти.

16.4. Советы

[1] Представляйте концепции классами; §16.1.

[2] Отделяйте интерфейс класса от его реализации; §16.1.

[3] Используйте открытые данные (**struct**) только тогда, когда они используются исключительно как данные, и нет инварианта для их членов; §16.2.4.

[4] Определяйте конструктор для инициализации объектов; §16.2.5.

[5] По умолчанию объявляйте конструктор с единственным аргументом с ключевым словом **explicit**; §16.2.6.

[6] Объявляйте функцию-член, которая не модифицирует состояние объекта, с ключевым словом **const**; §16.2.9.

[7] Конкретные типы являются простейшими видами классов; где только возможно, предпочитайте конкретные типы более сложным классам и более простым структурам данных; §16.3.

[8] Делайте функцию членом класса только, если ей нужен прямой доступ к представлению класса; §16.3.2.

[9] Используйте пространство имен, чтобы явно отразить ассоциацию класса и вспомогательных функций; §16.3.2.

[10] Функцию-член, не модифицирующую значения объекта, объявляйте с ключевым словом **const**; §16.2.9.1.

[11] Делайте функцию-член, требующую доступа к представлению класса, но не нуждающуюся в вызове от имени объекта, статической (**static**); §16.2.12.

17

Конструирование, очистка, копирование и перемещение

Невежество
чаще порождает уверенность,
чем знание.
— Чарльз Дарвин

- Введение
- Конструкторы и деструкторы

  Конструкторы и инварианты; Деструкторы и ресурсы; Деструкторы членов и базового класса; Вызов конструкторов и деструкторов; Виртуальные деструкторы
- Инициализация объектов класса

  Инициализация в отсутствие конструкторов; Инициализация конструкторами; Умолчательные конструкторы; Конструкторы со списком инициализаторов
- Инициализация членов и инициализация базового класса

  Инициализация членов; Инициализаторы базового класса; Делегирующие конструкторы; Внутриклассовые инициализаторы; Инициализация статических членов
- Копирование и перемещение
- Генерирование умолчательных операций

  Явное требование генерирования умолчаний; Умолчательные операции; Применение умолчательных операций; Объявление функций с **=delete**
- Советы

17.1. Введение

Эта глава фокусируется на технических аспектах жизненого цикла объектов класса. Как мы создаем объекты, как мы их копируем, как перемещаем и как очищаем ресурсы после того, как объекты исчезают? Каково надлежащее определение «копирования» и «перемещения»? Пример:

```
string ident(string arg)  // строка передается по значению (копируется в arg)
{
    return arg;           // возврат строки (перемещение значения строки из функции вызвавшему коду)
}
```

```
int main ()
{
    string s1 {"Adams"};        // инициализация строки (конструирование s1).
    s1 = indet(s1);             // копируем s1 в ident()
                                // перемещаем результат вызова ident(s1) в s1;
                                // значение s1 есть "Adams".
    string s2 {"Pratchett"};    // инициализация строки (конструирование s2)
    s1 = s2;                    // копируем значение s2 в s1
                                // как s1 так и s2 имеют значением "Pratchett".
}
```

Ясно, что после вызова **ident()** значение **s1** должно стать равным **"Adams"**. Мы копируем значение **s1** в аргумент **arg**, затем мы перемещаем значение **arg** из функции снова в **s1**. Затем мы конструируем **s2** со значением **"Prachett"** и копируем его в **s1**. В конце концов, по выходу из **main()** мы уничтожаем переменные **s1** и **s2**. Различие между копированием и перемещением состоит в том, что после копирования два объекта имеют одинаковые значения, а после перемещения источник этой операции может и не иметь оригинального значения. Перемещение можно использовать в случае, когда объект-источник не предполагается далее использовать. Это особо полезно при воплощении понятия перемещения ресурсов (§3.2.1.2, §5.2).

В нашем примере используется несколько функций:

- Конструктор, инициализирующий строку типа **string** строковым литералом (используется для **s1** и **s2**)
- Копирующий конструктор для копирования строки типа **string** (в аргумент **arg**)
- Перемещающий конструктор для значения типа **string** (из **arg** вовне **ident()** во временную переменную, хранящую результат вызова **ident(s1)**)
- Перемещающее присваивание для значения типа **string** (из временной переменной, хранящей результат вызова **ident(s1)** в **s1**)
- Копирующее присваивание для копирования значения типа **string** (из **s2** в **s1**)
- Деструктор для освобождения ресурсов из **s1**, **s2** и временной переменной, хранящей результат вызова **ident(s1)**

Оптимизатор может устранить часть работы. Например, в таком простом примере можно устранить временную переменную.

Конструкторы, операции присваивания копированием и перемещением напрямую поддерживают управление ресурсами. Объект может рассматриваться объектом своего типа после того, как отработает конструктор, и он остается объектом своего типа вплоть до момента, когда начнет работать деструктор. Взаимосвязь между временем жизни объекта и потенциальными ошибками в программе изучается в §13.2 и §13.3. В частности, здесь мы не рассматриваем вопрос о сконструированных наполовину объектах и об объектах, уничтоженных наполовину.

Конструирование объектов лежит в основе многих программных технологий. Широта вариантов применения находит свое отражение в диапазоне и гибкости языковых средств, поддерживающих инициализацию.

Для класса конструкторы, деструкторы, копирующие и перемещающие операции не являются логически изолированными. Мы должны определять их в тесной связке, или же встретимся с проблемами — логическими или проблемами с эффективностью. Если у класса **X** есть деструктор, выполняющий нетривиальную работу вроде освобождения динамически выделенной памяти или снятия блокировки, то скорее всего класс нуждается в полном комплекте функций:

```
class X {
    X(Sometype);              // "обычный конструктор": создает объект
    X();                      // конструктор по умолчанию
    X(const X&);              // копирующий конструктор
    X(X&&);                   // перемещающий конструктор
    X& operator=(const X&); // присваивание копированием: очистка целевого объекта и копиров.
    X& operator=(X&&);        // перемещ. присваивание: очистка целевого объекта и перемещ.
    ~X();                     // деструктор: очистка
    // ...
};
```

Объект копируется или перемещается в пяти случаях:

- Когда он служит источником присваивания
- Когда он является инициализатором объекта
- Когда объект служит аргументом функции
- Когда объект является возвратом функции
- Когда он является объектом исключения

Во всех этих случаях будет применяться копирующий или перемещающий конструктор.

Дополнительно к инициализации именованных объектов и объектов в свободной памяти конструкторы применяются для инициализации временных объектов (§6.4.2) и для реализации явных преобразований типа (§11.5).

Кроме «обычного конструктора» иные специальные функции могут генерироваться компилятором; см. §17.6.

Настоящая глава полна правил и технических подробностей. Это нужно для исчерпывающего понимания, хотя многие люди предпочитают учиться на примерах.

17.2. Конструкторы и деструкторы

Определяя конструктор, мы специфицируем в точности, как именно объект класса должен инициализироваться (§16.2.5, §17.3). Дополняя конструкторы, определим деструктор, гарантирующий «очистку» в точке уничтожения объекта (где он выходит из области видимости). Один из самых надежных приемов работы с ресурсами базируется на парах конструктор/деструктор. Пример:

```
struct Tracer {
    string mess;
    Tracer(const string& s): mess{s} {clog << mess;}
    ~Tracer() {clog << "~" << mess;}
};

void f(const vector<int>& v)
{
    Tracer tr {"in f()\n"};
    for (auto x : v) {
        Tracer tr {string{"v loop "}+to<string>(x)+'\n'}; // §25.2.5.1
        // ...
    }
}
```

Попробуем вызов

```
f({2,3,5});
```

Получим следующий выходной поток:

```
in_f()
v loop 2
~v loop 2
v loop 3
~v loop 3
v loop 5
~v loop 5
~in_f()
```

17.2.1. Конструкторы и инварианты

Член класса с именем, совпадающим с именем класса, называется конструктором. Например:

```
class Vector {
public:
    Vector(int s);
    // ...
};
```

Объявление конструктора включает список аргументов (как и для обычных функций), но не имеет возврата. Имя класса нельзя использовать для обычной функции-члена, члена данных, типа-члена и т.д. в пределах объявления класса. Например:

```
struct S {
    S();                    // хорошо
    void S(int);            // ошибка: у конструктора нет возврата
    int S;                  // ошибка: имя класса только у конструктора
    enum S {foo, bar};      // ошибка: имя класса только у конструктора
};
```

Задача конструктора состоит в инициализации объекта класса. Часто такая инициализация устанавливает инвариант класса, то есть нечто, что должно иметь место при вызове функций-членов (не из определения класса). Рассмотрим следующее:

```
class Vector {
public:
    Vector(int s);
    // ...
private:
    double* elem;       // elem указывает на массив из sz эл-ов типа double
    int sz;             // sz не отрицательное
};
```

Здесь инвариант класса формулируется посредством комментария: «**elem** указывает на массив из **sz** элементов типа **double**» и «**sz** нетрицательно». Конструктор должен воплотить это. Например:

```
Vector::Vector(int s)
{
    if (s<0) throw Bad_size{s};
    sz = s;
    elem = new double[s];
}
```

Здесь конструктор пытается установить инваринт, и если это не удается, он генерирует исключение. В таком случае объект не создается, а конструктор должен гарантировать, что никакие ресурсы не утекают (§5.2, §13.3). В общем случае, ресурс — это нечто, что

мы сначала захватываем, а потом (явно или неявно) возвращаем, как только нужда в нем отпадет. Примерами служат память (§3.2.1.2), блокировки (§5.3.4), дескрипторы файлов (§13.3) и дескрипторы потоков (§5.3.1).

Для чего стоит определять инвариант?

- Чтобы сфокусироваться на проектном решении для класса (§2.4.3.2)
- С целью уточнения поведения класса (§13.2)
- Для упрощения определений функций-членов (§2.4.3.2, §16.3.1)
- Для уточнения правил управления ресурсами в классе (§13.3)
- Ради улучшения документирования класса

В общем и целом, введение классового инварианта в конце концов упрощает работу.

17.2.2. Деструкторы и ресурсы

Конструктор инициализирует объект. Другими словами, он создает окружение, в котором выполняются операции с классом. Иногда для этого приходится захватывать ресурс — вроде файла, блокировки, или памяти, и его нужно будет потом освободить (§5.2, §13.3). В итоге, некоторые классы нуждаются в функции, которая будет автоматически вызвана в момент уничтожения объекта класса так же, как конструкторы автоматически вызываются в момент создания объекта класса. Поэтому такую функцию называют деструктором. Имя деструктора составляется из символа ~, за которым следует имя класса, например **~Vector()**. Деструктор не принимает аргументов, и в классе может быть только один деструктор. Деструктор вызывается неявно, когда автоматическая переменная выходит из своей области видимости, объект в свободной памяти удаляется, и т.д. Только в очень редких случаях пользователь должен явно вызывать деструктор (§17.2.4).

Деструктор в типичном случае освобождает захваченные ресурсы. Например:

```
class Vector {
public:
    Vector(int s): elem{new double[s]}, sz{s} {}; // конструктор: захват памяти
    ~Vector() {delete[] elem;}                    // деструктор: освобождение памяти
    // ...
private:
    double* elem;                  // elem указывает на массив из sz эл-ов типа double
    int sz;                        // sz не отрицательное
};
```

Например:

```
Vector* f(int s)
{
    Vector v1(s);
    // ...
    return new Vector(s+s);
}

void g(int ss)
{
    Vector* p = f(ss);
    // ...
    delete p;
}
```

Здесь **v1** уничтожается по выходе из **f()**. Кроме того, созданный в свободной памяти в рамках функции **f()** **Vector** уничтожается операцией **delete**. В обоих случаях активируется деструктор класса **Vector** для освобождения памяти, выделенной конструктором.

А что если конструктор не сможет выделить затребованную память? Например, если **s*sizeof(double)** или **(s+s)*sizeof(double)** превысит количество свободной памяти (измеряемой в байтах). В этом случае **new** сгенерирует исключение типа **std::bad_alloc** (§11.2.3) и механизм обработки исключений активирует соответствующие деструкторы, так что вся ранее выделенная память (и только она) будет освобождена (§13.5.1).

Скоординированная пара конструктор/деструктор служит стандартным механизмом реализации понятия объектов переменной длины в языке C++. Контейнеры стандартной библиотеки, такие как **vector** и **unordered_map**, применяют некие варианты этого подхода для хранения своих элементов.

Считается, что типы, в которых нет явно объявленного деструктора (например, встроенные типы), имеют деструктор, который ничего не делает.

Если программист объявляет в своем классе деструктор, то он должен решить, могут ли объекты класса копироваться или перемещаться (§17.6).

17.2.3. Деструкторы членов и базового класса

Конструкторы и деструкторы корректно соотносятся с классовыми иерархиями (§3.2.4, Глава 20). Конструктор строит объект класса «снизу-вверх»:

1. Сначала конструктор вызывает конструктор базового класса,
2. Затем он активирует конструкторы членов и
3. Наконец, он выполняет свое тело.

Деструктор «расправляется» с объектом в обратном порядке:

1. Сначала деструктор выполняет свое тело,
2. Затем он вызывает деструкторы членов и
3. Наконец, он вызывает деструктор базового класса.

В частности, виртуальный базовый класс конструируется перед всеми базовыми классами, которые могут его использовать, а уничтожается он после уничтожения перечисленных базовых классов (§21.3.5.1). Этот порядок гарантирует, что никакой член или базовый класс не используется до полной инициализации или используется после уничтожения. У программиста есть возможность нарушить это простое и важное правило, но только посредством намеренной передачи в качестве аргументов указателей на неинициализированные переменные. Разумеется, это может привести к катастрофическим последствиям.

Конструкторы вызывают конструкторы членов и базовых классов в порядке их объявления: если бы два конструктора использовали разный порядок, то деструктор не смог бы гарантировать точный обратный порядок уничтожения. См. также §17.4.

Если предполагается использовать класс так, что нужен умолчательный конструктор, то в случае отсутствия объявления иных конструкторов компилятор сам попытается сгенерировать умолчательный конструктор. Например:

```
struct S1 {
    string s;
};

S1 x;          // OK: x.s инициализируется в ""
```

Аналогично, может использоваться почленная инициализация в случае необходимости в инициализаторах. Пример:

```
struct X {X(int);};

struct S2 {
    X x;
};
```

```
S2 x1;          // ошибка:
S2 x2 {1};      // OK: x2.x инициализируется в 1
```

См. также §17.3.1.

17.2.4. Вызов конструкторов и деструкторов

Деструктор вызывается неявно по выходе из области видимости или по причине операции **delete**. В типичном случае нет необходимости в явном вызове деструктора; более того, чаще всего это вызовет катастрофические последствия. Однако, существуют редкие случаи, когда деструктор нужно вызывать явно. Рассмотрим контейнер, который (как **std::vector**) поддерживает пул памяти, в рамках которого он может расти или сжиматься (например, посредством **push_back()** и **pop_back()**). Когда мы добавляем элемент, контейнер должен вызвать конструктор элемента по конкретному адресу:

```
void C::push_back(const X& a)
{
    // ...
    new(p) X{a};        // X конструируется копированием значения a по адресу p
    // ...
}
```

Такой способ вызова конструктора известен как «размещающее **new**» (§11.2.4).

И наоборот, когда мы удаляем элемент, нужно вызвать деструктор для этого элемента:

```
void C::pop_back()
{
    // ...
    p->~X();            // уничтожение X по адресу p
}
```

Запись **p->~X()** обеспечивает вызов деструктора для ***p**. Такой стиль вызова деструктора никогда не применяется для объекта, уничтожаемого нормальным образом (когда объект выходит из области видимости или явно уничтожается операцией **delete**).

Более полный пример явной манипуляции объектами в блоке памяти рассматривается в §13.6.1.

Будучи объявленным в классе **X**, деструктор будет неявно вызван в случае выхода объекта из области видимости (или в случае операции **delete**). Из этого следует, что мы можем воспротивиться уничтожению объектов типа **X**, объявив деструктор с **=delete** (§17.6.4) или с ключевым словом **private**.

Из этих двух альтернатив использование **private** дает больше гибкости. Например, можно объявить класс, объекты которого могут уничтожаться лишь явно (неявно не могут):

```
class Nonlocal {
public:
    // ...
    void destroy() {this->~Nonlocal();}  // явное уничтожение
private:
    // ...
    ~Nonlocal();                         // не уничтожать неявно
};

void user()
{
    Nonlocal x;                 // ошибка: нельзя уничтожить Nonlocal
    X* p = new Nonlocal;        // OK
    // ...
```

```
    delete p;          // ошибка: нельзя уничтожить Nonlocal
    p.destroy();       // OK
}
```

17.2.5. Виртуальные деструкторы

Деструктор можно объявить с ключевым словом **virtual**, что в обычном случае стоит делать для классов с виртуальными функциями. Например:

```
class Shape {
public:
    // ...
    virtual void draw() = 0;
    virtual ~Shape();
};

class Circle {
public:
    // ...
    void draw();
    ~Circle();              // замещает ~Shape()
    // ...
};
```

Причина нужды в виртуальном деструкторе заключается в том, что объекты, манипулируемые обычно посредством интерфейса базового класса, уничтожаются также через этот интерфейс.

```
void user(Shape* p)
{
    p->draw();              // вызов соответствующего draw()
    // ...
    delete p;               // вызов соответствущего destructor
};
```

Если бы деструктор у типа **Shape** не был виртуальным, то операция **delete** потерпела бы неудачу в вызове соответствующего деструктора производного класса (т.е. **~Circle()**) и могла произойти утечка ресурсов.

17.3. Инициализация объектов класса

В этом разделе мы рассмотрим инициализацию объектов класса с конструкторами и без конструкторов. Мы также рассмотрим определение конструкторов, которые могут принимать однородные списки инициализации переменного размера (вроде **{1,2,3}** и **{1,2,3,4,5,6}**).

17.3.1. Инициализация в отсутствие конструкторов

Для встроенных типов нет возможности объявить конструктор. Однако мы можем выполнить инициализацию подходящим типом. Например:

```
int a {1};
char* p {nullptr};
```

Аналогично можно инициализировать объекты класса, у которого мы не объявляли конструкторов, если использовать:

- почленную инициализацию,
- инициализацию копированием или
- умолчательную инициализацию (без инициализаторов или пустым списком инициализации).

Например:

```
struct Work {
    string author;
    string name;
    int year;
};

Work s9 {"Beethoven",
    "Symphony No. 9 in D minor, Op. 125; Choral",
    1824
    };                          // почленная инициализация

Work currently_playing {s9};    // инициализация копированием
Work none {};                   // умолчательная инициализацияп
```

Все три члена объекта **currently_playing** являются копиями членов объекта **s9**.

Умолчательная инициализация с {} определяется как инициализация каждого члена посредством {}. Так что **none** инициализируется выражением {{},{},{}}, что дает {"","",0} (§17.3.3).

Там, где требующие аргументов конструкторы не объявляются, можно полностью отказаться от инициализатора. Пример:

```
Work alpha;

void f()
{
    Work beta;
    // ...
}
```

В таком случае правила не столь просты, как могло бы показаться. Для статически аллоцируемых объектов (§6.4.2) правило таково, как если бы вы использовали {}, так что значением **alpha** станет {"","",0}. Однако, для локальных объектов и объектов в свободной памяти умолчательная инициализации выполняется для членов, имеющих тип класса, а члены встроенных типов остаются неинициализированными, так что значением **beta** станет {"","", **unknown**}.

Причиной такого усложнения является желание повысить производительность для редких критических случаев. Например:

```
struct Buf {
    int count;
    char buf[16*1024];
};
```

Вы можете использовать **Buf** в качестве локальной переменной без ее предварительной инициализации перед использованием в качестве цели для операции ввода. Для большинства локальных переменных инициализация не критична в плане эффективности, а неинициализированные локальные переменные создают массу проблем. Если хотите гарантировать инициализацию и избежать сюрпризов, предоставляйте инициализатор, такой как {}. Пример:

```
Buf buf0;       // статическая аллокация, так что умолчательная инициализация

void f()
{
    Buf buf1;               // оставляем эл-ты неинициализированными
    Buf buf2 {};            // я действительно хочу обнулить элементы

    int* p1 = new int;      // *p1 не инициализировано
    int* p2 = new int{};    // *p2 == 0
    int* p3 = new int{7};   // *p3 == 7
    // ...
}
```

Естественно, почленная инициализация возможна лишь тогда, когда есть доступ к членам. Например:

```
template<class T>
class Checked_pointer {     // контроль доступа к T* члену
public:
    T& operator*();         // проверка на nullptr и возврат значения
// ...
};

Checked_pointer<int> p {new int{7}};    // ошибка: нет доступа к p.p
```

Если у класса есть закрытый нестатический член данных, то нужен конструктор для его инициализации.

17.3.2. Инициализация посредством конструкторов

Когда почленное копирование не подходит, нужно определить конструктор для инициализации объекта. В частности, конструктор часто используется для установления инварианта класса и для захвата ресурсов (§17.2.1).

Если для класса объявляется конструктор, то какой-то из конструкторов будет использоваться для его объектов. Будет ошибкой попытка инициализации объекта ненадлежащими инициализаторами, противоречащими конструкторам. Например:

```
struct X {
    X(int);
};

X x0;           // ошибка: нет инициализатора
X x1 {};        // ошибка: пустой инициализатор
X x2 {2};       // OK
X x3 {"two"};   // ошибка: неправильный тип инициализатора
X x4 {1,2};     // ошибка: неправильное число инициализаторов
X x5 {x4};      // OK: копирующий конструктор определен неявно(§ 17.6)
```

Заметьте, что умолчательный конструктор (§17.3.3) пропадает, как только вы определяете конструктор с аргументами; в конце концов, **X(int)** уведомляет, что требуется значение **int** для конструирования объекта класса **X**. В то же время, копирующий конструктор не исчезает (§17.3.3); полагается, что объект можно копировать. Если такое умолчательное копирование вызывает проблемы (§3.3.1), его можно явным образом запретить (§17.6.4).

Я применяю {} для отражения факта инициализации. Запись {} может применяться для передачи аргументов конструктору, если объект должен быть сконструирован. Пример:

```
struct Y : X {
    X m{0};                 // предоставляет умолчательный инициализатор для члена m
    Y(int a): X{a}, m{a} {}; // инициализация базы и члена (§ 17.4)
    Y() : X{0} {};          // инициализация базы и члена
};

X g {1};            // инициализация глобальной переменной

void f(int a)
{
    X def {};               // ошибка: нет умолчательного значения для X
    Y de2 {};               // OK: используется умолчательный конструктор
    X* p {nullptr};
    X var {2};              // инициализация локальной переменной
    p = new X{4};           // инициализация объекта в свободной памяти
    X a[] {1,2,3};          // инициализация элементов массива
    vector<X> v {1,2,3,4};  // инициализация элементов вектора
}
```

По этой причине {}-инициализация иногда называется универсальной инициализацией. Дополнительно, {}-инициализация однородна: где бы вы не инициализировали объект типа **X** значением **v** используя обозначение **{v}**, одно и то же значение типа **X** (**X{v}**) создается.

А вот нотации со знаками = и () для инициализации (§6.3.5) не являются универсальными. Например:

```
struct Y : X {
   X m;
   Y(int a) : X(a), m=a {}; // ошибка синтаксиса: нельзя использовать = для инициализации членов
};

X g(1); // инициализация глобальной переменной

void f(int a)
{
   Xdef();              // функция, возвращающая X (сюрприз!?)
   X* p {nullptr};
   X var = 2;           // инициализация локальной переменной
   p = new X=4;         // синтаксич. ошибка: нельзя использовать = для new
   X a[](1,2,3);        // ошибка: нельзя использовать () для инициализации массива
   vector<X> v(1,2,3,4); // ошибка: нельзя использовать () для элементов списка
}
```

Нотации с = и () не являются также и однородными, правда, к счастью, примеры таких случаев довольно туманны и редки. Если уж вам так хочется применять = или () для инициализации, то вам нужно помнить, где они разрешены и что они означают.

К конструкторам применимы обычные правила разрешения перегрузки (§12.3). Например:

```
struct S {
    S(const char*);
    S(double*);
};

S s1 {"Napier"};            // S::S(const char*)
S s2 {new double{1.0}};     // S::S(double*);
S s3 {nullptr};             // неоднозначно: S::S(const char*) или S::S(double*)?
```

Заметьте, что {}-инициализация не допускает сужения (§2.2.2). Это еще одна причина, по которой следует отдавать ей предпочтение.

17.3.2.1. Инициализация конструкторами

Используя ()-нотацию, вы требуете использовать конструктор для инициализации. То есть, вы можете гарантировать инициализацию именно конструктором, а не почленную инициализацию или инициализацию списком инициализации (§17.3.4), предлагаемую {}-нотацией. Например:

```
struct S1 {
    int a, b;  // нет конструктора
};

struct S2 {
    int a, b;
    S2(int a = 0, int b = 0) : a(aa), b(bb) {} // конструктор
};

S1 x11(1,2);    // ошибка: нет конструктора
S1 x12 {1,2};   // OK: почленная инициализация

S1 x13(1);      // ошибка: нет конструктора
S1 x14 {1};     // OK: x14.b становится 0

S2 x21(1,2);    // OK: используется конструктор
S2 x22 {1,2};   // OK: используется конструктор

S2 x23(1);      // OK: конструктор и один умолчательный аргумент
S2 x24 {1};     // OK: конструктор и один умолчательный аргумент
```

Однородная {}-инициализация стала возможной лишь в C++11, так что более ранний код использует инициализацию с () и =. Как следствие, последние варианты должны быть для вас более привычны. Тем не менее, я не знаю ни одной логической причины, по которой вы могли бы предпочесть ()-нотацию, за исключением редкого случая, когда нужно различать между инициализацией списком элементов и списком аргументов конструктора. Например:

```
vector<int> v1 {77};      // один элемент со значением 77
vector<int> v2(77);       // 77 элементов с умолчательным значением 0
```

Такая проблема — и необходимость сделать выбор — может возникать у типа с конструктором со списком инициализаторов (§17.3.4), в типичном случае у контейнера, у которого есть и «обычный конструктор», принимающий аргументы, имеющие тип его элементов. В частности, мы иногда должны использовать ()-инициализацию для контейнера **vector** с элементами типа **int** или чисел с плавающей запятой, но никогда для элементов строкового или указательного типов:

```
vector<string> v1 {77};       // 77 элементов с умолчательным значением ""
                              // (vector<string>(std::initializer_list<string>) не принимает {77})
vector<string> v2(77);        // 77 элементов с умолчательным значением ""

vector<string> v3 {"Booh!"}; // один элемент со значением "Booh!"
vector<string> v4("Booh!");  // ошибка: нет конструктора со строковым аргументом

vector<int*> v5 {100,0};      // 100 int*s инициализируются в nullptr (100 не int*)

vector<int*> v6 {0,0};        // 2 int* инициализируются в nullptr
vector<int*> v7(0,0);         // пустой вектор (v7.size()==0)
vector<int*>v8;               // пустой вектор (v7.size()==0)
```

Примеры с **v6** и **v7** могут быть интересны лишь адвокатам языка.

17.3.3. Умолчательные конструкторы

Конструктор, который можно вызвать без аргументов, называется умолчательным конструктором. Умолчательные конструкторы очень распространены. Пример:

```
class Vector {
public:
    Vector();       // умолчательный конструктор: нет элементов
    // ...
};
```

Умолчательный конструктор используется в случае, когда не предоставляются никакие аргументы, или в случае пустого списка инициализации:

```
Vector v1;          // OK
Vector v2 {};       // OK
```

Умолчательный аргумент (§12.2.5) может превратить конструктор с аргументами в умолчательный конструктор. Пример:

```
class String {
public:
    String(const char* p = "");   // умолчательный конструктор : пустая строка
    // ...
};

String s1;          // OK
String s2 {};       // OK
```

Типы **vector** и **string** из стандартной библиотеки имеют такие умолчательные конструкторы (§36.3.2, §31.3.2).

Про встроенные типы считается, что они располагают умолчательным и копирующим конструкторами. Однако, в случае встроенных типов умолчательный конструктор не вызывается для неинициализированных нестатических переменных (§17.3). Умолчательным значением для встроенных целых типов служит **0**, для типов с плавающей запятой это **0.0**, и **nullptr** для указателей. Пример:

```
void f()
{
    int a0;                 // не инициализировано
    int a1();               // объявление функции

    int a {};               // a становится 0
    double d {};            // d становится 0.0
    char* p {};             // p становится nullptr

    int* p1 = new int;      // неинициализированное целое
    int* p2 = new int{};    // int инициализируется в 0
}
```

Конструкторы для встроенных типов используется прежде всего для аргументов шаблонов. Например:

```
template<class T>
struct Handle {
    T* p;
    Handle(T* pp = new T{}): p{pp} {}
    // ...
};

Handle<int> px;     // будет сгенерировано int{}
```

Сгенерированный **int** будет проинициализирован значением **0**.

Ссылки и константные объекты должны быть инициализированы (§7.7, §7.5). Поэтому класс, содержащий такие члены, не может инициализироваться по умолчанию, кроме как если он предоставляет внутриклассовые инициализаторы членов (§17.4.4) или определяет умолчательный конструктор, инициализирующий эти члены (§17.4.1). Пример:

```
int glob {9};

struct X {
    const int a1 {7};      // OK
    const int a2;          // ошибка: требуется пользовательский конструктор
    const int& r {9};      // OK
    int& r1 {glob};        // OK
    int& r2;               // ошибка: требуется пользовательский конструктор
};

X x;            // error: no default constructor for X
```

Массив, стандартный **vector** и аналогичные контейнеры могут объявляться так, чтобы аллоцировать некоторое число элементов с умолчательной инициализацией. В таких случаях умолчательный конструктор однозначно требуется для классов, служащих типом элементов вектора **vector** или массива. Например:

```
struct S1 {S1();};                  // есть умолчательный конструктор
struct S2 {S2(string);};            // нет умолчательного конструктора

S1 a1[10];                          // OK: 10 умолчательных элементов
S2 a2[10];                          // ошибка: невозможно проинициализировать элементы
S2 a3[] {"alpha", "beta"};          // OK: два элемента: S2{"alpha"}, S2{"beta"}

vector<S1> v1(10);                  // OK: 10 умолчательных элемента
vector<S2> v2(10);                  // ошибка: невозможно проинициализировать элементы
vector<S2> v3 {"alpha", "beta"};    // OK: два элемента: S2{"alpha"}, S2{"beta"}

vector<S2> v2(10,"");               // OK: 10 эл-ов, каждый инициализируется в S2{""}
vector<S2> v4;                      // OK: нет элементов
```

Когда класс должен иметь умолчательный конструктор? Простой ответ — «когда он используется в качестве типа элементов массива и т.д.». Но более разумный вопрос звучит так: «для каких типов целесообразно определять умолчательное значение»? Или даже «есть для данного типа такое специальное значение, которое можно естественным образом использовать как умолчательное»? Для строк это пустая строка, у контейнеров может быть пустой набор элементов {}, у чисел есть **0**. Проблема выбора умолчания для типа **Date** (§16.3) возникает из-за того, что для этого типа трудно решить, какое значение является специальным. Вообще, лучше не слишком умничать в выборе умолчаний. Например, проблему контейнеров элементов, не имеющих разумных умолчательных значений, лучше всего решить таким образом, чтобы аллоцировать элементы только после того, как для них найдутся подходящие значения.

17.3.4. Конструкторы со списком инициализаторов

Конструктор, который принимает единственный аргумент типа **std::initializer_list**, называется конструктором со списком инициализаторов. Такой конструктор используется для конструирования объектов, используя {}-список в качестве инициализатора. Контейнеры стандартной библиотеки (например **vector** и **map**) обладают конструкторами со списком инициализаторов (§31.3.2, §31.4.3). Рассмотрим следующее:

```
vector<double> v = {1, 2, 3.456, 99.99};

list<pair<string, string>> languages = {
    {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};

map<vector<string>, vector<int>> years = {
    {{"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000}},
    {{"Martin", "Richards"} {1982, 2003, 2007}},
    {{"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004}}
};
```

Механизмом для принятия {}-списка служит функция (часто конструктор), принимающая аргумент типа **std::initializer_list<T>**. Пример:

```
void f(initializer_list<int>);

f({1,2});
f({23,345,4567,56789});
f({});          // пустой список

f{1,2};         // ошибка: отсутствуют () для вызова функции

years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

Список инициализаторов может иметь произвольную длину, но он обязан быть однородным. То есть, все элементы должны иметь тип аргумента шаблона, **T**, или же быть неявно приводимыми к **T**.

17.3.4.1. Разрешение неоднозначностей для конструкторов со списком инициализаторов

Когда у класса есть несколько конструкторов, для выбора одного из них используются обычные правила разрешения перегрузки (§12.3). При этом умолчательный конструктор и конструктор со списком инициализаторов имеют приоритет. Рассмотрим следующее:

```
struct X {
    X(initializer_list<int>);
    X();
    X(int);
};

X x0 {};  // пустой список (конструктор по умолчанию)
X x1 {1}; // одно целое (конструктор со списком инициализаторов)
```

Правила таковы:

- Если подходят и умолчательный конструктор, и конструктор со списком инициализаторов, то приоритет отдается первому.
- Если подходят конструктор со списком инициализаторов и обычный конструктор, то приоритет отдается конструктору со списком инициализаторов.

Первое правило, «предпочитаем умолчательный конструктор», есть просто отражение здравого смысла: следует выбрать простейший вариант из всех подходящих.

Второе правило, «предпочитаем конструктор со списком инициализаторов», необходимо во избежание разных результатов разрешения на основе разного числа элементов. Рассмотрим **std::vector** (§31.4):

```
vector<int> v1 {1};          // один элемент
vector<int> v2 {1,2};        // два элемента
vector<int> v3 {1,2,3};      // три элемента

vector<string> vs1 {"one"};
vector<string> vs2 {"one", "two"};
vector<string> vs3 {"one", "two", "three"};
```

Во всех случаях выбирается конструктор со списком инициализаторов. Если бы нам нужно было, чтобы выбирались конструкторы с одним или двумя аргументами, то нужно было применить ()-нотацию:

```
vector<int> v1(1);       // один элемент с умолчательным значением (0)
vector<int> v2(1,2);     // один элемент со значением 2
```

17.3.4.2. Использование initializer_list

Функция с аргументом типа **initializer_list** может осуществить доступ к нему как к последовательности, используя функции-члены **begin()**, **end()** и **size()**. Пример:

```
void f(initializer_list<int> args)
{
    for (int i = 0; i!=args.size(); ++i)
        cout << args.begin()[i] << "\n";
}
```

К сожалению, **initializer_list** не предоставляет возможности индексирования.

Аргумент типа **initializer_list** передается по значению. Это требование правил разрешения перегрузки (§12.3) и оно не приводит к излишним накладным расходам, поскольку объект типа **initializer_list<T>** является небольшим дескриптором, через который идет обращение к массиву элементов типа **T**.

Цикл можно эквивалентно переписать в следующем виде:

```
void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p)
        cout << *p << "\n";
}
```

или

```
void f(initializer_list<int> args)
{
    for (auto x : args)
        cout << x << "\n";
}
```

Чтобы в явном виде использовать тип **initializer_list**, нужно в исходный файл включать директивой **#include** заголовочный файл, в котором он определен: **<initializer_list>**. Но поскольку такие распространенные стандартные контейнеры, как **vector**, **map** и т.д. используют **initializer_list**, то их заголовочные файлы (**<vector>**, **<map>** и т.д.) уже включают упомянутый выше заголовочный файл, так что явно самим его нужно включать не так уж и часто.

Элементы у типа **initializer_list** неизменяемы. Даже не думайте пробовать изменять их значения. Например:

```
int f(std::initializer_list<int> x, int val)
{
    *x.begin() = val;    // ошибка: попытка изменить знач. эл-та списка инициализаторов
    return *x.begin();   // OK
}
```

```
void g()
{
    for (int i=0; i!=10; ++i)
        cout << f({1,2,3}, i) << '\n';
}
```

Если бы присваивание в f() увенчалось успехом, то получилось бы, что 1 из {1,2,3} подверглась изменению. Это нанесло бы серьезный урон некоторым фундаментальным концепциям. Поскольку элементы у initializer_list неизменяемы, то к ним нельзя применять перемещающий конструктор (§3.3.2, §17.5.2).

Контейнер может реализовывать конструктор со списком инициализаторов следующим образом:

```
template<class E>
class Vector {
public:
    Vector(std::initializer_list<E> s);    // конструктор со списком инициализаторов
    // ...
private:
    int sz;
    E* elem;
};

template<class E>
Vector::Vector(std::initializer_list<E> s)
    : sz{s.size()}                                  // устанавливаем размер вектора
{
    reserve(sz);                                    // резервируем надлежащий объем памяти
    uninitialized_copy(s.begin(), s.end(), elem);   // инициализир. эл-ты в elem[0:s.size())
}
```

Списки инициализаторов являются частью дизайна универсальной и однородной инициализации (§17.3).

17.3.4.3. Прямая инициализация и инициализация копированием

Различие между прямой инициализацией и инициализацией копированием (§16.2.6) сохраняется и для {}-инициализации. Для контейнера это означает, что различие имеет место как для контейнера самого, так и для элементов:

- У контейнера конструктор со списком инициализаторов может быть объявлен как с ключевым словом **explicit**, так и без него.
- Конструктор типа элементов в списке инициализаторов может быть или не быть **explicit**.

Для vector<vector<double>> можно увидеть различие между прямой инициализацией и инициализацией копированием применительно к элементам. Например:

```
vector<vector<double>> vs = {
    {10,11,12,13,14},           // OK: вектор из пяти эл-ов
    {10},                       // OK: вектор из одного эл-та
    10,                         // ошибка: vector<double>(int) задан явно

    vector<double>{10,11,12,13}, // OK: вектор из пяти эл-ов
    vector<double>{10},         // OK: вектор из одного элемента со значением 10.0
    vector<double>(10),         // OK: вектор из 10 элементов со значением 0.0
};
```

У контейнера некоторые из конструкторов могут быть **explicit**. Здесь примером может быть стандартный **vector**. Например, **std::vector<int>(int)** является **explicit**, а **std::vector<int>(initialize_list<int>)** не является **explicit**:

```
vector<double> v1(7);          // OK: v1 имеет 7 элементов ( используйте (), а не {} )
vector<double> v2 = 9;         // ошибка: нет конверсии int в vector

void f(const vector<double>&);
void g()
{
    v1 = 9;                    // ошибка: нет конверсии int в vector
    f(9);                      // ошибка: нет конверсии int в vector
}
```

Заменяя () на {}, получаем:

```
vector<double> v1 {7};         // OK: у v1 один элемент (со значением 7)
vector<double> v2 = {9};       // OK: у v2 один элемент (со значением 9)

void f(const vector<double>&);
void g()
{
    v1 = {9};                  // OK: теперь у v1 один элемент (со значением 9)
    f({9});                    // OK: f вызывается со списком {9}
}
```

Очевидно, что результаты кардинально отличаются.

Данный пример специально составлен так, чтобы продемонстрировать наиболее «смутные» случаи. Отметим, что очевидные случаи недоразумений не возникают из-за длинных списков. Например:

```
vector<double> v1 {7,8,9};     // OK: у v1 три элемента со значениями {7,8,9}
vector<double> v2 = {9,8,7};   // OK: у v2 три элемента со значениями {9,8,7}

void f(const vector<double>&);
void g()
{
    v1 = {9,10,11};            // OK: у v1 теперь три элемента со значениями {9,10,11}
    f({9,8,7,6,5,4});          // OK: f вызывается со списком {9,8,7,6,5,4}
}
```

Аналогично, потенциальные неоднозначности не возникают для элементов списка неинтегральных типов:

```
vector<string> v1 {"Anya"};          // OK: у v1 один элемент (со значением "Anya")
vector<string> v2 = {"Courtney"};    // OK: у v2 один элемент (со значением "Courtney")

void f(const vector<string>&);
void g()
{
    v1 = {"Gavin"};                  // OK: у v1 теперь один элемент (со значением "Gavin")
    f({"Norah"});                    // OK: f вызывается со списком {"Norah"}
}
```

17.4. Инициализация членов и инициализация базового класса

Конструкторы могут устанавливать инварианты и захватывать ресурсы. Чаще всего они делают это посредством инициализации членов класса и базовых классов.

17.4.1. Инициализация членов

Рассмотрим класс, который мог бы использоваться, чтобы хранить информацию для небольшой организации:

```
class Club {
    string name;
    vector<string> members;
    vector<string> officers;
    Date founded;
    // ...
    Club(const string& n, Date fd);
};
```

Конструктор класса **Club** берет имя клуба и дату его основания в виде аргументов. Аргументы для конструкторов членов специфицированы в списке инициализаторов членов в определении конструктора класса, содержащего члены. Пример:

```
Club::Club(const string& n, Date fd)
    : name{n}, members{}, officers{}, founded{fd}
{
    // ...
}
```

Список инициализаторов членов начинается с двоеточия, а индивидуальные инициализаторы членов разделяются запятыми.

Конструкторы членов вызываются до исполнения тела конструктора содержащего члены класса (§17.2.3). Эти конструкторы вызываются в порядке, в котором члены объявляются в классе, а не в порядке, в котором члены расположены в списке инициализаторов. Будет правильным, если вы во избежание недоразумений расположите члены в списке инициализаторов в том же порядке, в котором члены объявляются в классе. Деструкторы членов вызываются в порядке, обратном к вызову их конструкторов и после исполнения тела деструктора содержащего класса.

Если конструктор члена не нуждается в аргументах, то член не требует упоминания в списке инициализаторов членов. Пример:

```
Club::Club(const string& n, Date fd)
    : name{n}, founded{fd}
{
    // ...
}
```

Этот конструктор эквивалентен предыдущей его версии. В обоих случаях, **Club::officers** и **Club::members** инициализируются вектором без элементов.

Вообще хорошей идеей является явная инициализация членов. Не забудьте, что неявно проинициализированные члены встроенных типов остаются фактически неинициализированными (§17.3.1).

Конструктор может инициализировать члены и базовые классы своего класса, но не члены или базовые классы своих членов и базового класса. Например:

```
struct B {B(int); /* ... */};
struct BB : B {/* ... */};
struct BBB : BB {
    BBB(int i) : B(i) {}; // ошибка: попытка инициализации базы у базы
    // ...
};
```

17.4.1.1. Инициализация членов и присваивание

Инициализаторы членов существенны для типов, для которых смысл инициализации отличается от присваивания. Пример:

```
class X {
    const int i;
    Club cl;
    Club& rc;
    // ...
    X(int ii, const string& n, Date d, Club& c) : i{ii}, cl{n, d}, rc{c} {}
};
```

Член-ссылка или константный член обязаны инициализироваться (§7.5, §7.7, §17.3.3). Однако, для большинства типов у программиста есть выбор между применением инициализатора и использованием присваивания. В таком случае я предпочитаю синтаксис инициализаторов членов, чтобы явным образом отразить тот факт, что выполняется инициализация. Часто это дает и выигрыш в эффективности по сравнению с присваиванием. Например:

```
class Person {
    string name;
    string address;
    // ...
    Person(const Person&);
    Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
    : name{n}
{
    address = a;
}
```

Здесь **name** инициализируется копией **n**. С другой стороны, **address** сначала инициализируется пустой строкой, а затем ему присваивается копия **a**.

17.4.2. Инициализаторы базового класса

Если базовые классы требуют инициализаторов, они должны быть предоставлены в виде инициализаторов базы в конструкторе. При желании можно явно специфицировать умолчательное конструирование. Например:

```
class B1 {B1();};              // есть умолчательный конструктор
class B2 {B2(int);}            // нет умолчательного конструктора

struct D1 : B1, B2 {
    D1(int i): B1{}, B2{i} {}
};

struct D2 : B1, B2 {
    D2(int i): B2{i} {}        // B1{} используется неявно
};

struct D1 : B1, B2 {
    D1(int i) {}               // ошибка: B2 требует инициализатора типа int
};
```

Как и для членов, порядок инициализации соответствует порядку в объявлении, и рекомендуется располагать инициализаторы базовых классов в том же самом порядке (во избежание недоразумений). Базовые классы инициализируются до инициализации членов, а уничтожаются после уничтожения членов (§17.2.3).

17.4.3. Делегирующие конструкторы

Если два конструктора выполняют одинаковое действие, то вы можете все это повторить в коде, или определить некую функцию **init()**, которая выполнит повторяющееся действие. Оба таких решения распространены, ибо предыдущие версии языка C++ не предлагали ничего лучше. Пример:

```
class X {
    int a;
    validate(int x) {if (0<x && x<=max) a=x; else throw Bad_X(x);}
public:
    X(int x) {validate(x);}
    X() {validate(42);}
    X(string s) {int x = to<int>(s); validate(x);} // §25.2.5.1
    // ...
};
```

Альтернативой может послужить определение конструктора в терминах другого конструктора:

```
class X {
    int a;
public:
    X(int x) {if (0<x && x<=max) a=x; else throw Bad_X(x);}
    X(): X{42} {}
    X(string s): X{to<int>(s)} {}    // §25.2.5.1
    // ...
};
```

То есть, инициализатор члена, используя имя класса (имя конструктора), вызывает другой конструктор как часть инициализации. Такой конструктор принято называть делегирующим конструктором.

Вы не можете одновременно и делегировать и явно инициализировать член. Например:

```
class X {
    int a;
public:
    X(int x) {if (0<x && x<=max) a=x; else throw Bad_X(x);}
    X(): X{42}, a{56} {} // ошибка
    // ...
};
```

Делегирование вызовом другого конструктора в списке инициализаторов членов и базовых классов сильно отличается от явного вызова конструктора в теле конструктора. Рассмотрим следующее:

```
class X {
    int a;
public:
    X(int x) {if (0<x && x<=max) a=x; else throw Bad_X(x);}
    X() {X{42};} // вероятно ошибка
    // ...
};
```

Выражение **X{42}** просто создает новый неименованный объект (временный) и ничего с ним не делает.

Объект не считается сконструированным до тех пор, пока не закончит свою работу конструктор (§6.4.2). В случае использования делегирующего конструктора объект не считается сконструированным до тех пор, пока не завершит работу делегирующий конструктор — недостаточно завершения работы конструктора, которому была делегирована работа. Деструктор не будет вызван для объекта, если оригинальный конструктор не отработал штатно.

Если все, что вам нужно, это просто установить умолчательное значение члена (что не зависит от аргументов конструктора), то проще использовать инициализатор члена (§17.4.4).

17.4.4. Внутриклассовые инициализаторы

Можно указать инициализатор для нестатического члена данных прямо в рамках объявления класса. Пример:

```
class A {
public:
    int a {7};
    int b = 77;
};
```

По техническим причинам можно использовать {}-нотацию и = -нотацию инициализаторов, но нельзя использовать ()-нотацию.

По умолчанию, конструктор будет использовать эти внутриклассовые инициализаторы, так что приведенный пример эквивалентен следующему:

```
class A {
public:
    int a;
    int b;
    A() : a{7}, b{77} {}
};
```

Такое использование внитриклассовых инициализаторов может сэкономить силы на уменьшенном объеме набора исходного текста, но настоящее преимущество внутриклассовых инициализаторов обнаруживается в более сложных примерах со множественными конструкторами. Часто несколько конструкторов используют один и тот же инициализатор для члена. Например:

```
class A {
public:
    A(): a{7}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
    A(int a_val): a{a_val}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
    A(D d): a{7}, b{g(d)}, algorithm{"MD5"}, state{"Constructor run"} {}
    // ...
private:
    int a, b;
    HashFunction algorithm;      // криптографичвский хэш для всех A
    string state;                // строка, указыв. сост-е объекта в его жизненном цикле
};
```

Тот факт, что **algorithm** и **state** имеют одно и то же значение во всех конструкторах, может потеряться в большом коде и стать проблемой при поддержке такого кода. Чтобы явно отразить факт одинаковых значений, можно вынести его в отдельное место:

```
class A {
public:
    A(): a{7}, b{5} {}
    A(int a_val): a{a_val}, b{5} {}
    A(D d): a{7}, b{g(d)} {}
    // ...
private:
    int a, b;
    HashFunction algorithm {"MD5"};  // криптогр. хэш для всех A
    string state {"Constructor run"};  // строка сост-я объекта в его жизненном цикле
};
```

Если для члена имеется внутриклассовый инициализатор, но он также устанавливается и конструктором, то выполняется лишь инициализация конструктором. Так что можно продолжить упрощения:

```
class A {
public:
    A() {}
    A(int a_val): a{a_val} {}
    A(D d): b{g(d)} {}
    // ...
private:
    int a {7};                          // смысл 7 для a есть ...
    int b {5};                          // смысл 5 для b есть ...
    HashFunction algorithm {"MD5"};  // криптогр. хэш для всех A
    string state {"Constructor run"};  // строка сост-я жизн. цикла объекта
};
```

Как видно, умолчательные внутриклассовые инициализаторы помогают документировать наиболее общие случаи.

Внутриклассовые инициализаторы членов могут использовать имена, которые находятся в области видимости в точке их использования в объявлении члена. Рассмотрим следующий замысловатый технический пример:

```
int count = 0;
int count2 = 0;

int f(int i) {return i+count;}

struct S {
    int m1 {count2};    // то есть, ::count2
    int m2 {f(m1)};     // то есть, this->m1+::count; и, ::count2+::count
    S() {++count;}      // очень странный конструктор
};

int main()
{
    S s1;     // {0,0}
    ++count;
    S s2;     // {1,2}
}
```

Инициализация членов выполняется в порядке их объявления (§17.2.3), так что **m1** инициализируется значением глобальной переменной **count2**. Значение глобальной переменной берется в точке, где запускается конструктор для нового объекта типа **S**, так что оно может измениться (и в данном примере так и есть). Затем, **m2** инициализируется вызовом глобальной функции **f()**.

Вообще-то, это плохая идея прятать тонкие зависимости от глобальных данных в инициализаторы членов.

17.4.5. Инициализация статических членов

Статический член класса аллоцируется статически, а не как часть каждого объекта класса. И вообще, объявление статического члена действует так же, как объявление для определения вне класса. Например:

```
class Node {
    // ...
    static int node_count;    // объявление
};

int Node::node_count = 0;     // определение
```

Тем не менее, за исключением нескольких специальных случаев можно проинициализировать статический член в рамках объявления класса. Статический член должен быть квалифицирован ключевым словом **const** и быть интегрального типа или перечислением, или же **constexpr** литерального типа (§10.4.3), а инициализатор должен быть константным выражением. Пример:

```
class Curious {
public:
    static const int c1 = 7;          // OK
    static int c2 = 11;               // ошибка: не const
    const int c3 = 13;                // OK, но не static (§ 17.4.4)
    static const int c4 = sqrt(9);    // ошибка: внутрикласс. инициализатор не константа
    static const float c5 = 7.0;      // ошибка: используйте constexpr, а не const
    // ...
};
```

Если только вы используете инициализированный член способом, требующим, чтобы он хранился в памяти как объект, то тогда член должен быть уникальным образом где-нибудь определен. Инициализатор можно не повторять:

```
const int Curious::c1;            // не повторяйте инициализатор здесь
const int* p = &Curious::c1;      // OK: Curious::c1 уже был определен
```

Главной целью констант-членов является задача обеспечить константы символическими именами, необходимыми в других местах объявления класса. Например:

```
template<class T, int N>
class Fixed {  // массив фиксированного размера
public:
    static constexpr int max = N;
    // ...
private:
    T a[max];
};
```

Для целых, энумераторы (§8.4) предлагают альтернативу определения символических констант в рамках объявления класса. Пример:

```
class X {
    enum {c1 = 7, c2 = 11, c3 = 13, c4 = 17};
    // ...
};
```

17.5. Копирование и перемещение

Когда нам нужен трансфер значения из **a** в **b**, у нас обычно есть две опции:

- Копирование — это стандартная трактовка операции **x=y**; эффект состоит в том, что значения **x** и **y** становятся равны значению **y** до выполнения операции присваивания.
- Перемещение — оставляет **x** с предыдущим значением **y**, а **y** — в состоянии moved from («опустошенным»). Для наиболее интересных случаев, контейнеров, это «опустошенное» состояние есть «пусто».

Небольшая логическая несостыковка состоит в том, что для обоих случаев мы используем одинаковую нотацию.

В типичном случае операция перемещения не может генерировать исключений, в то время как операция копирования может (поскольку она часто испытывает нужду в захвате ресурса), но операция перемещения часто более эффективна, чем копирование. Когда вы программируете операцию перемещения, вы должны оставлять объект-источник в корректном (пусть и неспецифицированном) состоянии, поскольку этот объект со временем должен быть уничтожен, а деструктор не сможет работать с объектом в некорректном состоянии. Кроме того, алгоритмы стандартной библиотеки должны быть в состоянии в эти опустошенные объекты совершать присваивания (или перемещения). Таким образом, проектируя операции перемещения, не пытайтесь генерировать исключения, а также оставляйте объекты-источники в состоянии, для которого возможна работа деструктора и операции присваивания.

Для копирования и перемещения имеются умолчательные реализации (§17.6.2), что спасает нас от утомительной работы в ряде случаев.

17.5.1. Копирование

Копирование для класса реализуется двумя операциями:

- Копирующим конструктором: **X(const X&)**
- Копирующим присваиванием: **X& operator=(const X&)**

Вы можете определить эти две операции с более проблемными типами аргументов, такими как **volatile X&**, но все же не стоит; вы только навредите себе и другим. Предполагается, что копирующий конструктор создает копию объекта без его модификации. Вы можете использовать тип **const X&** в качестве возврата операции копирующего присваивания. В общем, вы можете допускать отклонения, но я считаю, что лучше, чтобы две рассматриваемые операции имели стандартные типы.

Рассмотрим простой пример двумерной матрицы:

```
template<class T>
class Matrix {
    array<int,2> dim;                    // два измерения
    T* elem;                             // указатель на dim[0]*dim[1] эл-ов типа T
public:
    Matrix(int d1, int d2): dim{d1, d2}, elem{new T[d1*d2]} {}  // без обработки ошибок
    int size() const {return dim[0]*dim[1];}

    Matrix(const Matrix&);               // копирующий конструктор
    Matrix& operator=(const Matrix&);    // присваивание копированием

    Matrix(Matrix&&);                    // перемещающий конструктор
    Matrix& operator=(Matrix&&);         // перемещающее присваивание

    ~Matrix() {delete[] elem;}
    // ...
};
```

Сначала обратим внимание на то, что умолчательное копирование (почленное копирование) будет иметь катастрофические последствия: элементы **Matrix** не будут скопированы, копия объекта **Matrix** будет ссылаться на те же самые элементы, что и у исходной матрицы, а деструктор будет дважды удалять одни и те же элементы (§3.3.1).

На самом деле программист может придать любой смысл операциям копирования, а традиционный вариант копирования для контейнеров состоит в копировании его элементов:

```
template<class T>
Matrix::Matrix(const Matrix& m)                          // копирующий конструктор
    : dim{m.dim},
    elem{new T[m.size()]}
{
    uninitialized_copy(m.elem, m.elem+m.size(), elem);   // копирование элементов
}

template<class T>
Matrix& Matrix::operator=(const Matrix& m)               // присваивание копированием
{
    if (dim[0]!=m.dim[0] || dim[1]!=m.dim[1])
        throw runtime_error("bad size in Matrix =");
    copy(m.elem, m.elem+m.size(), elem);                 // копируем элементы
}
```

Копирующий конструктор и копирующее присваивание отличаются тем, что конструктор должен инициализировать ранее неинициализированную память, в то время как копирующая операция присваивания должна корректно работать с объектом, который ранее уже был сконструирован и может обладать ресурсами.

Для типа **Matrix** у копирующей операции присваивания есть такая особенность, что если при копировании элемента генерируется исключение, то целевой объект операции может оказаться в смешанном состоянии — будет смесь из старых значений и новых. То есть присваивание для **Matrix** поддерживает базовую гарантию, но не строгую гарантию (§13.2). Если такое поведение неприемлемо, то можно обратиться к фундаментальной технике, когда сначала делается копия, а потом идет обмен представлениями:

```
Matrix& Matrix::operator=(const Matrix& m)  // присваивание копированием
{
    Matrix tmp {m};                         // получаем копию
    swap(tmp,*this);                        // свопируем tmp-представление и предст-е *this
    return *this;
}
```

А вызов **swap()** будет выполнен лишь в случае успеха создания копии. Очевидно, что этот вариант **operator=()** работает лишь в случае, если **swap()** не использует присваивания (**std::swap()** не использует); см. §17.5.2.

Обычно копирующий конструктор должен скопировать все нестатические члены (§17.4.1). Если копирующему конструктору не удается скопировать элемент (например, из-за нехватки необходимого ресурса), он может сгенерирвать исключение.

Обратите внимание на то, что я не стал защищать копирующее присваивание для **Matrix** от самоприсваивания, **m=m**. Это объясняется тем, что самоприсваивание элементов безопасно. Кроме того, специально обрабатывать редкий случай нужно лишь тогда, когда вы точно уверены, что это нужно сделать.

17.5.1.1. Опасайтесь умолчательных конструкторов

Когда вы программируете копирующую операцию, не забывайте копировать все члены и базовые классы. Рассмотрим следующее:

```
class X {
        string s;
        string s2;
        vector<string> v;

        X(const X&)             // копирующий конструктор
            : s{a.s}, v{a.v}    // вероятно ошибка
        {
        }
        // ...
};
```

Здесь я «забыл» скопировать член **s2**, так что он получил умолчательное значение "". Это неправильно (скорее всего). И маловероятно, что я сделал бы на практике такую ошибку в столь маленьком примере. Но в случае больших классов это вполне возможно. Еще хуже то, что в процессе поддержки и модификации программы кто-то добавляет новый член, но забывает внести его в список членов для копирования. В этом состоит причина предпочтения умолчательных копирующих операций, генерируемых компилятором (§17.6).

17.5.1.2. Копирование базовых классов

С точки зрения копирования, базовый класс есть просто член: копируя объект производного класса, не нужно забывать скопировать и его часть от базового класса. Пример:

```
struct B1 {
        B1();
        B1(const B1&);
        // ...
};

struct B2 {
        B2(int);
        B2(const B2&);
        // ...
};

struct D : B1, B2 {
        D(int i): B1{}, B2{i}, m1{}, m2{2*i} {}
        D(const D& a): B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}
        B1 m1;
        B2 m2;
};

D d {1};        // конструируем с аргументом типа int
D dd {d};       // копирующий конструктор
```

Порядок инициализации обычный (базовый класс перед членами), но для копирования хорошо бы сделать этот процесс независимым от порядка.

Виртуальный базовый класс (§21.3.5) может оказаться базовым классом нескольких классов в иерархии. Умолчательный копирующий конструктор (§17.6) будет корретно выполнять копирование. Если вы пожелаете определить свой собственный копирующий конструктор, то стандартной техникой является повторяющееся копирование виртуального базового класса. Когда базовый объект небольшой, а виртуальный базовый класс повторяется не слишком часто в иерархии, то такая техника может оказаться эффективнее попытки избежать повторяющихся копирований.

17.5.1.3. Смысл копирования

Что должны делать копирующий конструктор и копирующее присваивание для того, чтобы считаться правильными копирующими операциями? Помимо правильного объявления типа, должна быть еще и правильная семантика. Рассмотрим копирование **x=y** двух объектов одного и того же типа. Чтобы быть совместимым с ориентированным на значения программированием (§16.3.4) и, что еще важнее, использоваться согласованно со стандартной библиотекой (§31.2.2), операция должна удовлетворять двум критериям;

- Эквивалентность: После выполнения операции **x=y** остальные операции над **x** и **y** должны приводить к одинаковому результату. В частности, если для типа определена операция ==, то должно быть **x==y** и **f(x)==f(y)** для любой функции **f()**, которая зависит лишь от величин **x** и **y** (а не от их адресов).
- Независимость: После выполнения операции **x=y** операции над **x** не должны неявным образом модифицировать состояние **y**, например **f(x)** не должно изменять значения **y** до тех пор, пока **f(x)** не ссылается на **y**.

Такое поведение демонстрируют типы **int** и **vector**. Копирующие операции, реализующие свойства эквивалентности и независимости ведут к более простому коду, который проще поддерживать. Однако, весьма распространен код, нарушающий эти правила, а программисты не до конца осознают, что в таком коде и кроются причины их затруднений. Рассмотренные нами понятия эквивалентности и независимости являются частью характеристик регулярного типа (§24.3.1).

Сначала обратим внимание на эквивалентность. Это свойство люди редко нарушают намеренно и умолчательные копирующие операции не нарушают его; они выполняют почленное копирование (§17.3.1, §17.6.2). Однако трюки, опирающиеся на зависимость копирования от «опций», время от времени появляются на свет и вызывают массу недоразумений. Также нередко объекты содержат члены, которые не рассматриваются в качестве части его значения. Например, при копировании стандартного контейнера его аллокатор не копируется, ибо он считается частью именно контейнера, а не его значения. Такие части объектов, не являющиеся частью их значений, не должны влиять на результат операций сравнения. В особенности, **x=y** должно предполагать **x==y**. Более того, срезка (§17.5.1.4) может приводить к «копиям», ведущим себя отличным образом, что чаще всего является серьезной ошибкой.

Теперь рассмотрим требование независимости. Большая часть случаев, имеющих проблемы с независимостью, имеют отношение к объектам, содержащим указатели. Умолчательным смыслом копирования является почленное копирование. При этом копируются указатели, а не объекты, на которые они указывают. Например:

```
struct S {
    int* p;     // указатель
};

S x {new int{0}};

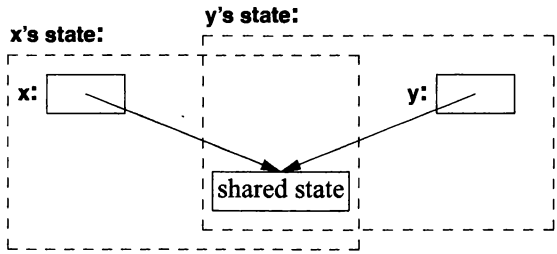
void f()
{
    S y{x};              // "копируем" x

    *y.p = 1;            // изменяем y; влияет на x
    *x.p = 2;            // меняем x; влияет на y
    delete y.p;          // влияет на x и y
    y.p = new int{3};    // OK: меняем y; не влияет на x
    *x.p = 4;            // упс: пишем в неаллоцированную память
}
```

Здесь я нарушил правило независимости. После копирования **x** в **y** мы можем манипулировать частью состояния **x** через **y**. Часто это называют поверхностным копирование (shallow copy), и тоже часто, к сожалению, превозносят его за эффективность. Соответственно, альтернативу в виде полного копирования, называют глубоким копированием (deep copy). А реальной альтернативой глубокому копированию является вовсе не поверхностное копирование. Им является перемещающая операция, которая минимизирует копирование без увеличения сложности (§3.3.2, §17.5.2).

Поверхностное копирование оставляет два объекта (здесь это **x** и **y**) в «спутанном» состоянии, которое разделяется между двумя объектами, что и служит источником ошибок. В итоге, об этих объектах нельзя рассуждать по-отдельности. К примеру, из исходного кода не так просто понять, что два присваивания объекту ***x.p** могут иметь совершенно разные результаты.

Два «спутанных» объекта можно отобразить графически следующим образом:



Отметим, что «спутанность» может возникать по разным причинам. Часто трудно вскрыть «спутанность» до тех пор, пока не проявят себя проблемы, возникающие из-за «спутанности». Например, тип, вроде **S**, может по неосторожности быть использован в определении другого, вполне корректного в остальном отношении классе. Оригинальный автор типа **S** мог быть готов использовать этот тип с полным знанием дела, а вот тот, кто наивно поверил в этот тип, может счесть копирование его полного значения приемлемым, после чего столкнется с неприятным сюрпризом.

С проблемой времени жизни разделенного подобъекта можно справиться введением некоторой формы «сборки мусора». Например:

```
struct S2 {
    shared_ptr<int> p;
};

S2 x {new int{0}};

void f()
{
    S2 y {x};                   // "копирование" x

    *y.p = 1;                   // меняем y, влияет на x
    *x.p = 2;                   // меняем x; влияет на y
    y.p.reset(new int{3});      // меняем y; влияет на x
    *x.p = 4;                   // меняем x; влияет на y
}
```

Операция поверхностного копирования и результирующие спутанные объекты служат одной из основных причин для введения сборки мусора. Спутанные объекты порождают код, которым чрезвычайно трудно управлять без некоторой формы сборки мусора (например, **shared_ptr**).

В то же время, **shared_ptr** это по-прежнему указатель, так что объекты, содержащие **shared_ptr**, невозможно рассматривать в полной изоляции. Кто может и кто должен обнов-

лять указуемый объект? Как? Когда? Если же мы имеем дело с многопоточной системой, то нужна ли синхронизация для доступа к разделяемым данным? В общем, спутанные объекты (например, порождаемые поверхностным копированием) служат источником чрезмерной сложности и ошибок, с которыми можно лишь частично справиться при помощи сборки мусора.

Заметьте, что неизменяемое разделенное состояние не создает проблемы. Если мы не сравниваем адреса, то мы не сможем выявить, что два одинаковых значения представлены в двух копиях. Например, объекты, передаваемые по значению, редко подвергаются записи. Отсюда вытекает техника копирование-по-записи: при копировании независимость не требуется до тех пор, пока не настанет время записи в разделяемое состояние, так что можно отложить полное копирование до такого момента. Рассмотрим следующее:

```
class Image {
public:
    // ...
    Image(const Image& a);      // копирующий конструктор
    // ...
    void write_block(Descriptor);
    // ...
private:
    Representation* clone();    // копируем *rep
    Representation* rep;
    bool shared;
};
```

Предположим, что **Pepresentation** может быть объемным и что **write_block()** является дорогостоящей операцией по сравнению с проверкой на **bool**. Тогда, в зависимости от вариантов использования **Image** может иметь смысл реализовать копирующий конструктор посредством поверхностного копирования:

```
Image::Image(const Image& a) // поверх-е копир-е и подготовка к копированию по записи
    : rep{a.rep},
    shared{true}
{
}
```

Мы защищаем аргумент этого копирующего конструктора посредством копирования **Representation** перед записью:

```
void write_block(Descriptor d)
{
    if (shared) {
        rep =clone();       // копия *rep
        shared = false;     // больше нет совместного использования
    }
    // ... можем безопасно писать в собственную копию rep ...
}
```

Как и любые иные вспомогательные приемы, техника копирование-по-записи не является панацеей. Она является эффективной комбинацией простоты настоящего копирования и эффективности поверхностного копирования.

17.5.1.4. Срезка

Указатель на производный класс неявно конвертируется в указатель на базовый класс. Применительно к копирующей операции это простое правило (§3.2.4, §20.2) приводит в ловушку. Рассмотрим следующее:

```
struct Base {
    int b;
    Base(const Base&);
    // ...
};

struct Derived : Base {
    int d;
    Derived(const Derived&);
    // ...
};

void naive(Base* p)
{
    B b2 = *p;  // возможна срезка: вызов Base::Base(const Base&)
    // ...
}

void user()
{
    Derived d;
    naive(&d);
    Base bb = d; // срезки: вызов Base::Base(const Base&), а не Derived::Derived(const Derived&)
    // ...
}
```

Переменные **b2** и **bb** содержат копии базовой части (типа **Base**) переменной **d**, то есть копии **d.b**. Член **d.d** не копируется. Этот феномен называется срезкой (slicing). Это, например, может быть именно тем, что вам нужно (например, как в случае копирующего конструктора типа **D** из §17.5.1.2, где мы передаем избранную информацию базовому классу), но в общем случае это еще один вид тонкой ошибки. Если вы хотите избежать срезки, то в вашем распоряжении два основных средства:

1. Запретить копирование базового класса: применить **delete** для копирующих операций (§17.6.4).
2. Предотвратить конвертацию указателя на производный класс в указатель на базовый класс: сделайте базовый класс **private** или **protected** (§20.5).

В первом случае инициализация **b2** и **bb** даст ошибку; во втором случае ошибочными будут вызов **naive()** и инициализация **bb**.

17.5.2. Перемещение

Традиционным способом придать переменной **b** значение переменной **a** является копирование. Для целого значения в памяти это может быть единственным, что имеет смысл; это как раз то, что аппаратура может выполнить посредством единственной инструкции. А вот с общей логической точки зрения дело обстоит иначе. Рассмотрим очевидную реализацию функции **swap()**, обменивающей два значения:

```
template<class T>
void swap(T& a, T& b)
{
    const T tmp = a;   // помещаем копию a в tmp
    a = b;             // копию b в a
    b =tmp;            // копию tmp в b
};
```

После инициализации **tmp** мы имеем две копии значения переменной **a**. После выполнения присваивания мы имеем две копии значения переменной **b**. После присваивания **b**

мы имеем две копии значения **tmp**. Затем мы уничтожаем **tmp**. Все это выглядит как большой объем работы, и часто так и бывает. Например:

```
void f(string& s1, string& s2,
       vector<string>& vs1, vector<string>& vs2,
       Matrix& m1, Matrix& m2)
{
       swap(s1, s2);
       swap(vs1.vs2);
       swap(m1, m2);
}
```

Что, если **s1** содержит тысячу символов? Что, если **vs2** имеет тысячу элементов, содержащих по тысяче символов каждый? А что, если **m1** является матрицей из 1000*1000 элементов типа **double**? Цена копирования таких сруктур данных может оказаться весьма немалой. Функция **swap()** из стандартной библиотеки тщательно запроектирована так, чтобы избегать таких накладных расходов для типов **string** и **vector** (используется факт, что **string** и **vector** являются дескрипторами для своих элементов). Но в общем случае оптимизацию придется делать для бесконенчного множества типов, которые создают пользователи.

Здесь самая суть проблемы состоит в том, что нам на самом деле не нужно копирование: нам просто нужно обменять значения.

Еще одна аналогия: вы просите у меня телефон позвонить и я отдаю его вам. Но я не создаю копию телефона для этого. Если я одалживаю машину, то я просто передаю ключи, а не создаю еще одну копию машины. Но как только я передал объект, я сам им больше не владею. Мы говорим об идеи перемещения владения или перемещения объектов. Многие компьютерные объекты напоминают физические объекты: блокировки, сокеты, файловые дескрипторы, потоки, длинные строки и большие вектора.

Чтобы помочь пользователю избежать логических проблем с копированием и повысить производительность, язык C++ вводит прямую поддержку перемещения, кроме традиционной поддержки копирования. В частности, мы теперь можем определить перемещающий конструктор и перемещающее присваивание, чтобы выполнять перемещение, а не копирование аргумента. Снова рассмотрим простую двумерную матрицу **Matrix** из §17.5.1:

```
template<class T>
class Matrix {
       std::array<int,2> dim;
       T* elem; // указатель на sz элементов типа T

       Matrix(int d1, int d2): dim{d1, d2}, elem{new T[d1*d2]} {}
       int size() const {return dim[0]*dim[1];}

       Matrix(const Matrix&);                  // копирующий конструктор
       Matrix(Matrix&&);                       // перемещающий конструктор

       Matrix& operator=(const Matrix&); // присваивание копированием
       Matrix& operator=(Matrix&&);      // присваивание перемещением

       ~Matrix(); // деструктор
       // ...
};
```

Обозначение **&&** специфицирует rvalue-ссылку (§7.7.2).

Идея, лежащая в основе перемещающего присваивания, состоит в желании работать с lvalue-значениями отдельно от rvalue-значений: копирующие присваивания и копирующие конструкторы для lvalue-значений и перемещающее присваивание и перемещающий конструктор для rvalue-значений. Для возвращаемых значений выбирается перемещающий конструктор.

Мы можем определить перемещающий конструктор для **Matrix** так, чтобы он брал представление источника и заменял его пустым **Matrix** (уничтожать который весьма недорого). Например:

```
template<class T>
Matrix<T>::Matrix(Matrix&& a)        // перемещающий конструктор
    : dim{a.dim}, elem{a.elem}       // захватываем представление у a
{
    a.dim = {0,0};                   // очищаем представление a
    a.elem = nullptr;
}
```

Для перемещающего присваивания можно просто делать обмен значениями. Причина состоит в том, что источник обмена вот-вот будет уничтожен, так что можно позволить деструктору выполнить за нас всю очистку:

```
template<class T>
Matrix<T>& Matrix<T>::operator=(Matrix&& a)      // присваивание перемещением
{
    swap(dim, a.dim);                            // обмен представлениями
    swap(elem, a.elem);
    return *this;
}
```

Перемещающие конструкторы и перемещающие присваивания принимают неконстантные (rvalue) ссылочные аргументы: они могут и часто реально пишут в свои аргументы. При этом аргументы перемещающих операций должны оставаться в состоянии, чтобы с ними могли работать деструкторы (и желательно, чтобы работа эта выполнялась крайне эффективно).

Для дескрипторов ресурсов перемещающие операции часто оказываются более простыми и эффективными, чем копирующие операции. В частности, перемещающие операции в типичном случае не генерируют исключений; они не захватывают ресурсы и не выполняют сложных операций. В этом они сильно отличаются от многих копирующих операций (§17.5).

Как компилятор узнает, что можно применить перемещающую операцию вместо копирующей? В некоторых случаях, как в случае с возвратами из функций, правила языка говорят, что это можно делать (так как следующим действием элемент будет уничтожен). Но в общем случае мы сами должны это специфицировать, предоставляя аргумент в виде rvalue-ссылки. Например:

```
template<class T>
void swap(T& a, T& b)  // "идеальное свопирование" (почти)
{
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```

В стандартной библиотеке имеется функция **move()**, возвращающая rvalue-ссылку на свой аргумент (§35.5.1). Таким образом, **std::move(x)** ничего не перемещает (несмотря на

свое смущающее имя, которое прижилось по историческим причинам); вместо этого она создает условия для перемещения **x**.

Контейнеры стандартной библиотеки (§3.3.2, §35.5.1) и иные типы стандартной библиотеки, такие как **pair** (§5.4.3, §34.2.4.1) и **unique_ptr** (§5.2.1, §34.3.1) обладают перемещающими операциями. Более того, операции, которые вставляют новые элементы в контейнеры стандартной библиотеки, такие как **insert()** и **push_back()**, имеют версии, принимающие rvalue-ссылки (§7.7.2). В итоге, стандартные контейнеры и алгоритмы стандартной библиотеки получают большую производительность, чем в случае применения лишь копирующих операций.

Что имеет место в случае, когда мы пытаемся обменять значения для типов, не обладающих перемещающим конструктором? Ничего особенного: приходится платить неизбежную цену за копирование. В общем случае, программист несет ответственность за избыточное копирование. Это не дело компилятора решать, что является избыточным, а что необходимым. Чтобы реализовать оптимизацию от копирования в сторону перемещения, программист должен предоставить перемещающие операции (явно или неявно; см. §17.6).

Для встроенных типов, таких как **int** и **double***, полагается, что они имеют перемещающие операции, основанные на копировании (вырожденный случай). Как всегда, следует проявлять осторожность для структур, содержащих указатели (§3.3.1). В частности, не предполагайте, что перемещенный указатель будет иметь значение **nullptr**.

Наличие перемещающих операций влияет на эффективность возврата больших значений из функций. Рассмотрим следующее:

```
Matrix operator+(const Matrix& a, const Matrix& b)
    // res[i][j] = a[i][j]+b[i][j] для каждого i и j
{
    if (a.dim[0]!=b.dim[0] || a.dim[1]!=b.dim[1])
        throw std::runtime_error("unequal Matrix sizes in +");

    Matrix res{a.dim[0], a.dim[1]};
    constexpr auto n = a.size();
    for (int i = 0; i!=n; ++i)
        res.elem[i] = a.elem[i]+b.elem[i];
    return res;
}
```

Тип **Matrix** имеет перемещающий конструктор, так что данный возврат «по значению» прост и эффективен. Без перемещающих операций мы бы столкнулись с проблемами, связанными с эффективностью, с которыми пришлось бы как-нибудь справляться. Мы могли бы тогда рассмотреть следующее:

```
Matrix& operator+(const Matrix& a, const Matrix& b)   // осторожно!
{
    Matrix& res = *new Matrix;    // аллоцируем в свободной памяти
    // res[i][j] = a[i][j]+b[i][j] для каждого i и j
    return res;
}
```

Применение **new** внутри **operator+()** не столь очевидно и заставляет пользователя операции **+** иметь дело с проблемами хитроумного обращения с памятью:

- Как объект, созданный при помощи **new**, должен уничтожаться?
- Нужна ли сборка мусора?
- Может нужно применить заранее (статически) сформированный набор объектов **Matrix** вместо динамического аллоцирования операцией **new**?
- Нужно ли представление **Matrix** с подсчетом ссылок?

- Может лучше перепроектировать интерфейс операции сложения для **Matrix**?
- Должен ли пользователь **operator+()** не забывать удалять результат?
- Что произойдет с заново выделенной памятью, если в результате вычисления будет сгенерировано исключение?

Видно, что все альтернативы не столь элегантны или общеупотребительны.

17.6. Генерирование умолчательных операций

Ручное написание стандартных операций, таких как копирование или написание деструкторов, может быть утомительным и чреватым ошибками, так что компилятор мог бы помочь нам, сгенерировав эти операции за нас. По умолчанию класс предоставляет:

- Умолчательный конструктор: **X()**
- Копирующий конструктор: **X(const X&)**
- Копирующее присваивание: **X& operator=(const X&)**
- Перемещающий конструктор: **X(X&&)**
- Перемещающее присваивание: **X& operator=(X&&)**
- Деструктор: **~X()**

По умолчанию, компилятор генерирует перечисленные операции, если они используются в программе. Однако, если программист явно определяет одну или более из этих операций, то генерация связанных операций подавляется:

- Если программист объявляет любой конструктор для класса, то умолчательный конструктор для этого класса не генерируется.
- Если программист объявляет копирующую операцию, перемещающую операцию или деструктор для класса, то никакие копирующие, перемещающие операции и деструктор для этого класса не генерируются.

Последнее правило из-за совместимости со старыми версиями реализовано не полностью: копирующий конструктор и копирующее присваивание генерируются несмотря на явное объявление деструктора. Это отклонение помечено в ISO-стандарте как «устаревшее» (§ iso.D), так что современные компиляторы могут выдавать в таких случаях предупреждения.

При необходимости мы можем в явном виде специфицировать, какие функции генерируются (§17.6.1), а какие не генерируются (§17.6.4).

17.6.1. Явное требование генерирования умолчаний

Так как генерация умолчательных операций может подавляться, то должен быть способ вернуть эту возможность на место. Кроме того, многие люди предпочитают видеть весь набор доступных операций в тексте программы. Мы можем написать:

```
class gslice {
    valarray<size_t> size;
    valarray<size_t> stride;
    valarray<size_t> d1;
public:
    gslice() = default;
    ~gslice() = default;
    gslice(const gslice&) = default;
    gslice(gslice&&) = default;
    gslice& operator=(const gslice&) = default;
    gslice& operator=(gslice&&) = default;
    // ...
};
```

На самом деле этот фрагмент реализации **std::gslice** (§40.5.6) эквивалентен следующему:

```
class gslice {
    valarray<size_t> size;
    valarray<size_t> stride;
    valarray<size_t> d1;
public:
    // ...
};
```

Я предпочитаю последний вариант, но менее опытные программисты могут предпочесть первый вариант, помогающий не забыть ни о каких функциях.

Применение **=default** часто лучше, чем написание собственной реализации умолчательной семантики. Кто-то, полагающий, что написать хоть что-то лучше, чем ничего, может написать следующее:

```
class gslice {
    valarray<size_t> size;
    valarray<size_t> stride;
    valarray<size_t> d1;
public:
    // ...
    gslice(const gslice& a);
};

gslice::gslice(const gslice& a)
    : size{a.size},
    stride{a.stride},
    d1{a.d1}
{
}
```

Это не только затрудняет чтение определения **gslice**, но и открывает возможности для совершения ошибок. Например, я могу забыть скопировать один из членов и позволить ему быть инициализированным по умолчанию (а не скопированным). Также, когда пользователь предоставляет собственную функцию, компилятор уже не может знать ее семантику, так что оптимизация компилятором этой функции снижается. А для умолчательных операций оптимизация весьма существенна.

17.6.2. Умолчательные операции

По умолчанию каждая сгенерированная компилятором операция сводится к применению операции к каждому базовому классу и к каждому нестатическому члену данных класса. То есть мы получаем почленное копирование, почленное умолчательное конструирование и т.д. Например:

```
struct S {
    string a;
    int b;
};

S f(S arg)
{
    S s0 {};        // умолчательное конструирование: {"",0}
    S s1 {s0};      // конструирование копированием
    s1 = arg;       // копир. присваивание
    return s1;      // конструирование перемещением
}
```

При копирующем конструировании **s1** выполняется копирование **s0.a** и **s0.b**. При возврате **s1** выполняется перемещение **s1.a** и **s1.b**, при котором **s1.a** оставляется в виде пустой строки, а **s1.b** оставляется неизменным.

Отметим, что значение перемещенной переменной встроенного типа остается неизменным. Если нам нужно что-то иное для члена класса, нам придется самим написать перемещающие операции для этого класса.

По умолчанию, состояние перемещенного объекта таково, что умолчательные деструктор и копирующее присваивание работают корректно. Не требуется, чтобы так же корректно работали иные операции над перемещенным объектом. Для обеспечения строгих гарантий вам придется написать самим операции соответствующим образом.

17.6.3. Применение умолчательных операций

Здесь мы представим несколько примеров, демонстрирующих, как копирование, перемещение и деструкторы связаны логическим образом. Если они логически не согласованы, то компилятору будет трудно обнаружить связанные с ними ошибки.

17.6.3.1. Умолчательные конструкторы

Рассмотрим следующий пример:

```
struct X {
    X(int);     // требуется int для инициализации X
};
```

Объявлением конструктора, который требует целого аргумента, программист явным образом указывает, что пользователь обязан предоставить целый аргумент для инициализации объекта типа **X**. Теперь мы имеем:

```
X a {1};  // OK
X b {};   // ошибка: нет умолчательного конструктора
```

Если нам нужен также и умолчательный конструктор, мы можем либо сами его определить, либо объявить, что нам нужен такой, который сгенерирует компилятор.

Пример:

```
struct Y {
    string s;
    int n;
    Y(const string& s); // инициализируем Y строкой string
    Y() = default;      // допускаем умолчательную иниц-ю с умолчат. смыслом
};
```

«Умолчательный» (то есть сгенерированный) умолчательный конструктор конструирует по умолчанию каждый член. Здесь **Y()** устанавливает пустую строку значением для **s**. А такая умолчательная инициализация для членов встроенных типов оставляет их (на самом деле) неинициализированными. Увы и ах! Вся надежда на предупреждение от компилятора.

17.6.3.2. Поддержка инвариантов

Часто класс имеет инвариант. В таком случае мы бы хотели, чтобы копирующие и перемещающие операции сохраняли инвариант, а деструктор освобождал бы все вовлеченные в это дело ресурсы. Но, к сожалению, сам компилятор вряд ли может разобраться в том, что программист считает инвариантом класса. Рассмотрим пример:

```
struct Z {      // инвариант:
                // my_favorite это index элемента my_favorite вектора elem
                // largest указывает на элемент с наибольшим значением в elem
    vector<int> elem;
    int my_favorite;
    int* largest;
};
```

Программист сформулировал инвариант посредством комментария, но компилятор не умеет читать комментарии. Более того, здесь программист не оставил даже намека на то, как нужно инвариант устанавливать и поддерживать. В частности, здесь не объявлены ни конструкторы, ни операции присваивания. Такой инвариант неявный. В результате **Z** можно копировать и перемещать умолчательными операциями:

```
Zv0;                        // нет инициализации
Z val {{1,2,3},1,&val[2]}; // OK, но потенциально возможны ошибки
Z v2 = val;                 // копирование: v2.largest указывает на val
Z v3 = move(val);  // перемещение: val.elem становится пустым; v3.my_favorite вне диапазона
```

Это какая-то мешанина. Корень проблемы состоит в том, что **Z** плохо спроектирован из-за того, что критически важная информация спрятана в комментарии (или вообще отсутствует). Правила, по которым генерируются умолчательные операции, являются эвристическими, нацеленными на возможность отлавливания стандартных ошибок. Везде, где только имеется возможность:

1. Устанавливайте инвариант в конструкторе (включая захват ресурсов).
2. Поддерживайте инвариант копирующими и перемещающими операциями.
3. Выполняйте всю необходимую очистку в деструкторе.

17.6.3.3. Ресурсные инварианты

Большая часть из очевидных инвариантов имеет отношение к ресурсам. Рассмотрим пример простого класса **Handle**:

```
template<class T> class Handle {
    T* p;
public:
    Handle(T* pp): p{pp} {}
    T& operator*() {return *p;}
    ~Handle() {delete p;}
};
```

Объект типа **Handle** конструируется на основе указателя на объект, аллоцированый операцией **new**. **Handle** обеспечивает доступ к указуемому объекту, а впоследствии уничтожает объект. Например:

```
void f1()
{
    Handle<int> h {new int{99}};
    // ...
}
```

У **Handle** конструктор принимает один аргумент: это подавляет генерацию умолчательного конструктора. Это хорошо, поскольку умолчательный конструктор мог оставить **Handle<T>::p** неинициализированным:

```
void f2()
{
    Handle<int> h; // ошибка: нет умолчательного конструктора
    // ...
}
```

Отсутствие умолчательного конструктора спасает нас от проблемы вызова операции **delete** для произвольного адреса в памяти.

Также в **Handle** объявляется деструктор: это подавляет генерацию копирующих и перемещающих операций. Это снова спасает нас от неприятных проблем. Рассмотрим следующее:

```
void f3()
{
    Handle<int> h1 {new int{7}};
    Handle<int> h2 {h1};          // ошибка: нет копирующего конструктора
    // ...
}
```

Если бы **Handle** имел умолчательный копирующий конструктор, то **h1** и **h2** имели бы по копии указателя и оба удаляли бы его (операцией **delete**). Соответственно, результат был бы неопределенным и скорее всего катастрофическим (§3.3.1). Так как генерация копирующих операций трактуется лишь как устаревшая (а не запрещенная), то если вы проигнорируете предупреждение, рассмотренный пример пройдет через этап компиляции. Но в любом случае, если в классе хранится указатель, умолчательные копирующие и перемещающие операции должны быть под подозрением. Если этот указатель ответственен за владение ресурсом, то почленное копирование неверно. А если указатель не имеет отношения ко владению чем-либо и почленное копирование приемлемо, то явное использование **=default** и хороший комментарий будут к месту.

Если нам нужно копирующее конструирование, то можно написать что-нибудь в таком роде:

```
template<class T>
class Handle {
    // ...
    Handle(const T& a): p{new T{*a.p}} {}   // клонирование
};
```

17.6.3.4. Частичные инварианты

Примеры кода, в которых программисты пытаются опереться на инварианты, но реализуют их конструкторами и деструкторами лишь частично, не так уж и редки на практике. Рассмотрим следующее:

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {} // всегда 9 позиций

    Tic_tac_toe& operator=(const Tic_tac_toe& arg)
    {
        for(int i = 0; i<9; ++i)
            pos.at(i) = arg.pos.at(i);
        return *this;
    }

    // ... иные операции ...

    enum State {empty, nought, cross};
```

```
private:
    vector<State> pos;
};
```

Этот пример является фрагментом реальной программы. Он использует «магическое число» **9** в рамках реализации копирующего присваивания, осуществляющего доступ к аргументу **arg** без проверки того факта, что он на самом деле имеет девять элементов. Кроме того, здесь явным образом определяется копирующее присваивание, но не коопирующий конструктор. Я не рассматриваю этот код как пример хорошего кода.

Поскольку мы определили копирующее присваивание, то мы должны определить и деструктор. Его вполне можно объявить с **=default**, поскольку все, что он должен сделать — это гарантировать уничтожение члена **pos**, а это должно делаться даже и при отсутствии определения копирующего присваивания. Вот здесь мы и замечаем, что то, что мы определили для копирующего присваивания, было бы получено и по умолчанию, так что и эту операцию лучше объявить с **=default**. Добавляем для полноты копирующий конструктор и вот что мы имеем в результате:

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}   // всегда 9 позиций
    Tic_tac_toe(const Tic_tac_toe&) = default;
    Tic_tac_toe& operator=(const Tic_tac_toe& arg) = default;
    ~Tic_tac_toe() = default;

    // ... иные операции ...

    enum State {empty, nought, cross};
private:
    vector<State> pos;
};
```

Глядя на этот код, мы замечаем, что эффект от всех этих **=default** состоит в том, что устраняются перемещающие операции. Разве этого мы хотели? Вероятно нет. Когда мы объявили копирующее присваивание с **=default**, мы избавились от зависимости от магической константы **9**. Но как нам добавить перемещающие операции? Простейший путь — убрать явные **=default**, и тогда мы увидим, что **Tic_tac_toe** стал совершенно ординарным типом:

```
class Tic_tac_toe {
public:
    // ... иные операции ...
    enum State {empty, nought, cross};
private:
    vector<State> pos {Vector<State>(9)}; // всегда 9 позиций
};
```

Вывод из рассмотрения этого примера старого кода таков: нужно с подозрением относиться к классам со странными комбинациями из умолчательных операций; их иррегулярность свидетельствует о больших провалах в проектировании класса. Для каждого класса мы должны спрашивать:

1. Действительно нужен умолчательный конструктор?
2. Нужен ли деструктор?
3. Нужны ли копирующие операции (например, из-за неприятия умолчательного почленного копирования)?
4. Нужны ли перемещающие операции?

И никогда не следует рассматривать эти операции по-отдельности вне связи друг с другом.

17.6.4. Объявление функций с =delete

Мы можем явно специфицировать, что некоторая функция из класса удалена, то есть она не существует во всех смыслах и ее нельзя вызвать явно или неявно. Основное применение такого средства — гарантировать отсутствие функций, которые могли бы «возникнуть по умолчанию». Например, иногда желательно предотвратить копирование классов, адресуемых указателями на базовый класс, во избежание срезки (§17.5.1.4):

```
class Base {
    // ...
    Base& operator=(const Base&) = delete;      // запрет копирования
    Base(const Base&) = delete;

    Base& operator=(Base&&) = delete;           // запрет перемещения
    Base(Base&&) = delete;
};

Base x1;
Base x2 {x1}; // ошибка: нет копирующего конструктора
```

Обычно в отношении копирующих или перемещающих операций удобнее применить **=default** (§17.6.1), то есть сказать, что мы хотим, а не **=delete**, сообщая о том, чего мы не хотим. Но в принципе можно указанным способом удалять любые функции, которые можно объявить. Например, мы можем устранить специализацию из набора возможных специализаций шаблона функции:

```
template<class T>
T* clone(T* p)                      // возврат копии *p
{
    return new T{*p};
};

Foo* clone(Foo*) = delete;          // не пытайтесь клонировать Foo

void f(Shape* ps, Foo* pf)
{
    Shape* ps2 = clone(ps);         // чудесно
    Foo* pf2 = clone(pf);           // ошибка: clone(Foo*) запрещено
}
```

Другой пример — устранение нежелательного приведения типов. Например:

```
struct Z {
    // ...
    Z(double);          // можно инициализировать типом double
    Z(int) = delete;    // но не integer
};

void f()
{
    Z z1 {1};           // ошибка: Z(int) запрещено
    Z z2 {1.0};         // OK
}
```

Еще одно применение — контроль за тем, где можно аллоцировать класс:

```
class Not_on_stack {
    // ...
    ~Not_on_stack() = delete;
};

class Not_on_free_store {
    // ...
    void* operator new(size_t) = delete;
};
```

Не существует локальной переменной, которая не может быть уничтожена (§17.2.2) и вы не можете аллоцировать объект в свободной памяти, если вы объявите с **=delete** классовую операцию аллоцирования памяти (§19.2.5). Например:

```
void f()
{
    Not_on_stackv1;          // ошибка: нельзя уничтожить
    Not_on_free_store v2;    // OK

    Not_on_stack* p1 = new Not_on_stack;          // OK
    Not_on_free_store* p2 = new Not_on_free_store; // ошибка: нельзя аллоцировать
}
```

Теперь мы никогда не сможем удалить (операцией **delete**) объект типа **Not_on_stack**. Альтернативно можно сделать деструктор частным (**private**; §17.2.2).

Разница в явном удалении функции и в ее «необъявлении» существует отчетливо. Например, в первом случае при попытке вызова функции компилятор распознает ситуацию и генерирует ошибку. Во втором случае возможны варианты — например компилятор может обратиться к глобальной функции **operator new()**.

17.7. Советы

[1] Проектируйте конструкторы, присваивания и деструктор как связанный друг с другом набор операций; §17.1.

[2] Используйте конструктор для установки инварианта класса; §17.2.1.

[3] Если конструктор захватывает ресурс, то классу требуется деструктор для освобождения ресурса; §17.2.2.

[4] Если в классе объявлена виртуальная функция, то ему нужен виртуальный деструктор; §17.2.5.

[5] В отсутствие конструктора в классе он может инициализироваться с помощью почленной инициализации; §17.3.1.

[6] Отдавайте предпочтение {}-инициализации перед = и ()-инициализацией; §17.3.2.

[7] Определяйте для класса умолчательный конструктор тогда и только тогда, когда для него имеется естественное умолчательное значение; §17.3.3.

[8] Если класс контейнерный, то определяйте для класса конструктор со списком инициализаторов; §17.3.4.

[9] Инициализируйте члены и базовые классы в порядке их объявления; §17.4.1.

[10] Если в классе есть член для подсчета ссылок, он наверное нуждается в копирующих операциях (копирующем конструкторе и копирующем присваивании); §17.4.1.1.

[11] Отдавайте предпочтение почленной инициализации перед присваиваниями в конструкторе; §17.4.1.1.

[12] Для предоставления умолчательных значений используйте внутриклассовые инициализаторы; §17.4.4.

[13] Если класс является дескриптором ресурсов, он вероятно нуждается в копирующих и перемещающих операциях; §17.5.

[14] Следите за тем, чтобы в копирующем конструкторе копировались все элементы, требующие копирования (остерегайтесь умолчательных инициализаторов); §17.5.1.1.

[15] Копирующие операции должны обладать свойствами эквивалентности и независимости; §17.5.1.3

[16] Опасайтесь «спутанных» структур данных; §17.5.1.3.

[17] Предпочитайте перемещающую семантику и технику копирование-по-записи, а не поверхностное копирование; §17.5.1.3.

[18] Если класс является базовым, то внесите защиту от срезки; §17.5.1.4.

[19] Если классу нужны копирующие операции или деструктор, он наверняка нуждается и в конструкторе, деструкторе, копирующем присваивании и копирующем конструкторе; §17.6.

[20] Если в классе имеется член-указатель, то класс скорее всего нуждается в деструкторе и в неумолчательных копирующих операциях; §17.6.3.3.

[21] Если класс является дескриптором ресурсов, ему нужен конструктор, деструктор и неумолчательные копирующие операции; §17.6.3.3.

[22] Если умолчательные конструктор, присваивание или деструктор приемлемы, то позвольте компилятору сгенерировать их (не пишите их сами); §17.6.

[23] Выражайте инварианты явно; используйте конструкторы для установки инварианта, а присваивания для их поддержания; §17.6.3.2.

[24] Проверяйте копирующие присваивания на случай самоприсваивания; §17.5.1.

[25] Добавляя новый член к классу, проверяйте конструкторы на необходимость их обновления с целью инициализации нового члена; §17.5.1.

18

Перегрузка операций

Когда я использую слово, оно означает ровно то,
что я выбрал для него означать — не больше и не меньше.
— Хампти Дампти

- Введение
- Функции определения операций

  Бинарные и унарные операции; Стандартный смысл операций; Операции и пользовательские типы; Передача объектов; Операции в пространствах имен
- Тип комплексных чисел

  Операции-члены и операции не-члены; Смешанная арифметика; Приведение типов; Литералы; Функции доступа; Вспомогательные функции
- Приведение типов

  Операции приведения типа; **explicit**-операции приведения; Неоднозначности
- Советы

18.1. Введение

Во многих технических областях были разработаны и внедрены способы краткого отражения часто используемых концепций. Например, из-за хорошего знакомства выражение

```
x+y*z
```

для нас более привычно, чем

```
multiply y by z and add the result to x
```

Трудно переоценить значение компактной нотации для общеупотребительных операций.

Как и многие другие языки программирования C++ поддерживает набор операций для встроенных типов. Однако, многие операции должны распространяться не только на встроенные типы, они должны поддерживать типы, определяемые пользователем. Например, в случае комплексной арифметики, матричной алгебры, логических сигналов или в случае символьных строк в C++ применяются классы, представляющие перечисленные концепции. Если определить для этих классов традиционные операции, то манипулировать объектами с их помощью будет намного удобнее и прозрачнее, чем если пользоваться лишь функциональной нотацией. Рассмотрим следующее:

```
class complex {                    // упрощенный complex
    double re, im;
public:
    complex(double r, double i): re{r}, im{i} {}
    complex operator+(complex);
    complex operator*(complex);
};
```

Здесь представлена простая реализация концепции комплексных чисел. Тип **complex** представляется парой чисел с плавающей запятой двойной точности, и которым можно манипулировать с помощью операций + и *. Программист определяет функции **complex::operator+()** и **complex::operator*()**, придающие смысл операциям + и *, соответственно. К примеру, если **b** и **c** имеют тип **complex**, то **b+c** означает **b.operator+(c)**. Теперь мы можем имитировать традиционные комплексные выражения:

```
void f()
{
    complex a = complex{1,3.1};
    complex b {1.2, 2};
    complex c {b};

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1,2);
}
```

Имеет место традиционное правило приоритета операций, так что второй оператор означает **b=b+(c*a)**, а не **b=(b+c)*a**.

Отметим, что грамматика C++ такова, что {}-нотация допустима лишь для инициализаторов и в правой части операций присваивания:

```
void g(complex a, complex b)
{
    a = {1,2};                     // OK: правая сторона присваивания
    a += {1,2};                    // OK: правая сторона присваивания
    b = a+{1,2};                   // синтаксическая ошибка
    b = a+complex{1,2};            // OK
    g(a,{1,2});                    // OK: аргумент функции как инициализатор
    {a, b} = {b, a};               // синтаксическая ошибка
}
```

В первый момент такое лимитирование может показаться странным, но проблемы с разбором грамматики вынуждают принять это ограничение.

Большинство распространенных примеров перегрузки операций относятся к числовым типам. Однако, польза от операций, определямых пользователем, распространяется далеко за рамки числовых типов. Например, проектирование общих и абстрактных интерфейсов часто требует применения операций, таких как **->**, **[]** и **()**.

18.2. Функции определения операций

Можно определять функции, задающие смысл следующих операций (§10.3):

```
+     -     *     /     %     ^     &
|     ~     !     =     <     >     +=
-=    *=    /=    %=    ^=    &=    |=
<<    >>    >>=   <<=   ==    !=    <=
>=    &&    ||    ++    --    ->*   ,
->    []    ()    new   new[] delete delete[]
```

Следующие операции не могут переопределяться пользователем:

:: операция разрешения области видимости (§6.3.4, §16.2.12)

. выбор члена (§8.2)

.* выбор члена по указателю на член (§20.6)

Эти операции принимают имя, а не значение в качестве второго операнда, и обеспечивают основные средства для ссылки на члены. Если разрешить перегрузку этих операций, то возникнут излишне тонкие конструкции [Stroustrup,1994].

Именованные «операции» не могут перегружаться из-за того, что они сообщают фундаментальные факты о своих операндах:

sizeof размер объекта (§6.2.8)

alignof выравнивание объекта (§6.2.9)

typeid **type_info** для объекта (§22.5)

И, наконец, нельзя перегружать условную тернарную операцию:

?: условная операция (§9.4.1)

Дополнительно, пользовательские литералы (§19.2.6) определяются в нотации **operator""**. Это некий синтаксический трюк, ибо не существует никакой операции, обозначаемой как "". Кроме того, **operator T()** определяет преобразование (приведение) к типу **T** (§18.4).

Невозможно определить новый знак операции, так что в таком случае приходится ограничиваться функциональной нотацией, например **pow()** вместо несуществующего обозначения ** (возведение в степень). В противном случае можно было бы «налететь» на неоднозначности. Действительно, в попытке организовать возведение в степень с помощью ** мы столкнулись бы с вопросами: должна ли операция ** ассоциироваться слева (как в языке Fortran) или справа (как в языке Algol)? Выражение **a**p** должно интерпретироваться как **a*(*p)** или как **(a)**(p)**?

Имя функции, определяющей операцию, состоит из ключевого слова **operator**, за которым следует обозначение собственно операции, например **operator<<**. Функции определения операций в остальном ничем не отличаются от других функций. Использование операций в выражениях является при этом способом сократить запись и улучшить «читаемость». Например:

```
void f(complex a, complex b)
{
    complex c = a + b;          // фактически сокращение
    complex d = a.operator+(b); // явный вызов
}
```

Два представленных здесь варианта записи эквивалентны.

18.2.1. Бинарные и унарные операции

Бинарную операцию можно определить либо нестатической функцией-членом, принимающей единственный аргумент, либо глобальной функцией с двумя аргументами. Для любой бинарной операции **@**, **aa@bb** может интерпретироваться как **aa.operator@(bb)** или как **operator@(aa, bb)**. Если в программе определены оба варианта, то применяется обычное правило разрешения перегрузки (§12.3). Пример:

```
class X {
public:
    void operator+(int);
    X(int);
};
```

```
void operator+(X, X);
void operator+(X, double);

void f(X a)
{
    a+1;      // a.operator+(1)
    1+a;      // ::operator+(X(1), a)
    a+1.0;    // ::operator+(a,1.0)
}
```

Унарная операция, префиксная или постфиксная, может определяться как нестатическая функция-член без аргументов, либо как глобальная функция с одним аргументом. Для любой префиксной унарной операции **@**, **@aa** может интерпретироваться как **aa.operator@ ()** или как **operator@(aa)**. Если определены оба варианта, то применяется правило разрешения перегрузок (§12.3). Для любой постфиксной унарной операции **@**, **aa@** может интерпретироваться как **aa.operator@(int)** или как **operator@(aa, int)**. Объяснение будет приведено позже в §19.2.4. Как всегда, в случае обоих вариантов, будет применяться правило разрешения перегрузки (§12.3). Объявлять операции можно лишь в их стандартном синтаксисе, определенном в грамматике языка (§ iso.A). Например, нельзя определить унарную операцию% или тернарную операцию +. Рассмотрим следующее:

```
class X {
public:          // члены (с неявным this):

    X* operator&();          // префиксная унарная & (взятие адреса)
    X operator&(X);          // бинарная & (И)
    X operator++(int);       // постфиксный инкремент (см. § 19.2.4)
    X operator&(X, X);       // ошибка: тернарная
    X operator/();           // ошибка: унарная /
};

// функции не-члены:

X operator-(X);              // префиксный унарный минус
X operator-(X, X);           // бинарный минус
X operator--(X&, int);       // постфиксный декремент
X operator-();               // ошибка: нет операнда
X operator-(X, X, X);        // ошибка: тернарная
X operator%(X);              // ошибка:  унарная %
```

Операция **[]** рассматривается в §19.2.1, операция **()** в §19.2.2, операция **->** в §19.2.3, операции **++** и **--** в §19.2.4, а операции выделения памяти и ее освобождения рассматриваются в §11.2.4 и §19.2.5.

Операции **operator=** (§18.2.2), **operator[]** (§19.2.1), **operator()** (§19.2.2) и **operator->** (§19.2.3) должны быть нестатическими функциями-членами.

Смысл операций **&&**, **||** и, (запятая) включает в себя правило последовательности: первый операнд вычисляется до вычисления второго (для операций **&&** и **||** второй операнд вообще вычисляется не всегда). Это правило не имеет места в случае, когда операции определяются пользователем; они трактуются точно так же, как и остальные бинарные операции.

18.2.2. Стандартный смысл операций

Смысл некоторых операций состоит в том, что они эквивалентны некоторой комбинации других операций над теми же самыми аргументами. Например, если **a** является целым, **++a** означает **a+=1**, что в свою очередь означает **a=a+1**. Такие соотношения не

имеют места для операций, определяемых пользователем, если только пользователь не определит зависимости явно. К примеру, компилятор не будет генерировать определение **Z::operator+=()** из определений **Z::operator+()** и **Z::operator=()**.

Операции = (присваивание), & (взятие адреса) и, (операция следования; §10.3.2) имеют предопределенный стандартный смысл, когда применяются к объектам классов. Этот предопределенный смысл можно устранить (§17.6.4):

```
class X {
public:
    // ...
    void operator=(const X&) = delete;
    void operator&() = delete;
    void operator,(const X&) = delete;
    // ...
};

void f(X a, X b)
{
    a = b;      // ошибка: нет operator=()
    &a;         // ошибка: нет operator&()
    a, b;       // ошибка: нет operator,()
}
```

Как альтернатива, им можно придать новый смысл подходящими определениями.

18.2.3. Операции и пользовательские типы

Функция определения операции должна быть либо функцией-членом, либо должна принимать хотя бы один аргумент пользовательского типа (кроме операций **new** и **delete**). Это правило гарантирует, что пользователь не может менять смысл выражений, кроме как в случае, если выражение содержит объект пользовательского типа. В частности, невозможно объявить функцию определения операции, если она работает исключительно с указателями. Это гарантирует, что язык C++ расширяемый, но не изменяемый (исключением являются операции =, & и, для объектов классов).

Функция определения операции, принимающая встроенный тип (§6.2.1) в качестве первого аргумента, не может быть функцией-членом. Например, рассмотрим сложение комплексной переменной **aa** и целого числа **2**: **aa+2** может при подходящем определении функции-члена интерпретироваться как **aa.operator+(2)**, а **2+aa** не может, ибо нет такого класса **int**, для которого определилась бы операция **+**, означающая **2.operator+(aa)**. Поскольку компилятор не понимает смысла пользовательской операции **+** и не знает, что она может быть коммутативной, и в итоге не может вывести эквивалентность **2+aa** и **aa+2**. В этом примере проблема снимается одной или более глобальными функциями (§18.3.2, §19.4).

Перечисления являются пользовательскими типами, так что мы можем определять для них операции. Например:

```
enum Day {sun, mon, tue, wed, thu, fri, sat};

Day& operator++(Day& d)
{
    return d = (sat==d)? sun : static_cast<Day>(d+1);
}
```

Каждое выражение проверяется на предмет неоднозначностей. Когда пользовательская операция обеспечивает подходящую интерпретацию, то далее выражение проверяется по правилу разрешения перегрузки (§12.3).

18.2.4. Передача объектов

Когда мы определяем операцию, мы обычно хотим добиться традиционной нотации, например **a=b+c**. Как следствие, мы лимитированы в способе передачи аргументов функции, определяющей операцию, и в способе возврата из нее. Например, мы не можем потребовать аргументов-указателей и тем самым заставить программиста работать с операцией взятия адреса, и не можем возвращать указатель, с тем, чтобы пользователь разыменовывал его: ***a=&b+&c**. Это неприемлемо.

Для аргументов у нас имеется два варианта:

- Передача по значению
- Передача по ссылке

Для объектов малого размера, скажем в один-четыре машинных слова, передача по значению вполне приемлема и часто обеспечивает наилучшую производительность. Реальная производительность программы, связанная с передачей аргументов, зависит от машинной архитектуры и таких особенностей компиляторов, как ABI (Application Binary Interface — Бинарный Интерфейс Приложений), а также от количества обращений к аргументу. Например, предположим, что тип **Point** представляется в виде пары целых чисел типа **int**:

```
void Point::operator+=(Point delta);      // передача по значению
```

Большие объекты мы передаем по ссылке. Например, поскольку **Matrix** (простая матрица элементов типа **double**; §17.5.1) явно больше, чем несколько машинных слов, мы применяем передачу по ссылке:

```
Matrix operator+(const Matrix&, const Matrix&);   // передача по конст. ссылке
```

В частности, мы используем константную ссылку при передаче больших объектов, что означает, что объекты не будут модифицироваться функцией (§12.2.1).

В типичном случае операция возвращает результат. Возврат указателя или ссылки на вновь созданный объект не является хорошей идеей: указатели порождают проблемы с нотацией, а ссылки на объекты в свободной памяти влекут за собой проблемы с управлением ресурсами. Вместо этого следует возвращать объекты по значению. Для больших объектов вроде **Matrix** следует определять перемещающие операции, чтобы сделать возврат эффективным (§3.3.2, §17.5.2). Например:

```
Matrix operator+(const Matrix& a, const Matrix& b)    // возврат по значению
{
    Matrix res {a};
    return res+=b;
}
```

Заметим, что операции, возвращающие один из своих объектов-аргументов, могут (и часто так делают) возвращать ссылку. К примеру, мы могли бы определить для типа **Matrix** операцию **+=** следующим образом:

```
Matrix& Matrix::operator+=(const Matrix& a)  // возврат по ссылке
{
    if (dim[0]!=a.dim[0] || dim[1]!=a.dim[1])
        throw std::exception("bad Matrix += argument");

    double* p = elem;
    double* q = a.elem;
    double* end = p+dim[0]*dim[1];
    while(p!=end)
        *p++ += *q++
    return *this;
}
```

Это особо часто применяется для определяющих операции функций, реализуемых в виде функций-членов.

Если функция просто передает объект другой функции, то следует использовать аргумент вида rvalue-ссылки (§17.4.3, §23.5.2.1, §28.6.3).

18.2.5. Операции в пространствах имен

Операция определяется либо в виде члена класса, либо она определяется в некотором пространстве имен (возможно в глобальном пространстве имен). Рассмотрим упрощенную версию ввода/вывода строк из стандартной библиотеки:

```
namespace std { // упрощенное std

    class string {
        // ...
    };

    class ostream {
        // ...
        ostream& operator<<(const char*);          // вывод строки в С-стиле
    };

    extern ostream cout;

    ostream& operator<<(ostream&, const string&);  // вывод std::string
} // namespace std

int main()
{
    const char* p = "Hello";
    std::string s = "world";
    std::cout << p << ", " << s << "!\n";
}
```

Естественно, что этот код выводит **Hello, world!** Но почему? Я не сделал доступным все из пространства имен **std**, применив

```
using namespace std;
```

Вместо этого я использовал префикс **std::** для **string** и **cout**. Другими словами, я сделал все от меня зависящее, чтобы не загрязнять глобальное пространство имен и не вносить ненужных зависимостей.

Операция вывода строк в С-стиле является членом **std::ostream**, так что по определению

```
std::cout << p
```

означает

```
std::cout.operator<<(p)
```

В то же время **std::ostream** не имеет функции-члена для вывода **std::string**, так что

```
std::cout << s
```

означает

```
operator<<(std::cout, s)
```

Операции, определенные в пространствах имен, могут быть идентифицированы на основе операндов так же, как функции определяются на основе типов аргументов

(§14.2.4). В частности, **cout** находится в пространстве имен **std**, так что **std** и рассматривается при поиске подходящего определения для **<<**. Таким образом компилятор находит и использует

```
std::operator<<(std::ostream&, const std::string&)
```

Рассмотрим бинарную операцию **@**. Если **x** имеет тип **X**, а **y** имеет тип **Y**, то **x@y** разыскивается последовательно следующим образом:

- Если **X** класс, то ищем **operator@** как функцию-член класса **X** или как функцию-член базового класса для **X**;
- Ищем объявления для **operator@** в контексте, окружающем **x@y**;
- Если **X** определяется в пространстве имен **N**, ищем объявления для **operator@** в **N**;
- Если **Y** определяется в пространстве имен **M**, ищем объявления для **operator@** в **M**.

Могут быть найдены несколько объявлений для **operator@**, так что используются правила разрешения перегрузок (§12.3) для определения наиболее подходящего варианта. Рассмотренный вариант розыска применяется только, если операция имеет по крайней мере один операнд пользовательского типа. Поэтому пользовательские преобразования типа (§18.3.2, §18.4) будут привлекаться к рассмотрению. Отметим, что синоним типа это просто синоним, а не отдельный пользовательский тип (§6.5).

Аналогично разрешается и вопрос с унарными операциями.

Следует отметить, что при поиске операций не отдается никакого преимущества членам над не-членами. Это отличается от случая поиска именованных функций (§14.2.4). Отсутствие сокрытия операций гарантирует, что операции над встроенными типами всегда доступны, и что пользователи могут модифицировать смысл операций без внесения изменений в существующие объявления классов. Например:

```
X operator!(X);

struct Z {
    Z operator!();                  // не скрывает ::operator!()
    X f(X x) {/* ... */ return!x;}  // вызов ::operator!(X)
    int f(int x) {/* ... */ return!x;} // вызов встроенного ! для int
};
```

В частности, стандартный **iostream** определяет **<<** для вывода встроенных типов, а пользователь может определить **<<** для вывода пользовательских типов без модификации класса **ostream** (§38.4.2).

18.3. Тип комплексных чисел

Реализация комплексных чисел из §18.1 слишком ограничена, что мало кого может удовлетворить. К примеру, можно было бы предположить, что следующее работает:

```
void f()
{
    complex a {1,2};
    complex b {3};
    complex c {a+2.3};
    complex d {2+b};
    b = c*2*c;
}
```

Дополнительно, мы можем нуждаться в операциях вроде **==** для сравнения и **<<** для вывода, а также подходящих математических функций, таких как **sin()** и **sqrt()**.

Класс **complex** является конкретным типом, и поэтому его дизайн следует рекомендациям из §16.3. Кроме того, пользователи комплексной арифметики настолько сильно опираются на операции, что определение **complex** вводит в игру большую часть из правил для перегрузки операций.

В классе **complex** для скаляров используется тип **double** и этот класс приблизительно эквивалентен **complex<double>** (§40.4) из стандартной библиотеки.

18.3.1. Операции-члены и операции не-члены

Я предпочитаю минимизировать число функций, непосредственно манипулирующих представлением объекта. Этого можно добиться, определяя в самом классе лишь операции, которые неизбежно модифицируют значение своего первого аргумента, такие как **+=**. А операции, производящие новое значение на основе значений своих аргументов, такие как **+**, определяются вне класса:

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a); // требуется доступ к представлению
    // ...
};

complex operator+(complex a, complex b)
{
    return a += b;                  // доступ к представлению через +=
}
```

Аргументы для **operator+()** передаются по значению, так что **a+b** не модифицирует значения аргументов.

Теперь мы можем написать:

```
void f(complex x, complex y, complex z)
{
    complex r1 {x+y+z};     // r1 = operator+(operator+(x, y), z)

    complex r2 {x};         // r2 = x
    r2 += y;                // r2.operator+=(y)
    r2 += z;                // r2.operator+=(z)
}
```

За исключением вопросов производительности вычисление **r1** и **r2** эквивалентно.

Составные операции присваивания, такие как **+=** и ***=**, имеют тенденцию к более простым определениям, чем **+** и ***. Сначала это многих удивляет, но следует принять во внимание, что для **+** три объекта вовлекаются в вычисления (два операнда и результат), в то время как лишь два объекта вовлекаются в операцию **+=**. В последнем случае эффективность улучшается устранением нужды во временных объектах. Пример:

```
inline complex& complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```

Здесь нет нужды во временном объекте для хранения результата сложения, а для компилятора это помогает организовать встраивание.

Очень умный компилятор сумеет сгенерировать близкий к оптимальному код и для операции +. Но мы не всегда располагаем очень умным оптимизатором кода, и не все типы столь просты, как тип **complex**. Поэтому в §19.4 мы обсуждаем способы определения операций с прямым доступом к представлению класса.

18.3.2. Смешанная арифметика

Чтобы справиться с выражением вроде **2+z**, где **z** имеет тип **complex**, нам нужно определить операцию +, принимающую операнды разных типов. В терминологии языка Fortran нам нужна смешанная арифметика. Этого можно добиться, просто добавляя соответствующие версии операций:

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a)
    {
        re += a.re;
        im += a.im;
        return *this;
    }

    complex& operator+=(double a)
    {
        re += a;
        return *this;
    }

    // ...
};
```

Три варианта для **operator+()** можно опредлить вне класса **complex**:

```
complex operator+(complex a, complex b)
{
    return a += b; // вызов complex::operator+=(complex)
}

complex operator+(complex a, double b)
{
    return {a.real()+b, a.imag()};
}

complex operator+(double a, complex b)
{
    return {a+b.real(), b.imag()};
}
```

Функции доступа **real()** и **imag()** опредляются в §18.3.6.
Имея определения для операции +, мы можем написать:

```
void f(complex x, complex y)
{
    auto r1 = x+y;      // вызов operator+(complex, complex)
    auto r2 = x+2;      // вызов operator+(complex, double)
    auto r3 = 2+x;      // вызов operator+(double, complex)
    auto r4 = 2+3;      // встроенное сложение целых
}
```

Я добавил сложение с целыми для полноты картины.

18.3.3. Приведение типов

Чтобы справиться с присваиваниями и инициализациями переменных типа **complex**, мы нуждаемся в приведениях скаляров (целые числа или числа с плавающей запятой) к типу **complex**. Например:

```
complex b {3}; // означает b.re=3, b.im=0

void comp(complex x)
{
    x = 4;    // означает x.re=4, x.im=0
    // ...
}
```

Можно было бы добиться этого введением конструктора, принимающего единственный аргумент. Конструктор, принимающий единственный аргумент, специфицирует преобразование типа своего аргумента в тип конструируемых объектов. Например:

```
class complex {
    double re, im;
public:
    complex(double r): re{r}, im{0} {} // строим complex из double
    // ...
};
```

Конструктор фактически определяет вещественную ось на комплексной плоскости.

Конструктор служит рецептом создания значения заданного типа. Конструктор используется тогда, когда ожидается значение указанного типа и когда такое значение может быть создано конструктором из значения, предоставленного как инициализатор или из присваиваемого значения. Таким образом, конструктор с единственным аргументом не требует явного вызова. Например:

```
complex b {3};
```

означает

```
complex b {3,0};
```

Пользовательское преобразование применяется неявно только в том случае, если оно уникально (§12.3). Если вы не хотите, чтобы конструктор использовался неявно, объявите его с ключевым словом **explicit** (§16.2.6).

Понятно, что нам нужен и конструктор с двумя аргументами типа **double**, а также полезен и умолчательный конструктор, инициализирующий комплексный объект значением **{0,0}**:

```
class complex {
    double re, im;
public:
    complex() : re{0}, im{0} {}
    complex(double r) : re{r}, im{0} {}
    complex(double r, double i) : re{r}, Im{i} {}
    // ...
};
```

Применяя умолчательные аргументы можно сократить код:

```
class complex {
    double re, im;
public:
    complex(double r =0, double i =0) : re{r}, Im{i} {}
    // ...
};
```

По умолчанию копирование реализуется копированием действительной и мнимой частей (§16.2.2). Пример:

```
void f()
{
    complex z;
    complex x {1,2};
    complex y {x};      // y также имеет значение {1,2}
    z = x;              // z также имеет значение {1,2}
}
```

18.3.3.1. Преобразования операндов

Мы определяем по три версии для каждой из стандартных арифметических операций:

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);
// ...
```

Это утомительно, а что утомительно, может быстро превратиться в ошибку. Тут имеется слишком много похожих друг на друга вариантов. Фактически, все они включают простое приведение типа аргумента к общему типу, после чего реализуется стандартный алгоритм.

Альтернативой предоставлению множества вариантов функций для всех комбинаций аргументов служит опора на приведение типа. Например, наш тип **complex** предоставляет конструктор, который конвертирует **double** в **complex**. Следовательно, мы можем продекларировать единственную версию операции сравнения на равенство для **complex**:

```
bool operator==(complex, complex);

void f(complex x, complex y)
{
    x==y;     // означает operator==(x, y)
    x==3;     // означает operator==(x, complex(3))
    3==y;     // означает operator==(complex(3), y)
}
```

Есть, конечно, соображения, по которым можно предпочесть определение различных функций. Например, когда конвертирование накладывает слишком большие расходы, а в других случаях можно реализовать более простой алгоритм для конкретных типов аргументов. В остальных случаях опора на конвертирование и наиболее общий вариант функции (плюс возможно небольшое количество критических вариантов) — покрывает комбинаторный взрыв вариантов, который вытекает из смешанной арифметики.

При наличии нескольких вариантов функции или операции компилятор должен отобрать правильный вариант на основе типов аргументов и имеющихся преобразований (стандартных и пользовательских). Если наилучшего выбора не существует, то ситуация трактуется как неоднозначная и ошибочная (см. §12.3).

Объект, сконструированный посредством явного или неявного вызова конструктора в рамках выражения является автоматическим и будет уничтожен при первой возможности (см. §10.3.4).

Никаких неявных пользовательских преобразований не применяется к левой части операции . (или ->). Это имеет место и в случае, когда . неявная. Пример:

```
void g(complex z)
{
    3+z;                    // OK: complex(3)+z
    3.operator+=(z);        // ошибка: 3 это не класс
    3+=z;                   // ошибка: 3 это не объект класса
}
```

Таким образом аппроксимируется концепция, что операция требует lvalue в качестве своего левого операнда посредством реализации этой операции в виде функции-члена. Но это лишь аппроксимация, поскольку можно обратиться ко временному объекту модифицирующей операцией, такой как **operator+=()**:

```
complex x {4,5}
complex z {sqr t(x)+={1,2}};   // как tmp=sqrt(x), tmp+={1,2}
```

Если мы не хотим неявных преобразований, мы можем применить **explicit** для их подавления (§16.2.6, §18.4.2).

18.3.4. Литералы

У нас есть литералы встроенных типов. К примеру, **1.2** и **12e3** — это литералы типа **double**. Для типа **complex** можно близко подобраться к подобным вещам, объявляя **constexpr**-конструкторы (§10.4). Например:

```
class complex {
public:
    constexpr complex(double r =0, double i =0) : re{r}, im{i} {}
    // ...
}
```

При этом значение типа **complex** может быть сконструировано из составных частей на этапе компиляции как в случае литералов встроенных типов. Например:

```
complex z1 {1.2,12e3};
constexpr complex z2 {1.2,12e3};  // гарантия инициализации во время компиляции
```

Когда конструкторы просты и встраиваемы, и в особенности когда они объявлены с ключевым словом **constexpr**, разумно трактовать их активизацию с литеральными аргументами как литералы.

Можно пойти еще дальше и ввести пользовательский литерал (§19.2.6) в поддержку нашего комплексного типа. В частности, можно определить **i** как суффикс, означающий «мнимую часть». Например:

```
constexpr complex<double> operator "" i(long double d) // мнимая часть
{
    return {0, d};  // complex пользовательский литерал
}
```

Это позволит нам написать следующее:

```
complex z1 {1.2+12e3i};

complex f(double d)
{
    auto x {2.3i};
    return x+sqrt(d+12e3i)+12e3i;
}
```

Этот пользовательский литерал позволяет нам пойти дальше того, что мы получили с **constexpr**-конструкторами: мы можем использовать пользовательский литерал в середи-

не выражений, где {}-нотация может использоваться лишь с квалификацией именем типа. Вышеприведенный код примерно эквивалентен следующему:

```
complex z1 {1.2,12e3};

complex f(double d)
{
    complex x {0,2.3};
    return x+sqrt(complex{d,12e3})+complex{0,12e3};
}
```

Я подозреваю, что выбор стиля литерала будет зависеть от вашего эстетического чутья и соглашений предметной области. В стандартной библиотеке тип **complex** использует **constexpr**-конструкторы, а не пользовательские литералы.

18.3.5. Функции доступа

До сих пор мы обеспечивали класс **complex** конструкторами и арифметическими операциями. Этого недостаточно для настоящего применения. В частности, нам часто нужно прочесть и изменить действительную или мнимую части:

```
class complex {
    double re, im;
public:
    constexpr double real() const {return re;}
    constexpr double imag() const {return im;}

    void real(double r) {re = r;}
    void imag(double i) {im = i;}
    // ...
};
```

Я не считаю хорошей идеей предоставлять индивидуальный доступ ко всем членам класса. Для многих типов индивидуальный доступ (часто называемый get- и set-функциями) — это приглашение к катастрофе. Он может скомпрометировать инвариант класса, и в типичном случае усложняет изменения в представлении. К примеру, рассмотрите возможность неправильного обращения с объектами в случае, если будут предоставлены get-и set-функции для типа **Date** из §16.3 или типа **String** из §19.3. Однако, для типа **complex** функции **real()** и **imag()** важны семантически: некоторые алгоритмы чище всего формулируются, если они могут манипулировать мнимой и действительной частями независимо. Например, имея **real()** и **imag()**, мы можем упростить операции, такие как **==**, как глобальные функции:

```
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

18.3.6. Вспомогательные функции

Если сложить все части вместе, то класс **complex** примет вид:

```
class complex {
    double re, im;
public:
    constexpr complex(double r =0, double i =0) : re(r), im(i) {}
```

```
    constexpr double real() const {return re;}
    constexpr double imag() const {return im;}

    void real(double r) {re = r;}
    void imag(double i) {im = i;}

    complex& operator+=(complex);
    complex& operator+=(double);
    // -=, *=, и /=
};
```

Дополнительно, нужно предоставить некоторое число полезных вспомогательных функций:

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);

// бинарный -, *, and /

complex operator-(complex);  // унарный минус
complex operator+(complex);  // унарный плюс

bool operator==(complex, complex);
bool operator!=(complex, complex);

istream& operator>>(istream&, complex&);   // ввод
ostream& operator<<(ostream&, complex);    // вывод
```

Заметьте, что члены **real()** и **imag()** существенны для определения сравнений. Определения остальных вспомогательных функций также полагаются на **real()** и **imag()**.

Например, можно предоставить функции, позволяющие пользователю действовать в терминах полярных координат:

```
complex polar(double rho, double theta);
complex conj(complex);

double abs(complex);
double arg(complex);
double norm(complex);

double real(complex);    // для удобства нотации
double imag(complex);    // для удобства нотации
```

Наконец, мы должны предоставить соответствующий набор математических функций:

```
complex acos(complex);
complex asin(complex);
complex atan(complex);
// ...
```

С точки зрения пользователя представленный здесь тип **complex** почти идентичен типу **complex<double>**, представленному в файле **<complex>** из стандартной библиотеки (§5.6.2, §40.4).

18.4. Приведение типа

Приведение типа можно выполнить

- Конструктором с единственным аргументом (§16.2.5).
- Операцией приведения типа (§18.4.1).

В обоих случаях приведение может быть

- Явным (**explicit**); то есть приведение выполняется только при прямой инициализации (§16.2.6) в качестве инициализатора, не используя знак **=**.
- Неявным; там, где это может быть применено без неоднозначностей (§18.4.3), например в качестве аргумента функции.

18.4.1. Операции приведения типа

Использование конструктора с единственным аргументом для выполнения приведения типа является удобным, но может иметь нежелательные эффекты. Кроме того, конструктор не может специфицировать

1. Неявное преобразование от пользовательского типа во встроенный тип (поскольку встроенные типы — не классы),
2. Приведение от нового класса к ранее определенному классу (без модификации объявления старого класса).

С этими проблемами можно справиться, определяя операцию приведения типа (для типа-источника). Функция-член **X::operator T()**, где **T** имя типа, определяет приведение от типа **X** к типу **T**. Например, можно определить 6-битовые неотрицательные целые, **Tiny**, которые можно свободно смешивать с обычными целыми в арифметических выражениях. Тип **Tiny** генерирует исключение **Bad_range**, если его операции приводят к переполнению или исчезновению порядка:

```
class Tiny {
    char v;
    void assign(int i) {if (i&~077) throw Bad_range(); v=i;}
public:
    class Bad_range {};

    Tiny(int i) {assign(i);}
    Tiny& operator=(int i) {assign(i); return *this;}

    operator int() const {return v;}      // конверсия в int функцию
};
```

Диапазон проверяется, когда **Tiny** инициализируется числом **int** и когда **int** присваивается переменной типа **Tiny**. Проверка диапазона не нужна при копировании **Tiny**, так что умолчательный копирующий конструктор и присваивание вполне подходят.

Чтобы обеспечить обычные целочисленные операции над переменными типа **Tiny**, мы определяем неявное преобразование от **Tiny** к **int**, **Tiny::operator int()**. Заметьте, что тип, к которому выполняется приведение, входит в имя операции и не может повторяться в качестве возврата функции преобразования:

```
Tiny::operator int() const {return v;}          // правильно
int Tiny::operator int() const {return v;}      // ошибка
```

В этом отношении операция приведения типа (преобразования) напоминает конструкторы.

Когда **Tiny** появляется там, где ожидается **int**, используется подходящий **int**. Например:

```
int main()
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2-c1;    // c3 = 60
    Tiny c4 = c3;       // no range check (not necessary)
```

```
    int i = c1+c2; // i = 64

    c1 = c1+c2;    // ошибка диапазона: c1 не может быть 64
    i = c3-64;     // i = -4
    c2 = c3-64;    // ошибка диапазона: c2 не может быть -4
    c3 = c4;       // без контроля диапазона (нет необходимости)
}
```

Функции преобразования оказываются особо полезными при работе со структурами данных, когда чтение тривиальное (реализуется операциями преобразования), в то время как присваивание и инициализация существенно менее тривиальны.

Типы **istream** и **ostream** полагаются на функции преобразования, чтобы следующие операторы могли работать:

```
while (cin>>x)
    cout<<x;
```

Операция ввода **cin>>x** возвращает тип **istream&**. Это значение неявно конвертируется в значение, индицирующее состояние потока **cin**. Это значение можно проверить в операторе **while** (см. §38.4.4). Однако, не является хорошей идеей определение неявного преобразования от одного типа к другому таким способом, что в процессе конвертирования теряется информация.

В общем случае, следует проявлять разумную осторожность при введении операций преобразования; в большом количестве они приводят к неоднозначностям. Они могут отлавливаться компилятором, но в любом случае — это скорее мешает, чем помогает. Неплохой идеей станет использование именованных функций на начальном этапе разработки, таких как **X::make_int()**. Если на практике такие функции становятся достаточно популярными, то тогда имеет смысл придать им форму операции вроде **X::operator int()**.

Если определены оба варианта, то возможны неоднозначности между пользовательскими операциями и встроенными операциями. Например:

```
int operator+(Tiny, Tiny);

void f(Tiny t, int i)
{
    t+i; // ошибка, неоднозначно: "operator+(t, Tiny(i))" или "int(t)+i"?
}
```

В общем, лучше выбрать что-то одно.

18.4.2. explicit-операции приведения

Существует тенденция, что если пользовательские операции определены, то их применяют всюду. Но можно объявить операцию приведения типа с ключевым словом **explicit**, так что использовать ее можно будет только для прямой инициализации (§16.2.6), где применялся бы и эквивалентный **explicit**-конструктор. Например, **unique_ptr** (§5.2.1, §34.3.1) из стандартной библиотеки объявляет явную (**explicit**) операцию приведения к **bool**:

```
template <typename T, typename D = default_delete<T>>
class unique_ptr {
public:
    // ...
    explicit operator bool() const noexcept; // содержит ли *this указатель (не равный nullptr)?
    // ...
};
```

В результате устраняются сомнительные случаи использования этой операции. Рассмотрим следующее:

```
void use(unique_ptr<Record> p, unique_ptr<int> q)
{
    if (!p)          // OK: хотим это использовать
        throw Invalid_uninque_ptr{};

    bool b = p;      // ошибка; подозрительное применение
    int x = p+q;     // ошибка; определенно мы этого не хотим
}
```

Если бы не было в определении операции ключевого слова **explicit**, то два последних определения прошли бы компиляцию. Значение **b** стало бы **true**, а значение **x** стало бы **1** или **2** (в зависимости от **q**).

18.4.3. Неоднозначности

Присвоение значения типа **V** объекту класса **X** допустимо, если имеется операция присваивания **X::operator=(Z)**, так что либо **V** есть **Z**, либо есть уникальное конвертирование **V** в **Z**. Инициализация трактуется аналогично.

В некоторых случаях значение желаемого типа может быть сконструировано посредством повторных вызовов конструкторов или операций преобразования. Это должно управляться посредством явных преобразований; неявные пользовательские преобразования допускают один уровень применения. Если возможны несколько способов конструирования значения желаемого типа, то такая ситуация трактуется как ошибочная. Например:

```
class X {/* ... */ X(int); X(const char*);};
class Y {/* ... */ Y(int);};
class Z {/* ... */ Z(X);};

X f(X);
Y f(Y);

Z g(Z);

void k1()
{
    f(1);        // ошибка: неоднозначно f(X(1)) или f(Y(1))?
    f(X{1});     // OK
    f(Y{1});     // OK

    g("Mack");       // ошибка: нужны две пользователь. конверсии; g(Z{X{"Mack"}})
    g(X{"Doc"});     // OK: g(Z{X{"Doc"}})
    g(Z{"Suzy"});    // OK: g(Z{X{"Suzy"}})
}
```

Пользовательские преобразования рассматриваются только, если вызов не может быть разрешен без них (то есть с применением лишь встроенных преобразований). Пример:

```
class XX {/* ... */ XX(int);};

void h(double);
void h(XX);
```

```
void k2()
{
    h(1);       // h(double{1}) or h(XX{1})? h(double{1})!
}
```

Вызов **h(1)** означает **h(double(1))**, поскольку здесь используется лишь стандартное преобразование, а не пользовательское (§12.3).

Правила для преобразований не слишком простые в реализации и документировании, но они по крайней мере безопаснее и надежнее иных возможных альтернатив. Однако, намного надежнее вручную разрешить неоднозначность, чем искать тонкие ошибки, вызванные неожиданными преобразованиями.

Тип возвращаемого значения не может использоваться при разрешении перегрузок. Например:

```
class Quad {
public:
    Quad(double);
    // ...
};

Quad operator+(Quad, Quad);

void f(double a1, double a2)
{
    Quad r1 = a1+a2;            // сложение вещественных с двойной точностью
    Quad r2 = Quad{a1}+a2;      // форсируем quad-арифметику
}
```

Поскольку типы у обоих частей операций инициализации или присваивания определены, то оба типа используются в разрешении неоднозначности. Например:

```
class Real {
public:
    operator double();
    operator int();
    // ...
};

void g(Real a)
{
    double d = a; // d = a.double();
    int i = a;    // i = a.int();

    d = a;    // d = a.double();
    i = a;    // i = a.int();
}
```

В таких случаях по-прежнему применяется синтаксический анализ снизу-вверх, но в каждый конкретный момент рассматривается единственная операция и ее аргумент.

18.5. Советы

[1] Определяйте операции для имитирования традиционных форм записи; §18.1.

[2] Переопределяйте или запрещайте копирование, если умолчательная версия для типа не подходит; §18.2.2.

[3] Для операндов больших размеров используйте константные ссылки в качестве аргументов; §18.2.4.

[4] Для результатов большого размера применяйте перемещающие конструкторы; §18.2.4.

[5] Для операций, требующих доступа к представлению, предпочитайте реализацию в виде функций-членов; §18.3.1.

[6] Для операций, не требующих доступа к представлению, предпочитайте реализацию в виде глобальных функций; §18.3.2.

[7] Используйте пространства имен для ассоциации вспомогательных функций с классами; §18.2.5.

[8] Для симметричных операций используйте глобальные функции; §18.3.2.

[9] Для операций, требующих lvalue в качестве левого операнда, используйте функции-члены; §18.3.3.1.

[10] Применяйте пользовательские литералы для имитации традиционной нотации; §18.3.4.

[11] Предоставляйте set- и get-функции для членов-данных только в том случае, если этого требует фундаментальная семантика класса; §18.3.5.

[12] Будьте осторожны с введением неявных преобразований; §18.4.

[13] Избегайте сужающих преобразований; §18.4.1.

[14] Не определяйте одних и тех же преобразований одновременно в виде конструктора и операции приведения типа; §18.4.3.

19

Специальные операции

Все мы специальные.
— Альбер Камю

- Введение
- Специальные операции

  Индексирование; Вызов функции; Разыменование; Инкремент и декремент; Выделение памяти и ее освобождение; Пользовательские литералы
- Класс строк

  Важные операции; Доступ к символам; Представление; Функции-члены; Полезные функции; Использование класса **String**
- Друзья

  Поиск друзей; Друзья и функции-члены
- Советы

19.1. Введение

Перегрузка операций не ограничивается арифметическими и логическими операциями. На самом деле, операции не менее важны и в других областях: в дизайне контейнеров (вроде **vector** и **map**; §4.4), для «интеллектуальных указателей» (**unique_ptr** и **shared_ptr**; §5.2.1), для итераторов (§4.5) и других классов, имеющих дело с управлением ресурсами.

19.2. Специальные операции

Операции

[] () -> ++ -- new delete

являются специальными в том смысле, что их переопределение в пользовательском коде несколько отличается от правил для стандартных унарных и бинарных операций, таких как **+**, **<** и **~** (§18.2.3). Операции **[]** (индексирование) и **()** (вызов функций) представляют несомненную пользу в случае перегрузки их пользователем.

19.2.1. Индексирование

Функция **operator[]()** может использоваться, чтобы придать индексам смысл для объектов класса. Второй аргумент (индекс) функции **operator[]** может иметь любой тип. Это позволяет определять типы **vector**, ассоциативные массивы и т.п.

В качестве примера определим простой ассоциативный массив:

```
struct Assoc {
    vector<pair<string, int>> vec;      // упрощенный complex

    const int& operator[] (const string&) const;
    int& operator[](const string&);
};
```

Тип **Assoc** хранит вектор элементов типа **std::pair**. Реализация использует тот же самый тривиальный и неэффективный метод поиска, что и в §7.7:

```
int& Assoc::operator[](const string& s)
    // поиск s; возврат ссылки на это значение (в случае успеха);
    // иначе, создаем pair {s,0} и возвращаем ссылку на это значение
{
    for (auto x : vec)
        if (s == x.first) return x.second;

    vec.push_back({s,0});           // начальное значение: 0

    return vec.back().second;       // вернуть последний элемент (§ 31.2.2)
}
```

Тип **Assoc** можно использовать следующим образом:

```
int main()      // сосчитать вхождение всех слов во входном потоке
{
    Assoc values;
    string buf;
    while (cin>>buf) ++values[buf];
    for (auto x : values.vec)
        cout << '{' << x.first << ',' << x.second << "}\n";
}
```

Типы **map** и **unordered_map** из стандартной библиотеки являются развитием идеи ассоциативных массивов (§4.4.3, §31.4.3) с менее наивными реализациями.

Функция **operator[]()** должна быть нестатической функцией-членом.

19.2.2. Вызов функции

Вызов функции, то есть нотация *выражение*(*список-выражений*), может интерпретироваться как бинарная операция, в которой *выражение* есть левый операнд, а *список-выражений* является правым операндом. Операция вызова, **()**, может перегружаться так же, как и многие другие операции. Например:

```
struct Action {
    int operator()(int);
    pair<int, int> operator()(int, int);
    double operator()(double);
    // ...
};

void f(Action act)
{
    int x = act(2);
    auto y = act(3,4);
    double z = act(2.3);
    // ...
};
```

Список аргументов для **operator()()** вычисляется и проверяется согласно стандартным правилам передачи аргументов. Практическая польза от перегрузки операции вызова функции наблюдается для типов с единственной операцией или с одной доминирующей операцией.

Наиболее очевидным примером использования операции **()** служат объекты, которые могут в некотором отношении вести себя как функции. Такие объекты часто называют объектами-функциями или функциональными объектами (§3.4.3). Функциональные объекты позволяют нам писать код, который принимает нетривиальные операции в качестве параметров. Например, можно определить класс с функцией **operator()()**, которая добавляет аргумент к сохраненному значению:

```
class Add {
    complex val;
public:
    Add(complex c): val{c} {}                          // сохранить значение
    Add(double r, double i): val{{r, i}} {}

    void operator()(complex& c) const {c += val;}      // прибавить значение к аргументу
};
```

Объект класса **Add** инициализируется комплексным числом, и будучи активированным операцией **()**, прибавляет это число к аргументу. Пример:

```
void h(vector<complex>& vec, list<complex>& lst, complex z)
{
    for_each(vec.begin(), vec.end(), Add{2,3});
    for_each(lst.begin(), lst.end(), Add{z});
}
```

Этот код прибавляет **complex{2,3}** к каждому элементу **vector** и прибавляет **z** к каждому элементу **list**. Заметим, что **Add{z}** создает объект, который последовательно используется в цикле **for_each()**: его **operator()()** вызывается для каждого элемента последовательности.

Все это работает, поскольку **for_each** является шаблоном, который применяет **()** к своему третьему аргументу, не зная точно, кем этот третий аргумент является на самом деле:

```
template<typename Iter, typename Fct>
Fct for_each(Iter b, Iter e, Fct f)
{
    while (b != e) f(*b++);
    return f;
}
```

На первый взгляд, такой пример кажется экзотическим, но он прост, эффективен и чрезвычайно полезен (§3.4.3, §33.4).

Заметьте, что лямбда-выражение (§3.4.3, §11.4) в своей основе является синтаксисом определения функционального объекта. Например, мы могли бы написать:

```
void h2(vector<complex>& vec, list<complex>& lst, complex z)
{
    for_each(vec.begin(), vec.end(),[](complex& a){a+={2,3};});
    for_each(lst.begin(), lst.end(),[](complex& a){a+=z;});
}
```

В таком случае каждое из лямбда-выражений генерирует эквивалент функционального объекта **Add**.

Другим популярным приложением для **operator()()** является его применение в качестве операции взятия подстроки и в качестве индексирующей операции для многомерных массивов (§29.2.2, §40.5.2).

Функция **operator()()** должна быть нестатической функцией-членом.

Часто операции вызова функции реализуются шаблонами (§29.2.2, §33.5.3).

19.2.3. Разыменование

Операция разыменования, **->**, может определяться как унарная постфиксная операция. Например:

```
class Ptr {
    // ...
    X* operator->();
};
```

Объекты класса **Ptr** могут использоваться для доступа к членам класса **X** в манере, аналогичной той, в которой используются указатели. Например:

```
void f(Ptr p)
{
    p->m = 7;      // (p.operator->())->m = 7
}
```

Трансформация объекта **p** в указатель **p.operator->()** не зависит от члена **m**, на который он указывает. В этом смысле **operator->()** является унарной постфиксной операцией. Но никакой новой операции при этом не определяется, так что имя члена по-прежнему требуется указывать после **->**. Например:

```
void g(Ptr p)
{
    X* q1 = p->;              // синтаксическая ошибка
    X* q2 = p.operator->();   // OK
}
```

Перегрузка **->** особо полезна для так называемых «интеллектуальных указателей», то есть для объектов, которые ведут себя как указатели, а дополнительно выполняют некоторую работу, будучи примененными к какому-либо объекту. Стандартные типы **unique_ptr** и **shared_ptr** (§5.2.1) реализуют операцию **->**.

В качестве примера мы можем определить класс **Disk_ptr** для доступа к объектам, хранящимся на диске. Конструктор класса принимает имя, с помощью которого объект разыскивается на диске; **Disk_ptr::operator->()** извлекает объект в память, а деструктор класса **Disk_ptr** запишет в конце концов обновленный объект обратно на диск:

```
template<typename T>
class Disk_ptr {
    string identifier;
    T* in_core_address;
    // ...
public:
    Disk_ptr(const string& s) : identifier{s}, in_core_address{nullptr} {}
    ~Disk_ptr() {write_to_disk(in_core_address, identifier);}

    T* operator->()
    {
        if (in_core_address == nullptr)
            in_core_address = read_from_disk(identifier);
    return in_core_address;
    }
};
```

Тип **Disk_ptr** можно использовать следующим образом:

```
struct Rec {
    string name;
    // ...
};

void update(const string& s)
{
    Disk_ptr<Rec> p {s};          // получить Disk_ptr для s

    p->name = "Roscoe";           // обновить s; при необходимости скачать с диска
    // ...
}   // у p деструктор пишет на диск
```

Естественно, более реалистичный вариант программы должен обрабатывать ошибки и применять менее наивный способ взаимодействия с диском.

Для обычных указателей использование **->** синонимично некоторым способам применениям * и []. Пусть имеется класс **Y**, для которого **->**, * и [] даны по умолчанию. Пусть указатель **p** имеет тип **Y***. Тогда:

```
p->m == (*p).m      // правда
(*p).m == p[0].m    // правда
p->m == p[0].m      // правда
```

Как всегда, нет таких гарантий для пользовательских операций. Где это необходимо, такую эквивалентность нужно запроектировать:

```
template<typename T>
class Ptr {
    Y* p;
public:
    Y* operator->() {return p;}          // разыменовать для доступа к члену
    Y& operator*() {return *p;}          // разыменовать для доступа к объекту
    Y& operator[](int i) {return p[i];}  // разыменовать для доступа к элементу
    // ...
};
```

Если вы определяете более одной из этих операций, то будет разумным обеспечить и эквивалентность точно так же, как разумно обеспечить эквивалентность **++x**, **x+=1** и **x=x+1** для переменной **x** типа **X**, если для этого класса определены операции **++**, **+=**, **=** и **+**.

Перегрузка операции **->** важна для целого класса программ и не является лишь любопытной экзотикой. Дело в том, что косвенное обращение является важным концептуальным понятием, и перегрузка операции **->** обеспечивает четкое и эффективное представление концепции косвенного обращения в программах. Итераторы (Глава 33) представляют собой наглядный пример сказанного.

Операция **->** должна определяться нестатической функцией-членом. Ее возврат должен быть указателем или объектом класса, к которому применима операция **->**. Поскольку тело определения функции-члена шаблона проверяется только в том случае, если эта функция используется в программе (§26.2.1), то мы можем определить **operator->()**, не беспокоясь о типах вроде **Ptr<int>**, для которого операция **->** не имеет смысла.

Несмотря на взаимосвязь между операциями **->** и **.** (операция «точка»), нет возможности перегрузить операцию точка.

19.2.4. Инкремент и декремент

Как только люди изобрели интеллектуальные указатели, они сразу же захотели предоставить и операции ++ и -- ради аналогии со встроенными типами. Это полезно для реализации интеллектуального указателя как более полного заменителя обычных указателей. Рассмотрим следующую традиционную программу:

```
void f1(X a)          // традиционное использование
{
     X v[200];
     X* p = &v[0];
     p--;
     *p = a;          // упс: p вне диапазона
     ++p;
     *p = a;          // OK
}
```

Здесь мы можем заменить **X*** объектом класса **Ptr<X>**, чтобы гарантировать его разыменование только в том случае, если он действительно указывает на **X**. Также хотелось бы гарантировать, что **p** может инкрементироваться и декрементироваться, только если он указывает на объект внутри массива, и что операции декремента и инкремента также производят объект внутри массива. То есть нам хотелось бы следующего:

```
void f2(Ptr<X> a)          // с проверкой
{
     X v[200];
     Ptr<X> p(&v[0], v);
     p--;
     *p = a;               // проверка на этапе выполнения: p вне диапазона
     ++p;
     *p = a;               // OK
}
```

Операции инкремента и декремента уникальны в том смысле, что они имеются в префиксном и постфиксном вариантах. Следовательно, мы должны предоставлять оба варианта для **Ptr<T>**. Пример:

```
template<typename T>
class Ptr {
     T* ptr;
     T* array;
     int sz;
public:
     template<int N>
          Ptr(T* p, T(&a)[N]); // привязка к массиву a, sz==N, начальное значение p
     Ptr(T* p, T* a, int s);   // привязка к массиву a размером s, началь. значение p
     Ptr(T*p);                 // привязка к единствен. объекту, sz==0, началь. значение p

     Ptr& operator++();        // префикс
     Ptr operator++(int);      // постфикс

     Ptr& operator--();        // префикс
     Ptr operator--(int);      // постфикс

     T& operator*();           // префикс
};
```

Аргумент типа **int** используется только для того, чтобы указывать на постфиксное применение. Этот **int** более никак не используется.

Теперь покажем реализации:

```
template<typename T>
Ptr& Ptr<T>::operator++()      // возврат текущего объекта после инкрементирования
{
    // ... проверка, можно ли адресовать ptr+1  ...
    return *++ptr;
}

template<typename T>
Ptr Ptr<T>::operator++(int)    // инкрементировать и вернуть Ptr со стар. знач-ем
{
    // ... проверка, можно ли  ptr+1
    Ptr<T> old {ptr, array, sz};
    ++ptr;
    return old;
}
```

Префиксная операция может возвращать ссылку на свой объект. Постфиксная операция должна создавать новый объект для возврата.

Применяя **Ptr**, напишем эквивалентный код:

```
void f3(T a)    // с проверкой
{
    T v[200];
    Ptr<T> p(&v[0], v,200);
    p.operator--(0);       // suffix: p--
    p.operator*() = a;     // ошибка на этапе выполнения: p вне диапазона
    p.operator++();        // prefix: ++p
    p.operator*() = a;     // OK
}
```

Можете дополнить класс **Ptr** самостоятельно в качестве упражнения. Шаблонный вариант указателя, ведущий себя корректно при наследовании, представлен в §27.2.2.

19.2.5. Выделение памяти и ее освобождение

Операция **new** (§11.2.3) захватывает память, активируя вызов **operator new()**. Аналогично, операция **delete** освобождает память вызовом **operator delete()**. Пользователь может переопределить глобальные функции **operator new()** и **operator delete()**, или же он может определить эти функции в конкретном классе.

Объявления для глобальных версий с применением синонима **size_t** (§6.2.8), выглядят следующим образом:

```
void* operator new(size_t);              // используем для индивидуального объекта
void* operator new[](size_t);            // используем для массива
void operator delete(void*, size_t);     // используем для индивидуального объекта
void operator delete[](void*, size_t);   // используем для массива

// другие версии, см. § 11.2.4
```

Итак, когда операция **new** должна выделить память в куче для объекта типа **X**, она вызывает **operator new(sizeof(X))**. Аналогично, когда нужно выделить память для **N** объектов типа **X**, вызывается функция **operator new[](N*sizeof(X))**. Решаться на замену глобальных **operator new()** и **operator delete()** стоит лишь чрезвычайно подготовленным программистам. В конце концов, некоторый код программы вполне может полагаться на умолчательное поведение этих операций.

Более аккуратным подходом будет определение этих операций в конкретном классе. Этот класс может быть базовым для многих других классов. Например, мы могли бы создать класс **Employee**, предоставляющий специализированный аллокатор и деаллокатор для внутреннего использования (и для своих производных классов):

```
class Employee {
public:
    // ...

    void* operator new(size_t);
    void operator delete(void*, size_t);

    void* operator new[](size_t);
    void operator delete[](void*, size_t);
};
```

Члены **operator new()** и **opeartor delete()** неявно статические. То есть они не располагают указателем **this** и не модифицируют объект. Они предоставляют область памяти, которую конструктор может инициализировать, а деструктор может очищать.

```
void* Employee::operator new(size_t s)
{
    // выделить s байт памяти и вернуть указатель на нее
}

void Employee::operator delete(void* p, size_t s)
{
    if (p) {    // удалять только если p!=0; см. § 11.2, § 11.2.3
        // полагаем, что p адресует s байт памяти, выделенной через Employee::operator new()
        // и освобождаем ее для повторного использования
    }
}
```

Использование отчасти загадочного аргумента **size_t** становится очевидным. Это размер удаляемого объекта. Удаление для **Employee** располагает значением аргумента **sizeof(Employee)**; при удалении **Manager**, производного от **Employee**, аргумент-размер равен **sizeof(Manager)**. Это позволяет специфичному для класса аллокатору не хранить информацию о размере при каждом выделении памяти. Но этот аллокатор может и хранить информацию о размере (а аллокатор общего назначения обязан это делать), и игнорировать аргумент **size_t** для **operator delete()**. В то же время, все это затрудняет оптимизацию для аллокатора общего назначения.

Как компилятор узнает правильный размер для операции **operator delete()**? Тип, указанный для операции **delete**, соответствует типу удаляемого объекта. Если мы удаляем объект по указателю на базовый класс, то базовый класс должен иметь виртуальный деструктор (§17.2.5), чтобы был предоставлен корректный размер:

```
Employee* p = new Manager; // потенциально проблема (потерян точный тип)
// ...
delete p;                  // надеемся, что Employee имеет виртуальный деструктор
```

В принципе, операция деаллокации затем выполняется деструктором (который знает размер для класса).

19.2.6. Пользовательские литералы

Язык C++ предоставляет множество литералов для встроенных типов (§6.2.6):

```
123       // int
1.2       // double
1.2F      // float
'a'       // char
1ULL      // unsigned long long
0xD0      // hexadecimal unsigned
"as"      // C-style string (const char[3])
```

Кроме того, можно определить литералы и для пользовательских типов, а также новые формы литералов для встроенных типов. Например:

```
"Hi!"s                                                  // строка, "не нуль-терминированный массив char"
1.2i                                                    // мнимое число
101010111000101b                                        // бинарное число
123s                                                    // секунды
123.56km                                                // не мили! (единицы)
12345678901234567890123456789012345678901234567890x    // расширенная точность
```

Такие *литералы, определяемые пользователем*, базируются на понятии *литеральных операций*, которые отображают литералы с заданным суффиксом в желаемый тип. Литеральная операция составляется из **operator""**, после чего следует суффикс. Пример:

```
constexpr complex<double> operator"" i(long double d)     // мнимый литерал
{
    return {0, d};  // complex это литеральный тип
}

std::string operator"" s(const char* p, size_t n)   // std::string литерал
{
    return string{p, n}; // требует аллоцирования в свободной памяти
}
```

Эти две операции определяют суффиксы **i** и **s**, соответственно. Я применяю **constexpr** для того, чтобы сделать возможным вычисление на этапе компиляции. Теперь мы можем написать:

```
template<typename T> void f(const T&);

void g()
{
    f("Hello");          // передача указателя на char*
    f("Hello"s);         // передача пятисимвольной строки
    f("Hello\n"s);       // передача шестисимвольной строки

    auto z = 2+1i;       // complex{2,1}
}
```

Базовая идея состоит в том, что после обнаружения того, что может быть литералом, компилятор проверяет суффикс. Механизм пользовательских литералов предоставляет пользователю возможность специфицировать новый суффикс и определить, что следует сделать с литералом, стоящим перед этим суффиксом. Нельзя переопределять смысл суффиксов для встроенных литералов.

Имеется четыре типа литералов, которым можно приписывать суффиксы с целью создания пользовательского литерала (§ iso.2.14.8):

- Целые литералы (§6.2.4.1): приемлемы для литеральной операции, принимающей аргументы типа **unsigned long long** или **const char***, или для шаблонной литеральной операции; например **123m** или **12345678901234567890X**

- Литералы с плавающей запятой (§6.2.5.1): приемлемы для литеральной операции, принимающей аргументы **long double** или **const char***, или для шаблонной литеральной операции; например **1234567890123456789.0.976543210x** или **3.99s**
- Строковые литералы (§7.3.2): приемлемы для литеральной операции, принимающей пару аргументов типа (**const char*, size_t**); например **"string"s** и **R"(Foo\bar)"_path**
- Символьные литералы (§6.2.3.2): приемлемы для литеральной операции, принимающей символьный аргумент типа **char, wchar_t, char16_t** или **char32_t**; например **'f'_runic** или **u'BEEF'_w**.

Для примера мы можем определить литеральную операцию для сбора цифр целых значений, которые не могут быть представлены никаким из встроенных целых типов:

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);

f(12345678901234567890123456789012345678901234567890x);
```

Здесь строка в С-стиле **"12345678901234567890123456789012345678901234567890"** передается функции **operator""x()**. Заметьте, что я не заключаю эти цифры в двойные кавычки. Я запросил строку в С-стиле для моей операции, и компилятор предоставляет ее на базе предоставленных мною цифр.

Чтобы перевести строку в С-стиле из текста программы в литеральную операцию, мы запрашиваем как саму строку, так и число символов. Например:

```
string operator"" s(const char* p, size_t n);

string s12 = "one two"s;          // вызывает operator ""("one two",7)
string s22 = "two\ntwo"s;         // вызывает operator ""("two\ntwo",7)
string sxx = R"(two\ntwo)"s;      // вызывает operator ""("two\\ntwo",8)
```

В «сырой» строке (§7.3.2.1) **"\n"** представляет два символа **'\'** и **'n'**.

Рациональное зерно в запросе числа символов заключается в том, что если мы хотим иметь «другой тип строк», то нам почти всегда нужно знать число символов.

Литеральная операция, принимающая лишь аргумент **const char*** (без размера), может применяться для целых литералов и литералов с плавающей запятой. Например:

```
string operator"" SS(const char* p);    // предупреждение: не работает как следует

string s12 = "one two"SS;               // ошибка: нет применимой литеральной операции
string s13 = 13SS;                      // OK, но зачем это делать?
```

Литеральная операция, конвертирующая числовые значения в строки, может смутить пользователей.

Шаблонная литеральная операция — это литеральная операция, принимающая свой аргумент как набор параметров шаблона, а не как аргумент функции. Например:

```
template<char...>
constexpr int operator"" _b3();     // основание 3, т.е., тернарное
```

Теперь мы можем написать:

```
201_b3   // значит operator"" b3<'2','0','1'>(); так что 9*2+0*3+1 == 19
241_b3   // значит operator"" b3<'2','4','1'>(); ошибка: 4 не тернарная цифра
```

Техника шаблонов с переменным числом параметров (§28.6) является единственным способом приписать нестандартный смысл цифрам во время компиляции.

Чтобы определить **operator"" _b3()**, нам нужны вспомогательные функции:

```
constexpr int ipow(int x, int n) // x в n-ю степень n>=0
{
    return (n>0)? x*ipow(n-1) : 1;
}

template<char c>                 // обработка случая с одной тернарной цифрой
constexpr int b3_helper()
{
    static_assert(c<'3',"not a ternary digit");
    return c;
}

template<char c, char... tail> // "обчистить" одну тернарную цифру
constexpr int b3_helper()
{
    static_assert(c<'3',"not a ternary digit");
    return ipow(3, sizeof...(tail))*(c-'0')+b3_helper(tail...);
}
```

Теперь мы можем определить литеральную операцию по основанию 3:

```
template<char... chars>
constexpr int operator"" _b3()      // основание 3, т.е., тернарное
{
    return b3_helper(chars...);
}
```

Множество суффиксов имеют тенденцию быть короткими (например, **s** для **std::string**, **i** для мнимых частей комплексных чисел, **m** для обозначения метра (§28.7.3)), так что они легко могут столкнуться друг с другом. Нужно использовать пространства имен для предотвращения возможных столкновений:

```
namespace Numerics {
    // ...

    class Bignum {/* ... */};

    namespace literals {
    Bignum operator"" x(char const*);
    }

    // ...
}

using namespace Numerics::literals;
```

Стандартная библиотека резервирует все суффиксы, которые не начинаются с начального знака подчеркивания; вам следует учесть это обстоятельство:

```
123km    // зарезервировано стандартной библиотекой
123_km   // доступно для использования
```

19.3. Класс строк

Относительно простой строковый класс, который мы здесь представим, призван проиллюстрировать дизайн классов, использующих конвенциальные операции. Наш класс **String** является упрощенной версией стандартного класса **string** (§4.2, Глава 36). Класс **String** обеспечивает семантику значений, доступ к символам с проверкой и без, потоковый ввод/вывод, поддержку циклов range-**for**, операции сравнения на равенство и операцию конкатенации. Я еще добавил для **String** литерал, что отсутствует у стандартного **std::string**.

Для того, чтобы было легче иметь дело с С-строками (включая строковые литералы (§7.3.2)), я представляю строки в виде нуль-терминированных массивов символов. Кроме того, я реализую оптимизацию коротких строк, при которой строки с небольшим числом символов хранятся непосредственно в объекте класса, а не в свободной памяти. Это улучшает эффективность в работе с небольшими строками. Опыт показывает, что в огромном числе реальных приложений роль небольших строк чрезвычайно высока. Но эта оптимизация особо значима в многопоточных приложениях, в которых разделение данных посредством указателей недоступно, а работа со свободной памятью слишком дорога.

Чтобы класс **String** мог эффективно расти при добавлении символов в конец строки, я реализую схему с хранением избыточного объема памяти, аналогичную той, что используется для **vector** (§13.6.1). Это делает **String** удобной целью для многих форм ввода.

Будет хорошей практикой попытка улучшить этот класс. И если у вас все получится (в теории), то он может заменить **std::string** (Глава 36).

19.3.1. Важные операции

Класс **String** предоставляет обычный набор конструкторов, деструктор и операции присваивания (§17.1):

```
class String {
public:
    String();                                   // умолчательный конструктор : x{""}

    explicit String(const char*p);              // конструктор из C-style строки: x{"Euler"}

    String(const String&);                      // копирующий конструктор
    String& operator=(const String&);           // присваивание копированием

    String(String&& x);                         // перемещающий конструктор
    String& operator=(String&& x);              // перемещающее присваивание

    ~String() {if (short_max<sz) delete[] ptr;} // деструктор

    // ...
};
```

Класс **String** имеет семантику значений. То есть, после присваивания **s1=s2** две строки **s1** и **s2** совершенно не связаны друг с другом, и последующие изменения одной из них никак не скажутся на другой. Альтернативной могла бы быть семантика указателей, для которой изменение одной из строк после показанного выше присваивания влияли бы и на другую строку. Всюду, где это имеет смысл, я предпочитаю семантику значений; такая же семантика у типов **vector**, **complex**, **Matrix** и **string**. Однако из-за семантики значений мы вынуждены передавать объекты типа **String** по ссылке тогда, когда нам не нужны копии, и вынуждены реализовывать перемещающую семантику (§3.3.2, §17.5.2) для оптимизации возврата.

Слегка нетривиальное представление **String** рассмотрено в §19.3.3. Отметим, что для него потребуются пользовательские версии копирующих и перемещающих операций.

19.3.2. Доступ к символам

Проектирование операций доступа для строк является непростой темой, ибо в идеале этот доступ должен выполняться операцией [] максимально эффективно и с проверкой корректности диапазона. К сожалению, трудно добиться всего этого одновременно. Здесь я буду следовать стандартной библиотеке, где используется традиционная нотация [] без проверки диапазона и применяется функция **at()** с проверкой диапазона:

```
class String {
public:
    // ...

    char& operator[](int n) {return ptr[n];}              // доступ к элементам без проверки
    char operator[](int n) const {return ptr[n];}

    char& at(int n) {check(n); return ptr[n];} // доступ к элементам с проверкой диапазона
    char at(int n) const {check(n); return ptr[n];}

    String& operator+=(char c);                           // добавить с в конец

    const char* c_str() {return ptr;}                     // доступ к строке в С-стиле
    const char* c_str() const {return ptr;}

    int size() const {return sz;}                         // число элементов
    int capacity() const                          // элементы плюс доступное пространство
        {return (sz<=short_max)? short_max : sz+space;}

    // ...
};
```

Предполагается применять [] в обычных случаях. Например:

```
int hash(const String& s)
{
    int h {s[0]};
    for (int i {1}; i!=s.size(); i++) h ^= s[i]>>1;    // доступ к s без проверки
    return h;
}
```

Здесь применение проверяемой версии **at()** излишне, поскольку мы корректно выполняем доступ к **s** в диапазоне от **0** до **s.size()-1**.

Мы используем **at()** там, где с нашей точки зрения есть шанс совершить ошибку. Например:

```
void print_in_order(const String& s, const vector<int>& index)
{
    for (x : index) cout << s.at(x) << '\n';
}
```

Я предоставляю **const** и не-**const** версии функций доступа, чтобы можно было работать как с константными объектами, так и с неконстантными.

19.3.3. Представление

Представление для **String** было выбрано в соответствии с тремя целями:

- Чтобы было легко конвертировать С-строку (например, литерал) в **String** и чтобы был простой доступ к символам как к С-строке

- Для минимизации применения свободной памяти
- Чтобы добавление символов в конец строки было эффективным

Ясно, что результат получается несколько более сумбурным по сравнению с простым представлением {указатель, размер}, но более реалистичным:

```
class String {
/*
    Простая строка с оптимизацией коротких строк

    size()==sz это число элементов
    если size()<= short_max, символы хранятся в самом объекте типа String;
    в противном случае используется свободная память.

    ptr указывает на начало последовательности символов
    последовательность символов терминирована нулем: ptr[size()]==0;
    это позволяет исполь. строковые функ-и С-библиот. и легко возвращать С-стр.: c_str()

    Для эффектив. добавки символов в конец, String растет удвоением объема аллокации;
    capacity() это объем памяти, доступный символам
    (за исключением терминирующего нуля): sz+space
*/

public:
    // ...
private:
    static const int short_max = 15;
    int sz;                         // проверка диапазона
    char* ptr;
    union {
        int space;                  // неиспользуемая часть аллоцированной памяти
        char ch[short_max+1];       // оставляем место для терминирующего нуля
};

void check(int n) const             // проверка диапазона
{
    if (n<0 || sz<=n)
        throw std::out_of_range("String::at()");
}

    // функции-члены:
    void copy_from(const String& x);
    void move_from(String& x);
};
```

Оптимизация коротких строк достигается использованием двух строковых представлений:

- Если **sz<=short_max**, то символы хранятся в самом объекте **String** в массиве с именем **ch**.
- Если!**(sz<=short_max)**, то символы хранятся в свободной памяти и можно выделять избыточную память для роста. Член с именем **space** равен числу таких символов.

В обоих случаях **ptr** указывает на элементы. Это важно для производительности: функциям доступа не нужно проверять, какой тип представления используется; они просто используют **ptr**. Только конструкторы, присваивания, перемещения и деструктор (§19.3.4) должны заботиться об этих альтернативах.

Мы используем массив **ch** только, если **sz<=short_max** и целое число **space**, если!**(sz<=short_max)**. Как следствие, было бы неоптимально выделять память и для **ch**,

и для **space** в объектах **String**. Поэтому я использую **union** (§8.3). В частности, я использую форму **union**, известную как анонимное объединение (§8.3.2), которая специально предназначена для того, чтобы класс мог управлять альтернативными представлениями объектов. Все члены анонимного объединения располагаются в одной и той же области памяти с одним и тем же стартовым адресом. Только один член может быть актуальным и использоваться в один и тот же момент времени, но в остальном они используются точно так же, как если бы они были независимыми членами в пространстве имен, окружающим анонимное объединение. Программист несет ответственность за то, чтобы члены анонимного объединения использовались корректно.

19.3.3.1. Вспомогательные функции

Кроме функций, предназначенных для общего употребления, я нашел удобным ввести три вспомогательные функции в качестве строительных блоков, помогающих мне с несколько нетривиальным представлением, а также для минимизации кода. Двум функциям нужен доступ к представлению класса **String**, так что я сделал их функциями-членами. Но я снабдил их спецификатором **private**, поскольку они не относятся к общеупотребительным и безопасным операциям. Во многих случаях реализация класса — это не объединение представления с открытыми функциями-членами. Важны также и вспомогательные функции, устраняющие дуплицирование кода, улучшающие дизайн и помогающие в поддержке кода.

Первая из вспомогательных функций перемещает символы во вновь выделенную память:

```
char* expand(const char* ptr, int n)     // расширяемся в свободной памяти
{
    char* p = new char[n];
    strcpy(p, ptr); // §43.4
    return p;
}
```

Эта функция не имеет прямого доступа к представлению **String**, и поэтому я не стал делать ее функцией-членом.

Вторая из функций используется копирующими операциями, чтобы дать объекту типа **String** копию членов другого объекта:

```
void String::copy_from(const String& x)
    // делаем *this копией x
{
    if (x.sz<=short_max) {                  // копируем *this
        memcpy(this,&x, sizeof(x));         // §43.5
        ptr = ch;
    }
    else {                                  // копируем элементы
        ptr = expand(x.ptr, x.sz+1);
        sz = x.sz;
        space = 0;
    }
}
```

Необходимую очистку целевого объекта **String** должен делать тот, кто вызывает **copy_from()**; функция **copy_from()** переписывает целевой объект в безусловном порядке. Я использую стандартную функцию **memcpy()** (§43.5) для копирования байт источника в целевой объект. Это весьма низкоуровневая функция. Ее следует использовать там, где нет никаких объектов с конструкторами и деструкторами, ибо эта функция о таком ничего не знает. Обе копирующие операции у типа **String** используют функцию **copy_from()**.

А вот аналогичная функция для перемещающих операций:

```
void String::move_from(String& x)
{
    if (x.sz<=short_max) {              // копируем *this
        memcpy(this,&x, sizeof(x));     // §43.5
        ptr = ch;
    }
    else {                              // захват элементов
        ptr = x.ptr;
        sz = x.sz;
        space = x.space;
        x.ptr = x.ch;                   // x = ""
        x.sz = 0;
        x.ch[0]=0;
    }
}
```

Эта функция также в безусловном порядке превращает целевой объект в копию своего аргумента. Но она не оставляет аргумент владеть свободной памятью. Я мог бы использовать **memcpy()** и в версии для длинных строк, но поскольку в этом случае используется лишь часть представления **String**, то я предпочел копировать используемые члены индивидуально.

19.3.4. Функции-члены

Умолчательный конструктор формирует пустую строку:

```
String::String()                    // умолчательный конструктор : x{""}
        : sz{0}, ptr{ch}            // ptr указует элементы, ch начальная позиция (§ 19.3.3)
    {
        ch[0] = 0;                  // термиирующий нуль
    }
```

Имея функции **copy_from()** и **move_from()**, несложно реализовать конструкторы, перемещающие операции и присваивания. Конструктор, принимающий С-строку, должен определить число символов и хранить их соответственно:

```
String::String(const char* p)
    : sz{strlen(p)},
    ptr{(sz<=short_max)? ch : new char[sz+1]},
    space{0}
{
    strcpy(ptr, p);  // копируем символы в ptr из p
}
```

Если аргумент является короткой строкой, **ptr** устанавливается так, чтобы указывать на **ch**; в противном случае выделяется память в куче. В обоих случаях символы копируются из строки-аргумента в память под управлением **String**.

Копирующий конструктор просто копирует представление своего аргумента:

```
String::String(const String& x)     // копирующий конструктор
    copy_from(x);                   // копируем представление из x
}
```

Я не беспокоюсь о том, чтобы оптимизировать случай, когда размер источника равен размеру целевого объекта (как это делается для **vector**; §13.6.3). Вряд ли это стоит отдельного внимания.

Аналогично, перемещающий конструктор передвигает представление из источника (и возможно устанавливает аргумент пустой строкой):

```
String::String(String&& x) // перемещающий конструктор
{
    move_from(x);
}
```

Как и копирующий конструктор, копирующее присваивание использует **copy_from()** для клонирования представления своего аргумента. Кроме того, приходится удалять (**delete**) владение свободной памятью целевым объектом, а также позаботиться о самоприсваивании (**s=s**):

```
String& String::operator=(const String& x)
{
    if (this==&x) return *this;      // разбираемся с самоприсваиванием
    char* p = (short_max<sz)? ptr : 0;
    copy_from(x);
    delete[] p;
    return *this;
}
```

Перемещающее присваивание удаляет владение свободной памятью целевым объектом (если имеется), а затем выполняет перемещение:

```
String& String::operator=(String&& x)
{
    if (this==&x) return *this;          // самоприсваивание: (x = move(x) это не нужно)
    if (short_max<sz) delete[] ptr;      // удаляем целевой объект
    move_from(x);                        // не генерирует исключения
    return *this;
}
```

Логически можно переместить объект в самого себя (**s=std::move(s)**), так что снова нужно позаботиться о таком вырожденном случае (хотя и крайне маловероятном).

Наиболее сложной операцией для **String** является операция **+=**, которая добавляет символ к концу строки, увеличивая размер на единицу:

```
String& String::operator+=(char c)
{
    if (sz==short_max) {             // расширяем до long string
        int n = sz+sz+2;             // удваиваем аллокацию (+2 из-за терминирующего нуля 0)
        ptr = expand(ptr, n);
        space = n-sz-2;
    }
    else if (short_max<sz) {
        if (space==0) {              // расширяем в свободную память
            int n = sz+sz+2;         // удваиваем аллокацию (+2 из-за терминирующего нуля)
            char* p = expand(ptr, n);
            delete[] ptr;
            ptr = p;
            space = n-sz-2;
        }
        else
            --space;
    }
    ptr[sz] = c;                     // добавить c в конец
    ptr[++sz] = 0;                   // увеличиваем размер и ставим терминирующий нуль

    return *this;
}
```

Здесь много чего делается: от **operator+=()** требуется отслеживать, какое представление (короткое или длинное) используется и есть ли свободное пространство памяти для внедрения символа. Если нужна дополнительная память, вызывается **expand()** для выделения новой памяти и перемещения туда старых символов. Старую память нужно удалить. Если же имеющейся памяти достаточно, то внедрить туда новый символ **c** и добавить терминальный **0** не составит труда.

Все члены **String** стараются не модифицировать представление раньше времени, то есть до тех пор, пока новый вариант не будет готов окончательно. В частности, они не вызывают **delete** ранее, чем все потенциальные операции **new** будут выполнены успешно. Фактически, члены **String** предоставляют строгую гарантию исключений (§13.2).

Самостоятельное написание кода для строкового класса, для класса векторов и т.д. является великолепным упражнением. Оно прекрасно показывает, как много усилий помогает сохранить стандартная библиотека, версиями которых для обозначенных классов и нужно далее пользоваться.

19.3.5. Полезные функции

Чтобы завершить наш проект класса **String** я добавил несколько полезных функций: функции ввода/вывода, поддержку для циклов range-**for**, сравнения и конкатенацию. Все это похоже на дизайн стандартного класса **std::string**. В частности, операция **<<** просто выводит символы без дополнительного форматирования, а операция **>>** пропускает начальные пробелы перед чтением:

```
ostream& operator<<(ostream& os, const String& s)
{
    return os << s.c_str();    // §36.3.3
}

istream& operator>>(istream& is, String& s)
{
    s = "";                    // очищаем целевую строку
    is>>ws;                    // опускаем пробель. символы (§38.4.5.1)
    char ch = ' ';
    while(is.get(ch) &&!isspace(ch))
        s += ch;
    return is;
}
```

Я реализовал == и != для сравнений:

```
bool operator==(const String& a, const String& b)
{
    if (a.size()!=b.size())
        return false;
    for (int i = 0; i!=a.size(); ++i)
        if (a[i]!=b[i])
            return false;
    return true;
}

bool operator!=(const String& a, const String& b)
{
    return!(a==b);
}
```

Нетрудно будет добавить операцию **<** и др.

Для поддержки циклов range-**for** нам нужны функции **begin()** и **end()** (§9.5.1). Опять-таки, мы можем реализовать их как глобальные (не-члены) функции:

```
char* begin(String& x)   // доступ к строке в C-стиле
{
    return x.c_str();
}

    char* end(String& x)
{
    return x.c_str()+x.size();
}

    const char* begin(const String& x)
{
    return x.c_str();
}

const char* end(const String& x)
{
    return x.c_str()+x.size();
}
```

Отталкиваясь от уже определенной операции +=, добавляющей символ в конец строки, нетрудно определить глобальную функцию конкатенации:

```
String& operator+=(String& a, const String& b) // конкатенация
{
    for (auto x : b)
        a+=x;
    return a;
}

String operator+(const String& a, const String& b) // конкатенация
{
    String res {a};
    res += b;
    return res;
}
```

Вот как используется конкатенация с С-строкой:

```
String s = "Njal ";
s += "Gunnar";      // конкатенируем: добавляем в конец s
```

Применение операции += интерпретируется как вызов **operator+=(s, String("Gunnar"))**. Я не стал «играть» в оптимизацию этой операции, ибо вряд ли это будет иметь серьезный смысл в реальном мире: я не стал для этого принимать во внимание размер строки-источника.

Определение суффикса для литералов тривиально:

```
String operator"" _s(const char* p, size_t)
{
    return String{p};
}
```

Теперь мы можем написать:

```
void f(const char*);      // C-style строка
void f(const String&);    // наша строка
```

```
void g()
{
    f("Madden's");          // f(const char*)
    f("Christopher's"_s);   // f(const String&);
}
```

19.3.6. Использование класса String

Напишем небольшую программу, которая просто тестирует операции класса **String**:

```
int main()
{
    String s ("abcdefghij");
    cout << s << '\n';
    s += 'k';
    s += 'l';
    s += 'm';
    s += 'n';
    cout << s << '\n';

    String s2 = "Hell";
    s2 += " and high water";
    cout << s2 << '\n';

    String s3 = "qwerty";
    s3 = s3;
    String s4 ="the quick brown fox jumped over the lazy dog";
    s4 = s4;
    cout << s3 << " " << s4 << "\n";
    cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";

    String buf;
    while (cin>>buf && buf!="quit")
        cout << buf << " " << buf.size() << " " << buf.capacity() << '\n';
}
```

Нашему классу **String** недостает многого из того, что вы могли бы счесть полезным и важным. Но главное в нем то, что он напоминает стандартный класс **std::string** (Глава 36) и иллюстрирует программные техники, применяемые в реализации стандартного библиотечного **string**.

19.4. Друзья

Обычное объявление функции-члена специфицирует три вещи:

1. Функция может иметь доступ к закрытой части объявления класса.
2. Функция находится в области видимости класса.
3. Функция вызывается от лица объекта класса (имеет указатель **this**).

Если объявить функцию-член с ключевым словом **static** (§16.2.12), то мы оставляем за этой функцией лишь первые два свойства. А если объявить глобальную функцию (не член) с ключевым словом **friend**, то мы придадим этой функции первое из перечисленных выше свойств.

К примеру, нам нужна операция для перемножения **Matrix** на **Vector**. Естественно, что классы **Vector** и **Matrix** прячут свои представления и предоставляют набор операций для работы со своими объектами. Но у нас сейчас потребовалась операция, работающая над

объектами обоих типов. При этом мы не хотим определять низкоуровневую операцию, которая имела бы возможность читать и писать в представления обоих классов. В итоге, нам лучше объявить эту функцию *другом* (*friend*) этих классов:

```
constexpr rc_max {4}; // размер строки и столбца

class Matrix;

class Vector {
    float v[rc_max];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

class Matrix {
    Vector v[rc_max];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};
```

Теперь **operator*()** может достигать представлений как **Vector**, так и **Matrix**. Вот простая реализация этой операции:

```
Vector operator*(const Matrix& m, const Vector& v)
{
    Vector r;
    for (int i = 0; i!=rc_max; i++) { // r[i] = m[i] * v;
        r.v[i] = 0;
        for (int j = 0; j!=rc_max; j++)
            r.v[i] += m.v[i].v[j] * v.v[j];
    }
    return r;
}
```

Объявление друга можно помещать как в закрытую, так и в открытую части объявления класса — это не имеет значения. Таким образом, объявление друга аналогично объявлению функции-члена класса, для которого она является другом. То есть она фактически входит в интерфейс класса, как и его функции-члены.

Функция-друг одного класса может быть другом и для другого класса. Пример:

```
class List_iterator {
    // ...
    int* next();
};

class List {
    friend int* List_iterator::next();
    // ...
};
```

Имеется способ сократить объявления всех функций одного класса друзьями другого класса. Пример:

```
class List {
    friend class List_iterator;
    // ...
};
```

Здесь все функции-члены класса **List_iterator** объявлены друзьями для класса **List**.

Для класса-друга мы заранее не знаем весь состав функций, получающих доступ к классу, для которого объявлена дружба. В этом отношении объявление класса-друга отличается от объявлений функций-членов и функций-друзей. Ясно, что объявлять классы-друзья стоит с осторожностью. Они годятся для отражения лишь крайне близко связанных концепций.

Можно объявить другом аргумент шаблона:

```
template<typename T>
class X {
    friend T;
    friend class T; // избыточный "класс"
    // ...
};
```

Часто возникает альтернатива: делать класс членом другого класса или же классом-другом (§18.3.1).

19.4.1. Поиск друзей

Друг должен быть изначально объявлен в окружающем пространстве имен или же определен в неклассовом пространстве имен, непосредственно окружающим класс, который «объявляет дружбу». Рассмотрим технический пример:

```
class C1 {};                    // будет другом для N::C
void f1();                      // будет другом для N::C

namespace N {
    class C2 {};                // будет другом для C
    void f2() {}                // будет другом для C

    class C {
        int x;
    public:
        friend class C1;        // OK (ранее определено)
        friend void f1();

        friend class C3;        // OK (определено в объемлющем пространстве имен)
        friend void f3();
        friend class C4;        // Впервые объявлено в N и полагается быть в N
        friend void f4();
    };

    class C3 {};                // друг C
    void f3() {C x; x.x = 1;}   // OK: друг C
} // namespace N

class C4 {};                    // не друг для N::C
void f4() {N:: C x; x.x = 1;}   // ошибка: x закрыт и f4() не друг для N::C
```

Функция-друг может идентифицироваться по ее аргументам (§14.2.4), даже если она не объявлена в самом близком окружающем пространстве имен. Например:

```
void f(Matrix& m)
{
    inver t(m); // друг Matrix – invert()
}
```

Таким образом, функция-друг должна быть явно объявлена в окружающем пространстве имен или же должна принимать аргумент этого класса (или производного от него). Например:

```
// нет f() в этой области видимости

class X {
    friend void f();              // без пользы
    friend void h(const X&);      // можно найти через аргумент
};

void g(const X& x)
{
    f();                          // нет f() в области видимости
    h(x);                         // друг X – h()
}
```

19.4.2. Друзья и функции-члены

Когда следует использовать функцию-друга и когда следует предпочесть функцию-член для реализации операции. Сначала следует минимизировать число функций, имеющих доступ к представлению класса, а потом следует тщательно оптимизировать эти функции. Обычно таких функций остается меньше, чем нам кажется с самого начала. Некоторые операции обязаны быть членами — например, конструкторы, деструкторы и виртуальные функции (§3.2.3, §17.2.5), а в остальных случаях возможен выбор. Так как все имена членов локальны в классе, то функцию, требующую доступа к представлению, следует делать членом, если только нет каких-либо специфических причин делать ее глобальной.

Рассмотрим класс **X**, который предоставляет альтернативные способы реализации операции:

```
class X {
    // ...
    X(int);

    int m1();                     // член
    int m2() const;

    friend int f1(X&);            // друг, не член
    friend int f2(const X&);
    friend int f3(X);
};
```

Функции-члены вызываются только от имени объектов класса; никаких пользовательских приведений типа не применяется к левым операндам операций. и **->** (см. §19.2.3). Например:

```
void g()
{
    99.m1();  // ошибка: X(99).m1() не испытывалась
    99.m2();  // ошибка: X(99).m2() не испытывалась
}
```

У глобальной функции **f1()** те же особенности, поскольку неявные приведения не используются для неконстантных ссылок (§7.7). К аргументам **f2()** и **f3()** преобразования применимы:

```
void h()
{
    f1(99);   // ошибка: f1(X(99)) не испытывалась: не-const X& аргумент
    f2(99);   // OK: f2(X(99)); const X& аргумент
    f3(99);   // OK: f3(X(99)); X аргумент
}
```

Операции, модифицирующей состояние классового объекта, следует быть членом или функцией, принимающей аргумент в виде неконстантной ссылки (или неконстантного указателя).

Операции, модифицирующие операнд (вроде =, *= и ++) наиболее естественно определяются как члены для пользовательских типов. И наоборот, если неявные приведения желательны для всех операндов операции, то реализующая операцию функция должны быть глобальной функцией (не членом), принимающей константную ссылку в качестве аргумента или же аргумент, не являющийся ссылкой. Это типично для функций, реализующих операции, не требующие lvalue-операндов при работе с фундаментальными типами (+, - и ||). Однако, такие операции часто нуждаются в доступе к представлению классов их операндов. Следовательно, бинарные операции являются общим источником функций-друзей.

Если только не определены преобразования типов, то нет никаких четких предпочтений для членов перед друзьями, принимающими аргумент-ссылку, и наоборот. Часто программист может иметь индивидуальныве предпочтения и выбрать, например, **m2=inv(m)** для получения инвертированной матрицы из **m**, а может предпочесть и вариант **m2=m.inv()**. Но если **inv()** инвертирует непосредственно **m**, а не производит новую матрицу, то этой функции следует быть членом.

При прочих равных реализуйте операции, требующие доступа к представлению, как функции-члены:

- Заранее неизвестно, что кто-нибудь впоследствии не определит операцию преобразования.
- Синтаксис вызова функции-члена ясно показывает пользователю, что объект может быть модифицирован; аргумент-ссылка не так очевиден.
- Выражения в теле функции-члена могут быть существенно короче таковых у глобальной функции (например, за счет неявного **this**).
- Имена членов локальны в классе, так что имеет место тенденция к их укорочению.

И наоборот, операции, не требующие прямого доступа к представлению, часто имеют наилучшую реализацию в виде глобальных функций (не членов), возможно в пространстве имен, которое логически раскрывает ее связь с классом (§18.3.6).

19.5. Советы

[1] Используйте **operator[]()** для индексирования или для выбора на основе одного значения; §19.2.1.

[2] Используйте **operator()()** для реализации семантики вызова, для индексирования и для выбора на основе нескольких значений; §19.2.2.

[3] Используйте **operator->()** для разыменования «интеллектуальных указателей»; §19.2.3.

[4] Предпочитайте префиксную операцию ++; §19.2.4.

[5] Определяйте глобальные **operator new()** и **operator delete()** только в случае крайней необходимости; §19.2.5.

[6] Определяйте **operator new()** и **operator delete()** функциями-членами ради контроля за выделением памяти и освобождением ее под объектами класса из иерархий; §19.2.5.

[7] Используйте пользовательские литералы для имитации традиционной нотации; §19.2.6.

[8] Размещайте литеральные операции в отдельных пространствах имен; §19.2.6.

[9] В общем случае предпочитайте стандартный **string** (Глава 36) всем своим попыткам усовершенствовать строковый класс; §19.3.

[10] Используйте функцию-друга, если вам нужна глобальная функция с доступом к представлению класса; §19.4.

[11] Предпочитайте функции-члены друзьям для разрешения доступа к реализации класса; §19.4.2.

20

Производные классы

Не плодите сущности сверх меры.
— Уильям Оккам

- Введение
- Производные классы
  Функции-члены; Конструкторы и деструкторы
- Классовые иерархии
  Поля типов; Виртуальные функции; Явная квалификация; Управление замещением; Использование членов базового класса; Ослабление типа возврата
- Абстрактные классы
- Управление доступом
  Защищенные члены; Доступ к базовым классам; Контроль доступа и **using**-объявления
- Указатели на члены
  Указатели на функции-члены; Указатели на члены-данные; Члены базового и производного классов
- Советы

20.1. Введение

Язык C++ заимствовал из языка Simula идею о классах и классовых иерархиях. Кроме того, он заимствовал идею о том, что классы предназначены для моделирования концепций программы и реального мира. В итоге, язык C++ предоставляет конструкции, которые напрямую поддерживают эти идеи проектирования. Конечно, средства языка C++ можно использовать по-разному: можно с их помощью подпереть проектные идеи, а можно лишь использовать их в узких вспомогательных целях.

Никакая концепция не существует в полной изоляции. Она связаны с другими концепциями и черпает мощь из этой связи. Действительно, попробуйте объяснить, что есть автомобиль? Вскоре вам придется работать с такими концепциями, как колеса, руль, мотор, водитель, дороги, топливо и т.д. Поскольку мы собираемся использовать классы для отражения концепций в программе, то нам нужно подумать, как представляются связи между концепциями. Естественно, в языке программирования нельзя отразить связи смежду концепциями так, как это делается в реальном мире. Нам нужно сузить возможность отражения связей и четко определить необходимые средства языка.

Понятие производного класса и связанные с этим механизмы языка предназначены для отражения иерархических взаимосвязей, то есть для отражения общности между классами. К примеру, концепции круга и треугольника связаны, поскольку обе они отра-

жают формы; то есть форма, это общее для двух концепций. В языке C++ мы определяем классы **Circle** и **Triangle** так, чтобы для них общим был класс **Shape**. В таком контексте класс **Shape** называют *базовым классом* (или суперклассом), а классы **Circle** и **Triange** называют *производными* от базового класса. Если в программе не отразить рассмотренной общности, то может быть потеряно что-то существенное. То, что мы будем рассматривать в настоящей главе, лежит в основе *объектно-ориентированного программирования*.

Язык поддерживает построение новых классов на основе существующих:

- *Наследование реализации*: осуществляется экономия на усилиях по реализации посредством совместного использования того, что уже реализовано в базовом классе
- *Наследование интерфейса*: чтобы позволить использовать разные производные классы взаимозаменяемым образом посредством общего интерфейса, предоставляемого базовым классом

Часто наследование интерфейса называют *полиморфизмом времени выполнения*. По контрасту, однородное использование классов, не связанных наследованием, посредством шаблонов (§3.4, Глава 23) часто называют *полиморфизмом времени компиляции*.

Обсуждение классовых иерархий распространяется на три главы:

- *Производные классы* (Глава 20): рассматриваются основные средства языка для поддержки объектно-ориентированного программирования: базовые и производные классы, виртуальные функции и контроль доступа.
- *Иерархии классов* (Глава 21): в основу положено рассмотрение вопроса о том, как базовые и производные классы помогают организовать код на базе классовых иерархий; также рассматриваются технические вопросы, в ключая множественное наследование.
- *Идентификация типа на этапе выполнения* (Глава 22): рассматриваются способы явной навигации по классовым иерархиям; особенное внимание уделяется операциям **dynamic_cast** и **static_cast**, а также операции **typeid**.

Краткое изложение базовой идеи организации типов в иерархии можно найти в Главе 3: базовые и производные классы (§3.2.2) и виртуальные функции (§3.2.3). В последующих главах эти фундаментальные идеи и связанные с ними средства языка рассматриваются более глубоко и подробно.

20.2. Производные классы

Рассмотрим для примера программу, которая работает с людьми, нанятыми для работы в некоторой фирме. У этой программы может быть следующая структура данных:

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short depar tment;
    // ...
};
```

Потом можно определить структуру для управляющих (менеджеров):

```
struct Manager {
    Employeeemp;            // запись служащего у менеджера
    list<Employee*> group;  // нанятые люди
    short level;
    // ...
};
```

Но менеджер также является работником (employee); данные, относящиеся к работникам, хранятся в поле **emp** объектов типа **Manager**. Это очевидно для читателя-человека, но тут нет ничего такого, что говорило бы компилятору о том, что **Manager** одновременно является и **Employee**. В итоге, нельзя использовать **Manager*** там, где требуется **Employee***. Нельзя поместить **Manager** в список работников, не разработав для этого специальный код, который вряд ли будет простым и элегантным.

Правильным решением будет явное объявление того факта, что **Manager** есть **Employee** с добавлением небольшой порции информации:

```
struct Manager : public Employee {
    list<Employee*> group;
    short level;
    // ...
};
```

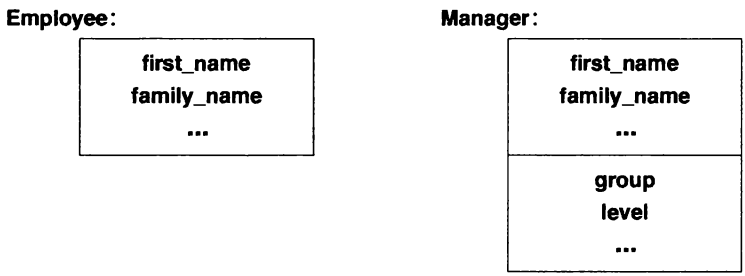
Тип **Manager** наследуется от типа **Employee**, так что **Employee** является базовым классом для **Manager**. У класса **Manager** имеются члены класса **Employee** (**first_name**, **department** и т.д.) плюс собственные члены (**group**, **level** и т.д.).

Наследование часто отражают графически стрелкой, направленной от изображения производного класса к базовому классу, что означает, что производный класс имеет отсылку к базовому классу (а не наоборот):



Производный класс наследует свойства базового класса, поэтому и говорят о механизме наследования. Часто применяют и другую терминологию — суперкласс и подкласс (для производного класса). Такая терминология немного смущает, ибо объекты производного класса содержат больше данных, чем объекты базового класса.

Объект производного класса часто изображают как объект базового класса, к которому в конец добавляют информацию, специфичную для производного класса:



Таким образом, дополнительная память под объект производного класса требуется ровно для дополнительных членов и нет никаких других накладных расходов.

Наследование делает **Manager** подтипом **Employee**, так что можно использовать **Manager** всюду, где приемлем **Employee**. Например, теперь мы можем составить список работников, часть из которых относятся к управляющим (то есть к типу **Manager**):

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist {&m1,&e1};
    // ...
}
```

Так как **Manager** есть одновременно и **Employee**, то можно использовать **Manager*** там, где используется **Employee***. Аналогично, можно использовать **Manager&** вместо **Employee&**. В обратном направлении сказанное неверно: **Employee** не обязательно является **Manager**, так что нельзя применять **Employee*** там, где требуется **Manager***. Обобщаем: если класс **Derived** имеет открытый базовый класс (§20.5) **Base**, то **Derived*** можно присваивать переменной типа **Base*** без явного приведения типов. В обратном направлении требуется явное приведение типов. Например:

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;          // OK: каждый Manager есть одновременно Employee
    Manager* pm = &ee;           // ошибка: не каждый Employee есть Manager

    pm->level = 2;               // катастрофа: ee не имеет level

    pm = static_cast<Manager*>(pe); // грубая сила: работает из-за того, что pe
                                    // указывает на Manager mm

    pm->level = 2;               // чудесно: pm указывает на Manager mm, имеющего level
}
```

Другими словами, объект производного класса можно трактовать как объект базового класса, когда объектами манипулируют через указатели или ссылки. Обратное неверно. Применение **static_cast** и **dynamic_cast** обсуждается в §22.2.

Класс должен быть определен, чтобы его можно было использовать в качестве базового (§8.2.2):

```
class Employee;                  // только объявление, не определение

class Manager : public Employee {    // ошибка:  Employee не определен
    // ...
};
```

20.2.1. Функции-члены

Простые структуры данных, такие как **Employee** и **Manager**, не настолько интересны и не слишком полезны. Нам требуются типы с соответствующим набором операций, и это нужно сделать так, чтобы не привязываться сильно к конкретным представлениям. Например:

```
class Employee {
public:
    void print() const;
    string full_name() const {return first_name + ' ' + middle_initial + ' ' + family_name;}
    // ...
private:
    string first_name, family_name;
    char middle_initial;
    // ...
};

class Manager : public Employee {
public:
    void print() const;
    // ...
};
```

Члены производного класса могут использовать открытые (а также **protected**; §20.5) члены базового класса так, как будто они объявляются в производном классе. Например:

```
void Manager::print() const
{
    cout << "name is " << full_name() << '\n';
    // ...
}
```

Но производный класс не может напрямую обращаться к закрытым членам базового класса:

```
void Manager::print() const
{
    cout << " name is " << family_name << '\n';      // ошибка!
    // ...
}
```

Эта версия **Manager::print()** не будет скомпилирована, поскольку **family_name** недоступен для **Manager::print()**.

И хотя такое ограничение может кого-то удивить, оно логичное. Иначе была бы скомпрометирована сама идея закрытых членов. А найти в коде все точки доступа к таким членам стало бы проблематично, ибо помимо членов и функций-друзей пришлось бы просматривать код программы повсюду, ибо где-угодно может появиться производный класс с искомым доступом. Поэтому для производных классов доступны открытые и защищенные (**protected**) члены базового класса (§20.5).

Наиболее ясной стратегией для производного класса будет использование лишь открытых членов базового класса. Например:

```
void Manager::print() const
{
    Employee::print();   // выводим информацию об Employee
    cout << level;       // выводим специфичную для Manager информацию
    // ...
}
```

Отметим, что нужно использовать::, поскольку функция **print()** переопределяется в **Manager**. Такое повторное использование имен вполне типично. В случае непонимания кто-то может написать:

```
void Manager::print() const
{
    print(); // упс!
    // выводим информацию, специфичную для Manager
}
```

Результатом будет серия рекурсивных вызовов, приводящих к краху программы.

20.2.2. Конструкторы и деструкторы

Как всегда, конструкторы и деструкторы играют важную роль:

- Объекты конструируются снизу-вверх, а уничтожаются сверху-вниз; §17.2.3.
- Каждый класс может инициализировать свои члены и базовый класс (но не напрямую); §17.4.1.
- В типичном случае деструкторы в иерархии должны быть виртуальными; §17.2.5.
- Копирующие конструкторы классов из иерархий следует использовать с осторожностью, чтобы избежать срезки; §17.5.1.4.

- Как разрешаются вызовы виртуальной функции и как выполняется работа операций **dynamic_cast** или **typeid** в конструкторе или деструкторе — зависит от стадии конструирования или деструкции; §22.4.

Когда я говорю о конструировании объектов снизу-вверх, я под этим понимаю, что все начинается с самого фундаментального (с базового класса) и далее строится то, что от этого зависит (производный класс). То есть строительство идет от корней (базового класса) в направлении листьев (производный класс).

20.3. Классовые иерархии

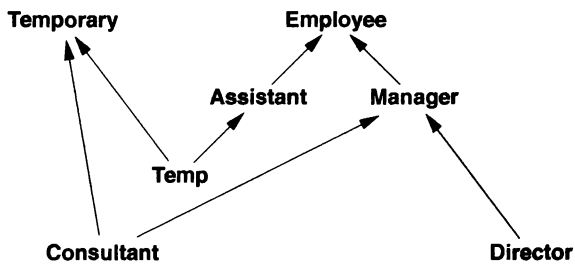
Производный класс сам может быть базовым. Например:

```
class Employee {/* ... */};
class Manager : public Employee {/* ... */};
class Director : public Manager {/* ... */};
```

Такие наборы связанных друг с другом классов традиционно называются *классовыми иерархиями*. Чаще всего иерархии являются деревьями, но могут образовывать и более сложные структуры графов. Например:

```
class Temporary {/* ... */};
class Assistant : public Employee {/* ... */};
class Temp : public Temporary, public Assistant {/* ... */};
class Consultant : public Temporary, public Manager {/* ... */};
```

Или в графической форме:



Таким образом, как это в деталях рассматривается в §21.3, язык C++ может отразить направленный ациклический граф классов.

20.3.1. Поля типов

С целью использовать производные классы не просто как удобные сокращения нотаций, мы должны решить следующую проблему: если у нас есть указатель **Base***, то на объект какого производного класса он реально указывает? Существуют четыре возможности решения проблемы:

1. Гарантировать единственный тип указуемого объекта (§3.4, Глава 23).
2. Поместить поле типа в базовый класс.
3. Использовать **dynamic_cast** (§22.2, §22.6).
4. Использовать виртуальные функции (§3.2.3, §20.3.2).

Если только вы не используете **final** (§20.3.4.2), решение номер один требует больших знаний о используемых типах, чем это может позволить себе компилятор. И воообще, это не очень хорошая идея быть умнее системы типов (особенно в комбинации с шаблонами),

но это может использоваться для реализации однородных контейнеров (вроде стандартных **vector** и **map**) с непревзойденной производительностью. Решения 2, 3 и 4 могут использоваться для построение гетерогенных списков, то есть списков указателей на объекты нескольких разных типов. Решение 3 является поддержанным языком вариантом решения 2. Решение 4 является специальным вариантом решения 2, безопасным по типам. Комбинации решений 1 и 4 особо интереересны; почти во всех ситуациях они приводят к более ясному коду, чем в случае решений 2 и 3.

Давайте сначала рассмотрим простое решение с полем типа, чтобы понять, почему его лучше всего избегать. Пример с работниками и менеджерами переопределим следующим образом:

```
struct Employee {
    enum Empl_type {man, empl};
    Empl_type type;

    Employee() : type{empl} {}

    string first_name, family_name;
    char middle_initial;

    Date hiring_date;
    short department;
    // ...
};

struct Manager : public Employee {
    Manager() {type = man;}

    list<Employee*> group;  // люди, которыми управляет менеджер
    short level;
    // ...
};
```

Теперь мы можем написать функцию, которая печатает информацию о каждом работнике:

```
void print_employee(const Employee* e)
{
    switch (e->type) {
    case Employee::empl:
        cout << e->family_name << '\t' << e->department << '\n';
        // ...
        break;
    case Employee::man:
    {   cout << e->family_name << '\t' << e->department << '\n';
        // ...
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
        break;
    }
    }
}
```

И потом можно использовать эту функцию для печати списка работников следующим образом:

```
void print_list(const list<Employee*>& elist)
{
    for (auto x : elist)
        print_employee(x);
}
```

Это прекрасно работает, особенно в случае небольших программ, создаваемых и поддерживаемых одним человеком. Однако, здесь имеется фундаментальная проблема, состоящая в том, что манипуляции программиста с типом непонятны компилятору. Это приводит к неприятным последствиям, так как функции вроде **print_employee()** могли бы использовать преимущества от общности классов, с которыми работают:

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::man) {
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
    }
}
```

В большой программе будет непросто разыскать все эти проверки типов в функциях, работающих со многими производными классами. Кроме того, довольно трудно понять все эти проверки. Наконец, если мы добавим новый тип работника, то это повлечет внесение изменений во все ключевые функции — во все, в которых выполняются проверки полей типов.

Другими словами, использование полей типов является порождающей ошибки техникой программирования, которая к тому же добавляет проблем и при поддержке кода. По мере роста объема программы растут и издержки на работу с полями типов, что противоречит стремлению к модульности и сокрытию данных. Каждая функция, работающая с полями типов, должна все знать о представлении и других деталях реализации каждого класса, производного от класса с полем типа.

Существует тенденция, что если данные доступны из производных классов через поля типа, то это провоцирует людей на то, чтобы добавлять таких данных все больше. В итоге получается, что базовый класс становится хранилищем всякой «полезной информации». Можно сказать, что в большой классовой иерархии, данные в базовом классе, к которым имеется доступ через поля типов, становятся своего рода «глобальными переменными» этой иерархии. Ради чистого дизайна и удобной поддержки кода отдельные части кода должны реализовываться отдельно и должны отсутствовать взаимные зависимости.

20.3.2. Виртуальные функции

Виртуальные функции позволяют преодолеть проблемы с полями типов, которые мы рассмотрели в предыдущем разделе. Виртуальные функции позволяют программисту объявить функции в базовом классе, которые можно будет уточнить в каждом производном классе. Компилятор и компоновщик могут гарантировать соответствие между объектами и соответствующими функциями. Например:

```
class Employee {
public:
    Employee(const string& name, int dept);
    virtual void print() const;
    // ...
```

```
private:
    string first_name, family_name;
    short depar tment;
    // ...
};
```

Ключевое слово **virtual** означает, что **print()** действует как интерфейс к одноименным функциям, определенным в базовом и производных классах. Компилятор гарантирует, что в случае объявления функции **print()** в производных классах, правильная версия этой функции будет вызвана для объектов (работников) в каждом конкретном случае.

Чтобы виртуальная функция работала как интерфейс, все объявления одноименной функции в производных классах должны иметь одинаковые аргументы. Допускаются малейшие изменения в тип возврата (§20.3.6).

Виртуальная функция обязана быть определена в классе, в котором она впервые объявляется (если только она не чисто виртуальная; см. §20.4). Например:

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    // ...
}
```

Виртуальную функцию можно использовать даже если никаких производных классов нет. И в производных классах, если в них нет нужды в иных версиях виртуальной функции, можно не объявлять и не переопределять виртуальную функцию. В определении производного класса просто предоставьте подходящий вам вариант. Пример:

```
class Manager : public Employee {
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    // ...
private:
    list<Employee*> group;
    short level;
    // ...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tlevel " << level << '\n';
    // ...
}
```

Про функцию из производного класса с тем же именем и теми же аргументами, что и у виртуальной функции из базового класса, говорят как о функции, *замещающей* версию базового класса. Можно заместить виртуальную функцию базового класса даже с типом возврата производного класса (§20.3.6).

За исключением случаев, когда мы явно обозначаем версию вызываемой виртуальной функции (как в вызове **Employee::print()**), замещающая виртуальная функция выбирается как наиболее подходящая для объекта, с которым выполняется вызов. Независимо от того, какой именно базовый класс используется для доступа к объекту, мы всегда получаем ту же самую функцию при использовавнии механизма виртуальных функций.

Глобальная функция **print_employee()** (§20.3.1) теперь не нужна, поскольку функции-члены **print()** заняли ее место. Теперь список работников можно распечатать следующим образом:

```
void print_list(const list<Employee*>& s)
{
    for (auto x : s)
        x->print();
}
```

Каждый работник будет обработан согласно его типу. Например:

```
int main()
{
    Employee e {"Brown",1234};
    Manager m {"Smith",1234,2};

    print_list({&e,&m});
}
```

Вот выдача этой программы:

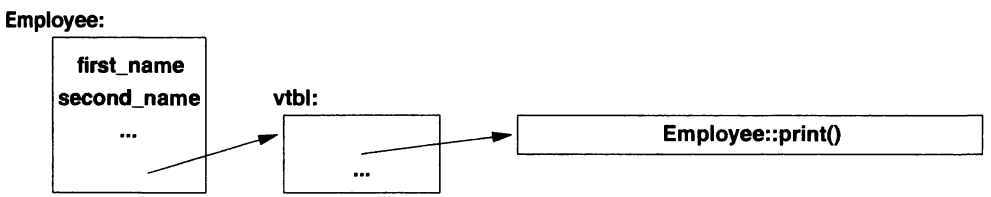
```
Smith 1234
    level 2
Brown 1234
```

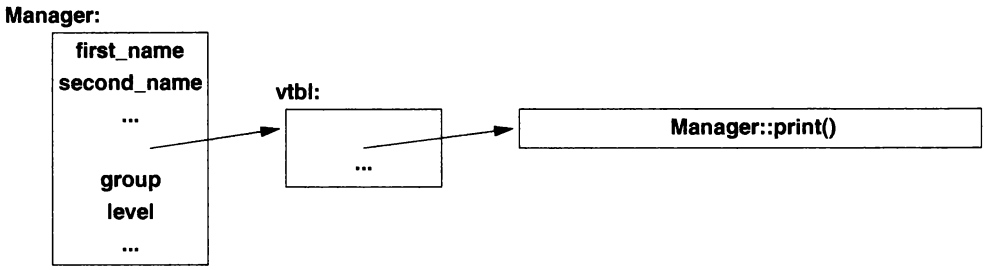
Отметим, что это будет работать, даже если **print_list()** был написан и скомпилирован до того, как конкретный производный класс **Manager** был вообще задуман. В этом состоит одна из ключевых идей классов. При правильном использовании они служат краеугольным камнем в основании объектно-ориентированного дизайна и обспечивают необходимую стабильность при эволюционировании программы.

Правильное функционирование функций из **Employee** независимо от того, какой именно подвариант **Employee** используется, называется *полиморфизмом*. Тип с виртуальными функциями называется *полиморфным типом* или (что более точно) типом с полиморфизмом времени выполнения. Для достижения полиморфизма времени выполнения нужно иметь виртуальную функцию и манипулировать объектом посредством указателя или ссылки. Когда с объектом работают напрямую (не через указатель или ссылку), тип объекта точно известен и нет никакой нужды в полиморфизме.

По умолчанию, функция, замещающая виртуальную функцию, сама является виртуальной. Мы можем повторить ключевое слово **virtual** в производном классе, но не обязаны это делать. Я не рекомендую повторять **virtual**. Если вам нужно большей наглядности, используйте ключевое слово **override** (§20.3.4.1).

Ясно, что для того, чтобы реализовать полиморфизм, компилятор должен хранить с каждым объектом дополнительную информацию и использовать ее для вызова правильной версии виртуальной функции **print()**. В типичной реализации дополнительный объем памяти часто содержит лишь указатель (§3.2.3); в обычном случае компилятор конвертирует имя виртуальной функции в индекс в таблице указателей на функции. Эту таблицу называют *таблицей виртуальных функций* или сокращенно **vtbl**. Каждый класс с виртуальными функциями имеет свою собственную **vtbl**, идентифицирующую виртуальные функции этого класса. Графически это можно отобразить следующим образом:





Функции из **vtbl** позволяют корректно использовать объект даже в том случае, когда размер объекта и раскладка данных в нем неизвестны вызывающему коду. Реализация вызывающего кода должна лишь знать местоположение **vtbl** для класса (например **Employee**) и индексы для каждой виртуальной функции. Дополнительные накладные расходы на вызов виртуальных функций невелики (не более 25%), так что вопросы производительности не должны препятствовать использованию виртуальных функций в случаях, когда вопросы производительности важны. А дополнительные расходы памяти сводятся к одному указателю на каждый объект плюс память под **vtbl** для класса.

Виртуальная функция, вызываемая из конструктора или деструктора, отражает тот факт, что объект не полностью сконструирован или лишь частично уничтожен (§22.4). Так что в типичном случае идея вызывать виртуальную функцию из конструктора или деструктора не слишком хороша.

20.3.3. Явная квалификация

При вызове функции с операцией разрешения::, как это сделано в случае **Manager::print()**, гарантируется, что механизм виртуальных функций не используется:

```
void Manager::print() const
{
    Employee::print(); // не виртуальный вызов
    cout << "\tlevel " << level << '\n';
    // ...
}
```

В ином случае **Manager::print()** пострадала бы от бесконечной рекурсии. У варианта вызова с квалифицированным именем есть еще одно достоинство. Если виртуальная функция объявлена с ключевым словом **inline**, то встраивание будет применено и для вызова с операцией::. Это позволяет программисту эффективно справляться с важным частным случаем, когда одна виртуальная функция вызывает другую для одного и того же объекта. Функция **Manager::print()** как раз служит примером такого случая. Поскольку тип объекта определяется вызовом **Manager::print()**, то нет необходимости снова динамически определять это для результирующего вызова **Employee::print()**.

20.3.4. Управление замещением

Если вы объявляете функцию в производном классе с тем же именем и параметрами, что и функция из базового класса, то вы тем самым замещаете ее в производном классе. Однако, для больших классовых иерархий бывает трудно наверняка знать, что вы на самом деле замещаете именно ту функцию, которую надо заместить. Рассмотрим следующее:

```
struct B0 {
    void f(int) const;
    virtual void g(double);
};

struct B1 :  B0 {/* ... */};
struct B2 :  B1 {/* ... */};
struct B3 :  B2 {/* ... */};
struct B4 :  B3 {/* ... */};
struct B5 :  B4 {/* ... */};

struct D :  B5 {
    void f(int) const;    // замещает f() в баз. классе
    void g(int);          // замещает g() в баз. классе
    virtual int h();      // замещает h() в баз. классе
};
```

Здесь иллюстрируются три ошибки, которые не столь тривиальны, когда они проявляются в больших классовых иерархиях, в которых классы **B0...B5** имеют множество членов, разбросанных по многим заголовочным файлам. Здесь:

- **B0::f()** не виртуальная, так что невозможно заместить ее, можно только скрыть (§20.3.5).
- **D::g()** не имеет одинаковых аргументов с **B0::g()**, так что если она и замещает что-то, то она не является виртуальной функцией **B0::g()**. Вероятнее всего, **D::g()** просто скрывает **B0::g()**.
- Здесь нет функции с именем **h()** в **B0**, и если **D::h()** и замещает что-нибудь, это в любом случае не функция из **B0**. Скорее всего она вводит новую виртуальную функцию.

Я не показал, что было в **B1..B5**, так что возможно там и происходит нечто иное из-за объявлений в этих классах. Лично я не дублирую **virtual** в случае небольших иерархий классов. Однако для больших классовых иерархий нужно контролировать ситуацию:

- **virtual**: Функция может быть замещена (§20.3.2).
- **=0**: Функция должна быть виртуальной и должна быть замещена (§20.4).
- **override**: Функция намерена заместить виртуальную функцию базового класса (§20.3.4.1).
- **final**: Функция не предназначена для замещения (§20.3.4.2).

При отсутствии каких-либо из перечисленных здесь управляющих элементов нестатическая функция-член является виртуальной, только если она замещает виртуальную функцию базового класса (§20.3.2).

Компилятор может предупредить относительно несогласованности в употреблении элементов управления замещением. Например, если в классе семь раз используется **override** при девяти виртуальных функциях базового класса, то вряд ли это можно легко понять программистам, поддерживающим код.

20.3.4.1. override

Можно явным образом отразить намерение о замещении:

```
struct D : B5 {
    void f(int) const override;    // ошибка: B0::f() не виртуальная
    void g(int) override;          // ошибка: у B0::f() аргумент типа double
    virtual int h() override;      // ошибка: нет функции h() для замещения
};
```

Отталкиваясь от данного определения (и предполагая, что промежуточные базовые классы **B1...B5** не предоставляют релевантных функций), все три объявления дают ошибки.

В больших классовых иерархиях со множеством виртуальных функций лучше всего использовать ключевое слово **virtual** только в случае введения новой виртуальной функции, и использовать **override** в случаях намерения заместить виртуальную функцию. Это проясняет намерения программиста.

Спецификатор **override** размещают в объявлении самым последним:

```
void f(int) const noexcept override;     // OK (если есть подходящая f() для замещения)
override void f(int) const noexcept;     // синтаксическая ошибка
void f(int) override const noexcept;     // синтаксическая ошибка
```

Такое требование кажется несколько нелепым, но такова плата за стабильность и совместимость на протяжении десятилетий.

Спецификатор **override** не входит в определение типа функции и не должен повторяться в определениях, расположенных вне определения класса. Например:

```
class Derived : public Base {
    void f() override;          // OK если Base имеет virtual f()
    void g() override;          // OK если Base имеет virtual g()
};

void Derived::f() override      // ошибка: замещение вне класса
{
    // ...
}

void g() // OK
{
    // ...
}
```

Как это не удивительно, **override** не является ключевым словом; в точности оно называется *констекстным ключевым словом*. То есть **override** имеет разный смысл в разных контекста, а в иных случаях может использоваться и как идентификатор. Например:

```
int override = 7;

struct Dx : Base {
    int override;

    int f() override
    {
        return override +::override;
    }
};
```

Не переусердствуйте с усложнениями кода; это только будет вредить при его поддержке. Причина в такой необычности спецификатора **override** опять-таки проистекает из огромного объема раннего кода, в котором **override** интенсивно использовалось в качестве идентификатора. Еще одним контекстным ключевым словом является **final** (§20.3.4.2).

20.3.4.2. final

В объявлении функции-члена у нас есть выбор: делать ее виртуальной или нет (имеет место по умолчанию). Для виртуальных функций мы предполагаем, что в производных

классах они будут уточняться и модифицироваться. Выбор часто базируется на семантике класса:

- Можно ли предвидеть нужду в дополнительных производных классах?
- Нужно ли будет разработчику производного класса переопределять функцию для достижения приемлемого эффекта?
- Чревата ли ошибками попытка замещения функции (например, потому что трудно добиться ожидаемой семантики виртуальной функции)?

Если ответ на все три вопроса есть «нет», то мы оставляем функцию в качестве невиртуальной ради достижения простоты дизайна и небольшого выигрыша в производительности. В стандартной библиотеки масса примеров такого подхода.

В более редком случае мы имеем классовую иерархию, которая стартует с виртуальных функций, но после определения ряда производных классов один из ответов становится «нет». Примером будет служить иерархия классов для синтаксического анализа конкретного языка программирования. Если мы захотим написать производный класс для работы с другим языком программирования, то нам не нужно замещать виртуальные функции, поскольку они не соответствуют новому языку программирования. В результате лучше закрыть наш дизайн от дальнейших модификаций пользователями. Например:

```
struct Node { // интерфейсный класс
    virtual Type type() = 0;
    // ...
};

class If_statement : public Node {
public:
    Type type() override final;     // запрещаем дальнейшее замещение
    // ...
};
```

В реалистичном примере будет еще несколько промежуточных классов между общим интерфейсом (здесь **Node**) и производным классом, представляющим специфическую конструкцию языка программирования (здесь **If_statement**). Однако, ключевым моментом в данном примере является тот факт, что **Node::type()** подразумевает замещение, а его заместитель — **If_statement::type()** не подразумевает (потому и объявлен с **final**). После объявления с **final** функция-член уже не может замещаться, а при попытке возникнет ошибка компиляции. Пример:

```
class Modified_if_statement : public If_statement {
public:
    Type type() override;     // ошибка: If_statement::type() имеет модификатор final
    // ...
};
```

Мы можем приписать всем виртуальным функциям класса **final**; просто добавляем **final** после имени класса. Например:

```
class For_statement final : public Node {
public:
    Type type() override;
    // ...
};

class Modified_for_statement : public For_statement { // ошибка: For_statement - final
    Type type() override;
    // ...
};
```

Хорошо это или плохо, но добавление **final** в объявление класса не только предотвращает замещение, но оно еще и предотвращает наследование для этого класса. Поэтому некоторые программисты могут применять это средство с целью оптимизации, поскольку улучшаются возможности для встраивания (§12.1.5). Я не советую так поступать, поскольку выигрыш в производительности незначительный, а порча дизайна классовой иерархии велика. Используйте **final** исключительно в случаях, когда это в точности соответствует дизайну вашей классовой иерархии для отражения семантических нужд.

Спецификатор **final** не влияет на тип функции и не может повторяться в определении функции, расположенном вне определения класса. Например:

```
class Derived : public Base {
    void f() final;  // OK если Base имеет virtual f()
    void g() final;  // OK если Base имеет virtual g()
    // ...
};

void Derived::f() final     // ошибка: final вне класса
{
    // ...
}

void g() final  // OK
{
    // ...
}
```

Как и **override** (§20.3.4.1), **final** является контекстным ключевым словом. То есть, **final** имеет специальный смысл в нескольких контекстах, но может использоваться и как простой идентификатор. Например:

```
int final = 7;

struct Dx : Base {
    int final;

    int f() final
    {
        return final +::final;
    }
};
```

Как и в случае **overried**, **final** является контекстным ключевым словом из-за многочисленного унаследованного кода.

20.3.5. Использование членов базового класса

Функции не перегружаются поверх границ областей видимости (§12.3.3). Например:

```
struct Base {
    void f(int);
};

struct Derived :  Base {
    void f(double);
};
```

```
void use(Derived d)
{
    d.f(1);              // вызов Derived::f(double)
    Base& br = d
    br.f(1);             // вызов Base::f(int)
}
```

Это часто приводит к сюрпризам, ибо всегда хочется добиться наилучшего соответствия. Для пространств имен можно использовать **using**-объявления для введения функции в область видимости. Пример:

```
struct D2 :  Base {
    using Base::f; // переносим все f из Base в D2
    void f(double);
};

void use2(D2 d)
{
    d.f(1);              // вызов D2::f(int), то есть, Base::f(int)
    Base& br = d
    br.f(1);             // вызов Base::f(int)
}
```

Это является простым следствием того факта, что класс сам является пространством имен (§16.2).

Несколько **using**-объявлений могут внести в область видимости имена из нескольких базовых классов. Пример:

```
struct B1 {
    void f(int);
};

struct B2 {
    void f(double);
};

struct D :  B1, B2 {
    using B1::f;
    using B2::f;
    void f(char);
};

void use(D d)
{
    d.f(1);     // вызов D::f(int), то есть, B1::f(int)
    d.f('a');   // вызов D::f(char)
    d.f(1.0);   // вызов D::f(double), то есть, B2::f(double)
}
```

Можно внести конструкторы в область видимости производного класса; см. §20.3.5.1. Имя, внесенное в область видимости производного класса с помощью **using**-объявления, имеет доступ, определяемый местоположением **using**-объявления; см. §20.5.3. Нельзя использовать **using**-директиву для внесения всех членов базового класса в производный класс.

20.3.5.1. Наследование конструкторов

Скажем, мне нужен вектор, который аналогичен **std::vector**, но имеет гарантированную проверку диапазона. Можно попробовать следующее:

```
template<class T>
struct Vector : std::vector<T> {
    T& operator[](size_type i) {check(i); return this->elem(i);}
    const T& operator[](size_type i) const {check(i); return this->elem(i);}

    void check(size_type i) {if (this->size()<i) throw range_error{"Vector::check() failed"};}
};
```

К сожалению, мы вскоре обнаружим, что приведенное определение весьма неполное. Например:

```
Vector<int> v {1, 2, 3, 5, 8}; // ошибка: нет конструктора со списком инициализаторов
```

Можно быстро выяснить, что **Vector** не может наследовать конструкторы из **std::vector**.

Это разумное правило: если класс добавляет члены-данные к базовому классу или же требует более строго инварианта класса, то наследование конструкторов было бы катастрофой.

Мы решаем проблему простым объявлением того факта, что конструкторы должны наследоваться:

```
template<class T>
struct Vector : std::vector<T> {
    using vector<T>::vector;        // наследуем конструкторы

    T& operator=[](size_type i) {check(i); return this->elem(i);}
    const T& operator=(size_type i) const {check(i); return this->elem(i);}

    void check(size_type i) {if (this->size()<i) throw Bad_index(i);}
};

Vector<int> v {1, 2, 3, 5, 8}; // OK: используем констр-р со списк. инициализаторов из std::vector
```

Такое применение **using**-объявления эквивалентно его применению для обычных функций (§14.4.5, §20.3.5).

При таком выборе можно «выстрелить себе в ногу», если вы определяете в производном классе новые члены-данные, требующие явной инициализации:

```
struct B1 {
    B1(int) {}
};

struct D1 : B1 {
    using B1:: B1;      // неявно объявляет D1(int)
    string s;           // string имеет умолчательный конструктор
    int x;              // мы "забыли" про инициализацию x
};

void test()
{
    D1 d {6};           // упс: d.x не инициализирован
    D1 e;               // ошибка:  D1 не имеет умолчательного конструктора
}
```

Причина, по которой **D1::s** инициализируется, а **D1::x** не инициализируется, состоит в том, что наследование конструктора эквивалентно конструктору, который инициализирует только базу. В таком случае, можно было бы написать эквивалентный код:

```
struct D1 : B1 {
    D1(int i) : B1(i) {}
    string s;       // string имеет умолчательный конструктор
    int x;          // мы "забыли" про инициализацию x
};
```

Один из способов «убрать пулю из ноги» состоит в том, чтобы добавить внутриклассовый инициализатор члена (§17.4.4):

```
struct D1 : B1 {
    using B1::B1;       // неявно объявляет D1(int)
    int x {0};          // x проинициализирован
};

void test()
{
    D1 d {6};           // d.x есть нуль
}
```

В общем случае не стоит слишком умничать; лучше ограничить наследование конструкторов простейшими случаями, когда не добавляется никаких новых членов данных.

20.3.6. Ослабление типа возврата

Имеется возможность ослабления требования о том, что тип возврата замещающей функции должен совпадать с типом возврата виртуальной функции, которую замещают. Оно состоит в том, что если у оригинальной функции тип возврата есть **B***, то у замещающей функции он может быть **D***, где **B** это открытый базовый класс для **D**. Аналогично, возврат типа **B&** может ослабляться до **D&**. Иногда это называют *ковариантным правилом для типов возврата*.

Это правило ослабления распространяется только на указательные и ссылочные типы возврата, но не на интеллектуальные указатели, такие как **unique_ptr** (§5.2.1). Кроме того, не существует правил ослабления для типов аргументов, ибо в этом случае произошло бы нарушение системы типов.

Рассмотрим иерархию классов, представляющих различные виды выражений. Дополнительно к операциям манипулирования выражениями базовый класс **Expr** предоставляет средства порождения новых объектов-выражений разных типов:

```
class Expr {
public:
    Expr();                     // умолчательный конструктор
    Expr(const Expr&);          // копирующий конструктор
    virtual Expr* new_expr() =0;
    virtual Expr* clone() =0;
    // ...
};
```

Идея состоит в том, чтобы **new_expr()** порождал умолчательный объект с типом выражения, а **clone()** создавала бы копию объекта. Обе функции возвращают объект специфического типа, производного от **Expr**. Они никогда не возвращают просто **Expr**, ибо этот класс намеренно сделан абстрактным классом.

Производный класс может заместить **new_expr()** и/или **clone()** для возврата объекта собственного типа:

```
class Cond : public Expr {
public:
    Cond();
    Cond(const Cond&);
    Cond* new_expr() override {return new Cond();}
    Cond* clone() override {return new Cond(*this);}
    // ...
};
```

Это означает, что имея объект типа **Expr**, пользователь может создать объект «того же самого типа». Например:

```
void user(Expr* p)
{
    Expr* p2 = p->new_expr();
    // ...
}
```

Указатель, присваиваемый **p2**, объявлен как указатель на **Expr**, но он будет указывать на объект типа, производного от **Expr**, такой как **Cond**.

Тип возврата у **Cond::new_expr()** и **Cond::clone()** есть скорее **Cond***, чем просто **Expr***. Это позволяет клонировать **Cond** без потери информации о типе. Аналогично, у производного класса **Addition** будет функция **clone()**, возвращающая **Addition***. Пример:

```
void user2(Cond* pc, Addition* pa)
{
    Cond* p1 = pc->clone();
    Addition* p2 = pa->clone();
    // ...
}
```

Если мы испрльзуем **clone()** для **Expr**, то результат нам известен как **Expr***:

```
void user3(Cond* pc, Expr* pe)
{
    Cond* p1 = pc->clone();
    Cond* p2 = pe->clone();      // ошибка: Expr::clone() возвращает Expr*
    // ...
}
```

Поскольку такие функции как **new_expr()** и **clone()** виртуальные и они (косвенно) создают объекты, то их называют *виртуальными конструкторами*. Каждая из функций просто использует конструктор для создания подходящего объекта.

Для создания объекта конструктор должен знать точный тип объекта, который он должен создать. Значит, конструктор не может быть виртуальным. Более того, конструктор не является обычной функцией. В частности, он взаимодействует с процедурами выделения памяти способом, недоступным обычным функциям. Поэтому вы не можете взять указатель на конструктор и передать его функции, создающей объект.

Вы можете написать функции, которые вызывают конструкторы для создания объектов. Это помогает создавать объекты, типы которых точно неизвестны. Класс **Ival_box_maker** (§21.2.4) служит примером класса, спроектированного специально для этой задачи.

20.4. Абстрактные классы

Многие классы напоминают **Employee** в том, что они полезны сами по себе, как интерфейсы для производных классов и как часть реализации производных классов. Но не все классы относятся к такому типу. Есть классы, вроде **Shape**, представляющие абстрактные концепции, для которых не может существовать реальных объектов. Класс **Shape** имеет смысл лишь как интерфейс для некоторых производных классов. Это видно и из того, что не существует разумных реализаций его виртуальных функций:

```
class Shape {
public:
    virtual void rotate(int) {throw runtime_error{"Shape::rotate"};}    // не элегантно
    virtual void draw() const {throw runtime_error{"Shape::draw"};}
    // ...
};
```

Попытка создать форму неспецифицированного вида конечно бессмысленная, но легальная:

```
Shape s; // глупо: "бесформенная форма"
```

Глупость в создании такого объекта заключается в том, что любая операция с объектом приводит к ошибке.

Лучшей альтернативой будет объявление виртуальных функций класса **Shape** как *чисто виртуальных*. Это достигается посредством псевдоинициализатора **=0**:

```
class Shape { // abstract class
public:
    virtual void rotate(int) = 0;          // чисто виртуальная функция
    virtual void draw() const = 0;         // чисто виртуальная функция
    virtual bool is_closed() const = 0;    // чисто виртуальная функция
    // ...
    virtual ~Shape();                      // виртуальная
};
```

Класс с одной или многими чисто виртуальными функциями называется *абстрактным классом*, для которого нельзя создавать объекты:

```
Shape s; // ошибка: переменная абстрактного класса Shape
```

Целью абстрактных классов является интерфейс к объектам, доступ к которым выполняется через указатели или ссылки (для обеспечения полиморфного поведения). Поэтому обычно бывает важным для абстрактного класса иметь виртуальный деструктор (§3.2.4, §21.2.2). Поскольку интерфейс, предоставляемый абстрактным классом, не может использоваться для создания объектов конструктором, то абстрактные классы обычно не имеют конструкторов.

Абстрактный класс используется только как интерфейс к другим классам. Пример:

```
class Point {/* ... */};

class Circle : public Shape {
public:
    void rotate(int) override {}
    void draw() const override;
    bool is_closed() const override {return true;}

    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

Чисто виртуальная функция, не определенная в производном классе, остается чисто виртуальной, так что производный класс также является абстрактным классом. Это позволяет строить реализации по этапам:

```
class Polygon : public Shape {    // абстрактный класс
public:
```

```
    bool is_closed() const override {return true;}
    // ... draw и rotate не замещены ...
};

Polygon b {p1, p2, p3, p4};  // ошибка: объявление объекта абстрактного класса Polygon
```

Класс **Polygon** по-прежнему абстрактный, поскольку мы не заместили **draw()** и **rotate()**. Только после того, как это будет сделано, мы получим класс, для которого можно создавать объекты:

```
class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    Irregular_polygon(initializer_list<Point>);

    void draw() const override;
    void rotate(int) override;
    // ...
};

Irregular_polygon poly {p1, p2, p3, p4}; // положим, что p1 .. p4 - Points, определенные где-либо
```

Абстрактный класс обеспечивает интерфейс без показа деталей реализации. Например, операционная система может скрывать детали аппаратных драйверов за абстрактным классом:

```
class Character_device {
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int ...) = 0;    // контроль I/O (ввода/вывода)

    virtual ~Character_device() {} // виртуальный деструктор
};
```

После этого мы можем специфицировать драйверы как классы, производные от **Character_device**, и манипулировать драйверами через этот интерфейс.

Стиль дизайна на основе абстрактных классов называется *наследованием интерфейса*, в отличие *наследования реализации*, которое поддерживается базовыми классами, обладающими ненулевым состоянием и/или определяющими свои функции-члены. Возможна комбинация обоих подходов. То есть можно определять базовые классы, имеющие состояние, и располагающие чисто виртуальными функциями. Однако, такая смесь стилей может смутить других программистов, что требует дополнительного внимания и может привести к проблемам при поддержке кода.

После введения абстрактных классов у нас появляются все средства для написания программ целиком в модульном стиле, используя классы как строительные блоки.

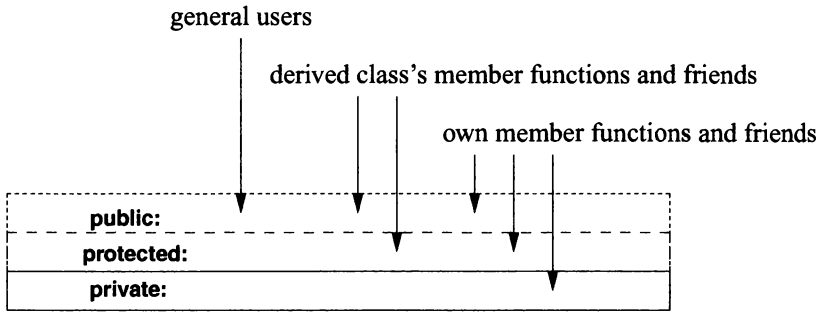
20.5. Управление доступом

Член класса может быть открытым (**public**), защищенным (**protected**) или закрытым (**private**):

- Если член класса закрытый (**private**), то его имя может использоваться только функциями-членами класса или друзьями класса, в котором они определены.

- Если член класса защищенный (**protected**), то его имя может использоваться функциями-членами класса или друзьями класса, в котором они определены, и то же самое справедливо для классов, производных от данного класса (§19.4).
- Если член класса открытый (**public**), то его имя может использоваться любыми функциями.

Такая схема отражает взгляд на три сорта функций, имеющих доступ к классу: функции, реализующие класс (его друзья и члены); функции, реализующий производный класс (и его друзья и члены) и остальные функции. Это можно изобразить графически:



Контроль доступа применим ко всем членам класса. Мы можем иметь закрытые функции-члены, типы, константы и т.д., а также закрытые члены-данные. Рассмотрим к примеру неинтрузивный список, нуждающийся в струтурах данных для отслеживания элементов. Список называется неинтрузивным, если он не требует модификации элементов (например, требованием, чтобы тип элементов имел поля связи). Информация и данные, используемые для организации списка могут быть закрытыми:

```
template<class T>
class List {
public:
    void insert(T);
    T get();
// ...
private:
    struct Link {T val; Link* next;};

    struct Chunk {
        enum {chunk_size = 15};
        Link v[chunk_size];
        Chunk* next;
    };

    Chunk* allocated;
    Link* free;
    Link* get_free();
    Link* head;
};
```

Определения открытых функций достаточно прямолинейны:

```
template<class T>
void List<T>::insert(T val)
{
    Link* lnk = get_free();
    lnk->val = val;
```

```
    lnk->next = head;
    head = lnk;
}

template<class T>
T List<T>::get()
{
    if (head == 0)
        throw Underflow{}; // Underflow это мой класс исключения

    Link* p= head;
    head = p->next;
    p->next = free;
    free = p;
    return p->val;
}
```

Как это нередко бывает, определения вспомогательных функций несколько более замысловаты:

```
template<class T>
typename List<T>::Link* List<T>::get_free()
{
    if (free == 0) {
        // ... аллоцировать новый кусок памяти и поместить Links в список free ...
    }
    Link* p = free;
    free = free->next;
    return p;
}
```

Внедрение в область видимости **List<T>** организуется посредством **List<T>::** в определении функции-члена. Однако, поскольку тип возврата у **get_free()** упоминается раньше упоминания **List<T>::get_free()**, то нужно использовать полное имя **List<T>:: Link** вместо сокращения **Link**. Альтернатива состоит в использовании нотации суффиксов для типов возврата (§12.1.4):

```
template<class T>
auto List<T>::get_free() -> Link*
{
    // ...
}
```

Глобальные функции (не друзья) не имеют такого доступа:

```
template<typename T>
void would_be_meddler(List<T>* p)
{
    List<T>::Link* q = 0;     // ошибка: List<T>::Link закрытый
    // ...
    q = p->free;    // ошибка: List<T>::free закрытый
    // ...
    if (List<T>::Chunk::chunk_size > 31) {  // ошибка: List<T>::Chunk::chunk_size закрытый
        // ...
    }
}
```

Для типа, определяемого ключевым словом **class**, члены по умолчанию закрытые; у типа с ключевым словом **struct** — по умолчанию открытые (§16.2.4).

Очевидной альтернативой типу в качестве члена будет помещение типа в окружающее пространство имен. Например:

```
template<class T>
struct Link2 {
    T val;
    Link2* next;
};

template<class T>
class List {
private:
    Link2<T>* free;
    // ...
};
```

Link неявно параметризуется параметром **T** типа **Link<T>**. Для **Link2** это нужно делать явно.

Если член-тип не зависит от параметров шаблонного класса, то внешняя версия может быть предпочтительной; см. §23.4.6.3.

Если внутренний класс не особо полезен сам по себе и окружающий класс нуждается в доступе к его представлению, то объявление класса-члена другом (§19.4.2) будет неплохой идеей:

```
template<class T> class List;

template<class T>
class Link3 {
    friend class List<T>;      // только List<T> имеет доступ к Link<T>
    T val;
    Link3* next;
};

template<class T>
class List {
private:
    Link3<T>* free;
    // ...
};
```

Компилятор может переупорядочить секции класса с разными спецификаторами доступа (§8.2.6). Например:

```
class S {
public:
    int m1;
public:
    int m2;
};
```

Компилятор может решить, что **m2** предшествует **m1** в раскладке объекта типа **S**. Такое переупорядочение может оказаться сюрпризом для программиста и оно зависит от реализации, так что без особой причины не используйте множественные спецификаторы доступа для членов-данных.

20.5.1. Защищенные члены

Конструируя классовую иерархию, мы можем определить функции, предназначенные для реализаторов производных классов, а не для обычных пользователей. Например, мы можем предоставить эффективную функцию без проверки для реализаторов производных классов и безопасную функцию с проверками для остальных. Для этого версию без проверки лучше всего снабдить спецификатором **protected**. Пример:

```
class Buffer {
public:
    char& operator[](int i);                    // доступ с проверкой
    // ...
protected:
    char& access(int i);                        // доступ без проверки
    // ...
};

class Circular_buffer : public Buffer {
public:
    void reallocate(char* p, int s);            // изменить положение и размер
    // ...
};

void Circular_buffer::reallocate(char* p, int s)    // изменить положение и размер
{
    // ...
    for (int i=0; i!=old_sz; ++i)
        p[i] = access(i);                       // без избыточной проверки
    // ...
}

void f(Buffer& b)
{
    b[3] = 'b';                 // OK (проверено)
    b.access(3) = 'c';          // ошибка: Buffer::access() защищенный
}
```

Другой пример — это **Window_with_border** (§21.3.5.2).

Производный класс может иметь доступ к защищенным членам базового класса только для объектов собственного типа:

```
class Buffer {
protected:
    char a[128];
    // ...
};

class Linked_buffer : public Buffer {
    // ...
};

class Circular_buffer : public Buffer {
    // ...
    void f(Linked_buffer* p)
    {
        a[0] = 0;       // OK: доступ Circular_buffer's к собственному защищенному члену
        p->a[0] = 0;    // ошибка: доступ к защищенному члену другого типа
    }
};
```

Это предотвращает тонкие ошибки, которые в ином случае возникали бы, когда один производный класс портил бы данные, принадлежащие другим производным классам.

20.5.1.1. Использование защищенных членов

Простая private/public-модель сокрытия данных хорошо работает для конкретных типов (§16.3). Но когда появляются производные классы, то возникают две категории пользователей класса: производные классы и «обычные пользователи». Функции-члены и дру-

зья, реализующие операции с классом, оперируют с классовыми объектами от лица этих пользователей. Модель с private/public доступом позволяет хорошо разграничивать реализаторов и обычных пользователей, но она не учитывает специфики производных классов.

Члены со спецификатором **protected** гораздо уязвимее, чем закрытые члены. Поэтому беспричинное объявление членов защищенными является ошибкой проектирования. Если поместить изрядный объем данных в общий класс с доступом к ним из производных классов, то это значит подвергать данные опасности.

По счастью, вас никто не заставляет работать с защищенными членами; по умолчанию члены закрытые. По моему опыту всегда имеются альтернативы такому решению, как поместить серьезный объем данных в общий класс и дать возможность работы с этими данными напрямую производным классам.

Но каких-либо сверхважных возражений против защищенных членов нет; **protected** позволяет открыто специфицировать операции для их использования в производных классах. Примером служит **Ival_slider** из §21.2.2.

20.5.2. Доступ к базовым классам

Как и в случае членов, спецификаторы **public**, **protected** и **private** можно применить непосредственно к базовому классу. Пример:

```
class X : public B {/* ... */};
class Y : protected B {/* ... */};
class Z : private B {/* ... */};
```

Различные спецификаторы наследования служат разным целям:

- **public**-наследование реализует идею о том, что производный класс является подтипом базового; например, **X** это вид (подтип) **B**.
- **private**-наследование сильно ограничивает интерфейс к базовому классу, что обеспечивает более сильные гарантии; к примеру **B** является деталью реализации **Z**. Хорошим примером также служит шаблон **Vector**, который добавляет проверку типа для **Vector<void*>** из §25.3.
- **protected**-наследование полезно в случае классовых иерархий, для которых дальнейшее наследование является нормой. Как и **private**-наследование, **protected**-наследование используется для представления деталей реализации. Примером служит **Ival_slider** из §21.2.2.

Если опустить спецификатор доступа, то в случае ключевого слова **class** мы получим **private**-наследование, а для ключевого слова **struct** — **public**-наследование. Например:

```
class XX :  B {/* ... */};      // B закрытый базовый класс
struct YY :  B {/* ... */};     // B открытый базовый класс
```

Люди скорее ожидают открытого наследования в умолчательном варианте Поэтому лучше явно использовать спецификатор наследования доступа.

Спецификатор наследования доступа определяет доступ к членам базового класса и процесс приведения типа от указателей и ссылок на производный класс к таковым на базовый класс. Рассмотрим класс **D**, производный от **B**:

- Если **B** закрытый базовый класс, то его открытые и защищенные члены могут использоваться только членами и друзьями класса **D**. Только друзья и члены **D** могут выполнять конвертирование из **D*** в **B***.
- Если **B** защищенный базовый класс, то его открытые и защищенные члены могут использоваться только членами и друзьями класса **D**, а также членами и друзьями

классов, производных от **D**. Только друзья и члены **D**, а также друзья и члены классов, производных от **D**, могут выполнять конвертирование из **D*** в **B***.

- Если **B** открытый базовый класс, его открытые члены могут использоваться любыми функциями. Кроме того, его защищенные члены могут использоваться членами и друзьями **D** и членами и друзьями классов, производных от **D**. Любая функция может выполнять конвертирование из **D*** в **B***.

Это практически повторяет правила доступа к членам (§20.5). При проектировании класса мы выбираем вариант доступа к базовому классу в той же манере, в какой мы проектируем доступ к членам. Смотри для примера **lval_slider** из §21.2.2.

20.5.2.1. Множественное наследование и контроль доступа

Если имя базового класса прослеживается по многим путям в решетке множественного наследования (§21.3), то оно доступно, если имеется доступ по всем путям. Пример:

```
struct B {
    int m;
    static int sm;
    // ...
};

class D1 : public virtual B {/* ... */};
class D2 : public virtual B {/* ... */};
class D12 : public D1, private D2 {/* ... */};

D12* pd = new D12;
B* pb = pd;         // OK: доступно через D1
int i1 = pd->m;     // OK: доступно через D1
```

Если единственная сущность доступна по многим путям, на нее можно ссылаться без неоднозначности. Например:

```
class X1 : public B {/* ... */};
class X2 : public B {/* ... */};
class XX : public X1, public X2 {/* ... */};

XX* pxx = new XX;
int i1 = pxx->m;    // ошибка, неоднозначно: XX::X1::B::m или XX::X2::B::m?
int i2 = pxx->sm;   // OK: есть только один B::sm в XX (sm статический член)
```

20.5.3. Контроль доступа и using-объявления

using-объявление (§14.2.2, §20.3.5) не может применяться для получения доступа к дополнительной информации. Это механизм для получения более удобного доступа. С другой стороны, если доступ имеется, то его можно презентовать другим пользователям. Пример:

```
class B {
private:
    int a;
protected:
    int b;
public:
    int c;
};
```

```
class D : public B {
public:
    using B::a;    // ошибка: B::a закрытый
    using B::b;    // делаем B::b общедоступным через D
};
```

Когда **using**-объявление комбинируется с закрытым или защищенным наследованием, то это можно использовать для презентации интерфейса к отдельным, но не ко всем средствам, предлагаемым классом. Например:

```
class BB : private B { // дает доступ к B::b и B::c, но не к B::a
public:
    using B::b;
    using B::c;
};
```

См. также §20.3.5.

20.6. Указатели на члены

Указатель на член является конструкцией, позволяющей программисту косвенно ссылаться на член класса. Операции **->*** и **.*** считаются наименее употребительными в языке C++. С помощью операции **->** мы обращаемся к члену класса с именем **m** через указатель **p**: **p->m**. При помощи операции **->*** мы обращаемся к члену, имя которого связано с указателем на член, **ptom**: **p->*ptom**. Это позволяет обращаться к членам с именами, передаваемыми в качестве аргументов. В любом случае **p** должен быть указателем на объект соответствующего класса.

Указателю на член нельзя присваивать значение **void*** или значение любого обычного указателя. Но можно присвоить нулевое значение указателя (**nullptr**), что будет означать «член не задан».

20.6.1. Указатели на функции-члены

Многие классы предоставляют простые и общие интерфейсы, которые предполагается использовать разными способами. К примеру, многие объектно-ориентированные графические пользовательские интерфейсы определяют набор запросов, на которые любые объекты, представленные на экране, должны быть готовы отреагировать. Кроме того, такие запросы должны прямо или косвенно генерироваться программами. Рассмотрим простой вариант указанной идеи:

```
class Std_interface {
public:
    virtual void start() = 0;
    virtual void suspend() = 0;
    virtual void resume() = 0;
    virtual void quit() = 0;
    virtual void full_size() = 0;
    virtual void small() = 0;

    virtual ~Std_interface() {}
};
```

Точный смысл каждой операции определяется объектом, для которого она вызвана. Часто между вызывающей программой и объектом, принимающим запрос, находится еще один программный (промежуточный) слой. В идеале такому промежуточному слою не нужно ничего знать об индивидуальных операциях вроде **resume()** или **full_size()**. В про-

тивном случае этот слой приходилось бы модернизировать каждый раз, когда операция изменяется хоть в чем-то. Отсюда вытекает, что промежуточный слой просто передает данные, представляющие операцию, от источника к получателю.

Выполнить это можно, передавая строку, представляющую операцию. К примеру, для активации операции **suspend()** можно передать строку **"suspend"**. Только кто-то должен подготовить такую строку и кто-то должен декодировать ее, чтобы решить, какую операцию активировать. Это достаточно утомительно. Вместо этого, можно просто послать целое число, указывающее на операцию. Например, **2** могло бы идентифицировать операцию **suspend()**. Но тут обратная ситуация: число 2 удобно для машины, но человеку оно мало о чем говорит, так что может запутать программиста. И снова нужно писать код, который выявляет, что **2** означает **suspend()**, после чего нужно вызвать **suspend()**.

Однако, мы можем воспользоваться указателем на член, чтобы непрямым образом ссылаться на член класса. Рассмотрим **Std_interface**. Если мы хотим вызвать **suspend()** для некоторого объекта без того, чтобы явно при этом упоминать **suspend()**, то нам нужен указатель на член, ссылающийся на **Std_interface::suspend()**. Кроме этого, нужен указатель или ссылка на объект, для которого нужно активировать указанную операцию. Вот тривиальный пример:

```
using Pstd_mem = void (Std_interface::*)();        // тип указателя на член

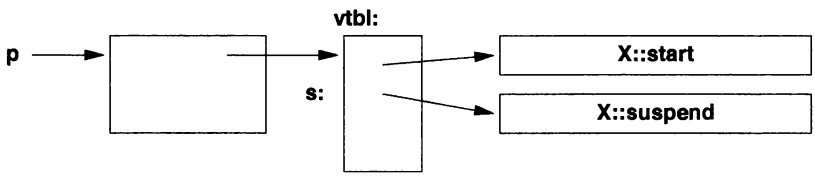
void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;          // указатель на suspend()
    p->suspend();                                  // прямой вызов
    p->*s();                                       // вызов через указатель на член
}
```

Можно указатель на член получить, прикладывая операцию взятия адреса, **&**, к полностью квалифицированному имени члена класса, например, **&Std_interface::suspend**. Переменная типа «указатель на член класса **X**» объявляется с применением декларатора в форме **X::***.

Указатель на член **m** может использоваться в комбинации с объектом. Выражение **p->*m** связывает **m** с объектом, на который указывает **p**, и **obj.*m** связывает **m** с объектом **obj**. Результат должен использоваться в соответствие с типом члена **m**. Нельзя сохранить результат операций **->*** или **.*** для более позднего использования.

Естественно, что в коде, для которого заранее известен член класса, который нужно вызвать, его можно вызвать напрямую и не морочиться с указателями на члены. Как и в случае указателей на обычные функции, указатели на члены используются, когда требуется ссылаться на функции без необходимости знания их имени. Но указатель на член не является просто указателем на конкретную область памяти, как это имеет место в случае указателя на переменную или на обычную функцию. Он больше похож на смещение в структуре или индекс массива, но реализации в любом случае различают случаи членов-данных, виртуальных функций, невиртуальных функций и т.д. Когда комбинируется указатель на член с указателем на объект нужного типа, то получается нечто, что идентифицирует конкретный член для конкретного объекта.

Вызов **p->*s()** можно отобразить графически следующим образом:



Поскольку указатель на виртуальный член (**s** в данном случае) есть некая форма смещения, то он не зависит от местоположения объекта в памяти. В результате, указатель на виртуальный член можно передавать между разными пространствами адресов, пока в обоих используется одинаковая раскладка объектов. Сказанное неверно для указателей на обычные функции и для указателей на невиртуальные члены.

Отметим, что функция, которая вызывается посредством указателя на функцию, может быть виртуальной. Например, когда мы вызываем **suspend()** посредством указателя на функцию, мы получаем правильный вариант **suspend()** для объекта, к которому применяется указатель на функцию. Это существенное свойство указателей на функции.

При написании интерпретатора мы могли бы использовать указатели на члены, чтобы вызывать функции, представленные строками:

```
map<string, Std_Interface*> variable;
map<string, Pstd_mem> operation;

void call_member(string var, string oper)
{
    (variable[var]->*operation[oper])();        // var.oper()
}
```

Статический член не ассоциируется с конкретным объектом, так что указатель на статический член является обычным указателем. Например:

```
class Task {
    // ...
    static void schedule();
};

void (*p)() = &Task::schedule;          // OK
void (Task::* pm)() = &Task::schedule; // ошибка: обычный указатель присваивается
                                        // указателю на член
```

Указатели на члены-данные рассматриваются в §20.6.2.

20.6.2. Указатели на члены-данные

Естественно, что указатели на члены применимы к членам-данным и к функциям-членам, имеющим аргументы и тип возврата. Например:

```
struct C {
    const char* val;
    int i;

    void print(int x) {cout << val << x << '\n';}
    int f1(int);
    void f2();
    C(const char* v) {val = v;}
};

using Pmfi = void (C::*)(int);      // указатель на функцию-член C с аргументом типа int
using Pm = const char* C::*;        // указатель на член данных C типа char*

void f(C& z1, C& z2)
{
    C* p = &z2;
    Pmfi pf = &C::print;
    Pm pm = &C::val;
```

```
    z1.print(1);
    (z1.*pf)(2);
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);

    pf = &C::f1;    // ошибка: несоответствие типа возврата
    pf = &C::f2;    // ошибка: несоответствие типа аргумента
    pm = &C::i;     // ошибка: не тот тип
    pm = pf;        // ошибка: не тот тип
}
```

Тип указателя на функцию проверяется так же, как и любой другой тип.

20.6.3. Члены базового и производного классов

У производного класса есть по крайней мере члены, которые он наследует от своих базовых классов. Но часто есть и другие члены. Из этого следует, что мы безопасно может присваивать указатель на член базового класса указателю на член производного класса, но не наоборот. Например:

```
class Text : public Std_interface {
public:
    void start();
    void suspend();
    // ...
    virtual void print();
private:
    vector s;
};

void (Std_interface::* pmi)() = &Text::print;   // ошибка
void (Text::*pmt)() = &Std_interface::start;    // OK
```

Это правило кажется противоположным правилу, которое разрешает присваивать указатель на производный класс указателю на его базовый класс. На самом деле оба правила предназначены для поддержания фундаментальной гарантии, что указатель не указывает на объект, который не обладает атрибутами, обещаемыми указателем. В данном случае, **Std_interface::*** может применяться к любому **Std_interface**, и большинство таких объектов предположительно не типа **Text**. Следовательно, у них нет члена **Text::print**, с которым мы пытались инициализировать **pmi**. Отказав в инициализации, компилятор спас нас от ошибки времени выполнения.

20.7. Советы

[1] Избегайте полей типов; §20.3.1.

[2] Осуществляйте доступ к полиморфным объектам посредством указателей или ссылок; §20.3.2.

[3] Используйте абстрактные классы для фокусирования дизайна на чистых интерфейсах; §20.4.

[4] Применяйте **override**, чтобы явно отразить намерение к замещению в больших классовых иерархиях; §20.3.4.1.

[5] Не следует часто применять **final**; §20.3.4.2.

[6] Для спецификации интерфейсов применяйте абстрактные классы; §20.4.

[7] Используйте абстрактные классы с тем, чтобы держать детали реализации раздельно с интерфейсами; §20.4.
[8] Класс с виртуальными функциями должен иметь виртуальный деструктор; §20.4.
[9] В типичном случае абстрактный класс не нуждается в конструкторе; §20.4.
[10] Закрытые члены применяйте для отражения деталей реализации; §20.5.
[11] Открытые члены предназначайте для интерфейса; §20.5.
[12] Осторожно применяйте защищенные члены и только в случае действительной необходимости; §20.5.1.1.
[13] Не объявляйте члены-данные со спецификатором **protected**; §20.5.1.1.

21

Классовые иерархии

Абстракция есть выборочное невежество.
— Эндрю Кениг

- Введение
- Дизайн классовых иерархий

  Наследование реализации; Наследование интерфейса; Альтернативные реализации; Локализация создания объектов
- Множественное наследование

  Множественные интерфейсы; Множественные классы реализации; Разрешение неоднозначности; Повторное использование базовых классов; Виртуальные базовые классы; Реплицирование или виртуальный базовый класс
- Советы

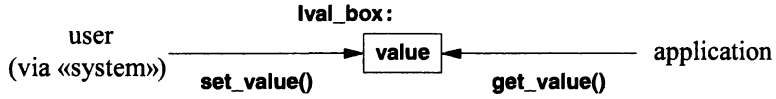
21.1. Введение

Настоящая глава сфокусирована на проблемах дизайна, а не на свойствах языка. Примеры соответствуют задачам графического интерфейса пользователя, но при этом я избегаю особенностей программирования на основе событий, что на самом деле типично для современных систем GUI. Нам нет необходимости углубляться в окончательные детали того, как некоторые действия на экране превращаются в реальные вызовы функций-членов, так как это ничего не добавит к рассмотрению проблемы классовых иерархий и просто отвлечет нас от основной темы. Если вы интересуетесь указанным вопросом, обратитесь к многочисленным библиотекам на эту тему.

21.2. Дизайн классовых иерархий

Рассмотрим проблему дизайна: как обеспечить возможность программе получить целое значение от пользователя. Это можно сделать невообразимо большим числом способов. Давайте начнем с того, что определим в нашей программе модель этой простой операции ввода.

Идея состоит в том, чтобы класс **Ival_box** знал диапазон целых значений, который он будет принимать. Программа будет запрашивать **Ival_box** насчет введенного значения и попросит сообщать пользователю при необходимости. Кроме того, программа попросит **Ival_box** сообщать о том, что пользователь ввел что-то новое после того, как программа опросила его в последний раз:



Поскольку существует много путей реализации указанной идеи, можно предположить, что могут существовать разные варианты для **Ival_box**, такие как ползунки, или простые текстовые поля, в которые пользователь будет вводить числа, а можно реализовать и голосовой ввод.

Самый общий подход будет состоять в том, чтобы построить некую «виртуальную систему взаимодействия с пользователем», которую будут использовать приложения. Ее можно будет реализовать на множестве разных операционных систем, чтобы помочь в переносимости приложений. Я буду рассматривать этот подход, который поможет мне продемонстрировать множество техник программирования и вопросов проектирования, причем эти техники весьма полезны, ибо используются во множестве реальных систем графического интерфейса, а также они полезны в иных предметных областях.

Ну и, дополнительно к игнорированию конкретной связи нашей системы интерфейса с библиотечными вызовами, я еще игнорирую здесь вопросы блокировок в мультипоточных системах GUI.

21.2.1. Наследование реализации

Наше первое решение заключается в иерархии классов, применяющих наследование реализации, что часто встречается в старых программах.

Класс **Ival_box** определяет базовый интерфейс для всех остальных вариантов графических элементов ввода и реализует умолчательное поведение, которое специфические виды элементов могут замещать в своем коде. Мы также объявляем и члены-данные, позволяющие реализовать базовое поведение:

```
class Ival_box {
protected:
    int val;
    int low, high;
    bool changed {false};    // изменения пользователя посредством set_value()
public:
    Ival_box(int ll, int hh): val{ll}, low{ll}, high{hh} {}

    virtual int get_value() {changed = false; return val;}        // для приложения
    virtual void set_value(int i) {changed = true; val = i;}      // для пользователя
    virtual void reset_value(int i) {changed = false; val = i;}   // для приложения
    virtual void prompt() {}
    virtual bool was_changed() const {return changed;}

    virtual ~Ival_box() {};
};
```

Мы приводим довольно неряшливую реализацию функций, предназначенную лишь для того, чтобы проиллюстрировать семантику. В более реалистичном варианте была бы выполнена проверка диапазона вводимых величин.

Программист может использовать классы элементов ввода следующим образом:

```
void interact(Ival_box* pb)
{
    pb->prompt(); // предупреждаем пользователя
    // ...
```

```
        int i = pb->get_value();
        if (pb->was_changed()) {
            // ... новое значение; делаем что-то ...
        }
        else {
            // ... иначе делаем то-то и то-то ...
        }
}

void some_fct()
{
        unique_ptr<Ival_box> p1 {new Ival_slider{0,5}};    // Ival_slider наследует от Ival_box
        interact(p1.get());

        unique_ptr<Ival_box> p2 {new Ival_dial{1,12}};
        interact(p2.get());
}
```

Большая часть кода написана в терминах указателей на «просто **Ival_box**», что видно из кода функции **interact()**. В результате приложению ничего не нужно знать про множество вариантов элементов ввода. Знание этого многообразия нужно лишь для малочисленных функций, создающих такие объекты. Это изолирует пользователя от изменений в реализации производных классов.

Я использую **unique_ptr** (§5.2.1, §34.3.1), чтобы не забывать удалять элементы ввода.

Ради упрощения я не рассматриваю вопрос о том, как программа ожидает ввода. Может она реально ждет ввода в **get_value()** (например, используя **get()** вместе с **future**; §5.3.5.1), а может она ассоциирует элемент ввода с событием и готовится реагировать обратным вызовом; наконец, программа может выполнять код для элемента ввода в отдельном потоке и опрашивать поток на предмет его состояния. Все это очень важно в дизайне системы пользовательского интерфейса. Но здесь такие вопросы просто отвлекут нас в сторону от основной темы.

Различные вариант элементов ввода определяются как производные классы от класса **Ival_box**. Например:

```
class Ival_slider : public Ival_box {
private:
        // ... графика для отображения спайдера и др., ...
public:
        Ival_slider(int, int);

        int get_value() override; // получить значение от пользователя и поместить его в val
        void prompt() override;
};
```

Данные-члены в **Ival_box** объявлены **protected** ради доступа к ним из производных классов. Таким образом, **Ival_slider::get_value()** может поместить значение в **Ival_box::val**. Мы знаем, что защищенные члены доступны из собственных функций-членов и из функций-членов производных классов, но не обычных пользователей (§20.5).

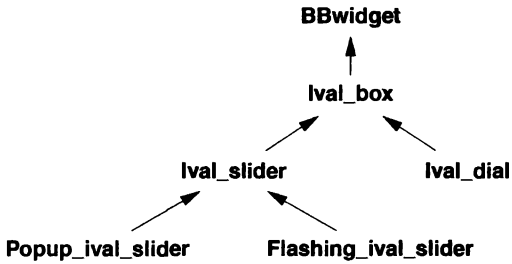
Дополнительно к **Ival_slider** мы может определять и иные варианты элементов ввода. Это может быть **Ival_dial**, при котором ввод выполняется нажатием на кнопку; **Flashing_ival_slider**, который мерцает, когда вы вызываетете для него **prompt()**; или **Popup_ival_slider**, который реагирует на **prompt()** тем, что появляется в некотором месте экрана так, чтобы пользователю было трудно его проигнорировать.

А откуда мы будем брать графическую начинку? Большинство графических интерфейсов предоставляют класс, определяющий базовые свойства тех сущностей, которые

будут появляться на экране. Таким образом, если мы используем систему от фирмы «Big Bucks Inc.», то мы должны будем сделать наши классы **lval_slider**, **lval_dial** и т.д. вариантами класса **BBwidget**. То есть, переписываем наш **lval_box** так, что он наследует от **BBwidget**. Тогда все наши классы будут наследовать все свойства от **BBwidget**. К примеру, любой **lval_box** может быть помещен на экран, он будет подчиняться правилам графического изображения, может изменяться в размере, перемещаться по экрану и т.д. согласно стандарту, установленному классом **BBwidget**. Наша классовая иерархия будет выглядеть следующим образом:

```
class lval_box : public BBwidget {/* ... */};        // переписан под BBwidget
class lval_slider : public lval_box {/* ... */};
class lval_dial : public lval_box {/* ... */};
class Flashing_ival_slider : public lval_slider {/* ... */};
class Popup_ival_slider : public lval_slider {/* ... */};
```

Или графически:



21.2.1.1. Критика

Представленный дизайн во многих отношениях работает хорошо, и для множества задач такой вид иерархии является приемлемым решением. Но есть и слабые места, которые могут заставить нас взглянуть на иные варианты.

Мы задним числом внедрили **BBwidget** в качестве базового класса для **lval_box**. Это не совсем верно (несмотря на широкое распространение в реальном коде). Применение **BBwidget** не входит в наш базовый дизайн для **lval_box**; это просто деталь реализации. Наследование **lval_box** от **BBwidget** выносит деталь реализации на уровень первичного дизайна. Это не может быть правильным. Мы просто решили на уровне нашей организации воспользоваться графической средой от «Big Bucks Inc.». Но что, если мы захотим также воспользоваться реализацией для **lval_box** от системы от «Imperial Bananas», «Liberated Software» или «Compiler Whizzes»? Нам придется поддерживать четыре версии программы:

```
class lval_box : public BBwidget {/* ... */};        // BB версия
class lval_box : public CWwidget {/* ... */};        // CW версия
class lval_box : public IBwidget {/* ... */};        // IB версия
class lval_box : public LSwindow {/* ... */};        // LS версия
```

Все это приведет к кошмару необходимости контроля версий.

В реальности нам не удастся найти простую самосогласованную схему с префиксом из двух букв. Более вероятно, что библиотеки поставщиков будут в разных пространствах имен и будут использовать разные системы терминологии для аналогичных концепций, такие как **BigBucks::Widget**, **Wizzies::control** и **LS::window**. Однако, это не касается напрямую нашей дискуссии, так что я не буду затрагивать проблему именования и вопросы использования пространств имен.

Другая проблема состоит в том, что все производные классы разделяют базовые данные, объявленные в **Ival_box**. Эти данные являются деталью реализации, прокравшейся в интерфейс класса **Ival_box**. С практической точки зрения эти данные во многих случаях некорректные. К примеру, для **Ival_slider** нет нужды хранить специфические данные. Они легко могут быть вычислены из позиции ползунка, когда кто-либо выполняет **get_value()**. В общем случае, когда работа ведется с двумя связанными логически, но раздельными данными, это только вопрос времени для возникновения ошибок. Кроме того, опыт показывает, что новички-программисты склонны к хаотическому использованию защищенных данных, что добавляет проблем при поддержке кода. Члены-данные предпочтительно иметь закрытыми, чтобы программисты производных классов не могли устроить мешанину с их использованием. Лучше всего, чтобы данные сосредотачивались в производных классах, где они могут выбираться точнее под специфические требования и не будут усложнять жизнь для несвязанных с ними производных классов. Почти во всех случаях, защищенный интерфейс должен состоять из функций, типов и констант.

Наследование от **BBwidget** дает выгоду в том, что средства этого класса становятся доступными пользователям **Ival_box**. К сожалению, изменения в **BBwidget** заставляют пользователей перекомпилировать или даже переписывать код, чтобы соответствовать изменениям. Большинство реализаций языка C++ предполагают перекомпиляцию всех производных классов при изменении лишь одного размера базового класса.

Наконец, может возникнуть ситуация, когда наша программа будет работать со смешанным интерфейсом (две системы разделяют один экран). Тогда при жестком внедрении в программу графической системы не будет необходимой гибкости.

21.2.2. Наследование интерфейса

Давайте начнем заново и построим классовую иерархию, решающую рассмотренные выше проблемы традиционных иерархий:

1. Система графического интерфейса должна быть деталью реализации, скрытой от тех пользователей, которые не хотят знать об этом.
2. Класс **Ival_box** не должен содержать данные.
3. Не должно требоваться перекомпиляции кода иерархией классов, использующих **Ival_box**, после изменения в системе интерфейса пользователя.
4. Разные **Ival_box** для разных систем должны иметь возможность сосуществовать в рамках одной программы.

Для реализации изложенного можно выбрать разные подходы. Здесь я представляю один подход, который хорошо реализуется на языке C++.

Сначала я определяю класс **Ival_box** как чистый интерфейс:

```
class Ival_box {
public:
      virtual int get_value() = 0;
      virtual void set_value(int i) = 0;
      virtual void reset_value(int i) = 0;
      virtual void prompt() = 0;
      virtual bool was_changed() const = 0;
      virtual ~Ival_box() {}
};
```

Все данные ушли. Ушли также и конструкторы, ибо нет ничего для инициализации. Вместо этого я добавил виртуальный деструктор, чтобы гарантировать корректную очистку данных, которые будут определены в производных классах.

Определение **Ival_slider** может быть следующим:

```
class Ival_slider : public Ival_box, protected BBwidget {
public:
    Ival_slider(int, int);
    ~Ival_slider() override;

    int get_value() override;
    void set_value(int i) override;
    // ...
protected:
    // ... функции, замещающие виртуаль. функции у BBwidget
    // например, BBwidget::draw(), BBwidget::mouse1hit() ...
private:
    // ... данные для slider ...
};
```

Производный класс **Ival_slider** наследует от абстрактного класса **Ival_box**, требующего реализовать его чисто виртуальные функции. Он также наследует от **BBwidget**, который предоставляет средства для этого. Поскольку **BBwidget** используется в качестве помощи в реализации, то при его наследовании используется спецификатор **protected** (§20.5.2). Из этого следует, что программист, использующий **Ival_slider**, не может напрямую использовать средства, определенные классом **BBwidget**. Интерфейс, который предоставляет **Ival_slider**, приобретен им от **Ival_box**, плюс его собственные добавки. Я применил при наследовании **protected** вместо более ограничивающего (и более безопасного) **private**, чтобы сделать **BBwidget** доступным классам, производным от **Ival_slider**. Я явно использовал **override**, поскольку данная иерархия не слишком простая и полезно увеличить наглядность, которая поможет избежать недоразумений впоследствии.

Одновременное наследование от нескольких классов называется множественным наследованием (§21.3). Заметим, что **Ival_slider** должен замещать функции как из **Ival_box**, так и из **BBwidget**. Как показано в §21.2.1.1, косвенное наследование **Ival_slider** от **BBwidget** посредством того, что **BBwidget** становится базовым для **Ival_box**, возможно, но при этом возникает неприяные побочные эффекты. Аналогично, делать класс реализации **BBwidget** членом класса **Ival_box** не выход, поскольку класс не может замещать виртуальные функции его членов. А если сделать **BBwidget*** членом **Ival_box**, то мы придем к совершенно иному дизайну со своими достоинствами и недостатками.

Многие люди воспринимают множественное наследование как нечто сложное и пугающее. Однако, использование одного класса в качестве класса реализации и другого класса (абстрактного) для фиксации интерфейса является общим местом для всех языков, поддерживающих наследование и проверку интерфейсов на этапе компиляции. Примером служат языки C# и Java.

У многих классов имеется необходимость в очистке ресурсов перед тем, как будут уничтожены объекты. Поскольку абстрактный класс **Ival_box** не может знать, потребуется ли очистка ресурсов для производных классов, лучше предположить, что таковая может потребоваться. Мы гарантируем очистку посредством объявления виртуального деструктора **Ival_box::~Ival_box()** в базовом классе и замещением его в производных классах. Например:

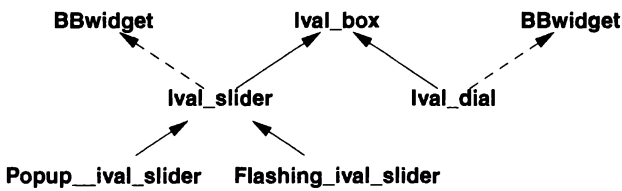
```
void f(Ival_box* p)
{
    // ...
    delete p;
}
```

Операция **delete** явным образом уничтожает объект, на который указывает **p**. Мы точно не знаем, на объект какого конкретного класса указывает **p**, но благодаря виртуальному деструктору класса **Ival_box** мы получим надлежащую очистку посредством деструктора соответствующего класса.

Теперь определим нашу иерархию элементов графического интерфейса следующим образом:

```
class Ival_box {/* ... */};
class Ival_slider
      : public Ival_box, protected BBwidget {/* ... */};
class Ival_dial
      : public Ival_box, protected BBwidget {/* ... */};
class Flashing_ival_slider
      : public Ival_slider {/* ... */};
class Popup_ival_slider
      : public Ival_slider {/* ... */};
```

Или графически:



Я использовал пунктирную линию, чтобы отразить факт **protected**-наследования (§20.5.1). Обычные пользователи не будут иметь доступа к базовым классам, от которых ведется **protected**-наследование (они справедливо рассматриваются как часть реализации).

21.2.3. Альтернативные реализации

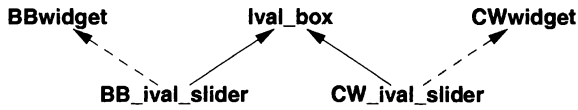
Представленный дизайн весьма чистый и удобный для поддержки, однако, он не решает проблемы контроля версий:

```
class Ival_box {/* ... */}; // общий
class Ival_slider
      : public Ival_box, protected BBwidget {/* ... */};   // для BB
class Ival_slider
      : public Ival_box, protected CWwidget {/* ... */};   // для CW
// ...
```

Здесь нет никакой возможности сосуществования **Ival_slider** с **BBwidget** и **Ival_slider** с **CWwidget**, даже если сами эти графические системы могут друг с другом ужиться. Очевидным решением будет определить множество разных **Ival_slider**-классов с различающимися именами:

```
class Ival_box {/* ... */};
class BB_ival_slider
      : public Ival_box, protected BBwidget {/* ... */};
class CW_ival_slider
      : public Ival_box, protected CWwidget {/* ... */};
// ...
```

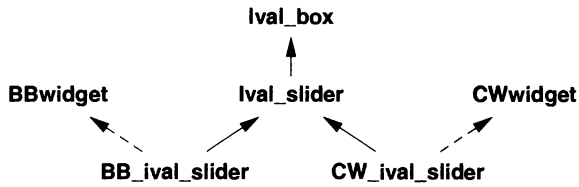
Или графически:



Ради дальнейшей изоляции наших классов **Ival_box** от деталей реализации, мы можем наследовать абстрактный класс **Ival_slider** от **Ival_box**, а затем уже наследовать специфические элементы-слайдеры:

```
class Ival_box {/* ... */};
class Ival_slider
    : public Ival_box {/* ... */};
class BB_ival_slider
    : public Ival_slider, protected BBwidget {/* ... */};
class CW_ival_slider
    : public Ival_slider, protected CWwidget {/* ... */};
// ...
```

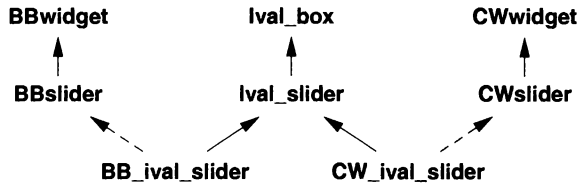
Или графически:



Обычно можно для удобства применить еще несколько более специфических классов в иерархии реализации. К примеру, если в системе от «Big Bucks Inc.» есть свой слайдерный класс, то мы можем наш слайдер наследовать непосредственно от **BBslider**:

```
class BB_ival_slider
    : public Ival_slider, protected BBslider {/* ... */};
class CW_ival_slider
    : public Ival_slider, protected CWslider {/* ... */};
```

Или графически:



Такое решение бывает полезно в случаях, когда наши абстракции не сильно отличаются от тех, что предоставляются системами. Тогда наследование от общих базовых классов вроде **BBwidget** применяется намного реже.

В итоге мы приходим к следующей иерархии, состоящей из наших оригинальных интерфейсных классов, определенных как производные:

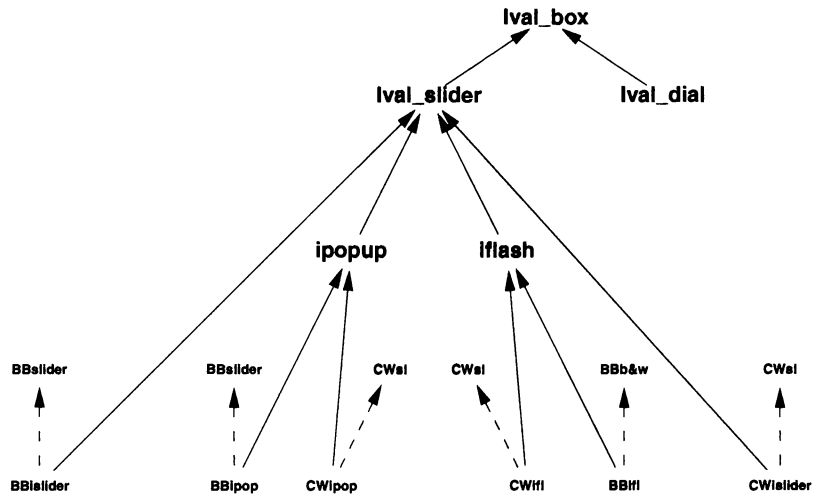
```
class Ival_box {/* ... */};
class Ival_slider
    : public Ival_box {/* ... */};
class Ival_dial
    : public Ival_box {/* ... */};
```

```
class Flashing_ival_slider
      : public Ival_slider {/* ... */};
class Popup_ival_slider
      : public Ival_slider {/* ... */};
```

За этим следуют реализации этой иерархии для различных графических систем интерфейса, выраженных как производные классы:

```
class BB_ival_slider
      : public Ival_slider, protected BBslider {/* ... */};
class BB_flashing_ival_slider
      : public Flashing_ival_slider, protected BBwidget_with_bells_and_whistles {/* ... */};
class BB_popup_ival_slider
      : public Popup_ival_slider, protected BBslider {/* ... */};
class CW_ival_slider
      : public Ival_slider, protected CWslider {/* ... */};
// ...
```

Применяя очевидные сокращения, представим эту иерархию графически:



Похоже, что оригинальная иерархия **Ival_box**-классов осталось нетронутой, будучи окруженной классами реализации.

21.2.3.1. Критика

Дизайн на основе абстрактных классов гибкий и почти столь же прост, как и дизайн на основе общего базового класса, определяющего пользовательскую систему интерфейса. В последнем дизайне оконный класс является корнем дерева. С точки зрения приложений, оба дизайна эквивалентны в том смысле, что код работает почти без изменений в обоих случаях. Вы можете работать с семейством **Ival_box**-классов, по большей части не беспокоясь о деталях реализации оконной системы. Например, нам не нужно переписывать **interact()** из §21.2.1, если мы переключимся с одной классовой иерархии на другую.

В обоих случаях реализации каждого **Ival_box**-класса должны переписываться, когда изменяется публичный интерфейс системы графического пользовательского инетрфейса. Тем не менее, в иерархии на базе абстрактных классов большая часть пользовательского кода защищена от изменений и не требует перекомпиляции. Это особо полезно в случае, когда поставщик иерархии реализации выпускает новый «почти совместимый» релиз.

Пользователи иерархии с абстрактными классами не могут случайно применить средства из иерархии реализации, поскольку им доступно лишь то, что явно специфицировано в **Ival_box**-иерархии; ничто не наследуется неявным образом от классов со специфическими реализациями.

Логический вывод из рассмотренного следующий: желательна система, которая пользователю представляется как иерархия абстрактных классов, а реализуется она классической иерархией. Другими словами:

- Используйте абстрактные классы для поддержки наследования интерфейса (§3.2.3, §20.1).
- Используйте базовые классы с реализациями виртуальных функций для поддержки наследования реализации (§3.2.3, §20.1).

21.2.4. Локализация создания объектов

Большая часть приложений может быть написана с применением **Ival_box**-интерфейса. Если производные интерфейсы эволюционируют ради предоставления дополнительных свойств по сравнению с простым **Ival_box**, то тогда большая часть приложений может быть написана, используя **Ival_box**, **Ival_slider** и подобных интерфейсов. Однако, создание объектов должно выполняться с использованием имен, специфичных для реализаций, таких как **CW_ival_dial** и **BB_flashibg_ival_slider**. Хотелось бы минимизировать число мест в программе, где используются столь специфичные имена. Локализации создания объектов не получится, если не выполнять ее систематически.

Как всегда, решение находится в косвенном обращении. Это можно сделать поразному. Простым способом будет введение абстрактного класса для представления операций создания объектов:

```
class Ival_maker {
public:
    virtual Ival_dial* dial(int, int) =0;                    // делаем dial
    virtual Popup_ival_slider* popup_slider(int, int) =0;    // делаем popup slider
    // ...
};
```

Для каждого интерфейса из **Ival_box**-семейства классов, о котором пользователь должен знать, класс **Ival_maker** предоставляет функцию, создающую объект. Такой класс часто называют фабричным классом, а его функции часто называют виртуальными конструкторами (§20.3.6).

Мы теперь представляем каждую систему пользовательского интерфейса классом, производным от **Ival_maker**:

```
class BB_maker : public Ival_maker {        // делаем BB версии
public:
    Ival_dial* dial(int, int) override;
    Popup_ival_slider* popup_slider(int, int) override;
    // ...
};

class LS_maker : public Ival_maker {        // делаем LS версии
public:
    Ival_dial* dial(int, int) override;
    Popup_ival_slider* popup_slider(int, int) override;
    // ...
};
```

Каждая функция создает объект типа желаемого интерфейса и типа реализации. Например:

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a, b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a, b);
}
```

Имея класс **Ival_maker**, пользователь может создавать объекты, не зная точно, какая система пользовательского интерфейса используется. Пример:

```
void user(Ival_maker& im)
{
    unique_ptr<Ival_box> pb {im.dial(0,99)};      // настраиваем соответствующий dial
    // ...
}

BB_maker BB_impl;          // для BB пользователей
LS_maker LS_impl;          // для LS пользователей

void driver()
{
    user(BB_impl);         // используем BB
    user(LS_impl);         // используем LS
}
```

Передача аргументов таким «виртуальным конструкторам» несколько нетривиальна. В частности, мы не можем замещать функции базового класса, представляющими интерфейсы с разными аргументами в производных классах. Отсюда вывод: требуется все-таки некоторая доля предвидения при конструировании интерфейса фабричного класса.

21.3. Множественное наследование

Как описано в §20.1, наследование дает два преимущества:

- Общие интерфейсы: помогает меньше дублировать код с использованием классов и позволяет делать такой код более однородным. Об этом говорят как о полиморфизме времени выполнения или наследовании интерфейса.
- Общая реализация: приводит к меньшему объему кода и к более однородному коду реализации. Часто называется наследованием реализации.

В классе можно комбинировать оба этих подхода.

Далее мы более подробно изучим общие случаи применения множественных базовых классов, а также более технические вопросы, связанные с множественным наследованием.

21.3.1. Множественные интерфейсы

Абстрактный класс (вроде **Ival_box**; §21.2.2) является очевидным способом представления интерфейса. Для абстрактного класса без изменяемого состояния мало разницы между вариантами, когда используется один или много базовых классов в классовой иерархии. Разрешение потенциальных неоднозначностей обсуждается в §21.3.3, §21.3.4

и §21.3.5. Любой класс без изменяемого состояния может использоваться в качестве интерфейса в решетке со множественным наследованием без каких-либо сложностей и накладных расходов.

Применение множественных абстрактных классов в качестве интерфейсов является практически универсальным в объектно-ориентированном проектировании (на любом языке, имеющим понятие интерфейса).

21.3.2. Множественные классы реализации

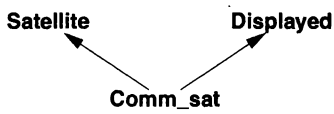
Рассмотрим задачу о моделировании тел, вращающихся вокруг Земли, где вращающиеся объекты представлены классом **Satellite**. Объект класса **Satellite** обладает орбитой, размером, параметрами плотности и т.д., и предоставляет операции для расчета орбиты, модифицирования атрибутов и т.д. Примерами спутников служат остатки от коммуникационных спутников, а также Международная Станция. Все эти объекты будут соответствовать классам, производным от класса **Satellite**. Эти производные классы будут добавлять члены-данные и функции-члены, а также будут замещать некоторые из виртуальных функций класса **Satellite** с целью подогнать их действие к конкретным обстоятельствам.

Теперь предположим, что я хочу отобразить результаты моделирования графически и что имеется графическая система, использующая стратегию производных объектов для их отображения с помощью общего базового класса, хранящего графическую информацию. Этот графический класс предоставляет операции для помещения объекта на экран, масштабирования и т.д. Ради простоты и ясности я буду ссылаться на этот класс как на класс **Displayed**.

Теперь определим класс для моделируемых коммуникационных спутников, класс **Comm_sat**:

```
class Comm_sat : public Satellite, public Displayed {
public:
    // ...
};
```

Или графически:



Дополнительно к операциям, чпециально предназначенных для **Comm_sat**, можно использовать операции от **Satellite** и **Displayed**. Например:

```
void f(Comm_sat& s)
{
    s.draw();              // Displayed::draw()
    Pos p = s.center();    // Satellite::center()
    s.transmit();          // Comm_sat::transmit()
}
```

Аналогично, можно передать **Comm_sat** функции, которая ожидает **Satellite**, и функции, ожидающей **Displayed**. Например:

```
void highlight(Displayed*);
Pos center_of_gravity(const Satellite*);
```

```
void g(Comm_sat* p)
{
    highlight(p);                    // передаем указатель части Displayed у Comm_sat
    Pos x = center_of_gravity(p);    // передаем указатель части Satellite у Comm_sat
}
```

Реализации должны обеспечить гарантию того, что функции, ожидающие **Satellite**, видят разные части **Comm_sat**, чем это делают функции, ожидающие **Displayed**. Например:

```
class Satellite {
public:
    virtual Pos center() const = 0; // центр гравитации
    // ...
};

class Displayed {
public:
    virtual void draw() = 0;
    // ...
};

class Comm_sat : public Satellite, public Displayed {
public:
    Pos center() const override;  // замещаем Satellite::center()
    void draw() override;         // замещаем Displayed::draw()
    // ...
};
```

Это гарантирует, что **Comm_sat::center()** и **Displayed::draw()** будут вызваны для **Comm_sat**, трактуемого как **Comm_sat** и **Displayed**, соответственно.

Почему бы не хранить **Satellite** и **Displayed** части **Comm_sat** совершенно раздельно? Я мог бы определить **Comm_sat** так, чтобы у него были члены **Satellite** и **Displayed**. Или я мог бы определить в **Comm_sat** члены типов **Satellite*** и **Displayed***, и позволить конструкторам устанавливать соответствующие соединения. Во многих случаях я так бы и сделал. Но система, которая вдохновила меня на этот пример, основывалась на идее класса **Satellite** с виртуальными функциями и на отдельно спроектированном классе **Displayed**, также с виртуальными функциями. И мы производим наши собственные спутники и свои собственные отображаемые (displayed) объекты посредством наследования. В частности, нам нужно замещать виртуальные функции классов **Satellite** и **Displayed** для уточнения поведения своих объектов. Это тот случай, когда трудно избежать множественного наследования от базовых классов с состоянием и реализацией.

Использование множественного наследования для склеивания несвязанных классов для реализации третьего класса эффективно и важно с практической точки зрения, но не слишком интересно. Главное, что эта техника освобождает программиста от написания множества перенаправляющих функций. Она не влияет существенно на общий дизайн программы и может иногда вступать в противоречие с желанием скрывать детали реализации. Но на практике техника программирования не обязана быть слишком умной, чтобы быть полезной.

В общем случае я предпочитаю иметь единственную иерархию реализации и (если необходимо) несколько абстрактных классов, предоставляющих интерфейсы. Это более гибко и помогает строить системы, удобные для эволюционирования. В то же время, этого не всегда можно добиться, особенно в случаях, когда предполагается использовать существующие классы, которые нежелательно модифицировать (они могут быть частью чьих-то библиотек).

Отметим, что если ограничиться лишь простым одиночным (не множественным) наследованием, то выбор программиста для реализации классов **Displayed**, **Satellite** и **Comm_sat** будет ограниченным. Тогда **Comm_sat** может быть вариантом **Satellite** или **Displayed**, но не обоими одновременно. В любом случае мы теряем в гибкости.

Наш пример с **Satellite** имеет реальную историческую подоплеку. Мы применили эти разработки для описания множественного наследования реализации. А ранее эта тема использовалась для проектирования коммуникационных систем на основе спутников, станций наблюдения на Земле и т.д.

21.3.3. Разрешение неоднозначности

Два базовых класса могут иметь функции-члены с одинаковыми именами. Например:

```
class Satellite {
public:
    virtual Debug_info get_debug();
    // ...
};

class Displayed {
public:
    virtual Debug_info get_debug();
    // ...
};
```

В случае с **Comm_sat** эти функции должны подвергнуться разрешению имен. Это можно сделать квалификацией имени функции именем класса:

```
void f(Comm_sat& cs)
{
    Debug_info di = cs.get_debug();    // ошибка: неоднозначность
    di = cs.Satellite::get_debug();    // OK
    di = cs.Displayed::get_debug();    // OK
}
```

Но разрешение неоднозначностей путем явной квалификации довольно хаотичное решение, так что лучше решать эти проблемы определением новой функции в производном классе:

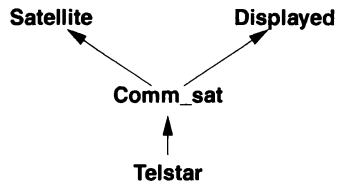
```
class Comm_sat : public Satellite, public Displayed {
public:
    Debug_info get_debug() // замещаем Comm_sat::get_debug() и Displayed::get_debug()
    {
        Debug_info di1 = Satellite::get_debug();
        Debug_info di2 = Displayed::get_debug();
        return merge_info(di1, di2);
    }
    // ...
};
```

Функция, объявленная в производном классе, замещает все функции с таким же именем и типом в базовых классах. Идеалом будет вызов функции, имеющий эффект, не зависящий от того, какой интерфейс использован для вызова (§20.3.2).

В реализации замещающей функции приходится применять явную квалификацию, чтобы определить правильную версию из базового класса. Квалифицированное имя, такое как **Telstar::draw**, может обращаться к **draw()**, объявленной в **Telstar** или в одном из его базовых классов. Пример:

```
class Telstar : public Comm_sat {
public:
    void draw()
    {
        Comm_sat::draw();        // находим Displayed::draw
        // ... собственная работа ...
    }
    // ...
};
```

Или графически:



Если **Comm_sat::draw** не получит разрешения в сторону функции **draw()**, объявленной в **Comm_sat**, то компилятор рекурсивно обратится к его базовым классам; то есть он будет просматривать **Satellite::draw** и **Displayed::draw**, а при необходимости обратится и к их базовым классам. Если найдется точное соответствие, то это имя и будет использоваться. А в ином случае имеем **Comm_sat::draw** несуществующим, либо имеет место неоднозначность.

Если бы вместо **Telstar::draw()** я применил вызов «просто» **draw()**, то результатом был бы бесконечный рекурсивный вызов функции **Telstar::draw()**.

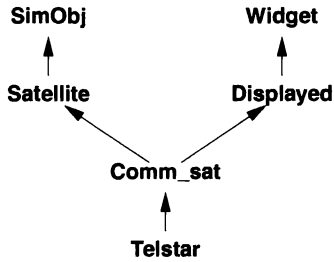
Я мог бы применить **Displayed::draw()**, но таким образом код был бы тонко скомпрометирован, если кто-нибудь добавил бы **Comm_sat::draw()**; в общем случае лучше ссылаться на непосредственный базовый класс, чем на косвенный базовый класс. Я мог бы использовать **Comm_sat:: Displayed::draw()**, но это избыточно. А в случае применения **Satellite::draw()** была бы ошибка компиляции, поскольку **draw** отсутствует в **Displayed**-ветви классовой иерархии.

В примере **get_debug()** предполагалось, что по крайней мере некоторые части **Satellite** и **Displayed** разрабатывались совместно. То есть полного совпадения имен, типов возврата, типов аргументов и семантики можно было избежать. Более вероятна была бы реализация сходной функциональности разными способами, так что пришлось бы потратить усилия на то, чтобы согласовать их. Мы также могли бы получить два класса **SimObj** и **Widget**, которые не могли бы модифицировать и которые не давали бы нам то, что нужно, а если бы и давали, то предоставляли бы несовместимые интерфейсы. В таком случае мы проектировали бы **Satellite** и **Displayed** в качестве интерфейсных классов, предоставляя согласующий слой, чтобы использовать наши высокоуровневые классы:

```
class Satellite : public SimObj {
    // средства SimObj более простые для имитирования Satellite
public:
    virtual Debug_info get_debug();    // вызов SimObj::DBinf() и извлечение информации
    // ...
};

class Displayed : public Widget {
    // средства Widget более простые для показа результатов имитирования Satellite
public:
    virtual Debug_info get_debug();    // читаем Widget данные и составляем Debug_info
    // ...
};
```

Или графически:



Интересно, что это в точности та же самая техника, которую мы использовали бы для разрешения неоднозначности в менее вероятном случае, когда два базовых класса предоставляют операции с точно совпадающими именами, но с разной семантикой: добавили бы интерфейсный слой. Рассмотрим классический пример класса с функцией-членом **draw()** для видеоигры с ковбоями:

```
class Window {
public:
    void draw();        // отображаем образ
    // ...
};

class Cowboy {
public:
    void draw();        // достаем пистолет из кобуры
    // ...
};

class Cowboy_window : public Cowboy, public Window {
    // ...
};
```

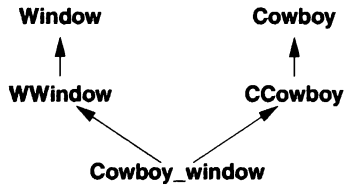
Как нам заместить **Cowboy::draw()** и **Window::draw()**? Эти две функции имеют совершенно разный смысл, но идентичны по имени и типам; мы должны заместить их двумя разными функциями. Не существует прямого решения этой проблемы средствами языка, но введение промежуточных классов помогает:

```
struct WWindow : Window {
    using Window::Window;                   // наследуем конструкторы
    virtual void win_draw() = 0;            // заставляем производный класс замещать
    void draw() override final {win_draw();}    // вывести изображение
};

struct CCowboy : Cowboy{
    using Cowboy::Cowboy;                   // наследуем конструкторы
    virtual void cow_draw() = 0;            // заставляем производный класс замещать
    void draw() override final {cow_draw();}    // достаем пистолет из кобуры
};

class Cowboy_window : public CCowboy, public WWindow {
public:
    void cow_draw() override;
    void win_draw() override;
    // ...
};
```

Или графически:



Если бы дизайнер класса **Window** был чуточку более аккуратным и объявил **draw()** как **const**, то вся проблема испарилась бы.

21.3.4. Повторное использование базовых классов

Когда каждый класс имеет единственный непосредственный базовый класс, то классовая иерархия имеет вид дерева, и класс существует в иерархии в единственном экземпляре. Когда у класса множественные базовые классы класс может появляться в иерархии несколько раз. Рассмотрим класс, предоставляющий функциональность хранения состояния в файле с последующим ее восстановлением:

```
struct Storable {    // хранилище
    virtual string get_file() = 0;
    virtual void read() = 0;
    virtual void write() = 0;

    virtual ~Storable() {}
};
```

Столь полезный класс будет использоваться в нескольких местах в классовой иерархии. Например:

```
class Transmitter : public Storable {
public:
    void write() override;
    // ...
};

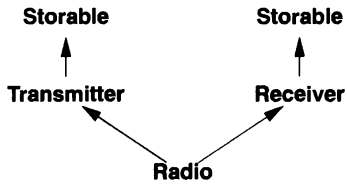
class Receiver : public Storable {
public:
    void write() override;
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    string get_file() override;
    void read() override;
    void write() override;
    // ...
};
```

Исходя из этого, мы можем представить два случая:

1. **Radio**-объект имеет два подобъекта класса **Storable** (один для **Transmitter** и еще один для **Receiver**).
2. **Radio**-объект имеет один подобъект класса **Storable** (общий для **Transmitter** и **Receiver**).

По умолчанию, то, что имеет место для написанного примера, это два подобъекта. Графически отображаем это следующим образом:



Виртуальная функция реплицируемого базового класса может быть замещена единственной функцией в производном классе. В типичном случае замещающая функция вызывает версию из базового класса, а затем уже выполняет работу, специфичную для производного класса:

```
void Radio::write()
{
    Transmitter::write();
    Receiver::write();
    // ... radio-специфичная информация ...
}
```

Приведение типа от реплицируемого базового класса к производному классу обсуждается в §22.2. Техника замещения каждой функции **write()** разными функциями из производных классов рассматривается в §21.3.3.

21.3.5. Виртуальные базовые классы

Пример с **Radio** из предыдущего раздела работает, поскольку класс **Storable** может безопасно и удобно реплицироваться. Причина этого проста: класс **Storable** является абстрактным, предоставляющим чистый интерфейс. Никаких данных для класса **Storable** внутри объектов нет. Это простейший случай, при котором наилучшим образом разделяются интерфейс и реализации. При этом класс без специальных усилий не сможет задать два подобъекта **Storable** в **Radio**.

А если бы **Storable** имел данные и было бы важно, чтобы они не реплицировались? Например, мы могли бы определить **Storable** так, чтобы харанилось имя файла, предназначенного для хранения объекта:

```
class Storable {
public:
    Storable(const string& s);      // храним в файле s
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
protected:
    string file_name;

    Storable(const Storable&) = delete;
    Storable& operator=(const Storable&) = delete;
};
```

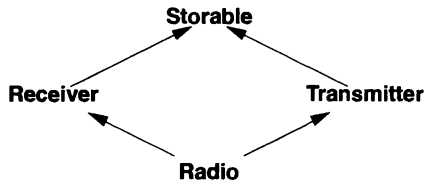
Получив это небольшое изменение в **Storable**, мы должны будем изменить дизайн **Radio**. Все части объекта должны совместно разделять единственную копию **Storable**. Иначе можно получить две части чего-то, производного от **Storable** несколько раз, используя разные файлы. Можно избежать реплицирования путем объявления базового класса с ключевым словом **virtual**: любая часть от виртуального наследования представляется одним и тем же разделяемым объектом. Например:

```
class Transmitter : public virtual Storable {
public:
    void write() override;
    // ...
};

class Receiver : public virtual Storable {
public:
    void write() override;
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    void write() override;
    // ...
};
```

Или графически:



Сравните эту диаграмму с рисунком для объекта **Storable** в §21.3.4, чтобы увидеть разницу между обычным наследованием и виртуальным наследованием. На графе наследования любой базовый класс, специфицированный с ключевым словом **virtual**, будет представлен единственным образом. С другой стороны, любой базовый класс без спецификатора **virtual** будет представляться подобъектом, представляющим этот класс.

Когда может потребоваться виртуальное наследование от класса, содержащего данные? Я представляю себе три случая, когда два класса из классовой иерархии разделяют общие данные:

1. Имеются нелокальные данные (глобальные или в пространстве имен).
2. Данные помещены в базовый класс.
3. Данные где-либо аллоцируются и двум классам предоставляются указатели на эти данные.

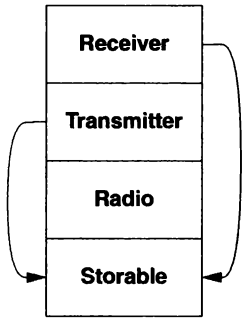
Случай 1 неоптимальный, поскольку мы не можем контролировать доступ к этим данным. Он ломает все представления об инкапсуляции и локализации.

Случай 2, когда данные помещаются в базовый класс, обычно самый простой. Однако, в случае одиночного наследования этот метод приводит к тому, что все полезные данные всплывают в иерархии все выше и выше в самый корень (в общий базовый класс). В результате все классы из иерархии получают доступ к данным. Получается, что этот случай логически близок к первому случаю со всеми его недостатками. В итоге, мы явно нуждаемся в общем базовом классе, который не является вершиной дерева — то есть виртуальный базовый класс.

Случай 3, то есть разделение данных, доступных по указателям, имеет определенный смысл. Но теперь конструкторы должны выделять память под общие данные, инициализировать их, а также предоставить указатели на эти данные всем нуждающимся в них объектам. Можно сказать, что в определенном смысле конструкторы делают то, что реализует виртуальный базовый класс.

Если вы не нуждаетесь в разделении данных, то вы можете обойтись без виртуального базового класса, и при этом ваш код будет чище и проще. Но если вы все-таки нуждаетесь в таких данных, то у вас есть выбор — применить виртуальный базовый класс или скрупулезно программировать все необходимые аспекты самостоятельно, реализуя собственный вариантр исполнения этой идеи.

Мы можем представить графически объект класса с виртуальным базовым классом:



21.3.5.1. Конструируем виртуальные базовые классы

Применяя виртуальные базовые классы, вы можете строить сложные решетки наследования. Естественно, что лучше сохранять такие решетки в более простых конфигурациях, но как бы ни были сложны эти решетки, язык C++ гарантирует, что конструктор виртуального базового класса будет вызван ровно один раз. При этом конструктор базового класса (виртуального или нет) будет вызван раньше, чем конструкторы производных классов. Во избежание хаоса конструктор виртуального класса вызывается из конструктора наиболее полного объекта («самого производного» класса). Это гарантирует единственный вызов конструктора виртуального базового класса, даже если он используется во многих местах классовой иерархии. Пример:

```
struct V {
    V(int i);
    // ...
};

struct A {
    A(); // умолчательный конструктор
    // ...
};

struct B : virtual V, vir tual A {
    B(): V{1} {/* ... */};        // умолчательный конструктор; нужно инициалйзир. базу V
    // ...
};

class C : virtual V {
public:
    C(int i) :  V{i} {/* ... */};        // нужно инициализировать базу V
    // ...
};

class D : virtual public B, virtual public C {
    // неявно получаем виртуальную базу V от B и C
    // неявно получаем виртуальную базу A от B
```

```
public:
    D() {/* ... */}                          // ошибка: нет умолчат. конструктора у C или V
    D(int i): C{i} {/* ... */};              // ошибка: у V нет умолчатель. конструктора
    D(int i, int j): V{i}, C{j} {/* ... */}  // OK
    // ...
};
```

Заметьте, что **D** может и должен предоставить инициализатор для **V**. И не имеет значения, что **V** не был явным образом упомянут как базовый класс для **D**. Виртуальный базовый класс всегда рассматривается как прямой базовый класс для наиболее производного класса. Тот факт, что как **B**, так и **C**, инициализировали **V**, не важен, поскольку компилятор не знает, какой из двух инициализаторов предпочесть. В итоге, применяется лишь инициализатор наиболее производного класса.

Конструктор виртуального базового класса вызывается раньше, чем конструкторы производных классов.

На практике, все это не так уж хорошо локализуется, как хотелось бы. В частности, если мы определим еще один производный от **D** класс, **DD**, тогда **DD** должен будет выполнить работу по инициализации виртуального базового класса. Все это должно предостеречь нас от неоправданно интенсивного использования виртуальных базовых классов.

Логической проблемы, существующей для конструкторов, не существует для деструкторов. Они просто вызываются в порядке, обратном порядку конструирования (§20.2.2). Деструктор виртуального базового класса вызывается ровно один раз.

21.3.5.2. Вызов функции-члена виртуального базового класса

Определяя функции для класса, имеющего виртуальный базовый класс, программист в общем случае не может знать, будут ли у базового класса другие производные классы. Это может породить проблемы, если требуется, чтобы функции базового класса могли вызываться лишь единожды для каждого вызова функции производного класса. Там, где это требуется, программист может самостоятельно имитировать схему, которая используется для конструкторов и при которой функция виртуального базового класса вызывается только из «наиболее производного» класса. К примеру, предположим, что у нас есть класс **Window**, который знает как отрисовывать свое содержимое:

```
class Window {
public:
    // базовый материал
    virtual void draw();
};
```

Кроме того, имеется много способов декорирования окон и добавления средств для работы с ними:

```
class Window_with_border : public virtual Window {
    // материал для рамки
protected:
    void own_draw(); // отобразить рамку
public:
    void draw() override;
};

class Window_with_menu : public virtual Window {
    // материал для меню
protected:
    void own_draw();   // отображение меню
```

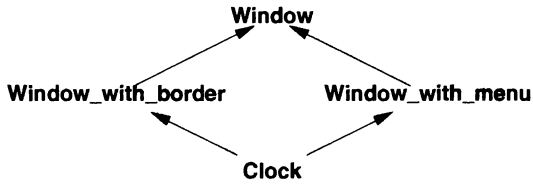
```
public:
    void draw() override;
};
```

Функции **own_draw()** не требуется делать виртуальными, поскольку они должны вызываться из виртуальной функции **draw()**, которая «знает» тип объекта, для которого она вызвана.

Теперь можно составить класс **Clock**:

```
class Clock : public Window_with_border, public Window_with_menu {
    // материал для часов
protected:
    void own_draw();          // отобразить циферблат часов и стрелки
public:
    void draw() override;
};
```

Или графически:



Можно определить функции **draw()**, используя функции **own_draw()**, так чтобы код, вызывающий **draw()**, обеспечивал единственный вызов функции **Window::draw()**. Это достигается независимо от конкретного вида **Window**, для которого вызывается **draw()**:

```
void Window_with_border::draw()
{
    Window::draw();
    own_draw();          // отобразить рамку
}

void Window_with_menu::draw()
{
    Window::draw();
    own_draw();          // отобразить меню
}

void Clock::draw()
{
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw();          // отобразить циферблат часов и стрелки
}
```

Заметьте, что квалифицированный вызов, такой как **Window::draw()**, не применяет механизм виртуальных функций. Вместо этого, напрямую вызывается явно поименованная функция, избегая бесконечной рекурсии.

Приведение от виртуального базового класса к производному классу обсуждается в §22.2.

21.3.6. Реплицирование или виртуальный базовый класс

Использование множественного наследования для реализации абстрактных классов, представляющих чистый интерфейс, влияет на способ проектирования программ. Класс **BB_ival_slider** (§21.2.3) является примером:

```
class BB_ival_slider
   : public Ival_slider,     // интерфейс
   protected BBslider        // реализация
{
   // реализация функций Ival_slider and и BBslider, используя средства BBslider
};
```

В этом примере два базовых класса играют логически разные роли. Один базовый класс является абстрактным и играет роль интерфейса, а второй базовый класс с защищенным наследованием, является конкретным классом, предоставляющим детали реализации. Эти роли классов отражаются в стиле их написания и в контроле доступа (§20.5). Здесь использование множественного наследования является существенным, поскольку производный класс нуждается в замещении виртуальных функций как из интерфейса, так и из реализации.

Рассмотрим опять, к примеру, **Ival_box**-классы из §21.2.1. Там в самом конце (§21.2.2) я сделал все **Ival_box**-классы абстрактными, чтобы привлечь внимание к их роли чистых интерфейсов. Это позволило мне разместить детали реализации в специальных классах реализации. К тому же, совместное использование деталей реализации выполнено в классической иерархии оконной системы.

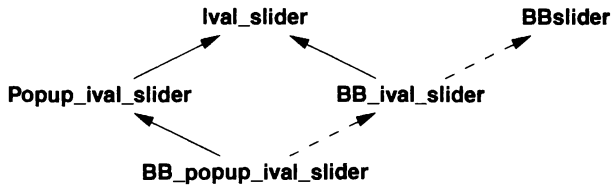
Когда мы используем абстрактный класс (без разделяемых данных) в качестве интерфейса, у нас есть выбор:

- Реплицировать интерфейсный класс.
- Сделать интерфейсный класс виртуальным, чтобы разделить его со всеми классами иерархии, которые на него ссылаются.

Применение **Ival_slider** в виде виртуального базового класса дает нам:

```
class BB_ival_slider
      : public virtual Ival_slider, protected BBslider {/* ... */};
class Popup_ival_slider
      : public virtual Ival_slider {/* ... */};
class BB_popup_ival_slider
      : public virtual Popup_ival_slider, protected BB_ival_slider {/* ... */};
```

Или графически:



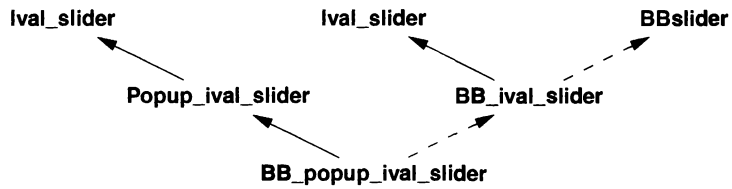
Легко представить дальнейшие интерфейсы, наследуемые от **Popup_ival_slider**, и дальнейшие классы реализации, наследуемые от таких классов и **BB_popup_ival_slider**.

У нас есть альтернатива, которая использует реплицируемые объекты **Ival_slider**:

```
class BB_ival_slider
      : public Ival_slider, protected BBslider {/* ... */};
```

```
class Popup_ival_slider
    : public Ival_slider {/* ... */};
class BB_popup_ival_slider
    : public Popup_ival_slider, protected BB_ival_slider {/* ... */};
```

Или графически:



Как это ни странно, но с точки зрения выигрыша в памяти или производительности ни у кого из рассмотренных вариантов нет выигрыша. Имеются лишь логические различия. В реплицируемом дизайне **Ival_slider** класс **BB_popup_ival_slider** не может неявным образом конвертироваться в **Ival_slider** (из-за неоднозначности):

```
void f(Ival_slider* p);
void g(BB_popup_ival_slider* p)
{
    f(p); // ошибка: Popup_ival_slider::Ival_slider или BB_ival_slider::Ival_slider?
}
```

С другой стороны, в реалистичных сценариях разделение с виртуальным базовым классом может вызвать неоднозначность при выполнении конвертации из базового класса (§22.2). Но с такими неоднозначностями легко можно справиться.

Как мы выбираем между виртуальными базовыми классами и реплицируемыми базовыми классами для наших интерфейсов? Чаще всего, у нас нет выбора, поскольку нужно согласовываться с уже существующим дизайном. Когда же реально выбор есть, мы можем принять во внимание, что решение с реплицируемыми базовыми классами имеет тенденцию получать объекты слегка меньших размеров (из-за отсутствия структур данных для поддержки общего владения) и что мы часто получаем доступ к интерфейсам из «виртуальных конструкторов» или «фабричных функций» (§21.2.4). Например:

```
Popup_ival_slider* popup_slider_factory(args)
{
    // ...
    return new BB_popup_ival_slider(args);
    // ...
}
```

Никакого явного приведения не требуется для перехода от реализации (здесь **BB_popup_ival_slider**) к его прямым интерфейсам (здесь **Popup_ival_slider**).

21.3.6.1. Замещение функций виртуального базового класса

Производный класс может заместить виртуальные функции своего прямого или непрямого виртуального базового класса. В частности, два разных класса могут замещать разные виртуальные функции виртуального базового класса. В таком случае несколько производных классов могут вносить лепту в реализацию интерфейса, представленного виртуальным базовым классом. Например, класс **Window** может иметь функции **set_color()** и **prompt()**. В таком случае **Window_with_border** может заместить **prompt()** для контроля взаимодействия с пользователем:

```
class Window {
    // ...
    virtual void set_color(Color) = 0;  // цвет фона
    virtual void prompt() = 0;
};

class Window_with_border : public virtual Window {
    // ...
    void set_color(Color) override;     // управление цветом фона
};

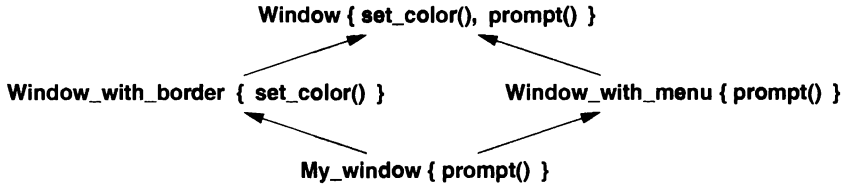
class Window_with_menu : public virtual Window {
    // ...
    void prompt() override;             // уведомление пользователей
};

class My_window : public Window_with_menu, public Window_with_border {
    // ...
};
```

А что, если разные производные классы будут замещать одну и ту же функцию? Это допускается исключительно в случае, когда замещающий класс наследуется от всех других классов, замещающих функцию. То есть, одна функция должна замещать все остальные. Например, **My_window** могло бы замещать **prompt()**, чтобы улучшить то, что предоставляет **Window_with_menu**:

```
class My_window : public Window_with_menu, public Window_with_border {
    // ...
    void prompt() override;       // не оставляем пользовательские взаимодействия базе
};
```

Или графически:



Если два класса замещают функцию базового класса, но ни один не замещает другого, то классовая иерархия ошибочна. Причина состоит в том, что никакая одиночная функция не может использоваться так, чтобы обеспечить согласованный смысл для всех вызовов независимо от того, какой класс они используют в качестве интерфейса. Или, используя терминологию реализации, нельзя создать таблицу виртуальных функций, поскольку вызов этой функции на полном объекте будет неоднозначным. Например, если бы **Radio** из §21.3.5 не объявлял **write()**, то объявления **write()** в **Receiver** и **Transmitter** были бы ошибочными в определениях **Radio**. Для **Radio** такой конфликт разрешается добавлением замещающей функции в наиболее производном классе.

Класс, предоставляющий некоторую, но не полную реализацию для виртуального базового класса, часто называют смесью.

21.4. Советы

[1] Используйте **unique_ptr** или **shared_ptr**, чтобы не забыть удалить объекты, созданные операцией **new**; §21.2.1.

[2] Избегайте членов-данных в базовых классах, предназначенных быть интерфейсами; §21.2.1.1.
[3] Используйте абстрактные классы для создания интерфейсов; §21.2.2.
[4] Снабжайте абстрактный класс виртуальным деструктором, чтобы гарантировать надлежащую очистку; §21.2.2.
[5] Для явного отражения факта замещения в больших классовых иерархиях используйте **override**; §21.2.2.
[6] Используйте абстрактные классы для поддержки наследования интерфейса; §21.2.2.
[7] Используйте базовые классы с членами-данных для поддержки наследования реализации; §21.2.2.
[8] Используйте обычное множественное наследование ради отражения факта объединения разных свойств; §21.3.
[9] Используйте множественное наследование для отделения реализации от интерфейса; §21.3.
[10] Используйте виртуальный базовый класс для отражения общности некоторых, но не всех классов иерархии; §21.3.5.

22

RTTI (идентификация типа на этапе выполнения)

Ранняя оптимизация есть корень всех зол.
— Дональд Кнут
Но мы не можем игнорировать эффективность.
— Джон Бентли

- Введение
- Навигация по классовой иерархии
  dynamic_cast; Множественное наследование; **static_cast** и **dynamic_cast**; Восстановление интерфейса
- Двойная диспетчеризация и «посетители»
  Двойная диспетчеризация; «Посетители»
- Конструирование и уничтожение объектов
- Идентификация типа
  Расширенная информация о типе
- Правильное и неправильное использование RTTI
- Советы

22.1. Введение

В самом общем случае класс конструируется из решетки производных классов. Такая решетка классов чаще всего называется *классовой иерархией*. Мы стараемся проектировать классы так, чтобы пользователи не заботились о том, как класс составлен из других классов. В частности, механизм вызова виртуальных функций гарантирует, что если мы вызываем функцию **f()** от имени объекта, то одна и та же функция вызывается независимо от того, какой класс из иерархии объявляет эту функцию и какой — определяет ее. Данная глава посвящена вопросу о том, как получить информацию о полном объекте, имея лишь интерфейс, предоставляемый базовым классом.

22.2. Навигация по классовой иерархии

Разумное применение классовой иерархии **lval_box**, определенной в §21.2, может состоять в том, чтобы предоставить ее системе, контролирующей экран, и которая возвра-

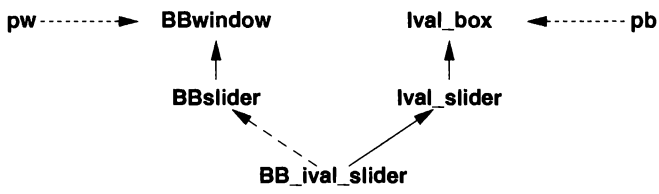
щала бы объекты прикладной программе, когда бы не произошла какая-либо экранная активность. Мы будем ссылаться на комбинацию GUI-библиотеки и средств операционной системы для управления выводом на экран просто как на систему. Объекты, которыми в обе стороны обмениваются система и прикладная программа, будем называть виджетами или элементами управления. Так работают многие системы пользовательского интерфейса. С точки зрения языка важно, что система ничего не знает о наших классах **Ival_box**. Системные интерфейсы реализованы в терминах их собственных классов и объектов, а не классов нашей программы. Тут есть один неприятный момент, который заключается в том, что мы теряем информацию о типе передаваемых и принимаемых объектов.

Восстановление потерянного типа объекта требует от нас возможности как-то просить объект о том, чтобы он раскрыл свой тип. Ведь любая операция над объектом требует, чтобы был объявлен указатель или ссылка типа, подходящего для объекта. Поэтому самой полезной операцией будет операция преобразования типа, которая вернет действительный указатель на объект, если объект относится к ожидаемому типу, или в противном случае вернет нулевой указатель. Это выполняет операция **dynamic_cast**. К примеру, допустим, что система вызывает **my_event_handler()** с указателем на **BBwindow**, для которого произошло событие активности. Я бы мог в ответ активизировать код моего приложения, используя **do_something()**:

```
void my_event_handler(BBwindow* pw)
{
    if (auto pb = dynamic_cast<Ival_box*>(pw)) {      // pw указывает на Ival_box?
        // ...
        int x = pb->get_value();                      // используем Ival_box
        // ...
    }
    else {
        // ... упс! Случилось неожиданное событие ...
    }
}
```

Можно дать следующее пояснение того, что здесь делает **dynamic_cast** — эта операция согласовывает (можно сказать «переводит») понятия пользовательского интерфейса системы на язык нашего приложения. Заметьте, что в этом примере нигде не упомянут истинный тип объекта. Объект имеет конкретный тип из иерархии **Ival_box**, скажем **Ival_slider**, реализованный конкретным вариантом **BBwindow**, скажем **BBslider**. В показанном примере взаимодействия системы и нашего приложения нет никакой необходимотси ракрывать истинный тип объекта. Для этого и существует интерфейс, отражающий существенные стороны взаимодействия. В частности, хорошо спроектированный интерфейс скрывает важные детали.

Графически, действие **pb=dynamic_cast<Ival_box*>(pw)** можно изобразить следующим образом:



Стрелки от **pw** и **pb** символизируют указатели на переденные объекты, в то время как остальные стрелки отражают связи наследования между разными частями объектов.

Использование информации о типе на этапе выполнения часто называют *RTTI* (Run-Time Type Information).

Приведение от базового класса к производному классу отражается в термине *downcast* (приведение вниз), поскольку традиционно дерево наследования изображают так, что корень (базовый класс) располагается сверху. Приведение в обратную сторону называют *upcast* (приведение вверх).

22.2.1. dynamic_cast

Операция **dynamic_cast** принимает два операнда: тип в угловых скобках и указатель (или ссылка) в круглых скобках. Рассмотрим следующее:

```
dynamic_cast<T*>(p)
```

Если **p** имеет тип **T*** или имеет тип **D***, где **T** является базовым классом для для **D**, то результат в точности тот же, что и в случае простого приведения **p** к **T***. Например:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
    // ...
};

void f(BB_ival_slider* p)
{
    Ival_slider* pi1 = p;                                  // OK
    Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);      // OK

    BBslider* pbb1 = p;                                    // ошибка: BBslider защищенная база
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);           // OK: pbb2 становится nullptr
}
```

Этот случай (upcast) не представляет интереса. Тем не менее, нужно знать, что **dynamic_cast** не дает случайного нарушения защиты для закрытых и защищенных базовых классов.

Основной случай применения операции **dynamic_cast** имеет место тогда, когда компилятор не может гарантировать корректность конвертации типа. В таком случае **dynamic_cast<T*>(p)** изучает объект, на который показывает **p**. Если этот объект класса **T**, тогда **dynamic_cast** возвращает значение указателя типа **T*** на этот объект; в противном случае возвращается **nullptr**. Отметим требование, чтобы конвертирование должно выполняться только для однозначно идентифицируемого объекта. Можно сконструировать примеры, когда приведение терпит неудачу и возвращается **nullptr**, поскольку объект, указуемый через **p**, имеет более одного подобъекта, представляющего базовые классы типа **T** (§22.2).

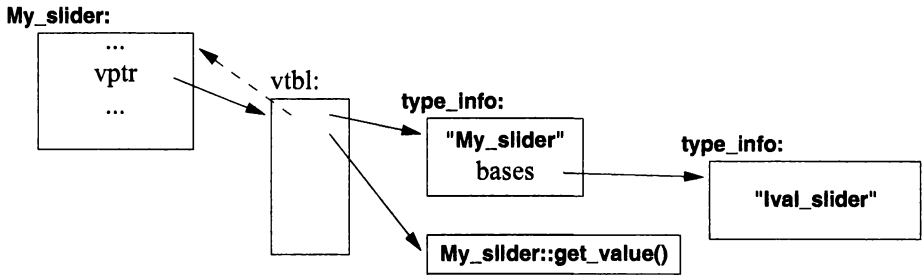
Операция **dynamic_cast** требует указатель или сслыку на полиморфный тип, чтобы можно было выполнить downcast или upcast. Например:

```
class My_slider: public Ival_slider { // полиморфная база (Ival_slider имеет виртуаль. функции)
    // ...
};

class My_date : public Date { // база не полиморфная (Date не имеет виртуальных функций)
    // ...
};

void g(Ival_box* pb, Date* pd)
{
    My_slider* pd1 = dynamic_cast<My_slider*>(pb); // OK
    My_date* pd2 = dynamic_cast<My_date*>(pd);     // ошибка: Date не полиморфная
}
```

Требование к типу указателя быть полиморфным упрощает реализацию операции **dynamic_cast**, поскольку при этом легко можно найти место для хранения информации о типе объекта. В типичном случае реализация присовокупит объект с информацией о типе (§22.5) к целевому объекту, поместив указатель на типовую информацию в таблицу виртуальных функций класса объекта (§3.2.3). Например:



Пунктирные линии отражают смещение, позволяющее найти начало полного объекта, отталкиваясь лишь от указателя на полиморфный подобъект. Понятно, что операцию **dynamic_cast** можно реализовать весьма эффективно: добавляется несколько сравнений объектов **type_info**, представляющих базовые классы.

Ограничение операции **dynamic_cast** полиморфными типами имеет и логическое обоснование. Действительно, если объект не имеет виртуальных функций, то им нельзя безопасно манипулировать без знания его точного типа. А если его тип точно известен, то тогда нам не нужна операция **dynamic_cast**.

Целевой тип операции **dynamic_cast** может и не быть полиморфным. Это позволяет обернуть конкретный тип в полиморфный, например при передаче с помощью объектной системы ввода/вывода (§22.2.4), а потом развернуть конкретный тип обратно. Например:

```
class Io_obj {        // базовый класс для объектной системы ввода/вывода
    virtual Io_obj* clone() = 0;
};

class Io_date : public Date, public Io_obj { };

void f(Io_obj* pio)
{
    Date* pd = dynamic_cast<Date*>(pio);
    // ...
}
```

Применение операции **dynamic_cast** для типа **void*** может использоваться для того, чтобы определить начало объекта полиморфного типа. Например:

```
void g(Ival_box* pb, Date* pd)
{
    void* pb2 = dynamic_cast<void*>(pb);   // OK
    void* pd2 = dynamic_cast<void*>(pd);   // ошибка: Date не полиморфная
}
```

Объект, представляющий базовый класс (такой как **Ival_box**), не обязательно идет первым подобъектом в объекте наиболее производного класса. Так что **pb** не обязательно содержит тот же адрес, что и **pb2**.

Такие приведения типов полезны лишь в самом низкоуровневом коде (ибо только они имеют дело с типом **void***). Операция **dynamic_cast** для **void*** неразрешима, ибо нет способа узнать, где следует искать **vptr**; §22.2.3).

22.2.1.1. Операция dynamic_cast для ссылок

Полиморфное поведение объекта реализуется тогда, когда им манипулируют посредством указателя или ссылки. Когда **dynamic_cast** применяется к указателю, в случае неудачи возвращается **nullptr**. Для ссылок это недостижимо и нежелательно.

Для указателей **nullptr** означает, что указатель не указывает ни на какой объект. Таким образом, результат операции **dynamic_cast** для указателей всегда может быть проверен явным образом. Для указателя **p** операцию **dynamic_cast<T*>(p)** можно представлять себе вопросом «Является ли объект, указуемый через **p**, если он имеется, объектом типа **T**?». Например:

```
void fp(Ival_box* p)
{
    if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {    // p указывает на Ival_slider?
        // ... используем слайдер ...
    }
    else {
        // ... *p не slider; работаем с альтернативами ...
    }
}
```

С другой стороны, мы всегда можем рассчитывать, что ссылка настроена на объект (§7.7.4). Поэтому операция **dynamic_cast<T&>(r)** для ссылки **r** является не вопросом, а утверждением: «Объект по ссылке **r** имеет тип **T**». В случае, когда объект по ссылке не имеет ожидаемого типа, генерируется исключение **bad_cast**. Пример:

```
void fr(Ival_box& r)
{
    Ival_slider& is = dynamic_cast<Ival_slider&>(r);    // r ссылка на Ival_slider!
    // ... используем слайдер ...
}
```

Разница в результатах для **dynamic_cast** отражает фундаментальное различие между указателями и ссылками. Если пользователю не хочется иметь дело с **bad_cast**, то можно предоставить подходящий обработчик. Пример:

```
void g(BB_ival_slider& slider, BB_ival_dial& dial)
{
    try {
        fp(&slider);    // указатель на BB_ival_slider переданный как Ival_box*
        fr(slider);     // ссылка на BB_ival_slider переданная как Ival_box&
        fp(&dial);      // указатель на BB_ival_dial переданный как Ival_box*
        fr(dial);       // dial переданный как Ival_box
    }
    catch (bad_cast) { // §30.4.1.1
        // ...
    }
}
```

Вызовы **fp()** и первый вызов **fr()** имеют нормальный возврат (в предположении, что **fp()** действительно может управиться с **BB_ival_dial**), но второй вызов **fr()** сгенерирует исключение **bad_cast**, которое будет перехвачено посредством **g()**.

Явные проверки на **nullptr** могут быть забыты. Если это вас беспокоит, можно написать конвертирующую функцию, которая сгенерирует исключение вместо возврата **nullptr**.

22.2.2. Множественное наследование

Когда используется лишь одиночное наследование, класс и его базовые классы образуют древовидную структуру с единственным началом в одном базовом классе. Когда используется множественное наследование, одного корня (вершины) нет. Само по себе это не вызывает никаких дополнительных трудностей. Другое дело, если класс появляется в иерархии более одного раза. В этом случае нужно проявлять осторожность, ссылаясь на объект, представляющий такой класс.

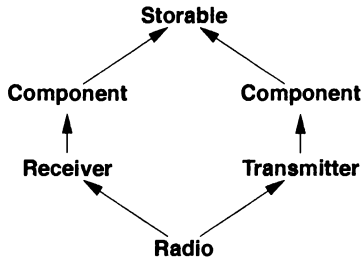
Естественно, лучше строить иерархии по-возможности более простыми. Однако, если мы имеем дело с нетривиальной иерархией, то все равно нам может потребоваться выполнить навигацию по такой иерархии, чтобы отыскать какой-либо специфический класс. Это бывает нужным в двух случаях:

- Требуется явным образом поименовать базовый класс для того, чтобы использовать его в качестве интерфейса, чтобы например разрешить неоднозначность или вызвать какую-то функцию, не полагаясь на механизм виртуальных функций (явно квалифицированный вызов; §21.3.3).
- Нужно получить указатель на один объект из иерархии на основе указателя на другой объект; например, на полный объект производного класса, отталкиваясь от указателя на базовый класс (downcast; §22.2.1), или получить указатель на объект базового класса исходя из указателя на другой базовый класс (§22.2.4).

Здесь мы рассмотрим вопрос о том, как вести навигацию по классовой иерархии, используя приведения типов ради получения указателя желаемого типа. Для иллюстрации имеющихся механизмов, рассмотрим решетку наследования, содержащую реплицируемый базовый класс и виртуальный базовый класс:

```
class Component
    : public virtual Storable { /* ... */ };
class Receiver
    : public Component { /* ... */ };
class Transmitter
    : public Component { /* ... */ };
class Radio
    : public Receiver, public Transmitter { /* ... */ };
```

Или графически:



Здесь объект **Radio** имеет два подобъекта класса **Component**. Следовательно, операция **dynamic_cast** от **Storable** к **Component** в пределах **Radio** будет неоднозначной и вернет **0**. Просто нет способа узнать, какой **Component** затребовал программист:

```
void h1(Radio& r)
{
    Storable* ps = &r;  // у Radio уникальный Storable
    // ...
```

```
    Component* pc = dynamic_cast<Component*>(ps); // pc = 0; у Radio – два Components
    // ...
}
```

В общем случае (и это типично) программист не знает полную решетку иерархии классов. Вместо этого код пишется на основе знания некоторой подрешетки. Например, программист мог бы иметь знание о **Transmitter**-части **Radio** и написать:

```
void h2(Storable* ps)    // ps может указывать (или нет) на Component
{
    if (Component* pc = dynamic_cast<Component*>(ps)) {
        // есть компонент!
    }
    else {
        // это был не Component
    }
}
```

Неоднозначность для указателя на **Radio**-объект не может в общем случае быть выявлена во время компиляции.

Необходимость в выявлении такой неоднозначности на этапе выполнения требуется только для виртуальных базовых классов. Для обычных базовых классов всегда имеется уникальный подобъект для приведения вниз (downcast; §22.2). Эквивалентная неоднозначность для виртуальных базовых классов возникает в случае приведения вверх (upcast), но эти неоднозначности ловятся на этапе компиляции.

22.2.3. static_cast и dynamic_cast

Операция **dynamic_cast** может выполнять приведение от полиморфного виртуального базового класса к производному классу или же к сестринскому классу (§22.2.1). А операция **static_cast** (§11.5.2) не анализирует объект и не может этого сделать:

```
void g(Radio& r)
{
    Receiver* prec = &r;                    // Receiver – обычная база для Radio
    Radio* pr = static_cast<Radio*>(prec);  // OK, без проверки
    pr = dynamic_cast<Radio*>(prec);        // OK, проверка run-time (во время выполнения)

    Storable* ps = &r;                      // Storable – виртуальная база для Radio
    pr = static_cast<Radio*>(ps);           // ошибка: нет конверсии для  virtual base
    pr = dynamic_cast<Radio*>(ps);          // OK, проверка run-time
}
```

Операция **dynamic_cast** требует полиморфного операнда, поскольку в неполиморфном объекте не хранится информация, которую можно использовать для розыска объектов, представляющих базовый класс. В частности, объект с ограничениями в раскладке, налагаемыми другими языками — языками Fortran или C — может использоваться в качестве виртуального базового класса. Для таких объектов будет доступна лишь статическая информация о типе.

Почему может возникнуть необходимость применять **static_cast** для навигации по классовым иерархиям? Во-первых, динамическое определение типа влечет некоторые потери в производительности (§22.2.1). Во-вторых, и это важнее, существуют миллионы строк кода, написанные до того, как операция **dynamic_cast** стала доступной. И в этом коде уже предприняты альтернативные способы узнать, что приведение типа будет успешным, так что для этого кода операция **dynamic_cast** избыточна. Такой код чаще всего был напи-

сан на языке C с применением приведений типа в C-стиле (§11.5.3), часто вызывающем туманные ошибки. Всюду, где можно, применяйте **dynamic_cast**.

Компилятор не может ничего предположить о памяти, адресуемой указателем типа **void***. Поэтому операция **dynamic_cast** не может работать с такими указателями, и поэтому придется применять **static_cast**. Пример:

```
Radio* f1(void* p)
{
    Storable* ps = static_cast<Storable*>(p);      // доверяем программисту
    return dynamic_cast<Radio*>(ps);
}
```

Обе операции, **dynamic_cast** и **static_cast**, учитывают ключевое слово **const** и контроль доступа. Например:

```
class Users : private set<Person> { /* ... */ };

void f2(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*>(pu);          // ошибка: нарушение доступа
    dynamic_cast<set<Person>*>(pu);         // ошибка: нарушение доступа

    static_cast<Receiver*>(pcr);            // ошибка: нет конверсии от const
    dynamic_cast<Receiver*>(pcr);           // ошибка: нет конверсии от const

    Receiver* pr = const_cast<Receiver*>(pcr);  // OK
    // ...
}
```

Невозможно выполнить приведение к закрытому базовому классу, используя **static_cast** или **reinterpret_cast**, а чтобы убрать **const** (или **volatile**) требуется **const_cast** (§11.5.2). Но даже тогда результат надежен только если объект изначально не был объявлен с **const** (или **volatile**) (§16.2.9).

22.2.4. Восстановление интерфейса

С точки зрения проектирования, операция **dynamic_cast** (§22.2.1) служит механизмом, позволяющим спросить объект, предоставляет ли он данный интерфейс.

Для примера рассмотрим простую объектную систему ввода/вывода. Пользователи хотят читать объекты из потока, выяснять их соответствие ожидаемому типу, после чего использовать их. Пример:

```
void user()
{
    // ... открываем файл с формами, и прикрепляем ss как istream для него ...

    unique_ptr<Io_obj> p {get_obj(ss)};     // читаем объект из потока

    if (auto sp = dynamic_cast<Shape*>(p.get())) {
        sp->draw();                         // используем Shape
        // ...
    }
    else {
        // упс: в файле не Shape
    }
}
```

Функция **user()** работает с фигурами (shapes) исключительно через абстрактный интерфейс **Shape** и поэтому может использовать любой вид фигур. Применение **dynamic_cast** важно потому, что система ввода/вывода работает и со многими другими видами объектов и она может открыть файл с объектами, о которых пользователь даже не догадывается.

Я применил **unique_ptr<Io_obj>** (§5.2.1, §34.3.1), чтобы не забыть удалить объект, аллоцированный функцией **get_obj()**.

Данная объектная система ввода/вывода предполагает, что любой объект, прочитанный или записанный, имеет тип класса, производного от **Io_obj**. Класс **Io_obj** должен быть полиморфного типа, чтобы пользователь функции **get_obj()** мог использовать **dynamic_cast** для восстановления «истинного типа» возвращаемого объекта. Например:

```
class Io_obj {
public:
    virtual Io_obj* clone() const =0;    // полиморфная
    virtual ~Io_obj() {}
};
```

Критической функцией в данной объектной системе ввода/вывода является функция **get_obj()**, которая читает данные из потока **istream** и создает объекты класса, базируясь на этих данных. Предположим, что данные, представляющие объект в потоке, предваряются строкой, идентифицирующей класс объекта. Задача **get_obj()** состоит в том, чтобы прочесть эту строку и вызвать функцию, которая может прочитать данные и создать объект правильного класса. Например:

```
using Pf = Io_obj*(istream&);      // указатель на ф-ю с возвратом Io_obj*

map<string, Pf> io_map;            // отображает строки на создающие функции

string get_word(istream& is);      // читаем слово из is; генерируем Read_error при неудаче

Io_obj* get_obj(istream& is)
{
    string str = get_word(is);     // читаем начальное слово
    if (auto f = io_map[str])      // ищем ф-ю для str
        return f(is);              // вызываем функцию
    throw Unknown_class{};         // нет соответствия для str
}
```

Тип **map** используется для хранения пар имен и функций, которые могут конструировать объекты класса с этим именем.

Мы могли бы наследовать класс **Shape** от **Io_obj**, как этого требует **user()**:

```
class Shape : public Io_obj {
    // ...
};
```

Но интереснее использовать класс **Shape** (§3.2.4) в неизмененном виде:

```
struct Io_circle :  Circle, Io_obj {
    Io_circle(istream&);                                  // инициализируем из входного потока
    Io_circle* clone() const { return new Io_circle{*this}; }     // копир. конструк-р
    static Io_obj* new_circle(istream& is) { return new Io_circle{is}; }    // для io_map
};
```

Это пример того, как класс может согласовываться с иерархией с использованием абстрактного класса с меньшим заглядыванием в будущее, чем это требовалось бы в случае построения узлового класса на первом месте (§21.2.2).

Конструктор **Io_circle(istream&)** инициализирует объект данными из своего аргумента. Функция **new_circle()** была помещена в **io_map**, чтобы класс стал известен объектной системе ввода/вывода. Например:

```
io_map["Io_circle"]=&Io_circle::new_circle;    // где либо
```

Другие фигуры конструируются аналогично:

```
class Io_triangle : public Triangle, public Io_obj {
    // ...
};

io_map["Io_triangle"]=&Io_circle::new_triangle;    // где либо
```

Если такая «возня» становится утомительной, то можно определить шаблон:

```
template<class T>
struct Io :  T, Io_obj {
public:
    Io(istream&);                                  // инициализируем из входного потока
    Io* clone() const override { return new Io{*this}; }
    static Io* new_io(istream& is) { return new Io{is}; }    // для io_map
};
```

Отталкиваясь от этого, определим **Io_circle**:

```
using Io_circle = Io<Circle>;
```

Мы по-прежнему должны определить **Io<Circle>::Io(istream&)** явным образом, поскольку он должен знать **Circle** в деталях. Заметьте, что **Io<Circle>::Io(istream&)** не имеет доступа к закрытым и защищенным данным у типа **T**. Идея состоит в том, что формат преобразования для типа **X** есть то, что требуется для конструирования **X** с помощью одного из конструкторов **X**. Информация из потока не обязательно является последовательностью значений членов типа **X**.

Шаблон **Io** служит примером того, как согласовать конкретный тип с классовой иерархией путем предоставления дескриптора, являющегося узлом в этой иерархии. Он наследует от своего параметра шаблона, что допускает приведение типа из **Io_obj**. Пример:

```
void f(io<Shape>& ios)
{
    Shape* ps = &ios;
    // ...
}
```

К сожалению, наследование от аргумента шаблона исключает возможность использования **Io** для встроенных типов:

```
using Io_date = Io<Date>;    // обертка конкретного типа
using Io_int = Io<int>;      // ошибка: нельзя наследовать от встроенного типа
```

С этой проблемой можно справиться, делая объект пользователя членом **Io_obj**:

```
template<class T>
struct Io: Io_obj {
    T val;

    Io(istream&);                                  // инициализируем из входного потока
    Io* clone() const override { return new Io{*this}; }
    static Io* new_io(istream& is) { return new Io{is}; }    // для io_map
};
```

Теперь у нас есть решение:

```
using Io_int = Io<int>;                    // обертка встроенного типа
```

Сейчас, когда значение стало членом, а не базовым классом, мы больше не можем напрямую конвертировать **Io_obj<X>** в **X**, так что мы создаем функцию для этого:

```
template<typename T>
T* get_val<T>(Io_obj* p)
{
    if (auto pp = dynamic_cast<Io<T>*>(p))
        return &pp->val;
    return nullptr;
}
```

Теперь функция **user()** принимает вид:

```
void user()
{
    // ... открываем файл с формами, и прикрепляем ss как istream для него ...

    unique_ptr<Io_obj> p {get_obj(ss)};          // читаем объект из потока

    if (auto sp = get_val<Shape>(p.get())) {
        sp->draw();                              // используем Shape
        // ...
    }
    else {
        // ... упс: в файле не Shape ...
    }
}
```

Полученная простая объектная система ввода/вывода не есть нечто, что удовлетворило бы абсолютно всех, но она умещается на одной странице, а ее ключевые механизмы находят много вариантов использования. Она является черновиком системы принимающего узла в коммуникационном канале, по которому в безопасной манере передаются произвольные объекты. Еще важно, что наша система предоставляет возможность вызова функции на основе предоставленной пользователем строки, содержащей ее имя, и возможность манипулирования объектами неизвестного типа через интерфейсы, распознаваемые на этапе выполнения с помощью механизма RTTI.

В общем случае, передающий конец канала с объектной системой ввода/вывода также будет использовать RTTI. Рассмотрим следующее:

```
class Face : public Shape {
public:
    Shape* outline;
    array<Shape*> eyes;
    Shape* mouth;

    // ...
};
```

Чтобы корректно писать в канал объект типа **Shape**, указуемый посредством **outline**, мы должны вычислить, к какому именно виду **Shape** относится объект. А это работа для **typeid()** (§22.5). В общем случае нам нужно хранить таблицу пар (указатель, уникальный идентификатор), чтобы иметь возможность передавать связанные структуры данных и избегать дублирования объектов, на которые «смотрят» более одного указателя (или ссылки).

22.3. Двойная диспетчеризация и «посетители»

Классическое объектно-ориентированное программирование базируется на выборе виртуальной функции на основе динамического типа (тип наиболее производного класса) объекта, располагая лишь указателем или ссылкой на интерфейс (базовый класс). В частности, язык C++ может выполнять такой поиск (часто называемый *динамической диспетчеризацией*) для одного типа за раз. Невозможность выбора функции на основе двух динамических типов ограничивает возможности. Кроме того, виртуальная функция должна быть функцией-членом. Это означает, что мы не можем добавить виртуальную функцию к классовой иерархии без модификации базовых классов, предоставляющих интерфейс, а также всех производных классов, на которых эта модификация скажется. Это может стать серьезной проблемой. В настоящем разделе мы рассмотрим пути решения этих проблем:

§22.3.1 *Двойная диспетчеризация* — рассказывает, как выбрать виртуальную функцию на базе двух типов.

§22.3.2 *Посетители* — рассказывает о том, как использовать двойную диспетчеризацию для добавления множества функций к классовой иерархии с единственной дополнительной для иерархии виртуальной функцией.

Более реалистичные примеры этой техники встречаются в случае, когда мы имеем дело со структурами данных, такими как вектора или графы, или указатели на объекты полиморфных типов. В таких случаях истинный тип объекта (элемент вектора или узел графа) может определяться лишь динамически посредством инспекции интерфейса, предоставляемого базовым классом.

22.3.1. Двойная диспетчеризация

Рассмотрим вопрос о том, как можно выделить функцию, базируясь на двух аргументах. Например:

```
void do_someting(Shape& s1, Shape& s2)
{
    if (s1.intersect(s2)) {
        // две формы перекрываются
    }
    // ...
}
```

Мы бы хотели, чтобы это работало для любых двух классов классовой иерархии с вершиной (корнем) **Shape**, такими как **Circle** и **Triangle**.

Идея заключается в том, чтобы выполнить вызов виртуальной функции для выбора правильной функции для **s1**, а затем выполнить второй вызов для выбора правильной функции для **s2**. Упрощая, я опустил выяснение того, действительно ли две фигуры имеют пересечение и написал лишь скелет кода. Сначала мы определяем **Shape** с функцией пересечения:

```
class Circle;
class Triangle;

class Shape {
public:
    virtual bool intersect(const Shape&) const =0;
    virtual bool intersect(const Circle&) const =0;
    virtual bool intersect(const Triangle&) const =0;
};
```

Затем нам нужно определить **Circle** и **Triangle**, чтобы заместить эти виртуальные функции:

```
class Circle : public Shape {
public:
    bool intersect(const Shape&) const override;
    virtual bool intersect(const Circle&) const override;
    virtual bool intersect(const Triangle&) const override
};

class Triangle : public Shape {
public:
    bool intersect(const Shape&) const override;
    virtual bool intersect(const Circle&) const override;
    virtual bool intersect(const Triangle&) const override;
};
```

Теперь каждый класс может обращаться с любыми классами из **Shape**-иерархии, так что нам нужно решить, что делать с каждой комбинацией:

```
bool Circle::intersect(const Shape& s) const { return s.intersect(*this); }
bool Circle::intersect(const Circle&) const { cout <<"intersect(circle, circle)\n"; return true; }
bool Circle::intersect(const Triangle&) const { cout <<"intersect(circle, triangle)\n"; return true; }

bool Triangle::intersect(const Shape& s) const { return s.intersect(*this); }
bool Triangle::intersect(const Circle&) const { cout <<"intersect(triangle, circle)\n"; return true; }
bool Triangle::intersect(const Triangle&) const { cout <<"intersect(triangle, triangle)\n"; return true; }
```

Интересными функциями здесь являются **Circle::intersect(const Shape&)** и **Triangle::intersect(const Shape&)**. Они нужны для работы с аргументом **Shape&**, поскольку этот аргумент должен ссылаться на производный класс. Трюк заключается в том, чтобы просто делать виртуальный вызов с аргументами в обратном порядке. Сделав это, мы попадем в одну из четырех функций, которые на самом деле могут выполнить вычисления с пересечением.

Можно протестировать наше решение, создав вектор пар значений **Shape*** и вызвав для них **intersect()**:

```
void test(Triangle& t, Circle& c)
{
    vector<pair<Shape*, Shape*>> vs { {&t,&t}, {&t,&c}, {&c,&t}, {&c,&c} };
    for (auto p : vs)
        p.first->intersect(*p.second);
}
```

Используя **Shape***, мы полагаемся на разрешение типов во время выполнения. Мы получаем:

```
intersect(triangle, triangle)
intersect(triangle, circle)
intersect(circle, triangle)
intersect(circle, circle)
```

Если вы сочтете это элегантным, то вам нужно повышать ваши стандарты, но работа все же выполняется. По мере роста классовой иерархии нужда в виртуальных функциях растет экспоненциально. Во многих случаях это недопустимо. Расширение числа аргументов до трех или более тривиально, но утомительно. Но хуже всего то, что каждая новая операция и каждый новый производный класс требует модификации каждого клас-

са в иерархии: эта техника двойной диспетчеризации чрезвычайно интрузивна. В идеале я бы предпочел простую функцию **intersect(Dhape&, Shape&)** с перегруженными вариантами, специфичными для желаемых комбинаций конкретных фигур. Такое возможно [Pirkelbauer, 2009], но не в C++11.

Некоторая неэлегантность двойной диспетчеризации не делает проблему, которую пытаются таким образом решить, менее важной. Часто нужно действие, такое как **intersect(x, y)**, которое засисит от типов двух (и более) операндов. Но есть и обходные пути. Например, пересечение прямоугольников — это простой и эффективный случай. А для многих приложений можно счесть достаточным работать с описанными прямоугольниками для каждой фигуры, а затем вычислять пересечения для этих прямоугольников. Например:

```
class Shape {
public:
    virtual Rectangle box() const = 0;      // прямоугольник охватывает shape
    // ...
};

class Circle : public Shape {
public:
    Rectangle box() const override;
    // ...
};

class Triangle : public Shape {
public:
    Rectangle box() const override;
    // ...
};

bool intersect(const Rectangle&, const Rectangle&);   // простое вычисление

bool intersect(const Shape& s1, const Shape& s2)
{
    return intersect(s1.box(), s2.box());
}
```

Другой прием заключается в предварительно вычисляемой таблице комбинаций типов [Stroustrup,1994]:

```
bool intersect(const Shape& s1, const Shape& s2)
{
    auto i = index(type_id(s1), type_id(s2));
    return intersect_tbl[i](s1, s2);
}
```

Вариации этой идеи широко используются на практике.

22.3.2. «Посетители»

Паттерн «посетителей» [Gamma,1994] является частичным решением проблемы экспоненциального роста виртуальных функций и замещающих их функций, а также неприятной интрузивности (слишком) простой техники двойной диспетчеризации.

Рассмотрим вопрос о том, как применить две (или более) операции к каждому классу из классовой иерархии. В самой основе, мы проделаем двойную диспетчеризацию для иерархии узлов и иерархии операций, чтобы выбирать правильную операцию для кор-

ректного узла. Операции называют *посетителями*; здесь они определены в классах, производных от класса **Visitor**. Узлы образуют иерархию классов с виртуальными функциями **accept()**, принимающими аргументы типа **Visitor&**. Для этого примера я использую иерархию из **Node**, описывающую конструкции языка, что типично для инструментов, базирующихся на абстрактных синтаксических деревьях:

```
class Visitor;

class Node {
public:
    virtual void accept(Visitor&) = 0;
};

class Expr : public Node {
public:
    void accept(Visitor&) override;
};

class Stmt : public Node {
public:
    void accept(Visitor&) override;
};
```

Пока что все хорошо: **Node**-иерархия просто предоставляет виртуальную функцию **accept()**, которая принимает аргумент типа **Visitor&**, представляющий собой то, что должно быть сделано с **Node** заданного типа.

Я здесь не применяю **const**, поскольку в общем случае операция с **Visitor** может изменять либо посещаемый **Node**, либо самого **Visitor**.

Операция **accept()** класса **Node** выполняет трюк с двойной диспетчеризацией и передает сам **Node** функции **accept()** класса **Visitor**:

```
void Expr::accept(Visitor& v) { v.accept(*this); }
void Stmt::accept(Visitor& v) { v.accept(*this); }
```

Класс **Visitor** объявляет свой набор операций:

```
class Visitor {
public:
    virtual void accept(Expr&) = 0;
    virtual void accept(Stmt&) = 0;
};
```

После этого мы можем определять операции, выполняя наследование от **Visitor** и замещая его функции **accept()**. Например:

```
class Do1_visitor : public Visitor {
    void accept(Expr&) { cout << "do1 to Expr\n"; }
    void accept(Stmt&) { cout << "do1 to Stmt\n"; }
};

class Do2_visitor : public Visitor {
    void accept(Expr&) { cout << "do2 to Expr\n"; }
    void accept(Stmt&) { cout << "do2 to Stmt\n"; }
};
```

Выполним тестирование, создав вектор пар указателей, чтобы гарантировать использование RTTI:

```
void test(Expr& e, Stmt& s)
{
    vector<pair<Node*, Visitor*>> vn {&e,&do1}, {&s,&do1}, {&e,&do2}, {&s,&do2}};
    for (auto p : vn)
        p.first->accept(*p.second);
}
```

На выходе получаем:

```
do1 to Expr
do1 to Stmt
do2 to Expr
do2 to Stmt
```

В противоположность простой двойной диспетчеризации паттерн «посетитель» широко применяется в реальном программировании. Он умеренно интрузивный (функция **accept()**) и существует во многих вариациях основной идеи. Но многие операции над иерархиями трудно выразить согласно паттерну «посетителей». Например, если операция требует доступа ко многим узлам графа с разными типами, то это трудно перевести на «язык посетителей». Как следствие, я считаю паттерн «посетителей» неэлегантным обходным путем решения проблемы. Есть и альтернативы [Solodkyy,2012], но не в C++11.

Большинство альтернатив «посетителям» в C++ базируются на идее явной итерации по однородным структурам данных (вектор или граф узлов, содержащих указатели на полиморфные типы). Для каждого элемента или узла вызов виртуальной функции может выполнить желаемую операцию, и можно выполнить оптимизацию на основе хранимой информации (см. например §27.4.2).

22.4. Конструирование и уничтожение объектов

Объект класса — это больше, чем просто область памяти (§6.4). Объект класса строится конструкторами на базе «сырой памяти», и возврат к сырой памяти происходит после исполнения деструктора. Конструирование выполняется снизу-вверх, а деструкция — сверху-вниз. Классовый объект до той степени объект, до которой он был сконструирован или уничтожен. Указанный порядок важен, чтобы можно было гарантировать, что к объекту нет доступа до того, как он проинициализирован. Неразумно выполнять доступ к базовой части объекта и членам на более ранних стадиях, пользуясь хитроумными манипуляциями с указателями (§17.2.3). Порядок конструирования и деструкции отражается в правилах RTTI, обработки исключений (§13.3) и виртуальных функций (§20.3.2).

Не стоит опираться на порядок конструирования и деструкции, но наблюдать за ним можно вызовами виртуальных функций, операций **dynamic_cast** (§22.2) и **typeid** (§22.5) в точках кода, где объект еще не полностью завершен. В подобной точке в конструкторе динамический тип объекта отражает лишь то, что к этому моменту построено. Например, если конструктор у **Component** в иерархии из §22.2.2 вызовет виртуальную функцию, будет вызвана версия для **Storable** или **Component**, но не версия для **Receiver**, **Transmitter** или **Radio**. В такой точке кода объект еще не **Radio**. Аналогично, вызов виртуальной функции в деструкторе отражает то, что еще не было уничтожено. В общем, лучше избегать вызова виртуальных функций из конструкторов и деструкторов.

22.5. Идентификация типа

Операция **dynamic_cast** в основном обеспечивает потребность в информации о типе объекта на этапе выполнения. Важно также, что гарантируется, что код с использованием

этого механизма работает корректно с классами, производными от классов, явно упомянутых программистом. Таким образом, **dynamic_cast** обладает гибкостью и расширяемостью на манер виртуальных функций.

Но иногда бывает нужным знать точный тип объекта. Например, нам может потребоваться узнать имя класса объекта. Этой цели служит операция **typeid**, которая в качестве результата возвращает объект, содержащий информацию об объекте-операнде. Если бы **typeid** был функцией, то ее объявление было бы чем-то вроде следующего:

```
class type_info;
const type_info& typeid(expression);          // псевдообъявление
```

Таким образом, **typeid** возвращает ссылку на тип из стандартной библиотеки, названный **type_info** и определяемый в файле **<typeinfo>**:

- Если операнд будет типом, то **typeid(type_name)** вернет ссылку на **type_info**, представляющий тип **type_name**; **type_name** должен быть полностью определенным типом (§8.2.2).
- Если операнд будет *выражением*, то **typeid(expr)** вернет ссылку на **type_info**, представляющий тип объекта, обозначенного с помощью **expr**; **expr** должен ссылаться на полностью определенный тип (§8.2.2). Если у **expr** значение **nullptr**, то **typeid(expr)** сгенерирует исключение **std::bad_typeid**.

Операция **typeid()** определяет тип объекта, на который ссылается указатель или ссылка:

```
void f(Shape& r, Shape* p)
{
    typeid(r);      // тип объекта, на который ссылается r
    typeid(*p);     // тип объекта, указуемый посредством p
    typeid(p);      // тип указателя, то есть, Shape
}
```

Если операнд операции **typeid()** является указателем или ссылкой на полиморфный тип со значением **nullptr**, то **typeid()** генерирует исключение **std::bad_typeid**. А если операнд операции **typeid()** не является полиморфным типом или не lvalue, результат определяется на этапе компиляции без необходимости вычисления операнда-выражения.

Если объект обозначен разыменованным указателем или ссылкой на полиморфный тип, то возвращенный **type_info** относится к самому производному классу объекта, то есть к типу, использованному при определении объекта. Например:

```
struct Poly {                   // полиморфный базовый класс
    virtual void f();
    // ...
};

struct Non_poly { /* ... */ };  // нет виртуальных функций

struct D1
    : Poly { /* ... */ };
struct D2
    : Non_poly { /* ... */ };

void f(Non_poly& npr, Poly& pr)
{
    cout << typeid(npr).name() << '\n';     // пишет что-то вроде "Non_poly"
    cout << typeid(pr).name() << '\n';      // имя Poly или класса, производного от Poly
}
```

```
void g()
{
    D1 d1;
    D2 d2;
    f(d2, d1); // пишет "Non_poly D1"
    f(*static_cast<Poly*>(nullptr),*static_cast<Null_poly*>(nullptr));    // упс!
}
```

Последний вызов напечатает **Non_poly** (поскольку **typeid(npr)** не оценивается) перед тем, как сгенерировать **bad_typeid**.

Определение **type_info** выглядит следующим образом:

```
class type_info {
    // данные
public:
    virtual ~type_info();                                  // полиморфный

    bool operator==(const type_info&) const noexcept;      // можно сравнить
    bool operator!=(const type_info&) const noexcept;

    bool before(const type_info&) const noexcept; // упорядочение
    size_t hash_code() const noexcept;            // для использования unordered_map
    const char* name() const noexcept;            // имя типа

    type_info(const type_info&) = delete;         // предотвращаем копирование
    type_info& operator=(const type_info&) = delete; // предотвращаем копирование
};
```

Функция **before()** позволяет сортировать **type_info**. Она позволяет использовать **type_id** в качестве ключей упорядоченных контейнеров (таких как **map**). Функция **hash_code()** позволяет использовать **typeid_id** в качестве ключей хэш-таблиц (таких как **unordered_map**).

Не гарантируется единственного **type_info** для каждого типа системы. Фактически, где используются динамические библиотеки, там реализации трудно обеспечить единственность объектов типа **type_info**. Как следствие, мы применяем операцию **==** к объектам **type_info** для проверки эквивалентности, а не к указателям на такие объекты.

Иногда нам нужно знать точный тип объекта, чтобы выполнить над ним некоторое действие. В идеале, такие манипуляции представлены виртуальными функциями, так что нет нужды знать точный тип объекта. Но в некоторых случаях нельзя предположить наличие общего интерфейса для всех манипулируемых объектов, так что приходится выяснять настоящий тип объектов (§22.5.1). Другим примером может служить выяснение имени класса для диагностической печати:

```
#include<typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

Символьное представление имени класса определяется реализацией. Эта С-строка находится в памяти, контролируемой системой, так что программист не должен применять к ней операцию **delete[]**.

22.5.1. Расширенная информация о типе

Объекты **type_info** содержат лишь минимальную информацию. Поэтому получение точного типа объекта часто лишь первый шаг на пути получения и использования более детальной информации об этом типе.

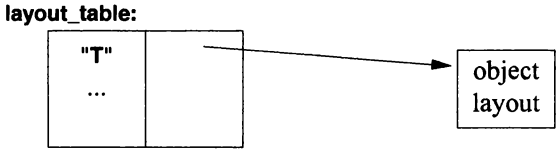
Рассмотрим, как реализация или инструмент могут сделать информацию о типах доступной пользователям во время выполнения программы. Предположим, что есть инструмент, который генерирует описание раскладки объектов для каждого класса, который я использую. Я могу разместить эти описания в контейнере **map**, чтобы пользовательский код мог найти такую информацию:

```
#include <typeinfo>

map<string, Layout> layout_table;

void f(B* p)
{
    Layout& x = layout_table[typeid(*p).name()]; // ищем Layout на базе *p имени
    // ... используем x ...
}
```

Результирующая структура данных выглядит следующим образом:



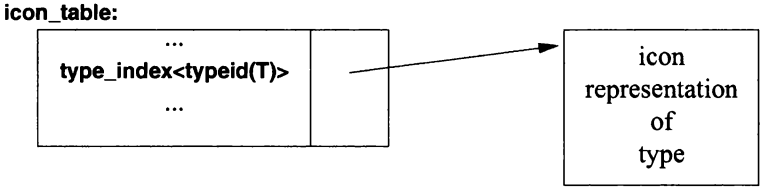
Кто-либо другой может предоставить другой тип информации:

```
unordered_map<type_index, Icon> icon_table;      // §31.4.3.2

void g(B* p)
{
    Icon& i = icon_table[type_index{typeid(*p)}];
    // ... используем i ...
}
```

Тип **type_index** является стандартным библиотечным типом для сравнения и хэширования объектов типа **type_info** (§35.5.4).

Результирующая структура данных выглядит следующим образом:



Связывая **typeid** с информацией без модификации системных заголовочных файлов позволяет разным людям или инструментам ассоциировать разную информацию с типами, независимо друг от друга.

22.6. Правильное и неправильное использование RTTI

Мы должны применять явную информацию о типе на этапе выполнения только в случае реальной необходимости. Статическая проверка (на этапе компиляции) надежнее, требует меньше накладных расходов и часто приводит к более структурированным програм-

мам. Интерфейсы, базирующиеся на виртуальных функциях, комбинируют статическую проверку типов с исследованием типа на этапе выполнения способом, который дает и безопасность типа, и гибкость. Однако, программисты часто недооценивают эти преимущества и используют RTTI там, где это не является подходящим решением. К примеру, RTTI можно использовать для написания отвратительного кода с оператором **switch**:

```
// неправильное использование RTTI:

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // ничего не делать
    }
    else if (typeid(r) == typeid(Triangle)) {
        // ... вращаем triangle (треугольник)...
    }
    else if (typeid(r) == typeid(Square)) {
        // ... вращаем квадрат (square) ...
    }
    // ...
}
```

Применение **dynamic_cast** вместо **typeid** улучшит этот код незначительно. В любом случае, этот код безобразен с синтаксической точки зрения, а также неэффективен, поскольку выполняет дорогие операции повторяющимся образом.

К сожалению, приведенный пример не является исключением и нередко используется на практике. Следует использовать виртуальные функции (§3.2.3, §20.3.2), а не RTTI, во всех случаях, когда требуется выполнять различение типов на этапе выполнения.

Многие примеры корректного применения RTTI относятся к случаю, когда какой-либо код выражен в терминах одного класса, а пользователь хочет добавить функциональность посредством наследования. Использование **Ival_box** из §22.2 служит хорошим примером. Если пользователь может и хочет модифицировать определения библиотечных классов, скажем **BBwindow**, тогда использования RTTI можно избежать; иначе он необходим. Даже если пользователь желает модифицировать базовые классы (например, чтобы добавить виртуальные функции), то такая модификация может привести к своим проблемам. Например, при этом может потребоваться неочевидная модификация виртуальных функций в классах, для которых эти функции не являются необходимыми. Пример использования RTTI для реализации простой системы объектного ввода/вывода можно найти в §22.2.4.

Для программистов, чей опыт состоит в применения языков, сильно зависящих от динамической проверки типов, таких как Smalltalk или Lisp, соблазнительно применять RTTI в комбинации с переобобщенными типами. Рассмотрим следующее:

```
// неправильное использование RTTI:

class Object {  // полиморфный
    // ...
};

class Container : public Object {
public:
    void put(Object*);
    Object* get();
    // ...
};
```

```
class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
    c->put(ps);                                  // помещаем Ship в контейнер
    // ...
    Object* p = c->get();                        // извлекаем Object из контейнера
    if (Ship* q = dynamic_cast<Ship*>(p)) {      // RTTI проверка, что Object есть Ship
        return q;
    }
    else {
        // ... что-либо еще (например, обработка ошибок) ...
    }
}
```

Здесь класс **Object** — это необязательный артефакт реализации. Он слишком обобщен, поскольку он не соответствует абстракции приложения, что вынуждает программиста использовать абстракцию уровня реализации (**Object**). Такие проблемы лучше преодолевать использованием шаблонных контейнеров, которые содержат единственный тип указателя:

```
Ship* f(Ship* ps, vector<Ship*>& c)
{
    c.push_back(ps);        // помещаем Ship в контейнер
    // ...
    return c.pop_back();    // извлекаем Ship из контейнера
}
```

Такой стиль кода менее подвержен ошибкам (лучше проверяется статически). В комбинации с виртуальными функциями такая техника подходит для многих случаев. В шаблоне шаблонный аргумент **T** занимает место **Object** и предоставляет возможность статической проверки типов (§27.2).

22.7. Советы

[1] Используйте виртуальные функции, чтобы гарантировать выполнение одной и той же операции независимо от использования того или иного интерфейса для объекта; §22.1.

[2] Используйте **dynamic_cast** там, где нельзя избежать навигации по классовой иерархии; §22.2.

[3] Используйте **dynamic_cast** для безопасной по типам навигации по классовой иерархии; §22.2.1.

[4] Применяйте **dynamic_cast** к ссылочному типу, если неудача в нахождении требуемого класса должна трактоваться именно как неудача; §22.2.1.1.

[5] Применяйте **dynamic_cast** к указательному типу, когда неудача в нахождении требуемого класса трактуется как законная альтернатива; §22.2.1.1.

[6] Используйте двойную диспетчеризацию или паттерн «посетитель» для того, чтобы выразить зависимость операции от двух динамических типов; §22.3.1.

[7] Не вызывайте виртуальные функции во время конструирования или деструкции; §22.4.

[8] Используйте **typeid** ради расширенной информации о типе; §22.5.1.

[9] Используйте **typeid** для того, чтобы найти тип объекта (а не интерфейс к объекту); §22.5.

[10] Вместо повторящегося кода из **typeid** или **dynamic_cast** в операторе **switch** применяйте виртуальные функции; §22.6.

23

Шаблоны

Здесь ваша цитата.
— Б. Страуструп

- Введение
- Простой шаблон строк
  Определение шаблона; Конкретизация шаблона
- Проверка типов
  Эквивалентность типов; Обнаружение ошибок
- Члены классовых шаблонов
  Члены-данные; Функции-члены; Члены-синонимы типов; Статические члены; Члены-типы; Друзья
- Шаблоны функций
  Аргументы функциональных шаблонов; Вывод аргументов для функциональных шаблонов; Перегрузка шаблонных функций
- Синонимы для шаблонов
- Организация исходного кода
- Советы

23.1. Введение

Шаблоны предоставляют прямую поддержку обобщенному программированию (§3.4) в форме использования типов в виде параметров. Механизм шаблонов языка C++ позволяет типу или значению быть параметром определения класса, функции или синонима типа. Синонимы обеспечивают способ представления целого семейства концепций с возможностью их комбинирования. Результирующие классы и функции сопоставимы с таковыми в ручном программировании в части производительности и потребления памяти.

Шаблоны зависят лишь от свойств параметров, которые используются фактически, и не требуют от типов, использованных как аргументы, ничего дополнительного. В частности, аргументы-типы шаблонов не обязаны быть частью иерархии наследования. Встроенные типы также допускаются и часто используются на практике.

Композиции, предлагаемые шаблонами, безопасны по типам (никакой объект не может использоваться способом, противоречащим его определению), но к сожалению требования шаблонов к своим аргументам не могут просто и прямо отражены в коде (§24.3).

Каждая существенная абстракция стандартной библиотеки представлена шаблоном (к примеру, **string**, **ostream**, **regex**, **complex**, **list**, **map**, **unique_ptr**, **thread**, **future**, **tuple** и **function**),

и таковы ключевые операции (например, сравнения для **string**, операция вывода **<<**, арифметические операции для **complex**, внедрения и удаления для **list**, а также **sort()**). Это делает главы настоящей книги, посвященные библиотеке (Часть IV), богатым источником примеров шаблонов и технологий программирования, основанных на шаблонах.

Мы здесь рассматриваем шаблоны в первую очередь под таким углом зрения, который помогает понять дизайн стандартной библиотеки, ее реализацию и использование. Приемы, использованные в стандартной библиотеке, предоставляют эффективность и гибкость, требующуюся при проектировании задач широкой проблематики. Эти техники помогают программистам прятать изощренные детали реализации за простыми интерфейсами и открывать сложности внутреннего устройства тогда, когда в этом имеется необходимость.

Перечисленные вопросы отражены в данной главе, а также в шести последующих главах. В настоящей главе мы сфокусируется на базовых средствах шаблонов и фундаментальных приемах работы с ними:

§23.2 *Простой шаблон строк*: базовые механизмы определения и использования шаблонов классов на примере шаблона строк.

§23.3 *Проверка типов*: основные правила эквивалентности типов и проверки типов, относящиеся к шаблонам.

§23.4 *Члены шаблонов класса*: как определяются и используются члены шаблонов класса.

§23.5 *Шаблоны функций*: как определить и использовать шаблоны функций. Как происходит разрешение перегрузки для шаблонов функций и обычных функций.

§23.6 *Синонимы шаблонов*: синонимы шаблонов предоставляют мощный механизм для сокрытия деталей реализации и «очищения» синтаксиса кода для работы с шаблонами.

§23.7 *Организация исходного кода*: как разместить код шаблонов в исходных файлах.

Глава 24, *Обобщенное программирование*, представляет основы обобщенного программирования, а также фундаментальную идею *концепций* (требований к параметрам шаблонов):

§24.2 *Алгоритмы*: пример основной методики разработки обобщенного алгоритма на основе конкретных примеров.

§24.3 *Концепции*: вводится и обсуждается фундаментальное понятие *концепции*, то есть набора требований, которые шаблон может наложить на свои шаблонные аргументы.

§24.4 *Конкретные концепции*: представлены приемы использования концепций, выраженных в форме предикатов времени компиляции.

Глава 25, *Специализации*: посвящена рассмотрению вопроса о том, как передаются аргументы шаблонам и вводится понятие специализации:

§25.2 *Параметры шаблонов и аргументы*: что может быть аргументом для шаблона: типы, значения и шаблоны. Как специфицировать и использовать аргументы шаблонов по умолчанию.

§25.3 *Специализации*: специальные версии, называемые специализациями, для специфических наборов аргументов шаблонов могут генерироваться компилятором или предоставляться программистом.

Глава 26, *Конкретизация*: рассматриваются вопросы, относящиеся к генерации специализаций шаблона и связыванию имен.

§26.2 *Конкретизация шаблонов*: правила, рассказывающие о том, как и когда компилятор генерирует специализацию из определения шаблона и как это специфицировать вручную.

§26.3 *Связывание имен*: правила, по которым определяется, к какой сущности относится имя, использованное в определении шаблона.

Глава 27, *Шаблоны и иерархии*: посвящена вопросу о том, как связаны приемы обобщенного программирования, основанные на шаблонах, и приемы объектно-ориентированного программирования, поддерживаемые классовыми иерархиями. Главный вопрос — как использовать их в комбинации.

§27.2 *Параметризация и иерархии*: шаблоны и иерархии классов являются двумя способами представления связанных абстракций. Какому способу отдать предпочтение?

§27.3 *Иерархии классовых шаблонов*: почему идея просто добавить шаблонные параметры к существующей классовой иерархии не слишком удачна?

§27.4 *Параметры шаблонов как базовые классы*: представлена техника композиции интерфейсов и структур данных для достижения безопасности типов и производительности.

Глава 28, *Метапрограммирование*: глава концентрируется на применении шаблонов как способа генерация классов и функций.

§28.2 *Функции типов*: функции, принимающие типы в качестве аргументов, или возвращающие типы в качестве результата.

§28.3 *Управляющие структуры времени компиляции*: как выразить выбор и рекурсию для функций типов, а также некоторые правила их использования.

§28.4 *Условные определения*: **enable_if**: рассматривается, как можно условно определить функции и перегрузить шаблоны, используя почти произвольные предикаты.

§28.5 *Список времени компиляции*: **tuple**: как можно создавать и работать со списками с элементами почти произвольных типов.

§28.6 *Шаблоны с произвольным числом параметров*: как в надежной статической манере определить шаблоны, принимающие произвольное число параметров произвольных типов.

§28.7 *Пример с единицами измерений*: используется простая техника метапрограммирования вкупе с иными приемами для разработки библиотеки вычислений, которая на этапе компиляции проверяется на предмет корректного использования системы измерений с килограммами, метрами и секундами.

Глава 29, *Дизайн матриц*: показывается, как разные средства шаблонов применяются в совокупности для решения сложной проектной задачи:

§29.2 *Шаблон* **Matrix**: как определить матрицу N-измерений, демонстрирующую гибкую и безопасную по типам инициализацию, операцию индексирования и взятие подматриц.

§29.3 *Арифметические операции для* **Matrix**: рассматриваются простые арифметические операции для N-мерных матриц.

§29.4 *Реализация типа* **Matrix**: некоторое количество полезных приемов реализации.

§29.5 *Решение линейных уравнений*: простой пример на матрицы.

Шаблоны уже были представлены в настоящей книге ранее (§3.4.1, §3.4.2) и широко использовались, так что я уверен, что вы обладаете начальными знаниями по шаблонам.

23.2. Простой шаблон строк

Будем рассматривать строки символов. Класс строк содержит символы и предоставляет такие операции, как индексирование, конкатенацию и сравнение, которые мы ассоциируем с понятием «строка». Мы хотим обеспечить такое поведение для разных видов символов. Например, строки знаковых или беззнаковых символов, Китайских символов, Греческих символов и т.д. Таким образом, мы хотим изобразить понятие «строка» с минимальными зависимостями от конкретных видов символов. Определение строки полагается на тот факт, что символы можно копировать, и еще на небольшое количество их свойств (§24.3). Мы можем произвести более общий тип строк, исходя из строки символов типа **char** (§19.3), делая символьный тип параметром:

```
template<typename C>
class String {
public:
    String();
    explicit String(const C*);
    String(const String&);
    String operator=(const String&);
    // ...
    C& operator[](int n) {return ptr[n];}      // доступ к элементам без проверки
    String& operator+=(C c);                   // добавить с в конец
    // ...
private:
    static const int short_max = 15;           // для оптимизации коротких строк
    int sz;
    C* ptr;                                    // ptr указывает на sz у С-ов
};
```

Префикс **template<typename C>** означает, что шаблон объявлен и что **C** является аргументом типа. После этого **C** используется точно так же, как любые другие имена типов. Область видимости **C** распространяется до конца объявления **template<typename C>**. В общем случае **C** это имя типа; оно не обязано быть именем класса. Люди с математическим образованием мргут прочесть **template<typename C>** как «для всех С», или как «для всех типов С». Но в С++ отсутствует общий механизм, с помощью которого налагаются необходимые требования к параметру шаблона **C**. То есть невозможно сказать «для всех С, таких как ...». Это означает, что невозможно специфицировать предполагаемый тип аргумента С (§24.3).

Имя класса с последующими угловыми скобками это по-прежнему имя класса и может использоваться так же, как и другие имена классов. Например:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

struct Jchar {/* ... */};          // Японский символ

String<Jchar> js;
```

Если не обращать внимание на специфический синтаксис, то имя **String<char>** работает точно так же, как если бы это было имя класса **String** из §19.3. Преобразуя **String** в шаблон, мы получаем возможность использовать средства, предназначенные для строк с символами **char**, применительно к строкам с любыми символами. К примеру, в случае использования контейнера **map** стандартной библиотеки и шаблона **String**, мы можем изобразить пример с подсчетом слов из §19.2.1 следующим образом:

```
int main()      // считаем количество каждого слова в потоке
{
    map<String<char>, int> m;
    for (String<char> buf; cin>>buf;)
        ++m[buf];
    // ... пишем результат ...
}
```

Версия для японских символов будет иметь следующий вид:

```
int main()      // считаем количество каждого слова в потоке
{
    map<String<Jchar>, int> m;
    for (String<Jchar> buf; cin>>buf;)
        ++m[buf];
    // ... пишем результат ...
}
```

В стандартной библиотеке есть шаблонный класс **basic_string**, аналогичный нашей шаблонизированной версии класса **String** (§19.3, §36.3). В стандартной билиотеке имя **string** является синонимом для **basic_string<char>** (§36.3):

```
using string = std::basic_string<char>;
```

Теперь программу для подсчета слов можно написать следующим образом:

```
int main()      // считаем количество каждого слова в потоке
{
    map<string, int> m;
    for (string buf; cin>>buf;)
        ++m[buf];
    // ... пишем результат ...
}
```

В общем случае синонимы типов (§6.5) помогают сокращать имена классов, генерируемые из шаблонов. Кроме того, синонимы помогают скрыть от нас лишние детали, например, что тип генерируется из шаблона.

23.2.1. Определение шаблона

Класс, генерируемый из шаблона класса, является обычным классом. Таким образом, использование шаблонов не предполагает никаких дополнительных механизмов времени выполнения по сравнению с классами, написанными вручную. Более того, код классов, генерируемых из шаблонов, может оказаться короче, поскольку при этом не генерируется код для членов, которые не используются фактически (§26.2.1).

Кроме шаблонов класса язык C++ предлагает шаблоны функций (§3.4.2, §23.5). Большая часть материала у меня будет касаться шаблонов класса, а подробное рассмотрение шаблонов функций будет отнесено к §23.5. Шаблон — это спецификация, рассказывающая о том, как генерировать нечто на основе подходящих аргументов шаблона; механизмы языка для выполнения этой задачи (конкретизация (§26.2) и специализация (§25.3)) не сильно дифференцируются для генерации класса или функции. То есть, если не имеет место обратное, то правила для шаблонов применяются одинаково для шаблонов класса и шаблонов функции. Шаблоны также могут определяться как синонимы (§23.6), но иные потенциально возможные конструкции, такие как, например, шаблоны пространств имен, не допустимы.

В данной книге я не делаю разницы между терминами шаблон класса и шаблонный класс, а также между шаблоном функции и шаблонной функцией. Я использую эти термины взаимозаменяемым образом.

При конструировании шаблона будет разумным создать и протестировать какой-нибудь конкретный класс вроде **String**, а уже потом превратить его в шаблон, например в **String<C>**. Это помогает решить многие проблемы с кодом и выявить многие ошибки в контексте конкретного класса. Такая работа знакома большинству программистов, так как при этом приходится работать с конкретным классом, а не с абстрактной концепцией. Позже мы будем работать с проблемами, вытекающими из обобщения, не отвлекаясь на то, чтобы одновременно иметь дело с более традиционными ошибками. Аналогично, пытаясь понять шаблон, удобно представлять себе вариант с конкретным аргументом, чтобы можно было вообразить его поведение в таком случае. Это отвечает представлению о том, что обобщенный вариант рождается как обобщение некоторого количества конкретных примеров, а не возникает сразу как абстрактная концепция (§24.2).

Члены шаблона класса объявляются и определяются точно так же, как это происходит в случае нешаблонного класса. Член не обязан быть определен непосредственно внутри шаблона класса. В таком случае, его определение должно присутствовать где-либо еще, как и для членов нешаблонных классов (§16.2.1). Члены шаблонного класса сами по себе являются шаблонами, параметризуемыми с помощью содержащего их классового шаблона. Когда такой член определяется вне их класса, он явным образом должен идентифицироваться как шаблон. Например:

```
template<typename C>
String<C>::String()        // String<C> конструктор
    : sz{0}, ptr{ch}
{
    ch[0] = {};                // терминирующий 0 соответствующего символьного типа
}

template<typename C>
String& String<C>::operator+=(C c)
{
    // ... добавляем c в конец этой строки ...
    return *this;
}
```

Параметр шаблона, такой как **C**, является параметром, а не именем специфического типа. Но это не влияет на то, как мы пишем код, используя этот идентификатор. В пределах области видимости **String<C>** квалификация с помощью **<C>** является излишней, так что **String<C>::String** служит именем для конструктора.

Может существовать лишь одна функция, определяющая функцию-член в программе. Однако, специализация (§25.3) позволяет предоставить альтернативную реализацию шаблона для набора специфических аргументов шаблона. Для функций, мы можем также воспользоваться перегрузкой, чтобы обеспечить разные определения для разных типов аргументов (§23.5.3).

Невозможно перегрузить имя шаблона класса, так что если классовый шаблон объявлен в некоторой области видимости, то нельзя там объявить какую-либо сущность с таким же именем. Например:

```
template<typename T>
class String {/* ... */};

class String {/* ... */};        // ошибка: двойное определение
```

Тип, используемый в качестве аргумента шаблона, должен предоставлять интерфейс, ожидаемый шаблоном. Например, тип для **String** должен предоставлять обычные опера-

ции копирования (§17.5, §36.2.2). Отметим, что нет никакого требования, чтобы разные аргументы для одного и того же шаблона были связаны наследованием. См. дополнительно §25.2.1 (параметры-типы для шаблонов), §23.5.2 (вывод параметров шаблона) и §24.3 (требование к аргументам шаблона).

23.2.2. Конкретизация шаблона

Процесс генерация класса или функции из шаблона с предоставленным списком аргументов часто называют *конкретизацией шаблона* (§26.2). Версия шаблона под специфический аргумент шаблона называется *специализацией*.

Обычно, это задача компилятора, а не программиста, гарантировать, чтобы специализации шаблона генерировались для каждого набора используемых шаблонных аргументов. Например:

```
String<char> cs;

void f()
{
    String<Jchar> js;

    cs = "It's the implementation's job to figure out what code needs to be generated";
}
```

Здесь реализация генерирует объявления для классов **String<char>** и **String<Jchar>**, определения для их деструкторов и умолчательных конструкторов, а также для **String<char>::operator=(char*)**. Остальные члены-функции в коде не используются и их определения не генерируются. Сгенерированные классы — суть такие же обычные классы, подчиняющиеся обычным для классов правилам. Аналогично, сгенерированные функции суть простые функции, подчиняющиеся правилам для функций.

Очевидно, что шаблоны являются мощным средством для генерации большого объема кода из относительно коротких определений. Следовательно, нужно внимательно следить за тем, чтобы не перегрузить память множеством практически идентичных определений функций (§25.3). С другой стороны, шаблоны могут применяться для того, чтобы достигать иначе не достигаемого качества кода. В частности, композиция шаблонов и встраивания может использоваться для того, чтобы избежать многих (прямых или косвенных) вызовов функций. Например, простые операции над критическими структурами данных (вроде **<** в **sort()** и **+** для скаляров в матричных вычислениях) редуцируются до единственных машинных инструкций в сильно параметризованных библиотеках.

23.3. Проверка типов

Конкретизация шаблона вовлекает в процесс сам шаблон и набор аргументов шаблона, и в результате генерируется код. Столь большое количество информации, имеющееся в момент конкретизации шаблона, позволяет собирать воедино и сведения об определении шаблона, и сведения о типах аргументов шаблона, так что в итоге может быть сгенерирован очень эффективный код. К сожалению, такая гибкость означает также и сложность в проверке типов, а также затрудняет выдачу сообщений об ошибках.

Проверка типов выполняется на коде, сгенерированном во время конкретизации (как если бы программист раскрывал шаблон вручную). Этот код может содержать много такого, о котором пользователь никогда и не слышал, и часто с ним приходится иметь дело слишком поздно.

Фундаментальной слабостью механизма шаблонов является невозможность явным образом выразить требования к аргументу шаблона. К примеру, мы не можем сказать следующее:

```
template<Container Cont, typename Elem>
  requires Equal_comparable<Cont::value_type, Elem>() // требования к типам Cont и Elem
int find_index(Cont& c, Elem e);                      // ищем индекс e в c
```

То есть, у нас в рамках языка C++ нет возможности сказать, что **Cont** должен быть типом, который может действовать как контейнер, и что тип **Elem** должен быть типом, позволяющим сравнивать значение элемента контейнера **Cont**. Ведется работа, которая позволит сделать это в будущих версиях C++ (без потери в гибкости и эффективности) [Sutton,2011], но сейчас мы вынуждены обходиться без этого.

Давайте сформулируем набор понятий, связанный с передачей аргументов шаблону. Можно думать о наборе требований к аргументам шаблона как о предикате. Например, мы можем трактовать фразу «**C** должен быть контейнером» как предикат, который принимает **C** как аргумент и возвращает **true**, если **C** является контейнером, и возвращает **false** в противном случае. Например, **Container<vector<int>>()** и **Container<list<string>>()** вернут **true**, в то время как **Container<int>()** и **Container<shared_ptr<string>>()** вернут **false**. Мы называем такой предикат *концепцией*. Пока что концепция не является конструкцией языка C++; это просто понятие, которое можно использовать, рассуждая про требования к аргументам шаблона, использовать в комментариях и иногда поддерживать в нашем коде (§24.3).

Для начала думайте о концепциях как об инструменте проектирования: специфицируйте **Container<T>()** в качестве набора комментариев, рассказывающих, какие свойства должны быть у типа **T**, чтобы **Container<T>()** возвращал **true**. К примеру:

- **T** должен обладать операцией индексирования ([]).
- **T** должен иметь функцию-член **size()**.
- **T** должен обладать членом типа **value_type**, служащим типом элементов.

Этот список не полон (не сказано, что должно быть аргументом у [] и что за возврат у этой операции) и не отвечает большинству семантических вопросов (что [] делает на самом деле?). Но даже частичный набор требований полезен; даже совсем простые вещи позволяют нам в ручном режиме проверить использование шаблона и обнаружить очевидные ошибки. Например, **Container<int>()** с очевидностью **false**, поскольку у типа **int** нет операции индексирования. Я еще раз вернусь к вопросу о проектировании концепций (§24.3), о поддержке концепций в коде (§24.4) и приведу пример набора полезных концепций (§24.3.2). Еще раз подчеркнем, что пока что язык C++ не поддерживает концепции напрямую, но это не значит, что концепции не существуют: для каждого работающего шаблона программист держит в уме некоторые концепции для аргументов. В свое время Деннис Ритчи хорошо сказал, что «C это язык с сильной типизацией, но со слабой проверкой». Вы почти то же самое можете сказать про шаблоны языка C++, за исключением того, что проверка аргументов шаблона выполняется в реальности, но слишком поздно во время компиляции.

23.3.1. Эквивалентность типов

Имея шаблон, мы можем генерировать типы на основе аргументов шаблона. Например:

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;
```

```
using Uchar = unsigned char;
using uchar = unsigned char;

String<Uchar> s4;
String<uchar> s5;
String<char> s6;

template<typename T, int N>        // §25.2.2
    class Buffer;
Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

Используя тот же самый набор аргументов для шаблона, мы всегда имеем дело с тем же самым сгенерированным типом. Но что значит «тот же самый» в данном контексте? Синонимы не дают нового типа, так что **String<Uchar>** и **String<uchar>** — это тот же тип, что и **String<unsigned char>**. И наоборот, поскольку **char** и **unsigned char** — это разные типы (§6.2.3), то **String<char>** и **String<unsigned char>** тоже разные типы.

Компилятор может оценивать константные выражения (§10.4), так что **Buffer<char,20-10>** понимается как тот же тип, что и **Buffer<char,10>**.

Типы, генерируемые из одного и того же шаблона разными аргументами шаблона, являются разными типами. В частности, типы, генерируемые из связанных друг с другом аргументов, не обязательно сами являются таковыми. Например, предположим, что **Circle** это подвид **Shape**:

```
Shape* p {new Circle(p,100)};              // Circle* конвертируется в Shape*
vector<Shape>* q {new vector<Circle>{}}; // ошибка: нет конверсии vector<Circle>* в vector<Shape>*
vector<Shape> vs {vector<Circle>{}};       // ошибка: нет конверсии vector<Circle> в vector<Shape>
vector<Shape*> vs {vector<Circle*>{}};     // ошибка: нет конверсии vector<Circle*> в vector<Shape*>
```

Если бы такие преобразования были разрешены, то возникали бы ошибки с типами (§27.2.1). Если нужны преобразования между сгенерированными классами, программист может их определить (§27.2.2).

23.3.2. Обнаружение ошибок

Шаблон сначала определяется, а затем используется в комбинации с набором аргументов шаблона. В определении шаблона происходит проверка на синтаксические ошибки, а также на иные ошибки, которые можно обнаружить независимо от аргументов шаблона. Например:

```
template<typename T>
struct Link {
    Link* pre;
    Link*suc                       // синтаксическая ошибка: отсутствует точка с запятой
    T val;
};

template<typename T>
class List {
    Link<T>* head;
public:
    List(): head{7} {}                     // ошибка: указатель инициализируется типом int
    List(const T& t) : head{new Link<T>{0, o, t}} {} // ошибка: неопредел. идентификатор o
    // ...
    void print_all() const;
};
```

Компилятор в состоянии поймать простые семантические ошибки в точке определения или позже в точке использования. Пользователи предпочитают более ранние обнаружения, однако не все «простые» ошибки легко детектировать. Здесь я сделал три ошибки:

- Простая синтаксическая ошибка: пропустил точку с запятой в конце объявления.
- Простая ошибка типов: Независимо от того, чем является параметр шаблона, указатель нельзя инициализировать числом **7**.
- Ошибка использования имени: Идентификатор **o** (следствие опечатки — вместо **0**) не может быть аргументом для **Link<T>** конструктора, поскольку такового имени нет в соответствующей области видимости.

Имя, использованное в определении шаблона, должно либо находиться в области видимости, или же каким-то очевидным образом зависеть от параметра шаблона (§26.3). Зависимость от параметра шаблона **T** может заключаться в явном использованиии этого имени, в использовании члена типа **T**, или принимается параметр типа **T**. Например:

```
template<typename T>
void List<T>::print_all() const
{
    for (Link<T>* p = head; p; p=p->suc)        // p зависит от T
        cout << *p;                             // << зависит от T
}
```

Ошибки, связанные с использованием параметров шаблона, не могут быть выявлены до того момента, пока шаблон не будет использоваться. Например:

```
class Rec {
    string name;
    string address;
};

void f(const List<int>& li, const List<Rec>& lr)
{
    li.print_all();
    lr.print_all();
}
```

Проверка **li.print_all()** проходит на ура, а проверка **lr.print_all()** выдает ошибку типа, поскольку для типа **Rec** не определена операция **<<**. Самой первой точкой обнаружения ошибок, имеющих отношение к параметрам шаблонов, является самое первое место в программе, где шаблон используется с конкретным типом параметра. Эта точка называется первой точкой конкретизации (§26.3.3). Реализациям разрешается откладывать такую проверку вплоть до компоновки готовой программы, а для некоторых ошибок это является и вообще самой первой возможной точкой. Независимо от того, когда выполняется проверка, один и тот же набор правил вовлекается в рассмотрение. Конечно же, пользователи предпочли бы более раннее обнаружение.

23.4. Члены классовых шаблонов

Как и обычный класс, шаблонный класс может иметь члены нескольких видов:

- Члены-данные (переменные и константы); §23.4.1
- Функции-члены; §23.4.2
- Члены — синонимы типов; §23.6
- Статические (**static**) члены (функции и данные); §23.4.4

- Члены-типы (например, класс-член); §23.4.5
- Шаблоны-типы (например, член-шаблон класса); §23.4.6.3

Кроме того, классовый шаблон может объявить друзей (**friend**), точно так же как и обычный класс; §23.4.7

Правила для членов классового шаблона такие же, как и для генерируемых из него классов.

23.4.1. Члены-данные

Как и обычный класс, шаблонный класс может иметь члены-данные любых типов. Нестатические члены-данные могут инициализироваться в их определениях (§17.4.4) или в конструкторах (§16.2.5). Например:

```
template<typename T>
struct X {
    int m1 = 7;
    T m2;
    X(const T& x): m2{x} {}
};

X<int> xi {9};
X<string> xs {"Rapperswil"};
```

Нестатические члены-данные могут быть константными, но к сожалению не могут быть константными выражениями (**constexpr**).

23.4.2. Функции-члены

Как и для обычного класса, нестатические функции-члены классового шаблона могут определяться внутри определения класса или вне его. Например:

```
template<typename T>
struct X {
    void mf1() {/* ... */}      // определено внутри определения класса
    void mf2();
};

template<typename T>
void X<T>::mf2() {/* ... */}    // определено вне определения класса
```

Аналогично, функция-член шаблона может быть виртуальной, или нет. Однако, виртуальная функция-член не может быть шаблонной функцией (§23.4.6.2).

23.4.3. Члены-синонимы типов

Члены-синонимы типов, вводимые с помощью **using** или **typedef** (§6.5), играют важную роль в дизайне классовых шаблонов. Они определяют связанные типы так, что к ним легко можно обратиться извне класса. Например, мы определяем контейнерный итератор и тип элементов как синонимы:

```
template<typename T>
class Vector {
public:
    using value_type = T;
    using iterator = Vector_iter<T>;    // Vector_iter определен где-то еще
    // ...
};
```

Имя шаблонного аргумента, то есть **T**, доступно лишь непосредственно самому шаблону, так что остальному коду для такого доступа нужно определить синоним.

Синонимы типов играют основную роль в обобщенном программировании, позволяя разработчику классов обеспечивать одинаковые имена в разных классах (и классовых шаблонах) с общей семантикой. Имена **value_type** и **iterator** позаимствованы из проектов контейнеров стандартной библиотеки (§33.1.3).

23.4.4. Статические члены

Статический член или функция, не определенные внутри класса, должны иметь в программе уникальное определение. Например:

```
template<typename T>
struct X {
    static constexpr Point p {100,250}; // Point должен быть литеральным типом (§ 10.4.3)
    static const int m1 = 7;
    static int m2 = 8;                  // ошибка: не const

    static int m3;
    static void f1() {/* ... */}
    static void f2();
};

template<typename T> int X<T>::m1 = 88;   // ошибка: два инициализатора
template<typename T> int X<T>::m3 = 99;

template<typename T> void X::<T>::f2() {/* ... */}
```

Как и для обычных (нешаблонных) классов, константные (или **constexpr**) статические члены-данные литеральных типов могут инициализироваться внутри класса, и их не нужно определять вне определения класса (§17.4.5, § iso.9.2).

Статический член требует определения только в случае, если он используется (§ iso.3.2, § iso.9.4.2, §16.2.12). Например:

```
template<typename T>
    struct X {
    static int a;
    static int b;
};

int* p = &X<int>::a;
```

Если в программе имеются лишь данный пример использования **X<int>**, мы получим ошибку «не определено» для **X<int>::a**, но не для **X<int>::b**.

23.4.5. Члены-типы

Как и для обычных классов мы можем в шаблоне определить типы в качестве членов. Как обычно, такой тип может быть классом или перечислением. Например:

```
template<typename T>
struct X {
    enum E1 {a, b};
    enum E2;      // ошибка: нижележащий тип не известен
    enum class E3;
    enum E4 : char;

    struct C1 {/* ... */};
    struct C2;
};
```

```
template<typename T>
enum class X<T>::E3 {a, b};              // требуется

template<typename T>
enum class X<T>::E4 : char {x, y};       // требуется

template<typename T>
struct X<T>::C2 {/* ... */};             // требуется
```

Внеклассовое определение члена-перечисления допускается лишь для перечислений, для которых известен нижележащий тип (§8.4).

Как обычно, энумераторы для обычных (не-**class**) перечислений помещены в область видимости перечисления; то есть энумераторы перечислений-членов находятся в области видимости их класса.

23.4.6. Члены-шаблоны

Класс или классовый шаблон могут иметь члены, которые сами по себе являются шаблонами. Это позволяет нам определять родственные (связанные друг с другом) типы с приятной возможностью контроля и гибкости. Например, комплексные числа наилучшим образом определяются как пары значений некоторого скалярного типа:

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    complex(): re{}, im{} {}                    // умолчательный конструктор
    template<typename T>
    complex(T rr, T ii =0): re{rr}, im{ii} {}

    complex(const complex&) = default;          // копирующий конструктор
    template<typename T>
        complex(const complex<T>& c) : re{c.real()}, im{c.imag()} {}
    // ...
};
```

В результате возможны математически разумные преобразования между комплексными типами, при запрете нежелательных сужающих преобразований (§10.5.2.6):

```
complex<float> cf;                       // умолчательное значение
complex<double> cd {cf};                 // OK: используется конверсия float в double
complex<float> cf2 {cd};                 // ошибка: нет неявной конверсии double->float

complex<float> cf3 {2.0,3.0};            // ошибка: нет неявной конверсии double->float
complex<double> cd2 {2.0F,3.0F};         // OK: используется конверсия float в double

class Quad {
    // нет конверсии в int
};

complex<Quad> cq;
complex<int> ci {cq};                    // ошибка: нет конверсии Quad в int
```

Отталкиваясь от определени **complex**, мы можем сконструировать **complex<T1>** из **complex<T2>**, или из пары значений **T2** тогда и только тогда, когда мы можем сконструировать **T1** из **T2**. Это кажется разумным.

Будучи осведомленным о том, что ошибка сужения **complex<double>** до **complex<float>** не будет поймана до конкретизации конструктора **complex<float>**, я использовал синтак-

сис {}-инициализации (§6.3.5) в инициализаторах членов конструктора. Такой синтаксис не допускает сужения.

Используя старый ()-синтаксис, мы оставили бы возможными ошибки сужения. Например:

```
template<typename Scalar>
class complex {     // старый стиль
    Scalar re, im;
public:
    complex(): re(0), im(0) {}
    template<typename T>
    complex(T rr, T ii =0): re(rr), im(ii) {}

    complex(const complex&) = default;          // копирующий конструктор
    template<typename T>
        complex(const complex<T>& c) : re(c.real()), im(c.imag()) {}
    // ...
};

complex<float> cf4 {2.1,2.9};                   // ух! сужение
complex<float> cf5 {cd};                        // ух! сужение
```

Я считаю это еще одной причиной, чтобы быть последовательным в использовании {}-нотации для инициализации.

23.4.6.1. Шаблоны и конструкторы

Чтобы минимизировать недоразумения, я явным образом добавил умолчательный конструктор. По техническим причинам шаблонный конструктор не используется для генерации копирующего конструктора, так что если явное определение для копирующего конструктора отсутствует, то он по умолчанию не появится. Аналогично, копирующие присваивания, перемещающие конструкторы и перемещающие присваивания (§17.5.1, §17.6, §19.3.1) должны определяться как нешаблонные операции, или же будут сгенерированы умолчательные версии.

23.4.6.2. Шаблоны и ключевое слово virtual

Член-шаблон не может быть **virtual** Пример:

```
class Shape {
    // ...
    template<typename T>
        virtual bool intersect(const T&) const =0;    // ошибка: виртуальный шаблон
};
```

Это должно запрещаться. Если это разрешалось бы, то традиционная техника реализации виртуальных функций через таблицу виртуальных функций (§3.2.3) не могла бы использоваться. Компоновщику пришлось бы каждый раз добавлять в эту таблицу новый вход, как только кто-то вызвал **intersect()** с новым типом аргумента. Усложнение компоновщика в таком направлении было сочтено неприемлемым. При этом управление динамическим связыванием потребовало бы технологий, сильно отличающихся от уже широко используемых.

23.4.6.3. Использование вложения

В общем случае идея хранить информацию столь локально, насколько это возможно, является хорошей идеей (имя легче найти и труднее ввести в противоречие с чем-либо еще в программе). Для реализации идеи вводят типы-члены. Однако, в случае шаблонов

приходится рассматривать, подходит ли параметризация типу-члену. Формально, член шаблона зависит от всех его параметров. Это может иметь нежелательные косвенные последствия в случаях, когда фактически поведение члена не зависит от каждого параметра шаблона. Широкоизвестным примером служит пример типа связи в связанном списке. Рассмотрим следующее:

```
template<typename T, typename Allocator>
class List {
private:
    struct Link {
        T val;
        Link* succ;
        Link* prev;
    };
    // ...
};
```

Здесь тип **Link** является деталью реализации **List**. Поэтому он выглядит отличным примером типа, который лучше всего определить в области видимости **List**, и даже сделать его закрытым (**private**). Это всегда считалось популярным дизайном и в общем случае работало неплохо. Но к удивлению, это могло приводить к дополнительным издержкам по сравнению с нелокальным типом для **Link**. Допустим, что никакой член **Link** не зависит от параметра **Allocator**, и что нам нужны **List<double, My_allocator>** и **List<double, Your_allocator>**. Теперь **List<double, My_allocator>::Link** и **List<double, Your_allocator>::Link** являются разными типами, так что код, их использующий, не может быть идентичным. Таким образом, делая **Link** членом в случае, когда используется лишь один из параметров **List**, вносит ненужный диссонанс в код. Теперь рассмотрим дизайн, в котором **Link** не является членом:

```
template<typename T, typename Allocator>
class List;

template<typename T>
class Link {
    template<typename U, typename A>
        friend class List;
    T val;
    Link* succ;
    Link* prev;
};

template<typename T, typename Allocator>
class List {
    // ...
};
```

Я сделал все члены для типа **Link** закрытыми и предоставил доступ для **List**. За исключением того, что имя **Link** стало нелокальным, сохранена дизайнерская идея, что **Link** является деталью реализации **List**.

Но что если вложенный класс не рассматривается деталью реализации? Может нам нужен ассоциированный тип, предназначенный для разного использования? Рассмотрим следующее:

```
template<typename T, typename A>
class List {
public:
    class Iterator {
        Link<T>* current_position;
```

```
    public:
        // ... обычные операции итератора ...
    };

    Iterator<T, A> begin();
    Iterator<T, A> end();
    // ...
};
```

Здесь тип-член **List<T, A>::iterator** (очевидно) не использует второй аргумент **A**. Однако, поскольку **iterator** является членом и поэтому формально зависит от **A**, мы не сможем написать функцию, которая обрабатывает объекты типа **List** независимо от того, как они конструируются с использованием аллокаторов:

```
void fct(List<int>::Iterator b, List<int>::Iterator e)  // ошибка: у List два аргумента
{
    auto p = find(b, e,17);
    // ...
}

void user(List<int, My_allocator>& lm, List<int, Your_allocator>& ly)
{
    fct(lm.begin(), lm.end());
    fct(ly.begin(), ly.end());
}
```

Вместо этого нам нужно написать шаблонную функцию с зависимостью от аргумента-аллокатора:

```
void fct(List<int, My_allocator>::Iterator b, List<int, My_allocator>::Iterator e)
{

    auto p = find(b, e,17);
    // ...
}
```

Но к сожалению, это рушит нашу функцию **user()**:

```
void user(List<int, My_allocator>& lm, List<int, Your_allocator>& ly)
{
    fct(lm.begin(), lm.end());
    fct(ly.begin(), ly.end());          // ошибка: fct принимает List<int, My_allocator>::Iterators
}
```

Мы могли сделать **fct** шаблоном и генерировать разные специализации для каждого случая использования **iterator**, но это привело бы к хаосу в коде [Tsafrir,2009]. Опять-таки, мы решили проблему, переместив **Link** во вне классового шаблона:

```
template<typename T>
struct Iterator {
    Link<T>* current_position;
};

template<typename T, typename A>
class List {
public:
    Iterator<T> begin();
    Iterator<T> end();
    // ...
};
```

Это делает итераторы для любого **List** с одинаковым первым шаблонным аргументом взаимозаменяемыми до тех пор, пока речь идет об их типах. В нашем случае, это именно то, что нам надо. Наша функция **user()** теперь работает так, как это было определено. Если бы **fct()** было определено как функциональный шаблон, то была бы единственная копия (конкретизация) определения **fct()**. У меня есть правило: «избегать вложенных типов в шаблонах, если только они не зависят естественным образом от всех параметров шаблона». Можно сказать, что это специфический случай общего правила избегать ненужных зависимостей в коде.

23.4.7. Друзья

Как показано в §23.4.6.3, шаблонный класс может обозначать функции в качестве друзей (с помощью ключевого слова **friend**). Рассмотрим примеры с **Matrix** и **Vector** из §19.4. В типичном случае и **Matrix**, и **Vector** будут шаблонами:

```
template<typename T> class Matrix;

template<typename T>
class Vector {
    T v[4];
public:
    friend Vector operator*<>(const Matrix<T>&, const Vector&);
    // ...
};

template<typename T>
class Matrix {
    Vector<T> v[4];
public:
    friend Vector<T> operator*<>(const Matrix&, const Vector<T>&);
    // ...
};
```

Знак **<>** после имени функции-друга нужен для того, чтобы стало ясно, что «друг» является шаблонной функцией. Без этого обозначения будет предполагаться нешаблонная функция. Операция умножения определяется так, чтобы можно было иметь доступ к данным из **Vector** и **Matrix** напрямую:

```
template<typename T>
Vector<T> operator*(const Matrix<T>& m, const Vector<T>& v)
{
    Vector<T> r;
    // ... используем m.v[i] и v.v[i] для прямого доступа к элементам ...
    return r;
}
```

Друзья не влияют на область видимости, в которой определяется шаблонный класс, а также они не влияют на область видимости, в которой шаблон используется. Вместо этого, функция-друзья и операции распознаются по схеме, основанной на типах их аргументов (§14.2.4, §18.2.5, § iso.11.3). Как и функция-член, функция-друг конкретизируется только в случае, если она используется (§26.2.1).

Как и остальные классы, классовый шаблон может обозначить класс другом. Например:

```
class C;
using C2 = C;
```

```
template<typename T>
class My_class {
    friend C;           // OK: C это класс
    friend C2;          // OK: C2 это синоним класса
    friend C3;          // ошибка: класса C3 нет в области видимости
    friend class C4;    // OK: вводим новый класс C4
};
```

Естественно, что в интересных случаях друзья зависят от параметров шаблона. Например:

```
template<typename T>
class my_other_class {
    friend T;                   // мой аргумент мой друг!
    friend My_class<T>;         // My_class с соответствующим аргументом – мой друг
    friend class T;             // ошибка: избыточный "класс"
};
```

Как всегда, «дружба» не наследуется и не транзитивна (§19.4). Например, **C** не становится другом для **My_other_class<int>** несмотря на то, что **My_class<int>** является другом и **C** является другом для **My_class<int>**.

Мы не можем напрямую сделать шаблон другом класса, но мы можем сделать объявление друга шаблоном. Например:

```
template<typename T, typename A>
class List;

template<typename T>
class Link {
    template<typename U, typename A>
        friend class List;
    // ...
};
```

К сожалению, нет способа сказать, что **Link<X>** должен быть только другом для **List<X>**.

Классы-друзья введены в язык для отражения небольших кластеров тесно связанных между собой концепций. Слишком сложные картины дружбы почти всегда являются ошибкой проектирования.

23.5. Шаблоны функций

Для большинства пользователей очевидными примерами шаблонов являются контейнерные классы, такие как **vector** (§31.4), **list** (§31.4.2) и **map** (§31.4.3). А вскоре появляется и необходимость в шаблонных функциях для манипуляции этими контейнерами. Сортировка для **vector** служит простым примером:

```
template<typename T> void sort(vector<T>&);     // объявление

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);       // sort(vector<int>&);
    sort(vs);       // sort(vector<str ing>&);
}
```

В момент вызова функционального шаблона типы аргументов функции определяют, какая версия шаблона используется; то есть аргументы шаблона выводятся из аргументов вызова функции (§23.5.2).

Естественно, функциональный шаблон должен быть где-то определен (§23.7):

```
template<typename T>
void sort(vector<T>& v) // определение
    // Shell sort (Knuth, Vol. 3, pg. 84)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (v[j+gap]<v[j]) {   // свопируем v[j] и v[j+gap]
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

Сравните это определение с функцией **sort()**, определенной в §12.5. Данная шаблонизированная версия чище и короче, поскольку она может рассчитывать на дополнительную информацию о типе подлежащих сортировке элементов. В типичном случае она и быстрее, поскольку не полагается на указатель на функцию для сравнения. Это означает, что никаких косвенных обращений нет и что встраивание простой операции < легко выполнить.

Дальнейшее упрощение состоит в том, чтобы использовать шаблон **swap()** (§35.5.2) из стандартной библиотеки ради упрощения кода до его натурального вида:

```
if (v[j+gap]<v[j])
swap(v[j], v[j+gap]);
```

При этом никаких дополнительных накладных расходов не возникает. Более того, **swap()** из стандартной библиотеки использует перемещающую семантику, так что можно будет наблюдать и ускорение (§35.5.2).

В этом примере операция < используется для сравнения. Однако, не у каждого типа есть такая операция. Это лимитирует использование данной версии функции **sort()**, но это легко преодолеть добавлением аргумента (§25.2.3). Например:

```
template<typename T, typename Compare = std::less<T>>
void sort(vector<T>& v)              // опеределение
    // Shell sort (Knuth, Vol. 3, pg. 84)
{
    Compare cmp;                     // создаем умолчательный объект Compare
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (cmp(v[j+gap], v[j]))
                    swap(v[j], v[j+gap]);
}
```

Теперь мы можем сортировать, используя умолчательную версию операции сравнения (<), или можно предоставить свою версию операции:

```
struct No_case {
    bool operator()(const string& a, const string& b) const; // сравнение без учета регистра
};
```

```
void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);                              // sort(vector<int>&)
    sort<int, std::greater<int>>(vi);      // sort(vector<int>&) применяя greater

    sort(vs);                              // sort(vector<str ing>&)
    sort<string, No_case>(vs);             // sort(vector<str ing>&) применяя No_case
}
```

К сожалению, правило, гласящее, что можно специфицировать только находящиеся сзади аргументы, заставляет нас специфицировать (а не выводить) тип элементов, когда мы специфицируем операцию сравнения.

Явная спецификация аргументов функциональных шаблонов разъясняется в §23.5.2.

23.5.1. Аргументы функциональных шаблонов

Функциональные шаблоны чрезвычайно важны для написания обобщенных алгоритмов, применимых к широкому набору контейнерных типов (§3.4.2, §32.2). Возможность вывода аргументов шаблона из аргументов вызова функции критически важна.

Компилятор может выводить аргументы-типы и нетиповые аргументы исходя из вызова при условии, что список аргументов функции уникально идентифицирует набор аргументов шаблона. Например:

```
template<typename T, int max>
struct Buffer {
    T buf[max];
public:
    // ...
};

template<typename T, int max>
T& lookup(Buffer<T, max>& b, const char* p);

Record& f(Buffer<string,128>& buf, const char* p)
{
    return lookup(buf, p);          // используем lookup() где T есть string и i есть 128
}
```

Здесь **T** для **lookup()** выводится так, что это **string**, а **max** выводится как **128**.

Заметьте, что параметры классовых шаблонов никогда не выводятся. Причина состоит в том, что гибкость, которую предоставляет множество конструкторов класса, делает этот вывод невозможным в большинстве случаев и весьма туманным во всех остальных. Вместо этого специализация (§25.3) обеспечивает механизм для неявного выбора между альтернативными определениями шаблона. Если нам требуется создать объект выводимого типа, мы часто можем сделать это, вызвав функцию, которая и выполнит вывод (и создание). Например, рассмотрим простой вариант для функции **make_pair()** (§34.2.4.1) из стандартной библиотеки:

```
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 a, T2 b)
{
    return {a, b};
}

auto x = make_pair(1,2);                        // x есть pair<int, int>
auto y = make_pair(string("New York"),7.7);     // y есть pair<str ing, double>
```

Если аргумент шаблона не может быть выведен из аргументов функции (§23.5.2), мы должны специфицировать его явным образом. Это делается точно так же, как у шаблонных классов явно специфицируются аргументы (§25.2, §25.3). Например:

```
template<typename T>
T* create();                    // создаем T и возвращаем указатель на него

void f()
{
    vector<int> v;              // класс, аргумент шаблона int
    int* p = create<int>();     // функция, аргумент шаблона int
    int* q = create();          // ошибка: невозможно вывести аргумент шаблона
}
```

Использование явной спецификации для определения возврата шаблонной функции очень распространено на практике. Это позволяет нам определять семейства функций для создания объектов (вроде функции **create()**) и функций приведения типа (см., например, §27.2.2). Синтаксис для **static_cast**, **dynamic_cast** и т.д. (§11.5.2, §22.2.1) соответствует синтаксису явно квалифицируемых функциональных шаблонов.

Умолчательные аргументы шаблонов можно использовать для упрощения явной квалификации в некоторых случаях (§25.2.5.1).

23.5.2. Вывод аргументов для функциональных шаблонов

Компилятор может вывести тип аргумента шаблона, **T** или **TT**, или нетиповой аргумент шаблона, **I**, из аргументов вызова шаблонной функции с типами, составленными из следующих конструкций (§ iso.14.8.2.1):

T	**const T**	**volatile T**
T*	**T&**	**T[constant_expression]**
type[I]	**class_template_name<T>**	**class_template_name<I>**
TT<T>	**T<I>**	**T<>**
T type::*	**T T::***	**type T::***
T (*)(args)	**type(T::*)(args)**	**T(type::*)(args)**
type (type::*)(args_TI)	**T (T::*)(args_TI)**	**type(T::*)(args_TI)**
T (type::*)(args_TI)	**type (*)(args_TI)**	

Здесь **args_TI** — это список параметров, из которого **T** или **I** могут быть определены посредством рекурсивного применения правил, а **args** — это список параметров, не допускающий вывода. Если не получается вывести все параметры указанным способом, то вызов считается неоднозначным. Например:

```
template<typename T, typename U>
void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
    f(p, g);    // T есть char, U есть int
    f(p, h);    // ошибка: невозможно вывести U
}
```

Аргументы первого вызова **f()** легко позволяют вывести аргументы шаблона. Для второго вызова **f()** мы видим, что **h()** не соответствует образцу **U(*)(U)**, поскольку аргумент у **h()** и тип возврата разные.

Если параметр шаблона можно вывести из более чем одного аргумента функции, то результатом должен быть одинаковый тип при каждом выводе. Иначе вызов считается ошибочным. Например:

```
template<typename T>
void f(T i, T* p);

void g(int i)
{
    f(i,&i);            // OK
    f(i,"Remember!");   // ошибка, неоднозначность: T есть int или T есть const char?
}
```

23.5.2.1. Вывод ссылок

Бывает полезно иметь разные действия, предпринимаемые для величин категории lvalue и rvalue. Рассмотрим класс, который содержит пары {integer, pointer} ({целое, указатель}):

```
template<typename T>
class Xref {
public:
    Xref(int i, T* p)                          // храним указатель: Xref владелец
        :index{i}, elem{p}, owner{true}
    {}

    Xref(int i, T& r)                          // храним указатель на r, владеет кто-то другой
        :index{i}, elem{&r}, owner{false}
    {}

    Xref(int i, T&& r)                         // перемещаем r в Xref, Xref владелец
        :index{i}, elem{new T{move(r)}}, owner{true}
    {}

    ~Xref()
    {
        if(owned) delete elem;
    }
    // ...
private:
    int index;
    T* elem;
    bool owned;
};
```

Так что:

```
string x {"There and back again"};

Xref<string> r1 {7,"Here"};                    // r1 владеет копией string{"Here"}
Xref<string> r2 {9, x};                        // r2 просто ссылается на x
Xref<string> r3 {3, new string{"There"}};      // r3 владеет string{"There"}
```

Здесь для **r1** выбирается **Xref(int, string&&)**, поскольку **x** это rvalue. Аналогично, для **r2** выбирается **Xref(int, string&)**, поскольку **x** это lvalue.

При выводе аргументов шаблона lvalue и rvalue различаются следующим образом: lvalue типа **X** выводится как **X&**, а rvalue — как **X**. Это отличается от связывания величин с нешаблонными аргументами-ссылками категории rvalue (§12.2.1), но это особенно полезно при продвижении аргументов (§35.5.1). Напишем «фабричную» функцию, которая создает **Xref** в свободной памяти и возвращает указатели типа **unique_ptr** на них:

```
template<typename T>
    T&& std::forward(typename remove_reference<T>::type& t) noexcept; // §35.5.1
template<typename T>
    T&& std::forward(typename remove_reference<T>::type&& t) noexcept;

template<typename TT, typename A>
unique_ptr<TT> make_unique(int i, A&& a)  // простой вариант make_shared (§ 34.3.2)
{
    return unique_ptr<TT>{new TT{i, forward<A>(a)}};
}
```

Мы хотим, чтобы **make_unique<T>(arg)** конструировало **T** из **arg** без размножения копий. Для этого важно, чтобы поддерживалось различение lvalue и rvalue. Рассмотрим следующее:

```
auto p1 = make_unique<Xref<string>>(7,"Here");
```

Здесь **"Here"** — это rvalue, и вызывается **forward(string&&)**, передавая rvalue, так что **Xref(int, string&&)** вызывается для перемещения из **string** содержимого **"Here"**.

А вот более интересный (тонкий) случай:

```
auto p2 = make_unique<Xref<string>>(9, x);
```

Здесь **x** это lvalue, так что вызывается **forward(string&)**, передавая lvalue: **T** для **forward()** выводится как **string&**, и что возврат имеет тип **string& &&**, что означает **string&** (§7.7.3). Таким образом, **Xref(int, string&)** вызывается с lvalue **x**, и при этом **x** копируется.

К сожалению, **make_unique()** не являются частью стандартной библиотеки, хотя и широко поддерживается на практике. Нетрудно определить **make_unique()** с произвольным числом аргументов, опираясь на шаблон с переменным числом параметров для продвижения (forwarding) (§28.6.3).

23.5.3. Перегрузка шаблонных функций

Мы можем объявить несколько функциональных шаблонов с одним и тем же именем, или даже можем объявить комбинацию из шаблонных и обычных функций. Когда перегруженная функция вызывается, требуется разрешение перегрузки для нахождения правильной функции или функционального шаблона, подлежащего вызову. Например:

```
template<typename T>
    T sqrt(T);
template<typename T>
    complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
    sqrt(2);    // sqrt<int>(int)
    sqrt(2.0); // sqrt(double)
    sqrt(z);    // sqrt<double>(complex<double>)
}
```

В том же ключе, в котором функциональный шаблон является обобщением понятия функции, правила для разрешения перегрузки в присутствии функциональных шаблонов обобщаются на этот случай. В основе, для каждого шаблона мы ищем специализацию, наилучшую для заданного набора аргументов. Затем мы применяем обычные правила разрешения перегрузки функций к этим специализациям и обычным функциям (§ iso.14.8.3):

1. Находим набор специализаций функционального шаблона (§23.2.2), которые примут участие в разрешении перегрузки. Это делается рассмотрением каждого функционального шаблона, решая, какие именно аргументы будут использоваться, если в текущей области видимости нет других функций или функиональных шаблонов с тем же самым именем. Для вызова **sqrt(z)** это приводит к тому, что **sqrt<double>(complex<double>)** и **sqrt<complex<double>>(complex<double>)** становятся кандидатами. См. также §23.5.3.2.
2. Если два функциональных шаблона могут быть вызваны и один из них более специализированный, чем другой (§25.3.3), то в дальнейших шагах нужно будет рассматривать лишь более специализированный вариант. Для вызова **sqrt(z)** это означает, что вариант **sqrt<double>(complex<double>)** будет иметь предпочтение перед вариантом **sqrt<complex<double>>(complex<double>)**: любой вызов, соответствующий **sqrt<T>(complex<T>)** также будет соответствовать и **sqrt<T>(T)**.
3. Выполняется разрешение перегрузки для отобранного набора функций, плюс обычные функции, для которых выполняется все как для обычных функций (§12.3). Если аргумент функционального шаблона был определен посредством вывода аргумента (§23.5.2), то этот аргумент не может продвигаться, не подвергается стандартным преобразованиям или преобразованиям, определяемым пользователем. Для **sqrt(2)**, точным соответствием будет **sqrt<int>(int)**, так что этот вариант имеет предпочтение перед **sqrt(double)**.
4. Если функция и специализация одинаково подходят, то предпочтение отдается функции. Следовательно, **sqrt(double)** будет выбрано вместо **sqrt<double>(double)** для вызова **sqrt(2.0)**.
5. Если не обнаружены никакие соответствия, то вызов считается ошибочным. Если мы приходим к нескольким одинаково подходящим вариантам, то вызов неоднозначный и это тоже ошибка.

Пример:

```
template<typename T>
T max(T, T);

const int s = 7;

void k()
{
    max(1,2);       // max<int>(1,2)
    max('a','b');   // max<char>('a','b')
    max(2.7,4.9);   // max<double>(2.7,4.9)
    max(s,7);       // max<int>(int{s},7) (тривиальная конверсия)

    max('a',1);     // ошибка: неоднозначно: max<char, char>() или max<int, int>()?
    max(2.7,4);     // ошибка: неоднозначно: max<double, double>() или max<int, int>()?
}
```

Проблема с двумя последними вызовами состоит в том, что мы не применяем продвижения и стандартные преобразования после того, как шаблонные параметры были уникальным образом определены. Не существует правила, по которому компилятор мог бы предпочесть один вариант разрешения другому. В большинстве случаев это правильно, что компилятор не допускает тонких вариантов решения. Альтернативой ошибкам неоднозначности были бы неоднозначные результаты, к которым приводят неожиданные варианты разрешения. Интуитивные предпочтения разных программистов в разрешении перегрузки столь различаются друг от друга, что на базе интуитивности можно было бы строить совершенно уж интуитивные правила разерешения перегрузки.

23.5.3.1. Разрешение неоднозначности

Мы могли бы разрешить две неоднозначности явной квалификацией:

```
void f()
{
    max<int>('a',1);          // max<int>(int('a'),1)
    max<double>(2.7,4);       // max<double>(2.7, double(4))
}
```

Альтернативным образом мы могли бы добавить подходящие объявления:

```
inline int max(int i, int j) {return max<int>(i, j);}
inline double max(int i, double d) {return max<double>(i, d);}
inline double max(double d, int i) {return max<double>(d, i);}
inline double max(double d1, double d2) {return max<double>(d1, d2);}

void g()
{
    max('a',1);      // max(int('a'),1)
    max(2.7,4);      // max(2.7,4)
}
```

Для обычных функций применяются обычные правила разрешения перегрузки (§12.3), и применение ключевого слова **inline** не приводит ни к каким дополнительным накладным расходам.

Определение для **max()** тривиально, так что мы могли бы реализовать сравнение непосредственно, а не вызовом специализаций **max()**. Однако, применение явной специализации шаблона является весьма простым способом разрешения и помогает избегать в программе повторов практически одинакового кода.

23.5.3.2. Неудачная подстановка аргументов

В процессе поиска наилучшего варианта совпадения для заданного набора аргументов функционального шаблона компилятор рассматривает вопрос о том, можно ли использовать аргумент способом, требуемым полным объявлением функционального шаблона (включая тип возврата). Например:

```
template<typename Iter>
typename Iter::value_type mean(Iter first, Iter last);

void f(vector<int>& v, int* p, int n)
{
    auto x = mean(v.begin(), v.end()); // OK
    auto y = mean(p, p+n); // ошибка
}
```

Здесь инициализация **x** проходит успешно, поскольку аргументы имеют соответствие и **vector<int>::iterator** имеет член, названный **value_type**. А вот инициализация **y** не проходит, поскольку несмотря на найденное соответствие аргументов, **int*** не имеет члена **value_type**, так что нельзя написать

```
int*::value_type mean(int*, int*);        // y int* нет члена value_type
```

Однако, а что если есть иное определение для **mean()**?

```
template<typename Iter>
typename Iter::value_type mean(Iter first, Iter last);     // #1

template<typename T>
T mean(T*, T*);      // #2
```

```
void f(vector<int>& v, int* p, int n)
{
    auto x = mean(v.begin(), v.end());      // OK: вызов #1
    auto y = mean(p, p+n);                  // OK: вызов #2
}
```

Это работает: обе инициализации проходят. Но почему мы не получили ошибку, когда пытались поставить в соответствие **mean(p, p+n)** первому определению шаблона? Аргументы соответствуют идеально, но подстановка в истинный аргумент шаблона (**int***) приводит к объявлению функции:

```
int*::value_type mean(int*, int*);    // у int* нет члена value_type
```

Это, естественно, просто мусор: у указателя нет члена **value_type**. По счастью, само по себе, рассмотрение возможного объявления не является ошибкой. Есть правило языка (§ iso.14.8.2), гласящее, что такая проблема с подстановкой не является ошибкой. В результате шаблон игнорируется; то есть шаблон не вносит свою специализацию в набор для разрешения перегрузки. Таким образом, **mean(p, p+2)** соответствует объявлению номер 2, которое и вызывается.

Без правила «ошибка в подстановке не есть причина для прекращения работы» мы получили бы ошибку компиляции, несмотря на то, что есть свободные от ошибок альтернативы (такие как #2). Более того, это правило дает нам в руки инструмент для выбора среди шаблонов. Техника, базирующаяся на этом правиле, описывается в §28.4. В частности, стандартная библиотека предоставляет **enable_if** для упрощения условных определений шаблонов (§35.4.2).

Таким образом, если в процесс генерирования функции-кандидата для разрешения функционального вызова компилятор обнаружит, что он генерирует специализацию, не имеющую смысла, такой кандидат не будет включаться в набор для разрешения перегрузки. Специализация шаблона считается бессмысленной, если она ведет к ошибке типа. В этом случае мы рассматриваем лишь объявление; определение шаблонной функции и определение членов класса не рассматриваются до тех пор, пока они не используются реально. Например:

```
template<typename Iter>
Iter mean(Iter first, Iter last)  // #1
{
    typename Iter::value_type = *first;
    // ...
}

template<typename T>
T* mean(T*, T*);                          // #2

void f(vector<int>& v, int* p, int n)
{
    auto x = mean(v.begin(), v.end());    // OK: вызов #1
    auto y = mean(p, p+n);                // ошибка: неоднозначно
}
```

Объявление **mean()** #1 корректно для вызова **mean(p, p+n)**. Компилятор не приступит к конкретизации тела **mean()** и устранит его из-за ошибки типа.

Здесь результатом является ошибка неоднозначности. Если бы **mean()** #2 не было, то было бы выбрано объявление #1 и мы бы натолкнулись на ошибку во время конкретизации. Итак, функция может быть отобрана в качестве наилучшего совпадения, хотя и будет затем страдать от ошибки компиляции.

23.5.3.3. Перегрузка и наследование

Правила разрешения перегрузки гарантируют, что функциональные шаблоны корректно взаимодействуют с наследованием:

```
template<typename T>
    class B {/* ... */};
template<typename T>
    class D : public B<T> {/* ... */};

template<typename T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb);      // f<int>(pb) of course
    f(pd);      // f<int>(static_cast<B<int>*>(pd));
                // использовано стандартное преобразование D<int>* в B<int>*
}
```

В этом примере функциональный шаблон **f()** принимает **B<T>** для любого типа **T**. Мы имеем аргумент типа **D<int>**, так что компилятор легко приходит к выводу, что выбирая для **T** тип **int**, вызов может быть однозначно разрешен в виде **f(B<int>*)**.

23.5.3.4. Перегрузка и невыводимые параметры

Аргумент функции, который не участвует в выводе параметра шаблона, трактуется в точности как аргумент нешаблонной функции. В частности, имеют место обычные правила преобразования типов. Рассмотрим следующее:

```
template<typename T, typename C>
T get_nth(C& p, int n);             // получаем n-ый элемент
```

Предположительно, данная функция возвращает величину **n**-го элемента контейнера типа **C**. Поскольку **C** должен выводиться из истинного аргумента вызова **get_nth()**, преобразования неприменимы к первому аргументу. Однако, второй аргумент обычный, так что рассматривается весь возможный диапазон преобразований. Например:

```
struct Index {
    operator int();
    // ...
};

void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth<int>(v,2);     // точное соответствие
    int i2 = get_nth<int>(v, s);    // стандарт. конверсия: short в int
    int i3 = get_nth<int>(v, i);    // пользовательская конверсия: Index в int
}
```

Часто такую нотацию называют *явной специализацией* (§23.5.1).

23.6. Синонимы для шаблонов

Мы можем определить синоним для типа, используя **using** или в варианте синтаксиса **typedef** (§6.5). Синтаксис с **using** является более общим в том отношении, что его можно использовать для синонима шаблона, часть параметров которого связана. Рассмотрим следующее:

```
template<typename T, typename Allocator = allocator<T>> vector;

using Cvec = vector<char>;              // оба аргумента связаны

Cvec vc = {'a', 'b', 'c'};              // vc есть vector<char, allocator<char>>

template<typename T>
using Vec = vector<T, My_alloc<T>>; // vector использует my allocator (2-ой аргумент связан)

Vec<int> fib = {0, 1, 1, 2, 3, 5, 8, 13}; // fib есть vector<int, My_alloc<int>>
```

В общем случае, если мы свяжем все аргументы шаблона, мы получим тип, а если мы свяжем лишь часть параметров, то мы имеем шаблон. Отметим, что то, что мы получаем при помощи **using**, всегда является синонимом. То есть, когда мы используем синоним, это абсолютно эквивалентно применению оригинального шаблона. Например:

```
vector<char, alloc<char>> vc2 = vc;          // vc2 и vc одного типа
vector<int, My_alloc<int>> verbose = fib;    // verbose и fib одного типа
```

Эквивалентность синонима и оригинального шаблона предполагает, что если специализируется шаблон, то вы (корректно) получаете специализацию при использовании синонима. Например:

```
template<int>
struct int_exact_traits {  // идея: int_exact_traits<N>::type есть тип точно в N бит
    using type = int;
};

template<>
struct int_exact_traits<8> {
    using type = char;
};

template<>
struct int_exact_traits<16> {
    using type = short;
};

template<int N>
using int_exact = typename int_exact_traits<N>::type; // синоним для удобства нотации

int_exact<8> a = 7;       // int_exact<8> есть int в 8 бит
```

Если специализации не используются через синоним, то мы не сможем утверждать, что **int_exac** это просто синоним для **int_exac_traits<N>::type**; они будут вести себя поразному. С другой стороны, вам не удастся определить специализацию синонима. Если бы такое было разрешено, то это могло сбить с толку человека, читающего код, который мог не понять, что именно специализировалось.

23.7. Организация исходного кода

Есть три разумных варианта организации кода, использующего шаблоны:

1. Включить определения шаблонов в код до того, как они будут использоваться в единице трансляции.
2. Включить объявления шаблонов (и только) перед их использованием в единице трансляции. Включить определения шаблонов позже в единице трансляции (потенциально позже их использования).

3. Включить объявления шаблонов (и только) перед их использованием в единице трансляции. Определить шаблоны в другой единице трансляции.

По техническим причинам опция номер 3 не реализуется. На настоящий момент чаще всего применяется подход с включением (обычно посредством **#include**) определений шаблонов в каждую единицу трансляции, где они используются, с надеждой на компилятор в плане оптимизации кода и на его способности устранить дублирование в объектном коде. Например, я мог бы предоставить шаблон **out()** в заголовочном файле **out.h**:

```
// файл out.h:

    #include<iostream>

    template<typename T>
    void out(const T& t)
    {
        std::cerr << t;
    }
```

Мы будем включать этот заголовочный файл всюду, где требуется **out()**. Например:

```
// файл user1.cpp:

    #include "out.h"
    // используется out()
```

и

```
// файл user2.cpp:

    #include "out.h"
    // используется out()
```

Таким образом, определение **out()** и все объявления, от которых оно зависит, включаются в несколько разных единиц компиляции. От компилятора зависит, когда генерировать код и как избавляться от избыточных определений. В такой стратегии шаблонные функции трактуются так же, как и встраиваемые функции.

У этой стратегии есть очевидная проблема, когда пользователи неожиданно могут оказаться в зависимости от объявлений, включенных исключительно ради определения **out()**. Проблему можно смягчить, воспользовавшись подходом [2] «включить определение шаблона позже», используя пространства имен, избегая макросов и, вообще, уменьшая общий объем включаемой информации. В идеале, хорошо минимизировать зависимость определения шаблона от его окружения.

В рамках объявленного подхода для нашего простого примера с **out()**, расщепим файл **out.h** на два файла. Объявления перемещаются в **.h**-файл:

```
// файл outdecl.h:

    template<typename T>
    void out(const T& t);
```

Определение перемещается в файл **out.cpp**:

```
// файл out.cpp:

    #include<iostream>

    template<typename T>
```

```
void out(const T& t)
{
    std::cerr << t;
}
```

А пользователь теперь включает оба файла:

// файл user3.cpp:

```
#include "out.h"
// используется out()
#include "out.cpp"
```

Это минимизирует шансы на существование нежелательных эффектов от реализации шаблонов на пользовательский код. Однако, при этом увеличиваются шансы на то, что что-то из пользовательского кода (например, макросы) окажет нежелательное воздействие на определения шаблонов.

Как обычно, невстраиваемые, нешаблонные функции и статические члены (§16.2.12) должны иметь уникальное определение в единице компиляции. Это подразумевает, что такие члены лучше не включать в шаблоны, подлежащие включению во вножество единиц трансляции. Как мы видели в примере с **out()**, определения шаблонных функций могут реплицироваться в разных единицах трансляции, так что остерегайтесь контекста, который может тонким образом влиять на смысл определений:

// файл user1.cpp:

```
#include "out.h"
// используется out()
```

и

// файл user4.cpp:

```
#define std MyLib
#include "out.c"
// используется out()
```

Неряшливое и чреватое ошибками использование макросов меняет определение **out** так, что определение в файле **user4.cpp** отличается от определения из файла **user1.cpp**. Это, конечно, ошибка, но такую ошибку реализация может и не обнаружить. Этот тип ошибки очень трудно обнаружить в большой программе, так что старайтесь минимизировать контекстные зависимости шаблонов и критически относитесь к макросам (§12.6).

Если требуется больший контроль над контекстом конкретизации, можете использовать явную конкретизацию и **extern template** (§26.2.2).

23.7.1. Компоновка

Правила компоновки шаблонов есть правила для компоновки сгенерированных классов и функций (§15.2, §15.2.3). Это предполагает, что если раскладка в памяти классового шаблона изменяется или изменяется определение встраиваемой шаблонной функции, то весь код, который использует этот класс или функцию, должен перекомпилироваться.

Для шаблонов, определенных в заголовочных файлах и включаемых «куда угодно», это повлечет огромный объем перекомпиляции, поскольку имеется тенденция вовлечения шаблонами больших объемов информации в заголовочные файлы. Если используются динамические библиотеки, то дополнительно нужно следить за тем, чтобы все случаи использования шаблона согласовывались друг с другом.

Иногда, бывает возможность уменьшить зависимость от изменений в больших и сложных библиотеках шаблонов, инкапсулируя их использование в функциях с нешаблонными интерфейсами. К примеру, я мог бы организовать некоторые вычисления, используя общую библиотеку для вычислений, поддерживающую широкий набор типов (Глава 29, §40.4, §40.5, §40.6). Но я часто знаю тип, вовлекаемый в вычисления. Например, в своей программе я систематически использую типы **double** и **vector<double>**. В этом случае я мог бы определить:

```
double accum(const vector<double>& v)
{
    return accumulate(v.begin(), v.end(),0.0);
}
```

Теперь я могу использовать простое нешаблонное объявление для **accum()** в моем коде:

```
double accum(const vector<double>& v);
```

Зависимость от **std::accumulate()** исчезла в направлении файла.**cpp**, что не видно в остальной части кода. Кроме того, я «страдаю» от избыточного времени компиляции (из-за **#include<numeric>**) тоже только в этом.**cpp** файле.

Отметим, что я упростил интерфейс к **accum()** по сравнению с **std::accumulate()**. Общность, являющаяся ключевым свойством хорошей библиотеки шаблонов, может приводить к сложностям в конкретном приложении.

23.8. Советы

[1] Используйте шаблоны для реализации алгоритмов, применимых ко многим типам аргументов; §23.1.

[2] Для формирования контейнеров используйте шаблоны; §23.2.

[3] Заметьте, что **template<class T>** и **template<typename T>** — суть синонимы; §23.2.

[4] Определяя шаблон, сначала разработайте и протестируйте нешаблонную версию; затем обобщите ее добавлением параметров; §23.2.1.

[5] Шаблоны безопасны по типам, но проверка часто осуществляется слишком поздно; §23.3.

[6] Проектируя шаблон, тщательно рассмотрите требования к аргументам шаблона; §23.3.

[7] Если классовый шаблон должен копироваться, придайте ему нешаблонный копирующий конструктор и нешаблонную операцию копирующего присваивания; §23.4.6.1.

[8] Если классовый шаблон должен перемещаться, придайте ему нешаблонный перемещающий конструктор и нешаблонную операцию перемещающего присваивания; §23.4.6.1.

[9] Виртуальная функция-член не может быть шаблонной функцией-членом; §23.4.6.2.

[10] Определяйте тип в виде члена шаблона, только если он зависит от всех аргументов классового шаблона; §23.4.6.3.

[11] Используйте функциональные шаблоны для вывода типов аргументов классового шаблона; §23.5.1.

[12] Перегружайте функциональные шаблоны для получения семантики, одинаковой для множества типов аргументов; §23.5.3.

[13] Используйте неудачу в подстановке аргументов для обеспечения правильного набора функций в программе; §23.5.3.2.

[14] Применяйте синонимы шаблонов для упрощения нотации и сокрытия деталей реализации; §23.6.

[15] Не существует раздельной компиляции шаблонов; включайте определения шаблонов в каждую единицу трансляции, где они используются; §23.7.

[16] Используйте обычные функции в качестве интерфейсов к коду, который не может иметь дело с шаблонами; §23.7.1.

[17] Раздельно компилируйте большие шаблоны и шаблоны с нетривиальными контекстными зависимостями; §23.7.

24

Обобщенное программирование

Настало время положить вашу работу
на твердый теоретический базис.
— Сэм Морган

- Введение
- Алгоритмы и обобщение
- Концепции

  Исследование концепций; Концепции и ограничения
- Делаем концепции конкретными

  Аксиомы; Многоаргументные концепции; Концепции-значения; Проверки ограничений; Проверка определения шаблона
- Советы

24.1. Введение

Для чего существуют шаблоны? Иными словами, какие приемы программирования раскрывают свою эффективность в случае использования шаблонов? Шаблоны предлагают:

- Возможность передавать типы (вместе с величинами и шаблонами) как аргументы без потери информации. Это обеспечивает отличные перспективы для встраивания, чем и пользуются современные реализации.
- Отложенную проверку типов (только во время конкретизации). Это помогает сопрячь информацию из разных контекстов.
- Возможность передачи константных значений в качестве аргументов. Это открывает возможность вычислений на этапе компиляции.

Другими словами, шаблоны обеспечивают мощный механизм вычислений на этапе компиляции и манипуляцию типами, которая может приводить к очень компактному и эффективному коду. Помните, что типы (классы) могут содержать как код, так и данные.

Самое первое и чаще всего используемое применение шаблонов состоит в поддержке *обобщенного программирования*, то есть программирования, сфокусированного на дизайне, реализации и использовании общих алгоритмов. Здесь слово «общий» означает, что алгоритм разрабатывается так, чтобы он мог использовать множество типов, пока они удовлетворяют требованиям алгоритма к его аргументам. В языке C++ шаблоны являются основным средством поддержки обобщенного программирования. Говорят, что шаблоны обеспечивают (на этапе компиляции) параметрический полиморфизм.

Существует множество определений обобщенного программирования. Так что этот термин неоднозначен. Тем не менее, в контексте языка C++ обобщенное программирование фокусируется на дизайне общих алгоритмов, используя для этого шаблоны.

Когда в программировании фокусируются на технике генерации (типов и функций) и опираются на функции типа для реализации вычислений на этапе компиляции, то все это называют шаблонным метапрограммированием, что является предметом рассмотрения Главы 28.

Проверка типов у шаблонов используется для проверки использования аргументов, а не для проверки явного интерфейса (объявление шаблона). При этом мы оперируем над величинами в присутствии определения операции. Значения «живут» в объектах и так объекты работают в языке C++; поэтому исключительно подходящие для объектов значения принимаются в рассмотрение. Но на этапе компиляции с применением шаблонов выполняется лишь то, что зависит от значений, а не от объектов. На этапе компиляции нет переменных. Таким образом, шаблонное программирование напоминает программирование в динамически типизируемых языках, но стоимость вычислений на этапе компиляции нулевая. Ошибки, которые обнаруживают себя в динамических языках как исключения в языке C++ превращаются в ошибки компиляции.

Ключевым аспектом обобщенного программирования, метапрограммирования и вероятно всех случаев использования шаблонов является тот факт, что при этом совершенно однородно манипулируют как встроенными типами, так и типами, определяемыми пользователем. Например, функция **accumulate()** не заботится о том, каков тип добавляемых величин: **int**, **complex<double>** или **Matrix**. Что единственно волнует эту функцию, так это вопрос о том, есть для них операция **+**. Применение типа в качестве аргумента шаблона не предполагает его присутствие в какой либо иерархии типов или же возможность какого-либо механизма самоидентификации типа на этапе выполнения. Это очень важно для высокопроизводительных вычислений.

В настоящем разделе мы фокусируемся на двух аспектах обобщенного программирования:

- *Обобщение*: обобщение алгоритма вплоть до применения максимально возможного диапазона типов аргументов (§24.2), так чтобы лимитировать алгоритм лишь свойствами, имеющими принципиальное значение.
- *Концепции*: тщательное и точное специфицирование требований алгоритма к своим аргументам (§24.3)

24.2. Алгоритмы и обобщение

Функциональный шаблон является обобщением обычной функции в том смысле, что он в состоянии выполнять действия на множестве типов данных и использовать множество операций, передаваемых в качестве аргументов. Алгоритм — это процедура или формула для решения проблемы: конечное множество действий для получения финального результата. Таким образом, функциональный шаблон часто называют алгоритмом.

Как осуществить переход от функции, которая выполняет конкретные операции над заданными типами, к алгоритму, выполняющему более общие операции над множеством типов данных? Наиболее эффективным приемом служит обобщение от одного или нескольких конкретных примеров. Этот прием помогает следить за переходом от частного к общему, а также не забывать про эффективность и разумность обобщения. Когда начинают сразу с общих идей, то часто получают путаные и неэффективные, плохо поддающиеся пониманию сторонними программистами программы.

Я буду иллюстрировать процесс обобщения на одном конкретном примере. Рассмотрим следующее:

```
double add_all(double* array, int n)
    // конкретный алгоритм для массива элементов типа double
{
    double s {0};
    for (int i = 0; i<n; ++i)
        s = s + array[i];
    return s;
}
```

Очевидно, что здесь вычисляется сумма значений типа **double** из аргумента-массива. Также рассмотрим и такой код:

```
struct Node {
    Node* next;
    int data;
};

int sum_elements(Node* first, Node* last)
    // алгоритм для списка элементов типа int
{
    int s = 0;
    while (first!=last) {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

А здесь вычисляется сумма значений типа **int** из односвязанного списка, реализованного с помощью узлов типа **Node**.

Два этих фрагмента отличаются стилем и деталями, но опытный программист сразу скажет, «Это просто две реализации накопительного алгоритма». Это популярный алгоритм. Как и любой другой популярный алгоритм он имеет множество названий. Но давайте мы сейчас разработаем общий алгоритм, отталкиваясь от двух представленных выше примеров. Причем будем это делать поэтапно, чтобы прочувствовать ход процесса обобщения. Сначала попробуем абстрагироваться от типов данных, чтобы не зависеть от

- типов **double, int**
- от дилеммы массив/связанный список.

Для этого я напишу некоторый псевдокод:

```
// псевдокод:

T sum(data)
    // как-нибудь параметризовано типом значений и типом контейнера
{
    T s = 0
    while (not at end) {
        s = s + current value
        get next data element
    }
    return s
}
```

Чтобы это конкретизировать, нам нужно предоставить три операции для доступа к данным контейнерной структуры:

- Не в конце
- Получить текущее значение
- Получить следующий элемент данных

Для реальных данных нам потребуются еще три операции:

- Инициализировать в нуль
- Добавить
- Вернуть результат

Конечно, все это не совсем строго, но мы можем превратить это в следующий код:

```
// конкретный STL-подобный код

template<typename Iter, typename Val>
Val sum(Iter first, Iter last)
{
    Val s = 0;
    while (first!=last) {
        s = s + *first;
        ++first;
    }
    return s;
}
```

Здесь я использовал знания о том, как обычно в библиотеке STL представляют последовательности значений (§4.5). Последовательность представляется как пара итераторов, поддерживающих три операции:

- * для доступа к текущему значению
- ++ для перехода к следующему элементу
- != для сравнения итераторов, чтобы удостовериться, что мы не на конце последовательности

Теперь у нас есть алгоритм (шаблонная функция), которую можно использовать и для массивов, и для связанных списков; как для **int**, так и для **double**. Пример с массивом работает тотчас же, поскольку **double*** является примером итератора:

```
double ad[] = {1,2,3,4};
double s = sum<double*>(ad, ad+4);
```

Если мы используем односвязанный список ручного изготовления, то мы должны предоставить для него итератор. После добавления небольшого количества операций, **Node*** становится итератором:

```
struct Node { Node* next; int data; };

Node* operator++(Node* p) { return p->next; }
int operator*(Node* p) { return p->data; }
Node* end(lst) { return nullptr; }

void test(Node* lst)
{
    int s = sum<int*>(lst, end(lst));
}
```

Я применяю **nullptr** в качестве конца итератора. Я использую явный аргумент шаблона (здесь это **<int>**), чтобы разрешить вызывающему коду специфицировать тип для аккумулирующей переменной.

Можно сказать, что то, что мы уже получили, является существенно более общим кодом, чем множество примеров из реальной жизни. Например, **sum()** будет работать для списков значений с плавающей запятой (любой точности), для массивов целых (всех диапазонов), а также для многих иных типов, таких как **vector<char>**. Важно также, что **sum()** столь же эффективен, как и те функции, с которых мы стартовали в самом начале. Нам не нужна общность ценой потери эффективности.

Опытный программист заметит, что можно обобщать **sum()** и далее. В частности, использование лишнего аргумента шаблона является неуклюжим, и нам требуется начальное значение **0**. С этим можно справиться, если позволить вызывающему коду предоставлять начальное значение, а потом выводить **Val**:

```
template<typename Iter, typename Val>
Val accumulate(Iter first, Iter last, Val s)
{
    while (first!=last) {
        s = s + *first;
        ++first;
    }
    return s;
}

double ad[] = {1,2,3,4};
double s1 = accumulate(ad, ad+4,0.0);      // аккумулируем в double
double s2 = accumulate(ad, ad+4,0);        // аккумулируем в int
```

Но почему +? Нам может потребоваться и перемножать элементы. Фактически, нам нужен некоторый (небольшой) набор разумных операций над элементами последовательности. В итоге мы приходим к дальнейшему обобщению:

```
template<typename Iter, typename Val, typename Oper>
Val accumulate(Iter first, Iter last, Val s, Oper op)
{
    while (first!=last) {
        s = op(s,*first);
        ++first;
    }
    return s;
}
```

Здесь мы используем аргумент **opt** для комбинирования значений элементов в рамках нашего аккумулятора. Пример:

```
double ad[] = {1,2,3,4};
double s1 = accumulate(ad, ad+4,0.0, std::plus<double>);          // как ранее
double s2 = accumulate(ad, ad+4,1.0, std::multiply<double>);
```

Стандартная библиотека предоставляет общие операции, такие как **plus** и **multiply** (умножить), как объекты-функции, применяемые в качестве аргументов. Стандартная библиотека предлагает дальнейшее обобщение **accumulate()**, которое позволяет пользователю предоставить альтернативу = для комбинирования результата «сложения» с аккумулятором (§40.6).

Конечно, процесс обобщения не прост. Его важно продумывать с учетом конкретных особенностей задачи (предметной области). Алгоритмы стандартной библиотеки получены в результате процесса обобщения, который делался с огромным вниманием к проблемам производительности.

24.3. Концепции

Каковы требования алгоритмов к своим аргументам? Другими словами, что шаблонный код ожидает от аргументов? Или наоборот, что должны обеспечить типы, чтобы стать приемлемыми в качестве аргументов шаблоном? Возможности вариативности безграничны, поскольку мы можем строить классы и шаблоны с произвольными свойствами, например:

- Тип обеспечивает операцию -, но не операцию +
- Тип предоставляет копирующие операции, но не предоставляет операции перемещения
- Тип с копирующими операциями, не выполняющими копирования (§17.5.1.3)
- Тип с операцией ==, которая выполняет сравнение, а другие типы делают это с помощью функции **compare()**
- Тип определяет сложение как функцию-член **plus()**, а другие типы осуществляют то же самое при помощи глобальной функции **operator+()**

Здесь-то и возникает хаос. Если кадый класс обладает уникальным интерфейсом, то становится трудно писать шаблоны, принимающие множество разных типов. И наоборот, если у шаблона уникальные требования к аргументам, то становится трудно определять типы, которые допустимы для шаблона. Приходится отслеживать и принимать во внимание множество интерфейсов; это не так уж сложно для небольшой программы, но неподъемно для реальных библиотек и программ. Нам нужно определить небольшое число *концепций* (наборов требований), которые можно было бы применить ко многим шаблонам и многим типам данных.

24.3.1. Исследование концепций

В качестве примера рассмотрим классовый шаблон **String** из §23.2:

```
template<typename C>
class String {
    // ...
};
```

Что требуется от типа **X**, так это чтобы его можно было использовать как аргумент для **String: String<X>**? С более общей точки зрения вопрос заключается в следующем: что значит быть символом в этом классе строк? Опытный программист и дизайнер даст на этот вопрос несколько ответов и начнет процесс разработки на их основе. Но мы сейчас попробуем дать ответ, исходя из общих принципов. Нам придется пройти через три стадии анализа:

1. Сначала мы рассмотрим нашу реализацию и выявим, какие свойства (операции, функции, члены-типы и т.д.) она использует исходя из типов параметров (а также смысла этих операций). Результирующий список составит минимальные требования для данной конкретной реализации.
2. Затем мы обратим внимание на возможные альтернативные реализации и составим список для них. Исходя из анализа списка, мы можем наложить более строгие ограничения на аргументы шаблона, чтобы учесть альтернативные реализации. Можно предпочесть реализацию, которая накладывает меньшее количество требований (или более простые требования).
3. Наконец, мы рассматриваем результирующий список (или списки) требуемых свойств и сравниваем их с теми, которрые мы используем для других шаблонов.

Постараемся найти простое, предпочтительно общее описание концепций, альтернативных огромным спискам требований к аргументам. Здесь фокус состоит в том, что мы пробуем извлечь пользу от общей работы по классификации. Результирующим концепциям можно назначить смысловые имена, которые также и легче запомнить. Желательно, чтобы они максимизировали взаимозаменяемость шаблонов и типов.

Первые два шага — по фундаментальным причинам — аналогичны тому, как мы ведем обобщение конкретных алгоритмов до их обобщенных версий (§24.2). Последний шаг противоречит соблазну предоставить каждому алгоритму набор требований к аргументам, который в точности соответствует его реализации. Такие списки требований слишком специфицированы и нестабильны: любое изменение в реализации подразумевает изменения требований, документируемых как часть интерфейса алгоритма.

Для **String<C>** сначала рассмотрим операции, действительно выполняющиеся над **C** реализацией **String** (§19.3). Это и будет минимальным набором требований для этой реализации:

1. Тип **C** копируется копирующими присваиваниями и копирующей инициализацией.
2. **String** сравнивает **C** посредством == и !=.
3. **String** создает массивы из **C** (что предполагает умолчательное конструирование для **C**).
4. **String** берет адрес **C**.
5. Все **C** уничтожаются, когда уничтожается **String**.
6. У **String** есть операции >> и <<, которые тем или иным способом должны читать и писать **C**.

Требования 4 и 5 являются техническими требованиями, которые мы обычно предполагаем для всех типов данных, так что я не буду рассматривать типы, им не удовлетворяющие; такие типы являются, как правило, слишком заумными артефактами. Первое требование — что значение можно копировать — не выполняется для некоторых важных типов, таких как **std::unique_ptr**, представляющего реальные ресурсы (§5.2.1, §34.3.1). Однако, оно верно почти для всех «обычных типов», так что можно его затребовать. Копирующее присваивание влечет за собой семантику одинакового поведения двух копий. Кроме того, требуется, чтобы операция == реализовывала обычную семантику.

Из требования к присваиваниям вытекает, что константный тип не может использоваться в качестве аргумента шаблона. Например, **String<const char>** может и не работать. Наличие присваивания означает также, что алгоритм может использовать временные переменные типа его аргументов, создавать контейнеры объектов таких типов и т.д. Но не предполагается, что мы не можем применять **const** в спецификациях интерфейсов. Например:

```
template<typename T>
bool operator==(const String<T>& s1, const String<T>& s2)
{
    if (s1.size()!=s2.size()) return false;
    for (auto i = 0; i!=s1.size(); ++i)
        if (s1[i]!=s2[i]) return false;
    return true;
}
```

Для **String<X>** мы требуем, чтобы объекты типа **X** могли копироваться. Кроме того, посредством константных типов аргументов **operator==()** обещает, что ничего не пишется в элементы типа **X**.

Должны ли мы требовать перемещающих операций для **C**? В конце концов, мы обеспечиваем перемещающие операции для **String<C>**. Да, можно было бы потребовать, но это требование не существенно: то, что нам нужно делать с **C**, достигается посредством копирования, а если какие-либо копирования неявно превращаются в перемещение (например, когда мы возвращаем значение типа **C**), то тем лучше. Потенциально полезные примеры, вроде **String<String<char>>** будут работать прекрасно (корректно и эффективно) без явного требования перемещающих операций.

Ну, пока что все хорошо. Но последнее требование выглядит излишним. Нам на самом деле нужно читать и писать любой тип строк? Может лучше сказать, что если мы пишем и читаем **String<X>**, то тогда **X** должен предоставить операции **>>** и **<<**? То есть вместо того, чтобы навесить это требование на параметр **C** для шаблона **String**, мы сделаем это лишь для тех **String**, которые мы на самом деле читаем и пишем.

Это является фундаментальным выбором в процессе дизайна: мы можем наложить требование на аргументы классового шаблона, или же на шаблонные аргументы для конкретных функций-членов класса. Последнее является более гибким, но и более трудоемким (нам нужно выразить требования для каждой функции, которая в этом нуждается).

Оглядываясь на уже предложенные ограничения, я вспомнил о еще двух операциях, типичных для «обычных символов» в «обычных строках»:

1. Упорядочение (например, **<**).
2. Преобразование в целое значение.

После нашего начального анализа мы понимаем, какие из «хорошо известных концепций» (§24.3.2) затрагивают наши списки. Центральной концепцией для «обычных типов» является *регулярность*. Регулярный тип — это тип, который

- вы можете копировать (присваиванием или инициализацией) с надлежащей семантикой копирования (§17.5.1.3),
- вы может конструировать по умолчанию,
- не имеют проблем в связи мельчайшими техническими требованиями (например, взятие адреса переменной),
- вы можете сравнивать на эквивалентность (используя **==** и **!=**).

Это выглядит отличным выбором для аргументов нашего шаблона **String**. Кстати, все встроенные типы регулярные.

А теперь решим, разумно ли опускать упорядочение (**<**) для **String**? Рассмотрим для этого способы, какими мы используем строки. Желаемое использование шаблона (такого как **String**) должно определить его требования к аргументам. Строки используются интенсивно, часто выполняется сравнение строк, в том числе неявно при сортировке, строки помещаются в контейнеры и т.д. Ну и **string** из стандартной библиотеки предоставляет **<**. Заглядывать в стандарт помогает наметить правильный путь почти всегда. Таким образом, будем для нашего класса **String** требовать и **Regular**, и возможность сортировки. Такова будет наша концепция **Ordered**.

Вообще проблема сортировки непростая. Для целых чисел имеет место естественный порядок следования. Символы могут конвертироваться в целые значения, и такая сортировка (лексикографическая) имеет место. Но не все типы имеют естественный порядок следования, например комплексные числа или изображения. А есть типы с несколькими естественными вариантами упорядочения (записи можно сортировать по имени или по адресу). Наконец, некоторые типы не имеют возможности быть упорядоченными в принципе. Рассмотрим к примеру

```
enum class rsp { rock, scissors, paper };
```

Игра «ножницы-бумага» критически зависит от соотношений

- **scissors<rock**
- **rock<paper**
- **paper<scissors**

Наш строковый класс **String** не должен принимать любой тип в качестве символьного типа; только типы, поддерживающие строковые операции (сравнения, сортировки и ввода/вывода).

Добавление умолчательного конструктора и операций == и < к нашим требованиям к аргументам шаблона позволяет нам определить несколько полезных операций. Действительно, чем больше мы требуем от типов аргументов шаблона, тем легче реализатору шаблона обеспечивать ту или иную функциональность. С другой стороны, важно не перегружать шаблон требованиями, которые редко будут полезны на практике и лишь для специфических операций; это приводит к излишним нагрузкам на реализатора типов аргументов и лимитирует набор типов, которые можно будет использовать для аргументов. Таким образом, для **String<X>** мы требуем:

- **Ordered<X>**
- **>>** и **<<** для **X** (и только), если используются **>>** и **<<** для **String<X>**
- Конвертируемость в целое, если мы определяем и используем операцию конвертирования для **X**

До сих пор мы выражали наши требования к типу символов для **String** в терминах синтаксических свойств, таких как «**X** должен предоставлять копирующие операции, == и <». Дополнительно, мы должны еще потребовать, чтобы эти операции имели правильную семантику (традиционный смысл для операций). Иногда семантика влечет и определенные соотношения для операций. Например, для стандартной библиотеки имеем (§31.2.2.1):

- Результат копирования эквивалентен чему-либо, что было эквивалентно оригиналу (из **a==b** полагается **T{a}==T{b}**), и копия независима от своего источника (§17.5.1.3).
- Сравнение «на меньше» обеспечивает строгое слабое упорядочение (§31.2.2.1).

Семантика определена в английских текстах и в математике, но у нас нет возможности отразить семантику в языке C++ (см. §24.4.1). Семантика для стандартной библиотеки выражена на английском языке в ISO-стандарте.

24.3.2. Концепции и ограничения

Концепции не являются просто наборами требований. Большинство списков требований к типам (или список типов) не определяют согласованную и полезную концепцию. Чтобы быть полезной концепцией, список требований должен отражать «чаяния» набора алгоритмов или набора операций шаблонного класса. Во многих предметных областях люди разработали концепции, отражающие фундаментальные свойства этих областей. Их, к удивлению, не столь и много. Например, в алгебре есть концепции полей, колец и т.д., а в STL есть концепции прямого итератора, двунаправленного итератора или итератора произвольного доступа. Выработка концепций для предметных областей не есть быстро решаемая задача. Вы не можете ожидать новых концепций каждый год. Концепции, используемые в настоящей книге, описываются в §24.4.4.

Концепции являются исключительно общей идеей, которая в своей основе не привязана к шаблонам. Даже в древнем варианте языка C — в K&R C — идея о типе «знаковый интегральный тип» подавалась как обобщение идеи о целых значениях в памяти компьютера. Но для нас концепциями являются требования к аргументам шаблонов, так что наиболее важные вещи, связанные с концепциями, происходят именно в связи с шаблонами.

Я рассматриваю концепцию как тщательно проработанную идею, отражающую фундаментальные свойства предметной области приложения. Как следствие, не может быть

много концепций, и они служат источником рекомендаций при разработке алгоритмов и типов. Тут полная аналогия с вилками и розетками; мы хотим, чтобы их было как можно меньше, иначе нам придется немало заплатить за их «изучение на практике». В общем, с идеалом концепции все не просто; идеал может конфликтовать с идеалом, заставляющим иметь минимальные интерфейсы к классам (§16.2.3), а также с желанием многих программистов писать код так, как им хочется. Однако, добиться удобства как у розеток и вилок без определенных усилий по стандартизации не удастся.

Я устанавливаю барьер для концепции весьма высоко: я требую общности, определенной стабильности, применимости во многих алгоритмах, семантической согласованности и др. Многие простые требования к аргументам шаблонов, которые нам хотелось бы наложить, не являются концепциями согласно моей идее. Мы же не всегда разрабатываем шаблоны как обобщенные алгоритмы или как типы, предназначенные для широкого использования. Часто мы пишем шаблоны, которые являются просто деталями реализации, так что их требования к аргументам опираются на специфические условия применения этих шаблонов (и не на какие другие). То есть, это просто ограничения, а не концепции.

В качестве примера рассмотрим библиотеку, предназначенную для экспериментирования со стратегиями балансировки бинарных деревьев. Дерево будет принимать **Balance** в качестве аргумента:

```
template<typename Node, typename Balance>
struct node_base { // база сбалансированного дерева
    // ...
}
```

Этот простой класс балансировки предоставляет три операции на узлах. Например:

```
struct Red_black_balance {
    // ...
    template<typename Node> static void add_fixup(Node* x);
    template<typename Node> static void touch(Node* x);
    template<typename Node> static void detach(Node* x);
};
```

Очевидно, что мы бы хотели сказать, что требуется для аргументов **node_base**, но балансировщик не предназначен для широкого использования; предполагается его использование в качестве детали конкретной реализации сбалансированных деревьев. Для начинающих семантика балансировщика видится критически зависимой от семантики **Node**. В таких аспектах семантика балансировщика отличается от концепции (вроде **Random_access_iterator**). Мы, однако, можем по-прежнему использовать минимальную спецификацию для балансировщика — «предоставляет для узлов эти три функции» — как ограничения на аргументы у **node_base**.

Отметим, что «семантика» прокралась в дискуссию о концепциях. Если удается прописать разумную спецификацию семантики, то мы имеем концепцию. В противном случае у нас имеются просто ограничения, которые полезны, но не стоит ожидать от них стабильности и широкого использования.

24.4. Делаем концепции конкретными

К сожалению, в языке C++ отсутствуют конструкции для явного отражения концепций. Но обдумывание концепций в голове и внесение их в код в качестве комментариев не является идеальным решением. Компилятор не понимает комментариев, так что программист не получит помощи от компилятора; программист сам должен следить за ком-

ментариями и принимать их во внимание. И тем не менее, несмотря на отсутствие прямого отражения концепций в языке, мы можем строить код, который будет проверять во время компиляции свойства аргументов шаблонов.

Концепция — это предикат, то есть мы понимаем концепцию как функцию времени компиляции, которая рассматривает набор аргументов шаблонов и возвращает **true**, если они удовлетворяют требованиям концепции, и **false**, если они не удовлетвояют этим требованиям. Таким образом, мы реализуем концепцию в виде **constexpr**-функции. В отличие от истинной концепции такие проверки ограничений не имеют дело с семантикой; проверяются только синтаксические свойства.

Рассмотрим наш **String**; предполагаем, что его аргумент по поводу типа символа является **Ordered**:

```
template<typename C>
class String {
    static_assert(Ordered<C>(),"String's character type is not ordered");
    // ...
};
```

Когда **String<X>** конкретизируется для типа **X**, компилятором будет выполнен **static_assert**. Если **Ordered<X>()** вернет **true**, то компиляция продолжится с генерацией кода, который бы производился и без assert-проверки. В ином случае будет выдано сообщение об ошибке.

Все это выглядит разумным моментом для дальнейшей проработки. Можно даже так написать:

```
template<Ordered C>
class String {
    // ...
};
```

Но это для будущего, а сейчас давайте посмотрим, как определить предикат **Ordered<T>()**:

```
template<typename T>
constexpr bool Ordered()
{
    return Regular<T>() && Totally_ordered<T>();
}
```

Итак, тип является **Ordered**, если он одновременно и **Regular**, и **Totally_ordered**. Попробуем углубиться, чтобы понять, что это значит:

```
template<typename T>
constexpr bool Totally_ordered()
{
    return Equality_comparable<T>()    // имеет == и !=
        && Has_less<T>()&& Boolean<Less_result<T>>()
        && Has_greater<T>() && Boolean<Greater_result<T>>()
        && Has_less_equal<T>() && Boolean<Less_equal_result<T>>()
        && Has_greater_equal<T>() && Boolean<Greater_equal_result<T>>();
}

template<typename T>
constexpr bool Equality_comparable()
{
    return Has_equal<T>() && Boolean<Equal_result<T>>()
        && Has_not_equal<T>() && Boolean<Not_equal_result<T>>();
}
```

Таким образом, тип **T** должен быть регулярным и обеспечивать шесть обычных операций сравнения. Операции сравнения должны возвращать результаты, которые могут конвертироваться в **bool**. Кроме того, операции сравнения должны иметь надлежащий математический смысл. Стандарт C++ в точности определяет, что это значит (§31.2.2.1, § iso.25.4).

Здесь **Has_equal** реализованы с использованием **enable_if** и приемов, описанных в §28.4.4. Так как я считаю концепции более фундаментальными, чем типы, то их имена я начинаю с заглавных букв, чтобы усилить к ним внимание программиста. Я также храню их в отдельном пространстве имен (**Estd**) в надежде на то, что аналогичные имена скоро станут частью языка или стандартной библиотеки.

Углубляясь далее в набор полезных концепций, мы можем определить **Regular**:

```
template<typename T>
constexpr bool Regular()
{
    return Semiregular<T>() && Equality_comparable<T>();
}
```

Функция **Equality_comparable** дает нам == и !=, а **Semiregular** является концепцией, которая выражает понятие типа, у которого нет необычных технических ограничений:

```
template<typename T>
constexpr bool Semiregular()
{
    return Destructible<T>()
        && Default_constructible<T>()
        && Move_constructible<T>()
        && Move_assignable<T>()
        && Copy_constructible<T>()
        && Copy_assignable<T>();
}
```

При этом **Semiregular** может копироваться и перемещаться. Этому удовлетворяет множество типов, но есть и исключения, например не может копироваться **unique_ptr**. В то же время, мне неизвестны полезные типы, которые могут копироваться, но не перемещаться. Типы, не копируемые и не перемещаемые, например **type_info** (§22.5), крайне редки.

Мы можем также использовать проверки ограничений для функций; например:

```
template<typename C>
ostream& operator<<(ostream& out, String<C>& s)
{
    static_assert(Streamable<C>(),"String's character not streamable");
    out << '"';
    for (int i=0; i!=s.size(); ++i) cout << s[i];
    out << '"';
}
```

Концепция **Streamable**, в которой нуждается операция << у **String**, требует от аргумента **C** предоставлять операцию вывода <<:

```
template<typename T>
constexpr bool Streamable()
{
    return Input_streamable<T>() && Output_streamable<T>();
}
```

Таким образом, **Streamable** проверяет, можно ли использовать потоки ввода/вывода стандартной библиотеки (§4.3, Глава 38) для этого типа.

Проверка концепций посредством шаблонов с проверкой ограничений имеет очевидные слабости:

- Проверка ограничений размещается в определениях, но реально они принадлежат объявлениям. Концепция является частью интерфейса к абстракции, но проверку ограничений можно выполнить лишь в ее реализации.
- Проверка ограничений выполняется как часть конкретизации шаблона. Поэтому проверка может выполняться позже, чем нам хотелось бы. В частности, мы бы предпочли гарантию того, чтобы это выполнялось в точке первого вызова, но это невозможно сделать без изменений в языке.
- Можно забыть вставить проверку ограничений (особенно для шаблонов функций).
- Компилятор не проверяет, использует ли реализация шаблона свойства, отраженные в концепции. В итоге, реализация шаблона может пройти проверку ограничений, но затем потерпеть неудачу в проверке типа.
- Мы не определяем семантические свойства таким образом, чтобы их мог понять компилятор (мы используем комментарии).

Добавление ограничений делает явными требования к аргументам шаблона, и если ограничения хорошо продуманы, то они приводят к более осмысленным сообщениям об ошибках. Если мы забудем про проверку ограничений, то мы оказываемся отброшены назад к обычной проверке типов в коде, генерируемом в момент конкретизации шаблона. Это неприятно, но не катастрофично.

Если мы захотим, то мы можем разместить проверки ограничений где-угодно. Например, чтобы гарантировать проверку какого-то типа в соответствии с концепцией, мы можем поместить проверку ограничений в пространство имен (например, в глобальное пространство). Пример:

```
static_assert(Ordered<std::string>,"std::string is not Ordered");        // будет успех
static_assert(Ordered<String<char>>,"String<char> is not Ordered");     // будет провал
```

Первый **static_assert** проверяет, относится ли стандартный тип **string** к **Ordered** (относится, так как он предоставляет ==, != и <). Второй **static_assert** проверяет то же самое в отношении нашего типа **String** (проверка не проходит, так как я забыл про <). Используя такие глобальные проверки, мы не зависим от того, используется ли реально конкретная специализация шаблона в программе. В зависимости от нашей цели это может оказаться преимуществом, или неприятностью. Такая проверка заставляет выполнить проверку типа в конкретной точке программы; обычно это полезно для локализации ошибки. Но в больших программах со множеством библиотек явные проверки быстро превращаются в проблему.

Быть **Regular** — это идеал для типа. Объекты регулярных типов можно копировать, внедрять их в контейнеры **vector** и в массивы, сравнивать их и т.д. А если тип является **Ordered**, то его объекты можно использовать во множествах, можно сортировать последовательности таких объектов и т.д. Так что мы возвращаемся назад и исправляем наш **String**, чтобы он стал **Ordered**. В частности, мы добавляем < для лексикографического упорядочения:

```
template<typename C>
bool operator<(const String<C>& s1, const String<C>& s2)
{
    static_assert(Ordered<C>(),"String's character type not ordered");
    bool eq = true;
```

```
    for (int i=0; i!=s1.size() && i!=s2.size(); ++i) {
        if (s2[i]<s1[i]) return false;
        if (s1[i]<s2[i]) eq = false;      // не s1==s2
    }
    if (s2.size()<s1.size()) return false; // s2 короче чем s1
    if (s1.size()==s2.size() && eq) return false;   // s1==s2
    return true;
}
```

24.4.1. Аксиомы

Как и в математике, аксима это то, что мы не можем доказать. Это то, что мы полагаем имеющим место быть. В контексте требований в аргументам шаблона мы используем термин аксиома в том смысле, что мы отсылаемся к семантическим свойствам. Аксиома формулирет требование класса или алгоритма к набору их входных данных. Аксиома отражает ожидания класса или алгоритма в отношении аргументов. Аксиома имеет отношение только к величинам, реально используемым в программе. Например, алгоритм может корректно избегать разыменования нулевых указателей или копирования значения с плавающей запятой NaN. В таком случае речь идет об аксиомах разыменованности указателей и возможности копировать величины с плавающей запятой. Можно также сформулировать аксиому, что отдельные значения (**nullptr** или NaN) нарушают предусловия, так что их не нужно использовать.

В языке C++ в настоящее время нет средств для выражения аксиом, но как и для концепций, мы можем так или иначе отразить все это в коде.

Рассмотрим, как мы можем выразить некоторые ключевые семантические требования к типу, чтобы он был регулярным:

```
template<typename T>
bool Copy_equality(T x)       // семантика конструирования копированием
{
  return T{x}==x;          // копия сравнивается на равенство с тем, чьей копией является
}

template<typename T>
bool Copy_assign_equality(T x, T& y)  // семантика присваивания
{
  return (y=x, y==x); // рез-т присваивания сравнив-ся на равенство с источником присваивания
}
```

Другими словами, копирующие операции на самом деле создают копии.

```
template<typename T>
bool Move_effect(T x, T& y)  // семантика перемещения
{
  return (x==y? T{std::move(x)}==y) : true) && can_destroy(y);
}

template<typename T>
bool Move_assign_effect(T x, T& y, T& z)    // семантика присваивания перемещением
{
  return (y==z? (x=std::move(y), x==z)) : true) && can_destroy(y);
}
```

Перемещающая операция дает значение, сравнимое как эквивалентное с тем, что сравнивалось для источника перемещающей операции, и что источник перемещения может быть уничтожен.

Такие аксиомы представлены в виде исполняемого кода. Мы можем использовать их для тестирования, но что более важно, это позволяет сильнее все обдумать, чем если бы мы просто писали комментарий. В своей основе, мы формулируем такие псевдо-аксиомы, используя первоочередную логику предикатов.

24.4.2. Многоаргументные концепции

Оглядываясь на концепции одного аргумента, мы замечаем, что они весьма похожи на обычную проверку типов, так что концепции соответствуют типу типов. Это справедливо, но это лишь одна часть истории. Часто оказывается, что нахождение взаимосвязей между типами аргументов существенно для корректной спецификации и использования. Рассмотрим стандартный алгоритм **find()**:

```
template<typename Iter, typename Val>
Iter find(Iter b, Iter e, Val x);
```

Аргумент шаблона **Iter** должен быть входным итератором, и мы можем легко определить проверку ограничений аргументов для этой концепции шаблона.

Пока что все хорошо, но **find()** зависит критически от сравнения **x** с элементами последовательности **[b:e)**. Нам нужно указать, что такое сравнение нужно; то есть, нам нужно утверждение, что **Val** и тип входного итератора сравнимы на эквивалентность. Это требует двухаргументной версии для **Equality_comparable**:

```
template<typename A, typename B>
constexpr bool Equality_comparable(A a, B b)
{
    return Common<T, U>()
        && Totally_ordered<T>()
        && Totally_ordered<U>()
        && Totally_ordered<Common_type<T, U>>()
        && Has_less<T, U>() && Boolean<Less_result<T, U>>()
        && Has_less<U, T>() && Boolean<Less_result<U, T>>()
        && Has_greater<T, U>() && Boolean<Greater_result<T, U>>()
        && Has_greater<U, T>() && Boolean<Greater_result<U, T>>()
        && Has_less_equal<T, U>() && Boolean<Less_equal_result<T, U>>()
        && Has_less_equal<U, T>() && Boolean<Less_equal_result<U, T>>()
        && Has_greater_equal<T, U>() && Boolean<Greater_equal_result<T, U>>()
        && Has_greater_equal<U, T>() && Boolean<Greater_equal_result<U, T>>();
};
```

Это избыточно для простой концепции. Я, однако, хотел явно отразить свойства всех операций и симметрию их использования.

Отсюда можно определить **find()**:

```
template<typename Iter, typename Val>
Iter find(Iter b, Iter e, Val x)
{
    static_assert(Input_iterator<Iter>(),"find() requires an input iterator");
    static_assert(Equality_comparable<Value_type<Iter>, Val>(),
        "find()'s Iterator and value arguments must match");

    while (b!=e) {
        if (*b==x) return b;
        ++b;
    }
    return b;
}
```

Многоаргументные концепции особенно полезны при создании обобщенных алгоритмов. Это та самая область, где вы найдете наибольшее число концепций.

24.4.3. Концепции-значения

Концепции могут выражать произвольные (синтаксические) требования для набора аргументов шаблона. В частности, аргумент шаблона может быть значением целого типа, и поэтому концепции могут учитывать целые аргументы. Например, мы можем написать проверку ограничений для тестирования факта, что аргумент-значение шаблона достаточно мал по величине:

```
template<int N>
constexpr bool Small_size()
{
    return N<=8;
}
```

Более реалистичный пример будет концепцией, для которой числовой аргумент является лишь одним из многих. Например:

```
constexpr int stack_limit = 2048;

template<typename T, int N>
constexpr bool Stackable() // T тип регулярный и N эл-ов типа T уместятся в малом стеке
{
    return Regular<T>() && sizeof(T)*N<=stack_limit;
}
```

Здесь реализуется понятие «достаточно малое значение, чтобы его разместить в стеке». Это можно использовать следующим образом:

```
template<typename T, int N>
struct Buffer {
    // ...
};

template<typename T, int N>
void fct()
{
    static_assert(Stackable<T, N>(),"fct() buffer won't fit on stack");
    Buffer<T, N> buf;
    // ...
}
```

По сравнению с фундаментальными концепциями типов, концепции-значения обычно невелики и разрабатываются по месту.

24.4.4. Проверки ограничений

Проверки ограничений, использованные в настоящей книге, можно найти на сайте поддержки этой книги. Они не являются частью стандарта, и я надеюсь, что в будущем они будут заменены на средства языка C++. Но они полезны для размышлений о шаблонах и типах и их роли в программном дизайне концепций в стандартной библиотеке. Они должны располагаться в отдельных пространствах имен во избежание потенциальных конфликтов с будущими средствами языка или с альтернативными реализациями идеи концепций. Я применяю пространство имен **Estd**, но это может быть синонимом

(§14.4.2). Вот несколько примеров проверок ограничений, которые могут быть вам полезными:

- **Input_iterator<X>**: **X** является итератором, который применяется лишь однажды для траверса последовательности (вперед посредством ++) с единичным чтением элемента.
- **Output_iterator<X>**: **X** вляется итератором, который применяется лишь однажды для траверса последовательности (вперед посредством ++) с единичной записью элемента.
- **Forward_iterator<X>**: **X** вляется итератором, который применяется для траверса последовательности (вперед посредством ++). Это то, что односвязанный список (например, **forward_list**) поддерживает естественным образом.
- **Bidirectional_iterator<X>**: **X** вляется итератором, который применяется для траверса последовательности вперед (используя ++) и назад (посредством --). Это то, что двусвязанный список (например, **list**) поддерживает естественным образом.
- **Random_access_iterator<X>**: **X** вляется итератором, который применяется для траверса последовательности (вперед и назад) и для произвольного доступа к элементам посредством индексирования и позиционирования с помощью += и -=. Это то, что массив реализует естественным образом.
- **Equality_comparable<X, Y>**: **X** может сравниваться с **Y** посредством == и !=.
- **Totally_ordered<X, Y>**: **X** и **Y** являются **Equality_comparable** и **X** может сравниваться с **Y** посредством <, <=, > и >=.
- **Semiregular<X>**: **X** может копироваться, конструироваться по умолчанию, аллоцироваться в свободной памяти и он свободен от мелких технических ограничений.
- **Regular<X>**: **X** является **Semiregular** и поддерживает операцию сравнения на равенство. Контейнеры стандартной библиотеки требуют от элементов быть **Regular**.
- **Ordered<X>**: **X** является **Regular** и **Totally_ordered**. Ассоциативные контейнеры стандартной библиотеки требуют от своих элементов быть упорядоченными, если только вы явно не предоставите операции сравнения.
- **Assignable<X, Y>**: **Y** может присваиваться **X**, используя =.
- **Predicate<F, X>**: Можно вызвать **F** для **X** с возвратом **bool**.
- **Streamable<X>**: **X** можно читать и писать, используя потоки стандартной библиотеки.
- **Moveable<X>**: **X** можно перемещать; то есть, этот тип имеет перемещающий конструктор и перемещающее присваивание. Дополнительно, тип **X** допускает взятие адреса и уничтожение.
- **Copyable<X>**: **X** является **Moveable** и также может копироваться.
- **Convertible<X, Y>**: **X** может неявно приводится к типу **Y**.
- **Common<X, Y>**: **X** и **Y** могут непротиворечивым образом приводиться к общему типу **Common_type<X, Y>**. Это формализация правила языка для совместимости операндов с операцией?: (§11.1.3). К примеру, **Common_type<Base*, Derived*>** есть **Base*** и **Common_type<int, long>** есть **long**.
- **Range<X>**: **X** может использоваться оператором range-**for** (§9.5.1), то есть, **X** должен предоставить члены **x.begin()** и **x.end()**, или их глобальные аналоги с надлежащей семантикой.

Очевидно, что эти определения являются неформальными. В большинстве случаев эти концепции базируются на предикатах стандартной библиотеки (§35.4.1); стандарт ISO C++ предоставляет формальные определения (§ iso.17.6.3).

24.4.5. Проверка определения шаблона

Шаблон с проверкой ограничений гарантирует, что тип обеспечивает свойства, требуемые концепцией. Если реализация шаблона реально использует больше свойств, чем это гарантируется концепцией, то возникает возможность ошибки. Например, **find()** из стандартной библиотеки требует пару итераторов в качестве аргументов, но мы можем (по неосторожности) определить **find()** следующим образом:

```
template<typename Iter, typename Val>
Iter find(Iter b, Iter e, Val x)
{
    static_assert(Input_iterator<Iter>(),"find(): Iter is not a Forward iterator");
    static_assert(Equality_comparable<Value_type<Iter>, Val>),
            "find(): value type doesn't match iterator");

    while (b!=e) {
        if (*b==x) return b;
        b =b+1;          // заметьте: не ++b
    }
    return b;
}
```

Теперь **b+1** является ошибкой, если только **b** не является итератором с произвольным доступом (или не является прямым итератором, что гарантируется проверкой ограничений). Однако, проверка ограничений не помогает обнаружить эту проблему. К примеру:

```
void f(list<int>& lst, vector<string>& vs)
{
    auto p = find(lst.begin(), lst.end(),1209);          // ошибка: list не предоставляет +
    auto q = find(vs.begin(), vs.end(),"Cambridge");     // OK: vector предоставляет +
    // ...
}
```

Вызов **find()** для **list** будет неудачным (поскольку **+** не определен для прямого итератора, предоставляемого для **list**), а вызов для **vector** пройдет успешно (поскольку **b+1** приемлем для **vector<string>::iterator**).

Проверка ограничений предоставляет пользователю сервис: реальные аргументы шаблона проверяются в соответствии с требованиями шаблона. С другой стороны, проверка ограничений не помогает разработчику шаблона, который хотел бы удостовериться, что реализация шаблона не использует никаких свойств, кроме тех, что специфицированы концепцией. Итак, как же мы проверяем реализацию параметризованного класса или обобщенного алгоритма?

Концепция формулирует строгую линию: реализация не должна использовать никаких свойств аргументов за пределами концепции. Поэтому нам нужно тестировать реализацию с аргументами, специфицированными концепцией, и ни с какими другими.

Итак, для примера с **find()** мы рассматриваем **Forward_iterator** и **Equality_comparable** или со стандартными определениями прямого итератора или эквивалентных концепций (§ iso.17.6.3.1, § iso.24.2.5). Затем мы решаем, что нам нужен тип **iterator**, который предоставляет по крайней мере:

- Умолчательный конструктор
- Копирующий конструктор и копирующее присваивание
- Операции **==** и **!=**
- Префиксную операцию **++**
- Тип **Value_type<Iterator>**

- Префиксную операцию *
- Возможность присвоить результат операции * величине типа **Value_type<Iterator>**
- Возможность присвоить **Value_type<Iterator>** результату операции *

Это некоторое упрощение по сравнению с прямым итератором из стандартной библиотеки, но этого достаточно для **find()**.

Исходя из этого списка, нам нужно найти или определить тип, который обеспечивает только желаемые свойства. Для прямого итератора (как это требуется для **find()**) **forward_list** из стандартной библиотеки идеально соответствует нашему списку. Это получилось потому, что прямой итератор был реализован в соответствии с идеей того, что он позволяет вести итерацию по односвязанному списку. Когда используется существующий тип, нужно быть осторожными, чтобы не связаться с типом, более гибким, чем требуется. Типичной ошибкой при тестировании алгоритма (такого как **find()**) будет выбор типа **vector**. Дело в том, что свергибкость и сверхобщность, делающие **vector** столь популярным, делают его неподходящим в качестве архетипа для многих простых алгоритмов.

Если не получается отыскать существующий тип, который подходит под наши нужды, то мы должны создать его сами. Это выполняется посредством прохода по списку требований; в результате создаются подходящие члены:

```
template<typename Val>
struct Forward {          // проверяем find()
    Forward();
    Forward(const Forward&);
    Forward operator=(const Forward&);
    bool operator==(const Forward&);
    bool operator!=(const Forward&);
    void operator++();
    Val& operator*();     // упрощено: не работаем с proxy для Val
};

template<typename Val>
using Value_type<Forward<Val>> = Val;       // упрощено; see § 28.2.4

void f()
{
    Forward<int> p = find(Forward<int>{}, Forward<int>{},7);
}
```

На этом уровне тестирования нет необходимости проверять, что эти операции действительно реализуют правильную семантику. Просто нужно проверить, что реализация не полагается на свойства, на которые она не должна полагаться.

Здесь я упростил ситуацию, опустив тестирование архетипа для аргумента **Val**. Вместо этого я просто использую тип **int**. Здесь было бы неуместно обсуждать взаимодействие архетипов для **Val** и архетипов для **Iter**, что сильно усложнило бы нашу работу, и не было бы слишком полезным.

Реализация теста с проверкой алгоритма **find()** относительно **std::forward_list** и **X** весьма нетривиальна, но это еще не самая сложная задача, стоящая перед разработчиком обобщенного алгоритма. Если использовать небольшой и хорошо специфицированный набор концепций, то можно сильно упростить задачу. Тесты могут и должны выполняться полностью на этапе компиляции.

Отметим, что наша простая спецификация и стратегия проверки ведут к **find()**, требующему, чтобы тип его итератора имел функцию типа **Value_type** (§28.2). Это позволяет использовать указатели в качестве итераторов. Это важно для многих параметров шабло-

нов, так как можно использовать встроенные типы и типы, определяемые пользователем (§1.2.2, §25.2.1).

24.5. Советы

[1] Шаблоны могут передавать типы аргументов без потери информации; §24.1.
[2] Шаблоны обеспечивают общий механизм для программирования этапа компиляции; §24.1.
[3] Проектируйте обобщенные алгоритмы посредством движения снизу-вверх от конкретных примеров; §24.2.
[4] Обобщайте алгоритмы, специфицируя требования к аргументам шаблонов в терминах концепций; §24.3.
[5] Не придавайте нетрадионных смыслов традиционным понятиям; §24.3.
[6] Используйте концепции как инструмент проектирования; §24.3.
[7] Нацеливайтесь на состыкуемость алгоритмов и типов аргументов, используя общие и регулярные требования к аргументам шаблонов; §24.3.
[8] Исследуйте концепции посредством минимизации требований алгоритмов к аргументам шаблона, а затем обобщайте для более широкого использования; §24.3.1.
[9] Концепция не является просто описанием требований конкретной реализации алгоритма; §24.3.1.
[10] По возможности выбирайте концепции из списка хорошо известных концепций; §24.3.1, §24.4.4.
[11] Умолчательная концепция для аргумента шаблона — это **Regular**; §24.3.1.
[12] Не все типы аргументов шаблона есть **Regular**; §24.3.1.
[13] Концепция требует семантического аспекта; она не является преимущественно синтаксическим понятием; §24.3.1, §24.3.2, §24.4.1.
[14] Воплощайте концепции в коде; §24.4.
[15] Реализуйте концепции как предикаты этапа компиляции и тестируйте их, используя **static_assert()** или **enable_if<>**; §24.4.
[16] Используйте аксиомы в качестве инструмента проектирования; §24.4.1.
[17] Используйте аксиомы как руководство для тестирования; §24.4.1.
[18] Некоторые концепции включают два или более аргументов шаблона; §24.4.2.
[19] Концепции — это не тип типов; §24.4.2.
[20] Концепции могут включать числовые величины; §24.4.3.
[21] спользуйте концепции как руководство к тестированию определений шаблонов; §24.4.5.

25

Специализации

Вы не знаете, что привело к беде.
Но что вы знаете наверняка, это то, что вы этого не знаете.
— Марк Твен

- Введение
- Параметры и аргументы шаблонов

  Типы как аргументы; Величины как аргументы; Операции как аргументы; Шаблоны как аргументы; Аргументы шаблонов по умолчанию
- Специализации

  Специализации интерфейса; Первичный шаблон; Порядок специализаций; Специализации шаблонных функций
- Советы

25.1. Введение

За последние два десятилетия шаблоны эволюционировали из относительно простой идеи в становой хребет практически всех современных технологий программирования на C++. Особенно шаблоны полезны для:

- улучшения безопасности типов (устранения явного приведения типов; §12.5)
- повышения уровня абстракции программ (при помощи стандартных контейнеров и алгоритмов; §4.4, §4.5, §7.4.3, Глава 31, Глава 32) и
- предоставления более гибкой и безопасной параметризации типов и алгоритмов (§25.2.3).

Все эти техники программирования критически полагаются на способность кода с шаблонами использовать аргументы шаблонов без значительной нагрузки по производительности. Большая часть техник также полагаются на возможность вывода типа, предлагаемого шаблонами. Все это лежит в основе возможности применения C++ для программирования в критически важных областях, таких как высокопроизводительные вычисления и программирование встраиваемых систем. За солидными примерами вы можете обратиться к стандартной библиотеке (Часть IV).

Данная глава и две последующих демонстрируют продвинутые или специализированные свойства языка, поддерживающие технологии, направленные на бескомпромиссную гибкость и производительность. Многое из рассмотренного применяется в реализации стандартной библиотеки. Как и большинство программистов, я предпочитаю забыть о продвинутых вопросах большую часть времени. Всюду, где это допустимо, я использую простые приемы и полагаюсь на библиотеки, которые разрабатывались экспертами на базе продвинутых возможностей языка C++.

Введение в шаблоны было дано в §3.4. Данная глава является частью последовательности глав, в которых рассматривается применение шаблонов:

- Глава 23 дает более детализированное введение в шаблоны.
- В Главе 24 обсуждается обобщенное программирование, наиболее общий случай применения шаблонов.
- Глава 25 (настоящая глава) показывает, как специализировать шаблон набором аргументов.
- Глава 26 фокусируется на вопросах реализации шаблонов, относящихся к связыванию имен.
- В Главе 27 обсуждается связь шаблонов и классовых иерархий.
- Глава 28 фокусируется на шаблонах, как на языке для генерирования классов и функций.
- Глава 29 представляет более крупные примеры технологий программирования на базе шаблонов.

25.2. Параметры и аргументы шаблонов

Шаблоны могут иметь параметры:

- Параметры-типы.
- Параметры-значения встроенных типов, таких как **int** (§25.2.2) и указатели на функции (§25.2.3).
- Параметры-шаблоны (§25.2.4).

Параметры-типы являлись до сих пор наиболее распространенными, но параметры-значения весьма полезны во многих случаях (§25.2.2, §28.3).

Шаблоны могут принимать либо фиксированное число параметров, либо переменное их число. Обсуждение шаблонов с переменным числом параметров отложим до §28.6.

Отметим, что принято использовать короткие имена с начальными заглавными буквами в качестве имен типовых аргументов шаблонов, например **T**, **C**, **Cont** и **Ptr**. Это приемлемо, поскольку такие имена становятся стандартными и ограничены относительно небольшими пространствами имен (§6.3.3). В то же время, если применять имена вроде **ALL_CAPS**, то есть вероятность конфликта с макросами (§12.6).

25.2.1. Типы как аргументы

Аргумент шаблона определяется как *параметр типа* при помощи префикса с ключевыми словами **typename** или **class** (с одинаковым результатом). Любой тип (встроенный или пользовательский) синтаксически применим к шаблонам, объявленных с параметром-типом. Например:

```
template<typename T>
void f(T);

template<typename T>
class X {
    // ...
};

f(1);                       // T выводится как int
f<double>(1);               // T есть double
f<complex<double>>(1);      // T есть complex<double>

X<double> x1;               // T есть double
X<complex<double>> x2;      // T есть complex<double>
```

Аргумент-тип никак не ограничивается, то есть в интерфейсе класса нет никаких ограничений на конкретные допустимые типы или типы из иерархии. Приемлемость типа зависит исключительно от того, как этот тип используется в шаблоне. Общие ограничения вы должны реализовывать как концепции (§24.3).

Встроенные типы и типы, определяемые пользователем, используются одинаково, когда применяются в качестве аргументов шаблона. Это существенно, поскольку позволяет определять шаблоны, работающие идентично в обоих случаях. Пример:

```
vector<double> x1;                  // vector из элементов типа double
vector<complex<double>> x2;         // vector из эл-ов типа complex<double>
```

В частности, отсутствуют временные накладные расходы или расходы по памяти в обоих этих случаях:

- Величины встроенных типов не упаковываются в специальные контейнерные объекты.
- Величины всех типов извлекаются непосредственно из **vector** без использования потенциально накладной функции **get()**.
- Обращение к величинам пользовательских типов не выполняется неявно через ссылки.

Чтобы использоваться как аргумент шаблона, тип должен находиться в пространстве имен и быть доступным. К примеру:

```
class X {
    class M {/* ... */};
    // ...
    void mf();
};

void f()
{
    struct S {/* ... */};
    vector<S> vs; // OK
    vector<X::M> vm;   // ошибка: X::M закрытый
    // ...
}

void M::mf()
{
    vector<S> vs; // ошибка: S нет в области видимости
    vector<M> vm;      // OK
    // ...
};
```

25.2.2. Величины как аргументы

Параметр шаблона, не являющийся типом или шаблоном, называется параметром-значением, а соответствующий передаваемый аргумент называется аргументом-значением. Например, целые аргументы удобны для предоставления размеров и предельных значений:

```
template<typename T, int max>
class Buffer {
    T v[max];
public:
    Buffer() {}
    // ...
};
```

```
Buffer<char,128> cbuf;
Buffer<int,5000> ibuf;
Buffer<Record,8> rbuf;
```

Простые и ограниченные контейнеры, такие как **Buffer**, полезны там, где важны компактность и эффективность во время выполнения. Они не используют свободной памяти, которую используют более общие **string** или **vector**, и не страдают от неявных приведений указателей, как для встроенных массивов (§7.4). В стандартной библиотеке **array** (§34.2.1) следует этой идее.

Аргумент для параметра-значения шаблона может быть (§ iso.14.3.2):

- Интегральным константным выражением (§10.4).
- Указателем или ссылкой на объект или функцию с внешним связыванием (§15.2).
- Неперегруженным указателем на член (§20.6).
- Нулевым указателем (§7.2.2).

Указатель, используемый в качестве аргумента шаблона, должен быть в форме **&of**, где **of** есть имя объекта или функции, или в форме **f**, где **f** это имя функции. Указатель на член должен быть в форме **&X::of**, где **of** это имя члена. В частности, строковый литерал неприемлем в качестве аргумента шаблона:

```
template<typename T, char* label>
class X {
    // ...
};

X<int,"BMW323Ci"> x1;          // ошибка: строковый литерал в качестве аргумента шаблона
char lx2[] = "BMW323Ci";
X<int, lx2> x2;                // OK: у lx2 внешнее связывание
```

Это ограничение, как и ограничение на числа с плавающей запятой для аргументов-значений, существует для упрощения реализации отдельно компилируемых единиц трансляции. Об аргументах-величинах лучше мыслить как о механизме передачи целых и указателей. Не нужно делать из этого что-то большее. К сожалению, интегральные типы (§10.4.3) не могут использоваться в аргументах-значениях. Еще аргументы-значения являются механизмом продвинутых вычислений на этапе компиляции (Глава 28).

Целый аргумент-значение должен быть константой. Например:

```
constexpr int max = 200;

void f(int i)
{
    Buffer<int, i> bx;         // ошибка: ожидается константное выражение
    Buffer<int, max> bm;       // OK: константное выражение
    // ...
}
```

И наоборот, параметр-значение является константой внутри шаблона, так что попытка изменить значение параметра является ошибкой. Например:

```
template<typename T, int max>
class Buffer {
    T v[max];
public:
    Buffer(int i) {max = i;}   // ошибка: попытка присвоить шаблону аргумент-значение
    // ...
};
```

В шаблоне параметр-тип может использоваться в качестве типа в том же списке параметров шаблона. Например:

```
template<typename T, T default_value>
class Vec {
    // ...
};

Vec<int,42> c1;
Vec<string,""> c2;
```

Это становится особо полезным, когда комбинируется с умолчательным аргументом шаблона (§25.2.5); например:

```
template<typename T, T default_value = T{}>
class Vec {
    // ...
};

Vec<int,42> c1;
Vec<int> c11;                    // умолчательное значение есть int{}, т.е., 0
Vec<string,"fortytwo"> c2;
Vec<string> c22;                 // умолчательное значение есть string{}; т.е., ""
```

25.2.3. Операции как аргументы

Рассмотрим слегка упрощенную версию стандартного контейнера **map** (§31.4.3):

```
template<typename Key, Class V>
class map {
    // ...
};
```

Как нам обозначить критерий сравнения для **Key**?

- Можно жестко встроить критерий сравнения в контейнер, поскольку контейнер не может (в общем случае) возлагать свои нужды на тип элементов. К примеру, по умолчанию **map** использует операцию **<** для сравнения, но не все **Key** имеют операцию **<**, которую мы захотели бы использовать.
- Мы не можем жестко внедрить критерий сравнения для упорядочения элементов в тип **Key**, поскольку существует много способов упорядочения элементов на базе ключа. Например, частым примером для **Key** служит тип **string**, а для этого типа имеется множество критериев (например, с учетом регистра или без).

Следовательно, критерий сортировки не встраивается в контейнерный тип или в тип элементов. Понятие критерий сортировки для **map** можно представить как:

1. Шаблонный аргумент-значение (например, указатель на функцию сравнения).
2. Шаблонный аргумент-тип, определяющий тип объекта сравнения.

На первый взгляд решение 1 кажется проще. К примеру:

```
template<typename Key, typename V, bool(*cmp)(const Key&, const Key&)>
class map {
public:
    map();
    // ...
};
```

От пользователя требуется предоставить критерий сравнения в виде функции:

```
bool insensitive(const string& x, const string& y)
{
    // сравнение без учета регистра (например, "hello" эквивалентно "HellO")
}
map<string, int, insensitive> m;    // сравнение посредством insensitive()
```

Но такое решение не очень гибкое. В частности, разработчик контейнера типа **map** должен решить, сравнивать в рамках неизвестного типа **Key**, используя указатель на функцию, или же использовать объект-функцию некоторого типа. К тому же, поскольку типы аргументов у операции сравнения должны зависить от типа **Key**, то трудно дать умолчательный критерий сравнения.

Как следствие, вторая альтернатива используется чаще и она применяется в стандартной библиотеке. Пример:

```
template<typename Key, Class V, typename Compare = std::less<Key>>
class map {
public:
    map() {/* ... */}                          // используем умолчательное сравнение
    map(Compare c): cmp{c} {/* ... */}         // замещение умолчания
    // ...
    Compare cmp {};                            // умолчательное сравнение
};
```

Наиболее общий случай, сравнение операцией «меньше-чем», используется по умолчанию. Если нужен иной критерий сравнения, то его можно предоставить как объект-функцию (§3.4.3):

```
map<string, int> m1;                // используем умолчательное сравнение (less<string>)

map<string, int, std::greater<string>> m2;    // сравниваем посредством greater<string>()
```

Объекты-функции сохраняют в себе состояние. Например:

```
Complex_compare f3 {"French",3};                // создаем объект сравнения (§ 25.2.5)
map<string, int, Complex_compare> m3 {f3};      // сравниваем посредством f3()
```

Мы также можем использовать указатели на функции, включая лямбды, которые могут конвертироваться в указатели на функции (§11.4.5). Пример:

```
using Cmp = bool(*)(const string&, const string&);
map<string, int, Cmp> m4 {insensitive};     // сравниваем с помощью указателя на функцию

map<string, int, Cmp> m4 {[](const string& a, const string b) {return a>b;}};
```

Передача операций сравнения путем объекта-функции имеет заметные преимущества по сравнению с передачей указателей на функции:

- Простая функция-член, определенная внутри класса, легко поддается встраиванию, в то время как попытка встраивания для вызова функции по указателю требует исключительного внимания от компилятора.
- Функциональный объект без членов-данных можно передать без дополнительных накладных расходов.
- Несколько операций могут быть переданы одним объектом без дополнительных накладных расходов.

Критерий сравнения для **map** является всего лишь примером. Однако, использованная при этом техника весьма общая, широко используется при параметризации классов и функций с «политиками». Примерами служат действия для алгоритмов (§4.5.4, §32.4),

аллокаторы для контейнеров (§31.4, §34.4) и тип **unique_ptr** (§34.3.1). Мы сталкиваемся с теми же альтернативами дизайна, когда требуется специфицировать аргументы для шаблонных функций, таких как **sort()**, и в стандартной библиотеки также учитываются альтернативы [2] для этих случаев (см., например, §32.4).

Если бы у нас в программе критерий сравнения использовался лишь один раз, то имел бы смысл использовать лямбду, чтобы жестче отразить версию с объектом-функцией:

```
map<string, int, Cmp> c3 {[](const string& x, const string& y) const {return x<y;}}; // ошибка
```

К сожалению, это работать не будет, ибо нет конверсии лямбды в тип функционального объекта. Можно поименовать лямбду, а затем использовать ее имя:

```
auto cmp = [](const string& x, const string& y) const {return x<y;}
map<string, int, decltype(cmp)> c4 {cmp};
```

Я вообще считаю, что именовать операции полезно как с точки зрения дизайна программы, так и с точки зрения ее поддержки. Кроме того, все что поименовано и определено нелокально, может пригодиться где-нибудь и в других местах программы.

25.2.4. Шаблоны как аргументы

Иногда требуется в качестве аргументов шаблона передать шаблон (а не конкретный класс или значение). Например:

```
template<typename T, template<typename> class C>
class Xrefd {
    C<T> mems;
    C<T*> refs;
    // ...
};

template<typename T>
    using My_vec = vector<T>;    // используем умолчательный аллокатор

Xrefd<Entry, My_vec> x1;         // храним кросс-ссылки на Entrys в векторе

template<typename T>
class My_container {
    // ...
};

Xrefd<Record, My_container> x2;  // храним кросс-ссылки на Records в My_container
```

Объявляя шаблон в качестве параметра шаблона, мы должны специфицировать требуемые им аргументы. К примеру, мы утверждаем, что параметр **C** шаблона **Xrefd** является шаблоном класса, принимающим единственный аргумент-тип. Если этого не сделать, то нельзя будет использовать специализации **C**. Причина, по которой шаблоны используются в качестве параметров шаблонов, состоит в нашем желании конкретизировать шаблоны множеством типов аргументов (как **T** и **T*** в предыдущем примере)

В качестве аргументов шаблонов могут использоваться лишь шаблоны классов.

Распространенный случай, когда шаблон нуждается в одном или двух контейнерах, лучше всего реализуется передачей контейнерных типов (§31.5.1). Пример:

```
template<typename C, typename C2>
class Xrefd2 {
    C mems;
```

```
    C2 refs;
    // ...
};

Xrefd2<vector<Entry>, set<Entry*>> x;
```

Здесь типы-значения **C** и **C2** могут быть получены простой функцией типов (§28.2), чтобы получить тип элементов контейнеров, например, **Value_type<C>**. Эта техника применяется в стандартной библиотеке для контейнерных адаптеров, таких как **queue** (§31.5.2).

25.2.5. Аргументы шаблонов по умолчанию

Явно специфицировать критерий сравнения при каждом использовании **map** весьма утомительно — особенно когда **less<Key>** в типичном случае является наилучшим выбором. В таком случае можно определить **less<Key>** в качестве умолчательного типа для шаблонного аргумента у **Container**, так что явно можно будет указывать лишь редкие варианты критериев сравнения:

```
template<typename Key, Class V, typename Compare = std::less<Key>>
class map {
public:
    explicit map(const Compare& comp ={});
    // ...
};

map<string, int> m1;                    // используем less<string> для сравнений
map<string, int, less<string>> m2;      // тот же тип что и m1

struct No_case {
    // определяем operator()() для сравнения строк без учета регистра
};

map<string, int, No_case> m3;           // m3 другого типа чем m1 и m2
```

Заметьте, как умолчательный конструктор у **map** создает умолчательный объект сравнения, **Compare{}**. Это типичный случай. Если мы хотим более проработанной конструкции, то это придется сделать явным образом. Например:

```
map<string, int, Complex_compare> m {Complex_compare{"French",3}};
```

Семантическая проверка аргумента по умолчанию для шаблонного параметра выполняется только в случае, если этот аргумент действительно используется. В частности, пока мы воздерживаемся от использования умолчательного шаблонного аргумента **less<Key>**, мы можем сравнивать значения типа **X**, для которых **less<X>** не компилируется. Это важно для дизайна стандартных контейнеров (например, **std::map**), которые полагаются на аргумент шаблона, специфицирующий умолчательные значения (§31.4).

Как и для умолчательных аргументов функций (§12.2.5), умолчательные аргументы шаблона могут специфицироваться и предоставляться лишь для замыкающих аргументов:

```
void f1(int x = 0, int y);      // ошибка: умолчательный аргумент не последний
void f2(int x = 0, int y = 1); // OK

f2(,2);     // синтаксическая ошибка
f2(2);      // вызов f(2,1);

template<typename T1 = int, typename T2>
class X1 {// ошибка: умолчательный аргумент не последний
    // ...
};
```

```
template<typename T1 = int, typename T2 = double>
class X2 {// OK
    // ...
};

X2<, float> v1;      // синтаксическая ошибка
X2<float> v2;        // v2 есть X2<float, double>
```

Решение не допускать «пустой» аргумент в смысле «использовать по умолчанию» — это сознательный компромисс между гибкостью и возможностью туманных ошибок.

Технология предоставления политики посредством шаблонного аргумента с последующим умолчанием по нему (для предоставления наиболее общей политики) принята в стандартной библиотеке (см. §32.4). Но как это не странно, она не применяется для **basic_string** (§23.2, Глава 36). Вместо этого стандартные строки полагаются на **char_traits** (§36.2.2), а стандартные контейнеры полагаются на аллокаторы (§34.4).

25.2.5.1. Умолчательные аргументы шаблонных функций

Очевидно, что умолчательные аргументы шаблонов полезны и для шаблонных функций. Например:

```
template<typename Target =string, typename Source =string>
Target to(Source arg)    // конвертация Source в Target
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                    // пишем arg в поток
        ||!(interpreter >> result)               // читаем result из потока
        ||!(interpreter >> std::ws).eof())       // stuff остался в потоке?
        throw runtime_error{"to<>() failed"};
    return result;
}
```

Шаблонный аргумент функции должен быть явно указан только, если он не может быть выведен или когда нет умолчательного значения, так что можно написать следующее:

```
auto x1 = to<string, double>(1.2); // очень явно
auto x2 = to<string>(1.2);         // Источник выводится в тип double
auto x3 = to<>(1.2);               // Цель по умолчанию string; Source выводится в double
auto x4 = to(1.2);                 // <> избыточны
```

Если все шаблонные аргументы умолчательны, то знак **<>** можно опустить (в точности как у специализаций шаблонных функций; §25.3.4.1).

Реализация функции **to()** несколько тяжеловесна для комбинаций простых типов вроде **to<double>(int)**, но улучшенные реализации могут представляться специализациями (§25.3). Заметьте, что **to<char>(int)** не будет работать, так как **char** и **int** не разделяют представления для **string**. Для конверсии между скалярными числовыми типами я предпочитаю **narrow_cast<>()** (§11.5).

25.3. Специализации

По умолчанию, шаблон предоставляет единичное определение, которое используется для любого шаблонного аргумента, которое пользователь может себе представить. Это не всегда имеет смысл для человека, программирующего шаблон. Я мог бы сказать, «если

аргумент шаблона является указателем, используйте это определение; в противном случае используйте другое определение», или «выдать ошибку, если только аргумент шаблона не есть указатель, выведенный из класса **My_base**». Многие из таких соображений дизайна могут быть отображены альтернативными определениями шаблона и выбором компилятором из этого набора на основе предоставленного аргумента шаблона в том месте, где он используется. Такие альтернативные определения шаблона называются *пользовательскими специализациями*, или просто *специализациями*. Рассмотрим возможное применение класса **vector**:

```
template<typename T>
class Vector { // общий векторный тип
    T* v;
    int sz;
public:
    Vector();
    explicit Vector(int);

    T& elem(int i) {return v[i];}
    T& operator[](int i);

    void swap(Vector&);
    // ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Здесь большинство **Vector** будут соответствовать некоторому указательному типу. Для этого существует несколько причин, но основная причина заключается в стремлении сохранить полиморфное поведение на этапе выполнения (§3.2.2, §20.3.2). В результате, все кто практикует объектно-ориентированное программирование и использует надежные по типам контейнеры (как и все контейнеры стандартной библиотеки), так или иначе придут к контейнерам указателей.

Умолчательное поведение большинства релизаций C++ демонстрирует репликацию кода для шаблонных функций. Это хорошо для эффективности в плане времени выполнения, но в случае недостаточного внимания может приводить к сильному разбуханию объема кода в критических случаях, как в примере для **Vector**.

По счастью есть очевидное решение. Контейнеры указателей могут разделять единственное представление. Это можно отразить в специализации. Сначала, мы определяем версию (специализацию) **Vector** для указателей на **void**:

```
template<>
class Vector<void*>{     // полная специализация
    void** p;
    // ...
    void*& operator[](int i);
};
```

Эта специализация затем может использоваться в качестве общей реализации для **Vector** с указателями. В ином случае можно реализовать **unique_ptr<T>** на базе единственной разделяемой реализации класса, хранящего **void***.

Префикс **template<>** говорит, что это специализация, которую можно специфицировать без шаблонного параметра. Аргументы шаблона, для которых следует использовать специализацию, указывают внутри **<>** после имени. То есть, **<void*>** сообщает, что это определение следует использовать в реализациях **Vector**, у которых **T** есть **void***.

Специализация **Vector<void*>** является полной специализацией. То есть, нет параметра, который нужно специфицировать или вывести для использования специализации; **Vector<void*>** используется следующим образом:

```
Vector<void*> vpv;
```

Чтобы определить специализации, которые используются для каждого вектора указателей и только для вектора указателей, можно написать:

```
template<typename T>
class Vector<T*> : private Vector<void*> {   // частичная специализация
public:
    using Base = Vector<void*>;

    Vector() {}
    explicit Vector(int i) :  Base(i) {}

    T*& elem(int i) {return reinterpret_cast<T*&>(Base::elem(i));}
    T*& operator[](int i) {return reinterpret_cast<T*&>(Base::operator[](i));}

    // ...
};
```

Схема специализации в виде **<T*>** после имени говорит о том, что эта специализация должна использоваться для любого типа указателей; то есть, эта специализация должна использоваться для любого **Vector** с любым шаблонным аргументом, который может быть выражен как **T***. Пример:

```
Vector<Shape*> vps;     // <T*> есть <Shape*> так что T есть Shape
Vector<int**> vppi;     // <T*> есть <Int**> так что T есть int*
```

Специализация с указанием параметра шаблона называется частичной специализацией.

Заметьте, что когда используется частичная специализация, параметр шаблона выводится из схемы специализации; параметр шаблона не есть просто актуальный аргумент шаблона. В частности, для **Vector<Shape*>** **T** есть **Shape**, а не **Shape***.

Имея частичную специализацию шаблона **Vector**, мы располагаем разделяемой реализацией для всех **Vector** с указателями. Класс **Vector<T*>** является просто интерфейсом к **Vector<void*>**, реализованного исключительно наследованием и расширением посредством встраивания.

Важно, чтобы такое уточнение реализации **Vector** достигалось без влияния на интерфейс, предоставляемый пользователям. Специализация является способом специфицирования альтернативных реализаций для различных вариантов применения общего интерфейса. Мы, конечно, могли бы дать общему **Vector** и вектору указателей разные имена. Но когда я так делал, то некоторые люди забывали использовать вариант для указателей и находили код более раздутым, чем они ожидали. Поэтому важно скрывать важные детали реализации за общим интерфейсом.

Некоторые компиляторы настолько умны, что могут выполнять такую специфическую оптимизацию даже без помощи со стороны программиста, но наша техника является общей и очень полезной.

25.3.1. Специализации интерфейса

Иногда специализация не является алгоритмической оптимизацией, а скорее модификацией интерфейса (или даже представления). Например, стандартный **complex** использует специализации для подстраивания набора конструкторов и типов аргументов для важных операций у важных специализаций (таких как **complex<float>** и **complex<double>**). Общий шаблон (§25.3.1.1) выглядит следующим образом:

```
template<typename T>
class complex {
public:
    complex(const T& re = T{}, const T& im = T{});
    complex(const complex&);                    // копирующий конструктор
    template<typename X>
        complex(const complex<X>&);             // конверсия из complex<X> в complex<T>

    complex& operator=(const complex&);
    complex<T>& operator=(const T&);
    complex<T>& operator+=(const T&);
    // ...
    template<typename X>
        complex<T>& operator=(const complex<X>&);
    template<typename X>
        complex<T>& operator+=(const complex<X>&);
    // ...
};
```

Отметим, что скалярные операции присваивания принимают ссылочные аргументы. Это неэффективно для типа **float**, так что **complex<float>** принимает их по значению:

```
template<>
class complex<float> {
public:
    // ...
    complex<float>& operator= (float);
    complex<float>& operator+=(float);
    // ...
    complex<float>& operator=(const complex<float>&);
    // ...
};
```

Для **complex<double>** применяется такое же решение. Дополнительно, предоставляются конверсии из **complex<float>** и **complex<double>** (как это описано в §23.4.6):

```
template<>
class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    explicit constexpr complex(const complex<long double>&);
    // ...
};
```

Заметьте, что эти специализированные конструкторы являются **constexpr**, делая **complex<double>** литеральным типом. Этого нельзя сделать для общего случая **complex<T>**. Кроме того, это определение извлекает пользу из знания того факта, что конверсия из **complex<float>** в **complex<double>** безопасна (нет сужения типов), и поэтому допускается неявный конструктор из **complex<float>**. Но конструктор из **complex<long double>** определен явно во избежание сужения.

25.3.1.1. Специализации реализации

Специализацию можно использовать для предоставления альтернативных реализаций шаблонного класса для специфического набора параметров. В этом случае специализация может даже давать представление, отличное от общего шаблона. Например:

```
template<typename T, int N>
class Matrix;              // N-размерная Matrix элементов типа T

template<typename T,0>
class Matrix {             // специализация для N==1
    T val;
    // ...
};

template<typename T,1>
class Matrix {             // специализация для N=1
    T* elem;
    int sz;                // число элементов
    // ...
};

template<typename T,2>
class Matrix {             // специализация для N=2
    T* elem;
    int dim1;      // число строк
    int dim2;      // число столбцов
    // ...
};
```

25.3.2. Первичный шаблон

Когда у нас есть и общее определение шаблона, и специализации, определяющие реализации для специфических наборов аргументов шаблона, мы ссылаемся на наиболее общий вариант шаблона как на *первичный шаблон*. Первичный шаблон задает интерфейс для всех специализаций (§ iso.14.5.5). То есть первичный шаблон определяет, можно ли использовать шаблон каким-то способом и принимает участие в разрешении перегрузки. Специализации рассматриваются после того, как рассмотрен вариант первичного шаблона.

Первичный шаблон нужно определить до всех определений специализаций. Пример:

```
template<typename T>
class List<T*> {
    // ...
};

template<typename T>
class List {    // ошибка: первичный шаблон после специализации
    // ...
};
```

Важнейшей информацией в первичном шаблоне является набор параметров шаблона, который пользователь должен предоставить при использовании шаблона или его специализаций. Если мы определили проверку ограничений для шаблона (§24.4), то они сосредоточены именно в первичном шаблоне, так как концепции — это то, о чем пользователь должен знать и заботиться, чтобы пользоваться шаблоном. Например:

```
template<typename T>
class List {
    static_assert(Regular<T>(),"List<T>: T must be Regular");
    // ...
};
```

По техническим причинам проверки ограничений нужно реплицировать в каждой специализации.

Объявление первичного шаблона достаточно для того, чтобы можно было определить специализацию:

```
template<typename T>
class List;                 // не определение

template<typename T>
class List<T*> {
    // ...
};
```

В случае его использования в программе, первичный шаблон должен быть где-то определен (§23.7). Если же первичный шаблон никогда не конкретизируется, то и определять его не нужно. Это можно использовать для определения шаблона, для которого допускается лишь фиксированный набор альтернативных аргументов. Если пользователь специализирует шаблон, то эта специализация должна находиться в области видимости при каждом использовании шаблона с типом, для которого он был специализирован. Например:

```
template<typename T>
class List {
    // ...
};

List<int*> li;

template<typename T>
class List<T*>{             // ошибка: специализация используется до определения
    // ...
};
```

Здесь **List** был специализирован для **int*** после того, как **List<int*>** был использован.

Существенно, чтобы каждое использование шаблона для заданного набора аргументов было реализовано той же самой специализацией. В противном случае, система типов будет разрушена и разные способы использования шаблона в разных местах программы будут давать разные результаты, а объекты в разных местах программы будут несовместимы. Естественно, что это недопустимо, и программист должен внимательно следить за тем, чтобы явные специализации были совместимыми на протяжении кода программы. По факту, многие реализации C++ позволяют отследить такие несовместимости, но в стандарте рекомендации на эту тему отсутствуют.

Все специализации шаблона должны объявляться в одном и том же пространстве имен с первичным шаблоном. В случае использования явно объявленная специализация (в противоположность сгенерированной из более общего шаблона) должна быть явно определена где-нибудь в программе (§23.7). В частности это означает, что при явной специализации шаблона отсутствуют другие определения, сгенерированные для данной специализации.

25.3.3. Порядок специализаций

Некоторая специализация является *более специализированной*, чем другая, если каждый список аргументов, соответствующий данной специализации, соответствует и другой, но не наоборот. Пример:

```
template<typename T>
    class Vector;              // общий первичный шаблон
template<typename T>
    class Vector<T*>;          // специализация для любого указателя
template<>
    class Vector<void*>;       // специализация для void*
```

Любой тип может использоваться в качестве аргумента шаблона для наиболее общего случая **Vector**, но только указатели могут использоваться для **Vector<T*>** и только **void*** для **Vector<void*>**.

Наиболее специализированной версии будет отдано предпочтение перед другими в объявлениях объектов, указателей и т.д. (§25.3).

25.3.4. Специализации шаблонных функций

Специализации полезны и для шаблонных функций (§25.2.5.1). Но поскольку можно перегружать функции, то используется меньше специализаций. Кроме того, C++ поддерживает лишь полные специализации для функций (§ iso.14.7), так что нам нужно использовать перегрузку там, где в противном случае мы могли бы попробовать специализации.

25.3.4.1. Специализации и перегрузки

Рассмотрим Shell-сортировку из §12.5 и §23.5. Эти версии сравнивают элементы, используя операцию < и свопируют элементы специализированным кодом. А вот более совершенное определение:

```
template<typename T>
bool less(T a, T b)
{
    return a<b;
}

template<typename T>
void sort(Vector<T>& v)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i!=n; ++i)
            for (int j=i-gap; 0<=j; j-=gap)
                if (less(v[j+gap], v[j]))
                    swap(v[j], v[j+gap]);
}
```

Это не исправляет алгоритм сам по себе, но допускает улучшение реализации. Теперь мы имеем **less** и **swap** в качестве именованных сущностей, для которых и можно предоставить усовершенствованные версии.

Как здесь написано, **sort()** не будет сортировать **Vector<char*>** корректным образом, потому что операция **<** будет сравнивать две величины типа **char***. То есть, будут сравниваться адреса первых символов в каждой строке. Вместо этого нам нужно сравнивать сами указуемые символы. Простая специализация **less()** для **const char*** позаботится об этом:

```
template<>
bool less<const char*>(const char* a, const char* b)
{
    return strcmp(a, b)<0;
}
```

Как и в случае классов (§25.3) префикс **template<>** говорит о том, что это специализация, специфицируемая без параметра шаблона. Конструкция **<const char*>** после имени функционального шаблона говорит о том, что эта специализация должна использоваться в случаях, в которых аргумент шаблона есть **const char***. Поскольку аргумент шаблона может выводиться из списка аргументов функции, мы не должны специфицировать его явным образом. Таким образом, мы можем упростить определение специализации:

```
template<>
bool less<>(const char* a, const char* b)
{
    return strcmp(a, b)<0;
}
```

При наличии префикса **template<>** второй случай **<>** является избыточным, так что мы можем написать:

```
template<>
bool less(const char* a, const char* b)
{
    return strcmp(a, b)<0;
}
```

Я предпочитаю более короткую форму объявления. Но можно пойти еще дальше. В последней версии различие между специализацией и перегрузкой стало исчезающе малым и по большей части несущественным, так что можно написать просто

```
bool less(const char* a, const char* b)
{
    return strcmp(a, b)<0;
}
```

Отметим, что мы специализировали **less()** до версии, которая корректна семантически и мы можем подумать, что можно сделать со **swap()**. Стандартная версия **swap()** подходит для нашего случая и уже оптимизирована для любых типов с эффективными перемещающими операциями.

Специализации удобны в случаях нерегулярности в типах аргументов, приводящих к тому, что общий алгоритм выдает неприемлемый результат (вроде **less()** для строк в С-стиле). Часто эти «нерегулярные типы» относятся к встроенным указателям и массивам.

25.3.4.2. Специализации, не являющиеся перегрузкой

Каким образом специализация отличается от перегрузки? С технической точки зрения они различаются, поскольку индивидуальные функции принимают участие в перегрузке, в то время как только первичный шаблон участвует в специализации (§25.3.1.1). Тем не менее, существуют примеры, где такое различие имеет практический смысл.

Существуют варианты использования специализаций функций. Например, мы можем выбирать из функций без аргументов:

```
template<typename T> T max_value(); // нет определения

template<> constexpr int max_value<int>() {return INT_MAX;}
template<> constexpr char max_value<char>() {return CHAR_MAX;}
//...

template<typename Iter>
Iter my_algo(Iter p)
{
  auto x = max_value<Value_type<Iter>>(); // для типов со специализированным max_value()
  // ...
}
```

Я использовал функцию типа **Value_type<>**, чтобы получить тип объекта, указуемого через **Iter** (§24.4.2).

Чтобы получить примерно такой же эффект с помощью перегрузки, мы должны передавать фальшивый (неиспользуемый) аргумент. Пример:

```
int max2(int) {return INT_MAX;}
char max2(char) {return INT_MAX;}

template<typename Iter>
Iter my_algo2(Iter p)
{
    auto x = max2(Value_type<Iter>{}); // для типов с перегруженным max2()
    // ...
}
```

25.4. Советы

[1] Используйте шаблоны для улучшения безопасности типов; §25.1.

[2] Используйте шаблоны для повышения уровня абстракции кода; §25.1.

[3] Используйте шаблоны для реализации гибкой и эффективной параметризации типов и алгоритмов; §25.1.

[4] Помните, что шаблонные параметры-значения должны быть константами времени компиляции; §25.2.2.

[5] Используйте объекты-функции в качестве аргументов для параметризации типов и алгоритмов «политиками»; §25.2.3.

[6] Используйте умолчательные аргументы шаблонов для предоставления простой нотации в простых случаях; §25.2.5.

[7] Специализируйте шаблоны для нерегулярных случаев (например, массивов); §25.3.

[8] Специализируйте шаблоны для оптимизации важных случаев; §25.3.

[9] Определяйте первичный шаблон раньше, чем какие-либо специализации; §25.3.1.1.

[10] Специализации должны находиться в области видимости для всех случаев их использования; §25.3.1.1.

26

Конкретизация

Для любой сложной проблемы
имеется простой, ясный ответ, и он неправильный.
— Х.Л. Менкен

- Введение
- Конкретизация шаблонов

  Когда требуется конкретизация?; Ручное управление конкретизацией
- Привязка имен

  Зависимые имена; Связывание в точке определения; Связывание в точке конкретизации; Множественные точки конкретизации; Шаблоны и пространства имен; Сверхагрессивный поиск на основе зависимости аргументов; Имена из базовых классов
- Советы

26.1. Введение

Одним из главных достоинств шблонов является то, что они являются чрезвычайно гибким механизмом комбинирования кода. Для того, чтобы породить код впечатляющего качества, компилятор комбинирует код (информацию) из

- определения шаблона и его лексического окружения,
- из аргументов шаблона и их лексического окружения, и
- из окружения кода, использующего шаблон.

Ключем к успеху служит возможность для компилятора видеть код во всех этих контекстах одновременно и связывать все вместе, используя всю имеющуюся информацию. А неприятность может заключаться в том, что определение шаблона не столь локализовано, как этого бы хотелось (при прочих равных). Иногда можно запутаться в вопросе, чему соответствует имя, использованное в определении шаблона:

- Это локальное имя?
- Это имя, связанное с аргументом шаблона?
- Это имя из базового класса иерархии?
- Это имя из пространства имен?
- Это глобальное имя?

В настоящей главе обсуждаются такие вопросы, как привязка имен, и ее влияние на стили программирования.

- Шаблоны были введены в §3.4.1 и §3.4.2.
- Глава 23 представила подробное введение в шаблоны и использование аргументов шаблонов.

- Глава 24 содержит введение в обобщенное программирование и в ключевое понятие концепций.
- Глава 25 представляет детали классовых шаблонов и шаблонов функций, а также вводит понятие специализации.
- В Главе 27 будут обсуждаться взаимосвязь между шаблонами и классовыми иерархиями (в поддержку обобщенного и объектно-ориентированного программирования).
- Глава 28 фокусируется на шаблонах как языке для генерации классов и функций.
- В Главе 29 дается большой пример того, как средства языка и стили программирования могут использоваться в комбинации.

26.2. Конкретизация шаблона

Имея определение шаблона и вариант его использования, реализация должна сгенерировать правильный код. Исходя из шаблона класса и набора аргументов шаблона компилятор должен породить определение класса и определения его функций-членов, реально используемых в программе (и только их; §26.2.1). Исходя из шаблона функции и набора аргументов шаблона компилятор должен сгенерировать функцию. Такой процесс часто называют *конкретизацией шаблона.*

Сгенерированные классы и функции называют *специализациями.* Чтобы различать специализации, сгенерированные компилятором, и наши ручные специализации, первые называют *сгенерированными специализациями*, а вторые — *явными специализациями* (или пользовательскими специализациями).

Чтобы использовать шаблоны в нетривиальных программах, программист должен разбираться в том, как имена из определения шаблона связаны с объявлениями, и как можно организовать исходный код (§23.7).

По умолчанию компилятор генерирует классы и функции из шаблонов в согласии с правилами связывания имен (§26.3). То есть, программист не обязан явным образом указывать, какие версии каких шаблонов должны быть сгенерированы. Это важно, ибо программисту нелегко определить необходимые версии шаблонов. Часто программист даже не знает, какие версии шаблонов используются в реализациях библиотек, а иногда имеет место ситуация, когда программист знает версии шаблонов, но не знает, какие типы аргументов шаблона использовались. Например, стандартный контейнер **map** (§4.4.3, §31.4.3) реализуется в терминах шаблона красно-черных деревьев с типами данных и операциями, которые остаются неизвестными большинству пользователей (кроме самых любопытных). В общем случае, набор генерируемых функций можно определить в результате рекурсивного прохода по шаблонам, используемых в библиотеках приложения. Компьютеры лучше выполняют такую работу.

С другой стороны, иногда важно для программиста определить, где именно код должен генерироваться из шаблона (§26.2.2). В таком случае программист получает детальный контроль над контекстом конкретизации.

26.2.1. Когда требуется конкретизация

Специализацию классового шаблона нужно генерировать только в том случае, когда нужно определение класса (§ iso.14.7.1). В частности, чтобы объявить указатель на некоторый класс, актуальное определение класса не нужно. Например:

```
class X;
X* p;          // OK: нет требуемого определения X
X a;           // ошибка: требуется определение X
```

Это различие важно иметь в виду, определяя классовые шаблоны. Шаблонный класс не конкретизируется, если только действительно не потребуется его определение. Например:

```
template<typename T>
class Link {
    Link* suc;          // OK: нет требуемого определения Link (пока что)
    // ...
};

Link<int>* pl;          // нет требующейся конкретизации Link<int>  (пока что)

Link<int> lnk;          // теперь мы должны конкретизировать Link<int>
```

Место, где шаблон используется, задает точку кокретизации (§26.3.3).

Реализация конкретизирует шаблонную функцию только, если функция реально используется (вызывается или же если берется ее адрес). Конкретизация шаблонного класса не подразумевает кокретизацию всех его функций-членов. Это дает программисту важную степень гибкости при определении шаблонного класса. Пример:

```
template<typename T>
class List {
    // ...
    void sort();
};

class Glob {
    // ... нет операций сравнения ...
};

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // ... используем операции для lb, но не lb.sort() ...
}
```

Здесь **List<string>::sort()** конкретизируется, а **List<Glob>::sort()** не конкретизируется. Это помогает сократить объем генерируемого кода и избавляет от необходимости перепроектирования программы. Если бы **List<Glob>::sort()** генерировалась, то нам пришлось бы добавить в **Glob** операции, необходимые для **List::sort()**, или пришлось бы переопределить **sort()** так, чтобы он не был членом **List** (это более приемлемый дизайн) или использовать какой-либо иной контейнер для типа **Glob**.

26.2.2. Ручной контроль конкретизации

Язык не требует со стороны пользователя каких-либо явных действий для достижения конкретизации шаблона. Тем не менее, он предоставляет два механизма, помогающих пользователю взять управление процессом на себя, если в этом будет необходимость. Такая необходимость возникает из желания

- оптимизировать процессы компиляции и компоновки путем устранения повторных (избыточных) конкретизаций,
- чтобы узнать точно, какая точка конкретизации реально используется, и устранить сюрпризы в случае сложных контекстов связывания имен.

Запрос явной конкретизации — это объявление специализации с префиксом **template**, за которым не следует знак **<**:

```
template class vector<int>;                    // класс
template int& vector<int>::operator[](int);    // функция-член
template int convert<int, double>(double);     // функция не член
```

Ключевое слово **template** без знака < стартует запрос на конкретизацию. Отметим, что ключевое слово **template** предваряет полное объявление; одного объявления имени не достаточно:

```
template vector<int>::operator[];              // синтаксическая ошибка
template convert<int, double>;                 // синтаксическая ошибка
```

Как и в случае вызова шаблонной функции можно опустить аргументы, которые могут быть выведены (§23.5.1). Например:

```
template int convert<int, double>(double);     // OK (избыточно)
template int convert<int>(double);             // OK
```

Когда классовый шаблон конкретизируется явно, все члены-функции также конкретизируются.

Выигрыш во времени компоновки и эффективность перекомпиляции в случае запросов на конкретизацию могут быть весьма существенными. Я знаю примеры, где сосредотачивание большинства случаев конкретизации шаблонов в одну единицу трансляции приводило к уменьшению времени компиляции с часов до минут.

Ошибкой является наличие двух определений для одной и той же специализации. Не имеет значения, являются ли такие множественные специализации пользовательскими (§25.3), неявно генерируемыми (§23.2.2) или же запрашиваемыми явно. Компилятор, однако, не обязан разбираться со множественными конкретизациями в разных единицах трансляции. Это открывает возможность умной композиции программ из библиотек, использующих явные конкретизации. Но не все реализации яыка являются столь умными. В таком случае, пользователи менее умных реализаций должны избегать множественных конкретизаций. Самым страшным последствием может стать отказ в компоновке, но не будет молчаливых изменений смысла программы.

В качестве дополнения к явным запросам на конкретизацию язык предоставляет возможность явного запроса на отсутствие конкретизации (**extern template**). Очевидным примером служит ситуация, когда можно запросить явную конкретизацию для специализации, но в иных единицах трансляции применить **extern template**. Это напоминает классическую ситуацию с одним определением и множеством объявлений (§15.2.3). Пример:

```
#include "MyVector.h"

extern template class MyVector<int>;   // подавляем неявную конкретизацию
                                       // явно конкретизирем где-то еще

void foo(MyVector<int>& v)
{
    // ... здесь используем вектор ...
}
```

Код «где либо» мог бы выглядеть следующим образом:

```
#include "MyVector.h"
template class MyVector<int>; // конкретизируем в данной единице трансляции;
                              // используем эту точку конкретизации
```

В дополнение к генерированию специализаций для всех членов класса явная конкретизация также задает единственную точку конкретизации так, что остальные точки кокретизации (§26.3.3) могут игнорироваться. Одним из способов добиться этого заключается в том, чтобы поместить явную кокретизацию в разделяемую библиотеку.

26.3. Связывание имен

Следует определять шаблонные функции так, чтобы минимизировать зависимости от нелокальной информации. Причиной является желание использовать шаблон для генерации функций и классов на основе неизвестных типов и в неизвестных контекстах. Даже незначительная контекстная зависимость может стать источником проблем, особенно если разработчик не хочет разбираться в мельчайших деталях реализации шаблона. Общее правило избегать глобальных имен насколько возможно приобретает особую важность в шаблонном коде. Мы стараемся сделать определение шаблона по-возможности самодостаточным и предоставлять то, что могло бы извлекаться из глобального контекста, в форме аргументов шаблона (§28.2.4, §33.1.3). Используйте концепции для документации зависимостей от аргументов шаблона (§24.3).

Однако, некоторые нелокальные имена должны использоваться, чтобы достичь наиболее элегантной формулировки шаблона. В частности, часто пишут набор взаимодействующих шаблонных функций вместо того, чтобы определить одну самодостаточную функцию. Иногда такие функции могут быть функциями-членами класса, но это не обязательно. В некоторых случаях нелокальные функции являются наилучшим выбором. Типичным примером послужат обращения функции **sort()** к функциям **swap()** и **less()** (§25.3.4). Алгоритмы стандартной библиотеки являются одним большим примером этого (Глава 32). Когда требуется, чтобы нечто было нелокальным, используйте пространства имен, а не глобальное пространство. Это помогает сохранить относительную локальность.

Операции с традиционными именами и семантикой, такие как **+**, *****, **[]** и **sort()**, служат дополнительными примерами источников нелокальных имен в определении шаблонов. Рассмотрим следующее:

```
bool tracing;

template<typename T>
T sum(std::vector<T>& v)
{
    T t {};
    if (tracing)
        cerr << "sum(" << &v << ")\n";
    for (int i = 0; i!=v.size(); i++)
        t = t + v[i];
    return t;
}

// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
    Quad c = sum(v);
}
```

Невинно выглядящая шаблонная функция **sum()** зависит от нескольких имен, не имеющих явной спецификации в ее определении; это имена **tracing**, **cerr** и операция **+**. В этом примере **+** определен в **<quad.h>**:

```
Quad operator+(Quad, Quad);
```

Важно то, что нет ничего, что имеет отношение к **Quad** в пространстве имен, в котором определяется **sum()**, а разработчик **sum()** может ничего не знать о **Quad**. Например,

операция + может быть определена в тексте программы позже определения **sum()**, и даже позже по времени.

Процесс нахождения объявлений для каждого имени, явно или неявно используемого в шаблоне, называется *связыванием имен*. Общая проблема в связи со связыванием имен в шаблонах проистекает из того, что три контекста вовлечены в конкретизацию шаблона, и их нельзя отчетливо разделить:

1. Контекст определения шаблона
2. Контекст объявления типов аргументов
3. Контекст использования шаблона

Определяя шаблонную функцию, мы хотим быть уверены в том, что имеется вполне достаточный для определения шаблона контекст, имеющий смысл в терминах действительных аргументов без извлечения случайных вещей из точки использования шаблона. Язык в качестве помощи разделяет имена из определения шаблона на две категории:

1. *Зависимые имена*: имена, зависящие от параметров шаблона. Такие имена связываются в точке конкретизации (§26.3.3). В примере с **sum()** определение + может быть обнаружено в контексте конкретизации, поскольку оно связано с типом аргументов шаблона.
2. *Независимые имена*: имена, не зависящие от параметров шаблона. Такие имена связываются в точке определения шаблона (§26.3.2). В примере с **sum()** шаблон **vector** определяется в стандартном заголовочном файле **<vector>**, а логическая переменная **tracing** — в области видимости, в которой компилятор сталкивается с определением **sum()**.

Зависимые и независимые имена должны быть либо в области видимости в точке их использования, или должны находиться в процессе просмотра и поиска зависимостей аргументов (§14.2.4).

Последующие разделы посвящены серьезным техническим деталям, относящимся к вопросу о том, как зависимые и независимые имена из определения шаблона связываются для специализаций. За полными разъяснениями обратитесь к § iso.14.6.

26.3.1. Зависимые имена

Простейшее определение того, что «**N** зависит от параметра шаблона **T**» будет таковым: «**N** является членом **T**». К сожалению, этого бывает недостаточно; примером служит сложение объектов **Quad** в подсчитывающем примере.

Следовательно, вызов функции является зависимым от аргумента шаблона тогда и только тогда, когда выполняются следующие условия:

1. Тип действительного аргумента зависит от шаблонного параметра **T** согласно правилам вывода типа (§23.5.2), например **f(T(1))**, **f(T)**, **f(g(t))** и **f(&t)**, в предположении, что **t** есть **T**.
2. Вызванная функция имеет параметр, зависящий от **T** согласно правилам вывода типа (§23.5.2), например **f(T)**, **f(list<T>&)** и **f(const T*)**.

В своей основе имя вызванной функции зависимое, если оно разыскивается на основе аргументов или формальных параметров. Например:

```
template<typename T>
T f(T a)
{
    return g(a);    // OK: a зависимое имя и поэтому таково и g
}
```

```
class Quad {/* ... */};
void g(Quad);

int z = f(Quad{2});        // g у f связывается с g(Quad)
```

Вызов, который случайно имеет аргумент, соответствующий действительному типу шаблонного параметра, не является зависимым. К примеру:

```
class Quad {/* ... */};

template<typename T>
T ff(T a)
{
    return gg(Quad{1}); // ошибка: нет gg() в области видимости и gg(Quad{1}) не зависит от T
}

int gg(Quad);

int zz = ff(Quad{2});
```

Если бы **gg(Quad{1})** рассматривалось как зависимое, то его смысл бы бы мистическим для читателя определения шаблона. Если программисту потребуется вызов **gg(Quad)**, то объявление для **gg(Quad)** должно помещаться перед определением **ff()**, так что **gg(Quad)** будет находиться в области видимости в момент, когда будет анализироваться **ff()**. Это такое же правило, как правило для нешаблонных определений функции (§26.3.2).

По умолчанию, зависимое имя полагается именем чего-либо, что не является типом. Таким образом, чтобы использовать зависимое имя в качестве типа, вы должны явно заявить об этом, используя ключевое слово **typename**. Пример:

```
template<typename Container>
void fct(Container& c)
{
  Container::value_type v1 = c[7]; // синтаксич. ошибка: полагается, что value_type не имя типа
  typename Container::value_type v2 = c[9];      // OK: value_type – имя типа
  auto v3 = c[11];                               // OK: пусть компилятор сам разберется
  // ...
}
```

Можно избежать столь неуклюжего использования ключевого слова **typename**, введя синоним типа (§23.6). Например:

```
template<typename T>
using Value_type<T> = typename T::value_type;

template<typename Container>
void fct2(Container& c)
{
    Value_type<Container> v1 = c[7]; // OK
    // ...
}
```

Применение имени шаблона после. (точка),—> или:: требует аналогичного использования ключевого слова **template**. Например:

```
class Pool {   // некоторый аллокатор
public:
    template<typename T> T* get();
    template<typename T> void release(T*);
    // ...
};
```

```
template<typename Alloc>
void f(Alloc& all)
{
    int* p1 = all.get<int>();          // синтакс. ошибка: get полагается именем не шаблона
    int* p2 = all.template get<int>(); // OK: get() полагается шаблоном
    // ...
}

void user(Pool& pool){
{
    f(pool);
    // ...
}
```

По сравнению с применением **typename** для явной идентификации имени в качестве имени типа применение ключевого слова **template** для явного обозначения имени как имени шаблона более редкое. Обратите внимание на расположение сбивающего с току ключевого слова: **typename** появляется перед квалифицированным именем, а **template** — непосредственно перед именем шаблона.

26.3.2. Связывание в точке определения

Когда компилятор видит определение шаблона, он выявляет, какие имена зависимые (§26.3.1). Если имя зависимое, то рассмотрение его объявления откладывается до момента конкретизации (§26.3.3).

Имена, не зависящие от аргумента шаблона, трактуются как имена, не входящие в определение шаблона; они должны быть в области видимости (§6.3.4) в точке определения. Например:

```
int x;
template<typename T>
T f(T a)
{
    ++x;          // OK: x в области видимости
    ++y;          // ошибка: y нет в области видимости, и y не зависит от T
    return a;     // OK: a зависимое
}

int y;

int z = f(2);
```

Если объявление находится, то это объявление используется даже, если «более хорошее» объявление найдется позже. К примеру:

```
void g(double);
void g2(double);

template<typename T>
int ff(T a)
{
    g2(2);    // вызов g2(double);
    g3(2);    // ошибка: g3() нет в области видимости
    g(2);     // вызов g(double); g(int) не в области видимости
    // ...
}

void g(int);
void g3(int);

int x = ff(a);
```

Здесь **ff()** вызовет **g(double)**. Определение **g(int)** появляется слишком поздно, чтобы его рассматривать — как и в случае нешаблонной **ff()** или когда **g** именовало бы переменную.

26.3.3. Связывание в точке конкретизации

Контекст для распознавания смысла зависимого имени (§26.3.1) определется в момент применения шаблона с заданным набором аргументов. Это называется точкой конкретизации для данной специализации (§ iso.14.6.4.1). Каждое использование шаблона с заданным набором аргументов формирует точку конкретизации. Для функционального шаблона эта точка находится в ближайшей глобальной области видимости или в области видимости пространства имен, включающей эту точку, сразу после объявления, содержащего этот акт использования. Например:

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);       // g связывается в точке конкретизации
}

void h(int i)
{
    extern void g(double);
    f(i);
}
// точка объявления f<int>
```

Точка конкретизации для **f<int>()** находится вне **h()**. Это существенно для гарантии того, что **g()**, вызываемое в **f()**, является глобальной **g(int)**, а не локальной **g(double)**. Неквалифицированное имя, использованное в определении шаблона, никогда не может связываться с локальным именем. Игнорирование локальных имен существенно для предотвращения мгоних неприятных ошибок.

Чтобы сделать возможными рекурсивные вызовы, точка объявления шаблонной функции должна располагаться позже объявлений с ее конкретизацией. Например:

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);           // g связывается в точке конкретизации
    if (i) h(a-1);  // h связывается в точке
}

void h(int i)
{
    extern void g(double);
    f(i);
}
// точка объявления f<int>
```

Здесь расположение точки конкретизации после определения **h()** необходимо, чтобы был разрешен (косвенно, рекурсивно) вызов **h(a-1)**.

Для шаблона класса или члена класса точка конкретизации располагается непосредственно перед объявлением, содержащим их использование:

```
template<typename T>
class Container {
    vector<T> v;          // элементы
    // ...
public:
    void sort();          // сортировка элементов
    // ...
};

// точка конкретизации Container<int>
void f()
{
    Container<int> c;     // точка использования
    c.sort();
}
```

Если точка конкретизации была бы позже **f()**, то вызов **c.sort()** не смог бы найти определение для **Container<int>**.

Опора на аргументы шаблона для вскрытия неявных зависимостей упрощает наше представление о шаблонном коде и даже позволяет нам добраться до локальной информации. Пример:

```
void fff()
{
    struct S {int a, b;};
    vector<S> vs;
    // ...
}
```

Здесь **S** это локальное имя, но поскольку мы используем его как явный аргумент, а не пытаемся «закопать» его в определении **vector**, то мы избегаем потенциальных тонких неясностей.

Итак, почему мы не избавляемся полностью от нелокальных имен в определениях шаблона? Это решило бы техническую проблему с поиском имен — но как и для обычных функций и определений классов — мы хотим свободно использовать все возможности относительно иных фкнкций и типов в нашем коде. Превращение всех зависимостей в аргументы может привести к запутанному коду. Например:

```
template<typename T>
void print_sorted(vector<T>& v)
{
    sort(v.begin(), v.end());
    for (const auto T& x : v)
        cout << x << '\n';
}

void use(vector<string>& vec)
{
    // ...
    print_sorted(vec);    // сортировка посредством std::sort, печать посредством std::cout
}
```

Здесь мы используем два нелокальных имени (**sort** и **cout**, оба из стандартной библиотеки). Чтобы избавиться от них, нужно добавить параметры:

```
template<typename T, typename S>
void print_sorted(vector<T>& v, S sort, ostream& os)
{
    sort(v.begin(), v.end());
```

```
    for (const auto T& x : v)
        os << x << '\n';
}

void fct(vector<string>& vec)
{
    // ...
    using Iter = decltype(vs.begin());          // тип итератора для vec
    print_sorted(some_vec, std::sort<Iter>, std::cout);
}
```

В этом тривиальном случае можно привести массу аргументов в пользу того, чтобы убрать зависимость от глобального имени **cout**. Однако в общем случае, что иллюстрируется с помощью **sort()**, добавление параметров может сделать код более гибким, но не более простым в понимании.

Кроме того, если бы правила связывания имен для шаблонов были бы более ограничительными, чем правила для нешаблонного кода, то тогда навыки написания шаблонного кода стали бы иной профессией, чем написание нешаблонного кода. Нешаблонный код и шаблонный перестали бы быть взаимозаменяемыми.

26.3.4. Множественные точки конкретизации

Специализация шаблона может генерироваться

- в любой точке конкретизации (§26.3.3),
- в любой последующей точке в единице трансляции,
- или в единице трансляции, специально предназначенной для генерации специализаций.

Это отражает три очевидные стратегии, которые реализации могут использовать для генерации специализаций:

1. Генерировать специализацию при первом использовании.
2. В конце единицы трансляции генерировать все необходимые для нее специализации.
3. Для всех единиц трансляции генерировать все необходимые программе специализации.

Все три стратегии имеют свои сильные и слабые стороны, а также возможны комбинации этих стратегий.

Итак, шаблон, используемый много раз с одним и тем же набором шаблонных аргументов, имеет несколько точек конкретизации. Программа становится неприемлемой, если возможно получить разные решения, выбрав разные точки конкретизации. То есть, если связывание для зависимых или независимых имен различаются, то программа недопустима. Пример:

```
void f(int);      // забота о целых (int)

namespace N {
    class X {};
    char g(X, int);
}

template<typename T>
void ff(T t, double d)
{
    f(d);             // f связывается с f(int)
    return g(t, d);   // g могло бы связываться с g(X, int)
}
```

```
auto x1 = ff(N::X{},1.1); // ff<N::X, double>; g может связываться с N::g(X, int), сужая 1.1 до 1

Namespace N {      // переоткрываем N, заботясь о double
    double g(X, double);
}

auto x2 = ff(N::X,2.2); // ff<N::X, double>; g связывается с N::g(X, double); наилучшее соотв-вие
```

Для **ff()** мы имеем две точки конкретизации. Для первого случая вызова мы могли бы генерировать специализацию в момент инициализации **x1** и получить вызов **g(N::X, int)**. Иначе мы могли бы подождать и сгенерировать специализацию в конце единицы трансляции и получить вызов **g(N::X, char)**. Как следствие, вызов **ff(N::X{},1.1)** является ошибочным.

Неряшливо вызывать перегруженную функцию из промежутка между двумя ее определениями. Глядя на большую программу, у программиста не найдется причины заподозрить проблему. В таком случае компилятор может помочь выявить двусмысленность. Такие проблемы могут возникать в разных единицах трансляции и обнаружение проблемы становится существенно более сложным (как для программиста, так и для компилятора). Релазации не могут гарантировать решение такой проблемы.

Чтобы избежать сюрпризов со связыванием имен, постарайтесь ограничить контекстные зависимости в шаблонах.

26.3.5. Шаблоны и пространства имен

Когда функция вызывается, ее объявление может быть найдено, даже если оно не находится в области видимости, при условии, что оно находится в том же самом пространстве имен, что и один из аргументов (§14.2.4). Это важно для функций, вызываемых в определениях шаблона, потому что это входит в механизм, посредством которого функции находятся во время конкретизации. Связывание зависимых имен выполняется (§ iso.14.6.4.2) просмотром

1. имен из области видимости в точке, в которой шаблон определяется, плюс
2. просмотром имен из пространства имен аргумента зависимого вызова (§14.2.4).

Пример:

```
namespace N {
    class A {/* ... */};
    char f(A);
}

char f(int);

template<typename T>
char g(T t)
{
    return f(t);      // выбираем f() в зависимости от того, что есть T
}

char f(double);

char c1 = g(N::A());      // заставляет вызвать N::f(N::A)
char c2 = g(2);           // заставляет вызвать f(int)
char c3 = g(2.1);         // заставляет вызвать f(int); f(double) не рассматривается
```

Здесь вызов **f(t)** явно зависимый, так что мы не можем связать **f** в точке определения. Для генерации специализации для **g<N::A>(N::A)** реализация просматривает в пространстве имен **N** функции **f()** и находит **N::f(N::A)**.

Функция **f(int)** находится потому, что она находится в области видимости в точке определения шаблона. Функция **f(double)** не находится, поскольку она не находится в области видимости в точке определения шаблона (§ iso.14.6.4.1), и зависимый от аргументов просмотр (§14.2.4) не находит глобальную функцию, которая принимает только аргументы встроенных типов. Я полагаю, что про это легко забыть.

26.3.6. Сверхагрессивный поиск на основе зависимости аргументов

Сверхагрессивный поиск на основе зависимости аргументов полезен, чтобы избежать большой многословности (§14.2.4). Например:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world" << endl;     // OK из-за ADL
}
```

Без поиска на основе аргументов манипулятор **endl** не был бы найден. Компилятор обнаруживает, что первый аргумент у операции **<<** имеет тип **ostream**, определенный в **std**. Поэтому он смотрит в **std** насчет **endl** и находит его (в **<iostream>**).

Однако, может получиться слишком большая агрессивность поиска на основе аргументов в случае комбинирования с неограниченными шаблонами. Рассмотрим следующее:

```
#include<vector>
#include<algorithm>
// ...

namespace User {
  class Customer {/* ... */};
  using Index = std::vector<Customer*>;

  void copy(const Index&, Index&, int deep); // глубок. или поверхност. копир-ние, в зависимости от глубины

  void algo(Index& x, Index& y)
  {
      // ...
      copy(x, y, false);  // ошибка
  }
}
```

Неплохим предположением будет считать, что автор **User** думал о **User::alg()** при вызове **User::copy()**. Однако ж, происходит другое. Компилятор обнаруживает, что **Index** это **vector**, определенный в **std**, и ищет подходящую функцию, доступную в **std**. В файле **<algorithm>** он находит:

```
template<typename In, typename Out>
Out copy(In, In, Out);
```

Очевидно, что этот общий шаблон идеально соответствует **copy(x, y, false)**. С другой стороны, **copy()** в **User** может быть вызван только с конверсией **bool**-в-**int**. Для этого примера, как и для других эквивалентных примеров, выбор компилятора будет сюрпризом для большинства программистов и источником неясных ошибок. Применение агрессивного поиска на основе аргументов возможно является недостатком дизайна языка. В конце концов, **std::copy()** требует пары итераторов (не просто двух аргументов одинакового типа, как

например двух переменных типа **Index**). Многие подобные проблемы могут быть решены использованием концепций (§24.3, §24.3.2). Например, если бы компилятор знал, что **std::copy()** требует двух итераторов, то возникла бы очевидная ошибка.

```
template<typename In, typename Out>
Out copy(In p1, In p2, Out q)
{
    static_assert(Input_iterator<In>(), "copy(): In is not an input iterator");
    static_assert(Output_iterator<Out>(), "copy(): Out is not an output iterator");
    static_assert(Assignable<Value_type<Out>, Value_type<In>>(), "copy(): value type mismatch");
    // ...
}
```

Хорошо, что компилятор обнаруживает, что **std::copy()** не является корректным кандидатом для этого вызова, и будет вызван **User::copy()**. К примеру (§28.4):

```
template<typename In, typename Out,
    typename = enable_if(Input_iterator<In>()
            && Output_iterator<Out>()
            && Assignable<Value_type<Out>, Value_type<In>>())>
Out copy(In p1, In p2, Out q)
{
    // ...
}
```

К сожалению, многие из таких шаблонов находятся в библиотеках, которые пользователь не может модифицировать (например, в стандартной библиотеке).

Хорошей идеей будет избегать излишне общих шаблонов функций в заголовочных файлах, которые к тому же содержат определения типов, но избежать этого трудно.

Что же может сделать пользователь, если в библиотеке содержится вызывающий проблемы неограниченный шаблон? Часто нам известно, из какого пространства имен происходит нужная нам функция, так что мы можем явно ее специфицировать. Например:

```
void User::algo(Index& x, Index& y)
{
    User::copy(x, y, false);    // OK
    // ...
    std::swap(*x[i],*x[j]);     // OK: лишь std::swap рассматривается
}
```

Если мы не хотим точно специфицировать пространство имен, но хотим, чтобы конкретная версия функции рассматривалась в процессе разрешения перегрузки, то мы может применить **using**-объявление (§14.2.2). К примеру:

```
template<typename Range, typename Op>
void apply(const Range& r, Op f)
{
    using std::begin;
    using std::end;
    for (auto& x : r)
        f(x);
}
```

Теперь стандартные **begin()** и **end()** находятся в наборе перегрузки, примененной в операторе range-**for**, предназначенном для траверса по **Range** (если только в **Range** есть члены **begin()** и **end()**; §9.5.1).

26.3.7. Имена из базовых классов

Когда шаблонный класс имеет базовый класс, он может обращаться к именам из базового класса. Как и в случае иных имен, имеются две возможности:

- Базовый класс зависит от аргумента шаблона.
- Базовый класс не зависит от аргумента шаблона.

Последний случай прост и рассматривается так же, как базовые классы в нешаблонных классах. Например:

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public B {
public:
    void h(int);
    void f()
    {
        g(2);       // вызов B::g(char)
        h(2);       // вызов X::h(int)
    }
    // ...
};
```

Как всегда, локальные имена скрывают остальные имена, так что **h(2)** связывается с **X::h(int)** и **B::h(char)** не рассматривается. Аналогично, вызов **g(2)** связывается с **B::g(char)** без внимания к функциям, объявленным вне **X**. То есть глобальная функция **g()** не рассматривается

Для базовых классов, зависящих от параметра шаблона, мы должны быть чуть более осторожными и явно специфицировать то, что нам нужно. Рассотрим следующее:

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public T {
public:
    void f()
    {
        g(2);       // вызов::g(int)
    }
    // ...
};

void h(X<B> x)
{
    x.f();
}
```

Почему здесь **g(2)** не дает вызов **B::g(char)**? Потому что **g(2)** не зависит от шаблонного параметра **T**. Этот вызов связывается в точке определения; имена из параметра шаблона **T** (используется как базовый класс) еще не известны и поэтому не рассматриваются. Если

мы хотим, чтобы имена из зависимого класса рассматривались, то мы должны сделать зависимость явной. Этого можно добиться тремя способами:

- Квалифицировать имя зависимым типом (вроде **T::g**).
- Сказать, что имя ссылается на объект этого класса (например, **this->g**).
- Внести имя в область видимости посредством **using**-объявления (вроде **using T::g**).

Пример:

```
void g(int);
void g2(int);

struct B {
    using Type = int;
    void g(char);
    void g2(char)
};

template<typename T>
class X : public T {
public:
    typename T::Type m;     // OK
    Type m2;                // ошибка (Type нет в области видимости)

    using T::g2();          // вносит T::g2() в область видимости

    void f()
    {
        this->g(2);         // вызов  T::g
        g(2);               // вызов  ::g(int); сюрприз?
        g2(2);              // вызов  T::g2
    }
    // ...
};

void h(X<B> x)
{
    x.f();
}
```

Только в точке конкретизации мы можем узнать, имеет ли тип аргумента, использованный для параметра **T** (здесь **B**), требуемые имена.

Легко забыть квалифицировать имена из базового класса, и квалифицированный код часто выглядит несколько многословным и неряшливым. Тем не менее, альтернативой будет то, что имя из шаблона будет иногда связываться с членом базового класса, а иногда — с глобальным именем, в зависимости от аргумента шаблона. Это явно не идеально, и поэтому неявное правило, чтобы шаблоны были самодостаточными до возможной степени, поддерживается правилами языка (§26.3).

Квалифицировать доступ к членам базового для шаблона класса может быть утомительным. В то же время, явная квалификация помогает программистам при поддержке кода. Общим примером является ситуация, когда целая иерархия классов шаблонизируется. Например:

```
template<typename T>
class Matrix_base {// память под матрицы, операции над элементами
    // ...
    int size() const {return sz;}
protected:
    int sz;         // число элементов
    T* elem;        // элементы матрицы
};
```

```
template<typename T, int N>
class Matrix : public Matrix_base<T> { // N-мерная матрица
    // ...
    T* data()                          // возврат указателя на хранилище элементов
    {
        return this->elem;
    }
};
```

Здесь квалификация **this->** нужна.

26.4. Советы

[1] Пусть компилятор/реализация генерируют специализации по необходимости; §26.2.1.

[2] Используйте явную конкретизацию, если вам нужен полный контроль окружения конкретизации; §26.2.2.

[3] Используйте явную конкретизацию, если вам нужно оптимизировать время для генерации специализаций; §26.2.2.

[4] Избегайте неясных контекстных зависимостей в определениях шаблона; §26.3.

[5] Имена должны находиться в области видимости в момент их использования в определении шаблона или доступные для поиска, зависящего от аргументов; §26.3, §26.3.5.

[6] Сохраняйте контекст конкретизации неизменным между точками конкретизации; §26.3.4.

[7] Избегайте слишком общих шаблонов, которые могут находиться в результате сверхагрессивного поиска в зависимости от аргументов; §26.3.6.

[8] Используйте концепции и/или **static_assert**, чтобы избежать использования неподходящих шаблонов; §26.3.6.

[9] Применяйте **using**-объявления для ограничения работы сверхагрессивного поиска в зависимости от аргументов; §26.3.6.

[10] Квалифицируйте имена из родительского класса с помощью **->** или **T::**; §26.3.7.

27

Шаблоны и иерархии

Евклид и пятая симфония Бетховена;
знание лишь одного из перечисленного отмечает вас
как полуобразованного.
— Стэн Келли-Бутл

- Введение
- Параметризация и иерархии
  Генерируемые типы; Конверсия шаблонов
- Иерархии классовых шаблонов
  Шаблоны как интерфейсы
- Параметры шаблона в качестве базовых классов
  Композиция структур данных; Линеаризация классовых иерархий
- Советы

27.1. Введение

Шаблоны и наследование являются являются механизмами построения новых типов, исходя из существующих типов, для специфицирования интерфейсов и вообще для написания полезного кода, который отражает разные формы общности:

- Шаблонный класс определяет интерфейс. Собственная реализация шаблона и его специализации могут быть доступны через этот интерфейс. Исходный код, реализующий шаблон, идентичен для всех типов параметров. Реализации различных специализаций могут сильно отличаться, но все они должны отражать семантику, специфицированную для первичного шаблона. Специализация может добавить функциональность к тому, что предлагает первичный шаблон.
- Базовый класс определяет интерфейс. Собственная реализация этого класса и реализации производных классов могут (используя виртуальные функции) быть доступны через этот интерфейс. Реализации разных производных классов могут разниться друг от друга, но все они должны реализовывать семантику, специфицированную для базового класса. Производный класс может добавлять функциональность к тому, что предлагает базовый класс.

С точки зрения перспектив дизайна оба подхода достаточно близки, так что для них допустимо одинаковое название. Оба позволяют однажды выразить какой-либо алгоритм, применимый ко множеству типов; в итоге люди называют их *полиморфными* (с греческого — много форм). Для их различения то, что предоставляется виртуальными функциями,

называется *полиморфизмом времени выполнения*, а то, что предоставляется шаблонами — называется *полиморфизмом времени компиляции* или *параметрическим полиморфизмом*.

Дуализм обобщенного и объектно-ориентированного подходов обманчив. Программисты в парадигме объектно-ориентированного подхода фокусируются на иерархии классов (типов) с интерфейсом в виде индивидуального класса (Глава 21). Программисты в обобщенной парадигме фокусируются на проектировании алгоритмов с концепциями для аргументов шаблонов, обеспечивая интерфейс, воспринимающий много типов (Глава 24). В идеальном случае программист должен овладеть обоими подходами, так чтобы можно было выбирать наиболее подходящий для конкретного случая. Часто оптимальный дизайн сочетает в себе оба подхода. Например, **vector<Shape*>** реализует контейнер, полиморфный во время компиляции, и который содержит элементы из полиморфной (времени выполнения) иерархии (объектно-ориентированной) (§3.2.4).

В общем случае, объектно-ориентированное программирование предполагает более глубокий взгляд в будущее, нежели обобщенное программирование, поскольку все типы в иерархии должны явным образом разделять интерфейс, задаваемый базовым классом. Шаблон примет любой тип, который допускает свою концепцию в виде аргумента, даже если нет явно заданной общности между этими типами. Например, **accumulate()** (§3.4.2, §24.2, §40.6.1) примет вектора целых (**int**) и списки из **complex<double>**, даже если нет явно объявленной связи между двумя этими типами элементов и нет объявленной связи между этими двумя типами последовательностей.

27.2. Параметризация и иерархии

Как показано в §4.4.1 и §27.2.2, комбинации шаблонов и классовых иерархий являются базисом для многих полезных техник программирования. Итак:

- Когда мы выбираем шаблон класса?
- Когда мы полагаемся на иерархию классов?

Рассмотрим эти вопросы с несколько упрощенной и абстрактной позиции:

```
template<typename X>
class Ct {          // интерфейс в терминах параметра
    X mem;
public:
    X f();
    int g();
    void h(X);
};

template<>
class Ct<A> {       // специализация (для A)
    A* mem;         // представление может отличаться от первичного шаблона
public:
    A f();
    int g();
    void h(A);
    void k(int);    // добавленная функциональность
};

Ct<A> cta;          // специализация для A
Ct<B> ctb;          // специализация для B
```

Исходя из этого, мы можем использовать **f()**, **g()** и **h()** на переменных **cta** и **ctb**, используя реализации для **Ct<A>** и **Ct<B>**, соответственно. Я использовал явную специализацию

(§23.5.3.4), чтобы показать, что реализации могут отклоняться от того, что предлагает первичный шаблон и что возможно добавлять функциональность. Более простой случай — без добавления функциональности — более распространен на практике.

Приблизительный эквивалент с применением иерархии выглядит следующим образом:

```
class X {
    // ...
};

class Cx {                  // интерфейс, выраженный в терминах типов области видимости
    X mem;
public:
    virtual X& f();
    virtual int g();
    virtual void h(X&);
};

Class DA : public Cx {  // производный класс
public:
    X& f();
    int g();
    void h(X&);
};

Class DB : public Cx {  // производный класс
    DB*p;           // представление может быть усилено по сравнению с базовым классом
public:
    X& f();
    int g();
    void h(X&);
    void k(int);            // добавленная функциональность
};

Cx& cxa {*new DA};      // cxa – интерфейс к DA
Cx& cxb {*new DB};      // cxb – интерфейс к DB
```

Исходя из этого, мы можем использовать **f()**, **g()** и **p()** на переменных **cxa** и **cxb**, используя реализации **DA** и **DB**, соответственно. Я использовал ссылки в иерархической версии, чтобы отразить тот факт, что мы должны манипулировать объектами производных классов через указатели или ссылки для достижения полиморфизма времени выполнения.

В любом случае мы манипулируем объектами, разделяющими общий набор операций. Исходя из этой упрощенной и абстрактной точки зрения мы наблюдаем следующее:

- Если типы в интерфейсе к сгенерированному или производному классам должны различаться, то преимущество получают шаблоны. Для получения доступа к различающимся интерфейсам к производным классам через базовый класс, мы должны применить какую-либо из форм явного приведения типов (§22.2).
- Если реализации сгенерированных или производных классов различаются лишь в параметре или же различаются лишь в нескольких специальных случаях, то преимущество имеют шаблоны. Нерегулярные реализации могут быть выражены посредством производных классов или специализациями.
- Если действительные типы используемых объектов не известны во время компиляции, то существенно применение классовых иерархий.
- Если иерархические связи требуются в отношении сгенерированных или производных классов, то преимущество получают иерархии. Базовый класс определяет

общий интерфейс. Конверсии между специализациями шаблона должны быть явно определены программистом (§27.2.2).

- Если нежелательно явное использование свободной памяти (§11.2), то преимущество у шаблонов.
- Если приоритет за эффективностью исполняемого кода, так что становится важным использование встраивания операций, то следует использовать шаблоны (поскольку использование иерархий предполагает опору на указатели или ссылки, которые не переваривают встраивание).

Комбинация шаблонов и классовых иерархий предоставляет свободу в дизайне и эффективность, недостижимую по отдельности. Например, указатель на базовый класс может использоваться в качестве аргумента шаблона, чтобы обеспечить полиморфизм времени выполнения (§3.2.4), а параметр шаблона может использоваться для того, чтобы специфицировать интерфейс базового класса для обеспечения безопасности по типам (§26.3.7, §27.3.1).

Невозможно иметь виртуальную шаблонную функцию (§23.4.6.2).

27.2.1. Генерируемые типы

Полезно понимать шаблон класса как спецификацию того, как будут сгенерированы конкретные типы. Другими словами, реализация шаблона является механизмом, который генерирует типы, основанные на спецификации. Следовательно, шаблоны классов часто называются *генераторами типов*.

С точки зрения правил языка C++ нет связи между двумя классами, сгенерированными из одного и того же шаблона. Например:

```
class Shape {
    // ...
};

class Circle : public Shape {
{
    // ...
};
```

Исходя из этих объявлений, многие думают, что должно иметь место отношение наследования между **set<Circle>** и **set<Shape>**, или по крайней мере между **set<Circle*>** и **set<Shape*>**. Это серьезная логическая ошибка, состоящая в том, что множество (контейнер set) объектов типа **Circle** гарантирует, что его члены действительно имеют тип **Circle**; а множество (контейнер set) объектов типа **Shape** не предоставляет подобной гарантии. Например:

```
class Triangle : public Shape {
    // ...
};

void f(set<Shape*>& s)
{
    // ...
    s.insert(new Triangle{p1, p2, p3});
}

void g(set<Circle*>& s)
{
    f(s);       // ошибка, несовпадение типа: s есть set<Circle*>, а не set<Shape*>
}
```

Это не будет скомпилировано, поскольку отсутствует встроенная конверсия из **set<Circle*>&** в **set<Shape*>&**. А ее и не должно быть. Гарантия того, что члены множества **set<Circle>** имеют тип **Circle**, позволяет надежно и эффективно использовать специфичные для **Circle** операции, такие как определение радиуса, к членам множества. Если бы мы позволили трактовать **set<Circle>** как **set<Shape>**, то мы не смогли бы обеспечить означенную выше гарантию. Например, **f()** помещает тип **Triangle*** в качестве аргумента у **set<Shape*>**. Если бы **set<Shape*>** мог бы быть **set<Circle*>**, фундаментальная гарантия, что **set<Circle*>** содержит только типы **Circle***, была бы нарушена.

С логической точки зрения мы могли бы трактовать неизменный **set<Circle*>** как неизменный **set<Shape*>**, потому что проблема вставки неподходящих элементов здесь отсутствует, поскольку в нашем случае мы не можем изменять множество. Поэтому мы могли бы предоставить преобразование (конверсию) от **const set<Circle*>** в **const set<Shape*>**. Язык не обеспечивает такую конверсию по умолчанию, но разработчик множества может предоставить свою.

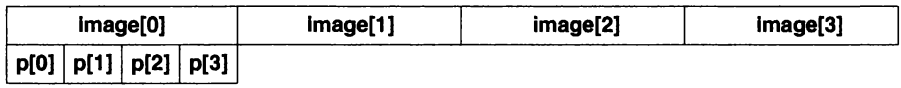
Комбинация массива и базового класса особо неприятна, потому что встроенный массив не обеспечивает безопасности по типам, присущей контейнерам. К примеру:

```
void maul(Shape* p, int n)    // Опасно!
{
    for (int i=0; i!=n; ++i)
        p[i].draw();      // выглядит невинно (но это не так)
}

void user()
{
    Circle image[10];     // image составлен из 10 Circle
    // ...
    maul(image,10);       // "maul" 10 Circle
    // ...
}
```

Как мы можем вызывать **maul()** с **image**? Во-первых, **image** конвертируется из **Circle[]** в **Circle***. Затем **Circle*** конвертируется в **Shape***. Неявное приведение имени массива к указателю на первый элемент массива является фундаментальным для программирования в С-стиле. Аналогично, неявное приведение указателя на произвродный класс к указателю на базовый класс является фундаментальным для объектно-ориентированного программирования. В комбинации они могут быть убийственными.

В примере выше предположим, что **Shape** является абстрактным классом с размером **4** и что **Circle** добавляет такие атрибуты, как центр и радиус. Тогда **sizeof(Circle)>sizeof(Shape)**, и когда мы взглянем на раскладку памяти для **image**, мы найдем что-то вроде следующего:



Когда **maul()** пытается активизировать виртуальную функцию на **p[1]**, то обнаруживается отсутствие указателя на виртуальную функцию там, где он ожидается, и вызов заканчивается неудачей.

Отметим, что мы не нуждаемся в явной конверсии, чтобы столкнуться с этой бедой:

- Так что нужно отдавать приоритет контейнерам перед встроенными массивами.
- Нужно рассматривать интерфейсы, такие как **void f(T* p, int count)**, как крайне подозрительные (тут нас могут поджидать проблемы).

- Нужно с подозрением относиться к операции точка (.), когда это касается чего-то, что предполагается быть полиморфным во время выполнения, если только это не применяется явным образом к ссылкам.

27.2.2. Конверсия шаблонов

Не существует никаких умолчательных связей между классами, генерируемыми из одного и того же шаблона (§27.2.1). Однако, для некоторых шаблонов мы хотели бы отразить такую связь. Например, когда мы определяет шаблон указателя, мы хотим отразить связь наследования между указуемыми объектами. Члены-шаблоны (§23.4.6) позволяют специфицировать связи, когда это нужно. Рассмотрим следующее:

```
template<typename T>
class Ptr {                              // указатель на T
    T* p;
public:
    Ptr(T*);
    Ptr(const Ptr&);                     // копирующий конструктор
    template<typename T2>
        explicit operator Ptr<T2>();     // конвертирует Ptr<T> в Ptr<T2>
    // ...
};
```

Мы хотели бы определить операции приведения, чтобы обеспечить связь наследования, к которой мы привыкли в связи со встроенными указателями, для определяемого пользователем типа **Ptr**. Например:

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps {pc};      // должно работать
    Ptr<Circle> pc2 {ps};    // должно дать ошибку
}
```

Мы хотим разрешить первую инициализацию, только если **Shape** является прямым или косвенным базовым классом для **Circle**. В общем случае, нам нужно определить операцию приведения от **Ptr<T>** к **Ptr<T2>** так, чтобы она принималась тогда (и только тогда), когда **T*** можно присвоить **T2***. Это можно сделать следующим образом:

```
template<typename T>
    template<typename T2>
        Ptr<T>::operator Ptr<T2>()
        {
            return Ptr<T2>{p};
        }
```

Оператор **return** пройдет компиляцию только в случае, если **p** (который имеет тип **T***) может быть аргументом конструктора **Ptr<T2>(T2*)**. Поэтому, если **T*** может неявно конвертироваться в **T2***, то приведение от **Ptr<T>** к **Ptr<T2>** будет работать. Например, можно написать:

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps {pc};      // OK: может конвертировать Circle* в Shape*
    Ptr<Circle> pc2 {ps};    // ошибка: не может конвертировать Shape* в Circle*
}
```

Следует определять только логически осмысленные приведения типов. Если есть сомнения, то лучше применить именованную функцию приведения, а не операцию приведения. Именованные функции приведения реже приводят к неоднозначностям.

Список шаблонных параметров шаблона не может комбинироваться с членом шаблона. Например:

```
template<typename T, typename T2>    // ошибочно
Ptr<T>::operator Ptr<T2>()
{
    return Ptr<T2>(p);
}
```

Альтернативно можно использовать **enable_if()** (§28.4).

27.3. Иерархии классовых шаблонов

В случае объектно-ориентированного дизайна базовый класс часто используется в качестве общего интерфейса к набору производных классов. Шаблон можно использовать для параметризации такого интерфейса, а в этом случае возникает искушение параметризовать всю иерархию производных классов теми же самыми параметрами шаблона. К примеру, мы могли бы параметризовать классический пример с формами фигур (§3.2.4) типами, призванными предоставить абстракцию для целевого «устройства» выхода:

```
template<typename Color_scheme, typename Canvas>      // спорный пример
class Shape {
    // ...
};

template<typename Color_scheme, typename Canvas>
class Circle : public Shape {
    // ...
};

template<typename Color_scheme, typename Canvas>
class Triangle : public Shape {
    // ...
};

void user()
{
    auto p = new Triangle<RGB, Bitmapped>{{0,0},{0,60},{30, sqrt(60*60-30*30)}};
    // ...
}
```

Что-то в этом роде приходит в голову в качестве первой идеи, которую программист использует в объектно-ориентированном проекте. Однако, следует соблюдать осторожность при смешении объектно-ориентированного подхода и шаблонов (обобщенное программирование).

Наша параметризованная иерархия форм избыточна для реальных случаев. С этим можно справиться при помощи умолчательных аргументов шаблона (§25.2.5). Но это не главная проблема. В том случае, когда только одна комбинация **Color_scheme** и **Canvas** используется в программе, объем генерируемого кода примерно такой же, как и в непараметризованном случае (компилятор предотвратит генерирование кода для неиспользуемых невиртуальных функций-членов классового шаблона). Но если используется **N** комбинаций **Color_scheme** и **Canvas**, тогда код для всех виртуальных функций будет ре-

плицироваться N раз. В результате будет получен массивный и запутанный код. Все это проистекает из-за того, что компилятор не знает, будет ли использоваться та или иная виртуальная функция, так что он генерирует код для всех таких функций, а также для функций, которые из них вызываются. Поэтому параметризация больших классовых иерархий с большим количеством виртуальных функций является плохой идеей.

В нашем примере с формами фигур параметры у **Color_scheme** и **Canvas** вряд ли сильно изменят интерфейс: большая часть функций-членов не располагает этими параметрами как часть типа. Эти параметры фактически являются деталями реализации, ускользающими из интерфейса (но с вероятными последствиями для эффективности кода). В реальности вся иерархия не нуждается в этих параметрах; лишь несколько конфигурационных функций и вероятно немного низкоуровневых функций рисования. В общем случае не стоит реализовывать сверхпараметризацию (§23.4.6.3): попытайтесь избежать параметров, которые влияют на малое число членов. Если лишь небольшое число членов-функций используют некоторый параметр, то лучше сделать эти функции шаблонами (с этим параметром). Например:

```
class Shape {
    template<typename Color_scheme, typename Canvas>
        void configure(const Color_scheme&, const Canvas&);
    // ...
};
```

Отдельный вопрос — как разделять конфигурационную информацию между разными классами и разными объектами. Ясно, что мы не можем просто хранить **Color_scheme** и **Canvas** в **Shape** без того, чтобы параметризовать сам **Shape** посредством **Color_scheme** и **Canvas**. Решение может состоять в том, чтобы добавить типу **Shape** член **Configuration***, где **Configuration** является базовым классом, обеспечивающим общий интерфейс к конфигурационной информации.

27.3.1. Шаблоны как интерфейсы

Классовый шаблон может использоваться с тем, чтобы предоставить гибкий и безопасный по типам интерфейс к общей реализации. Хорошим примером служит **vector** из §25.3:

```
template<typename T>
class Vector<T*>
    : private Vector<void*>
{
    // ...
};
```

В общем, эта техника может использоваться для предоставления безопасных по типам интерфейсов и для локализации приведений в реализациях вместо того, чтобы заставлять пользователей заниматься этим.

27.4. Параметры шаблона в качестве базовых классов

В классическом объектно-ориентированном программировании, использующим классовые иерархии, мы помещаем информацию, которая может отличаться для разных классов, в производные классы и осуществлять доступ к ним через виртуальные функции, объявленные в базовом классе (§3.2.4, §21.2.1). В таком случае мы можем писать общий код, не беспокоясь о различиях в реализациях. Однако, это не позволит нам модифици-

ровать типы, использованные в интерфейсе (§27.2). Кроме того, вызовы виртуальных функций достаточно дороги по сравнению с простыми операциями изнутри встраиваемых функций. Для компенсации мы можем передать специализированную информацию и операции базовому классу как шаблонные параметры. Фактически, аргумент шаблона может использоваться в качестве базового класса.

Последующие две подсекции книги адресованы изучению вопроса: «как можно скомбинировать порознь заданную информацию в один компактный объект с хорошо специфицированным интерфейсом?». Это фундаментальный вопрос и ответы на него чрезвычайно важны.

27.4.1. Композиция структур данных

Рассмотрим задачу написания библиотеки сбалансированных бинарных деревьев. Поскольку библиотека будет использоваться разными пользователями, то мы не можем вписать типы из пользовательских приложений в узлы нашего дерева. Тут есть несколько альтернатив:

- Мы могли бы поместить пользовательские данные в производный класс с доступом к ним посредством виртуальных функций. Но вызовы виртуальных функций относительно дороги, и поскольку интерфейс к данным не выражается посредством пользовательских типов, то по-прежнему нужны явные приведения типов для доступа к данным.
- Мы можем поместить **void*** в узлы дерева и позволить пользователям использовать их для доступа к данным, аллоцированным вне узлов дерева. Но это приведет к расходу памяти на указатели в каждом узле и приведет к потенциально дорогим операциям разыменования указателей.
- Мы могли бы поместить **Data*** в узлы дерева, где **Data** представляет собой «универсальный базовый класс» для наших структур данных. Это решит проблему проверки типов, но все равно тут присутствует комбинация дороговизны (в смысле эффективности) и неудобства двух предыдущих подходов.

Есть и другие альтернативы, так что рассмотрим следующее:

```
template<typename N>
struct Node_base {                          // не знает о Val (пользовательские данные)
    N* left_child;
    N* right_child;

    Node_base();

    void add_left(N* p)
    {
        if (left_child==nullptr)
            left_child = p;
        else
            // ...
    }
    // ...
};

template<typename Val>
struct Node :  Node_base<Node<Val>> { // используйте производный класс как часть его базы
    Val v;
    Node(Val vv);
    // ...
};
```

Мы передаем производный класс **Node<Val>** в качестве шаблонного аргумента его базовому классу (**Node_base**). Это позволяет **Node_base** использовать **Node<Val>** в своих интерфейсах даже без знания его истинного имени.

Заметьте, что раскладка **Node** весьма компактна. Например, для **Node<double>** она будет выглядеть примерно следующим образом:

```
struct Node_base_double {
    double val;
    Node_base_double* left_child;
    Node_base_double* right_child;
};
```

К сожалению, в таком дизайне пользователь должен знать операции у **Node_base** и структуру результирующего дерева. К примеру:

```
using My_node = Node<double>;

void user(const vector<double>& v)
{
    My_node root;
    int i = 0;

    for (auto x : v) {
        auto p = new My_node{x};
    if (i++%2)                          // выбираем, где вставить
        root.add_left(p);
    else
        root.add_right(p);
    }
}
```

Но, тем не менее, пользователю нелегко сохранять разумную структуру дерева. В типичном случае, мы бы хотели позволить самому дереву заботиться о реализации алгоритма балансировки. Однако, чтобы балансировать дерево для эффективного к нему доступа, балансировщик должен знать величины, которые пользователь будет хранить в дереве.

Как добавить балансировщик в наш проект? В принципе можно жестко закодировать стратегию балансировки в **Node_base** и позволить **Node_base** «подсматривать» пользовательскте данные. К примеру, реализации сбалансированных деревьев (вроде стандартного **map**), по умолчанию требуют, чтобы тип значений предоставлял операцию «меньше чем». Поэтому операции у **Node_base** могут просто использовать операцию <:

```
template<typename N>
struct Node_base {
    static_assert(Totally_ordered<N>(), "Node_base: N must have a <");

    N* left_child;
    N* right_child;
    Balancing_info bal;

    Node_base();

    void insert(N& n)
    {
        if (n<left_child)
            // ... что-нибудь делаем ...
        else
            // ... делаем что-нибудь иное ...
    }
    // ...
};
```

Это работает прекрасно. Фактически, чем больше информации об узлах мы встраиваем в **Node_base**, тем проще становится реализация. В частности, мы могли бы параметризовать **Node_base** типом значений, а не типом узлов (как это и делается в **std::map**). Но при этом мы не отвечаем на фундаментальный вопрос: как можно комбинировать информацию из порознь специфицированных источников.

Давайте предположим, что пользователь захочет манипулировать узлами типа **Node** (например, отсоединить узел от одного дерева и присоединить его к другому), так что мы не можем просто хранить пользовательские данные в анонимном узле. Давайте также предположим, что нам может потребоваться обращаться к разным алгоритмам балансировки и поэтому придется сделать балансировщика аргументом. Все это заставляет нас лицом к лицу столкнуться с фундаментальным вопросом. Простейшим ответом будет разрешить **Node** комбинировать тип значений с типом балансировщика. Самому **Node** нет нужды работать напрямую с балансировщиком, и поэтому он просто передает его **Node_base**:

```
template<typename Val, typename Balance>
struct Search_node : public Node_base<Search_node<Val, Balance>, Balance>
{
    Val val;                // данные пользователя
    search_node(Val v): val(v) {}
};
```

Мы дважды упомянули **Balance**, поскольку он является частью типа узла и потому что **Node_base** нужно создать объект типа **Balance**:

```
template<typename N, typename Balance>
struct Node_base :  Balance {
    N* left_child;
    N* right_child;

    Node_base();

    void insert(N& n)
    {
        if (this->compare(n, left_child))      // используем compare() из Balance
            // ... делаем что-нибудь ...
        else
            // ... делаем иное ...
    }
    // ...
};
```

Можно было бы применить **Balance** для объявления члена, а не делать его базовым классом. Однако, некоторые важные балансировщики не требуют данных в каждом узле, так что делая **Balance** базовым классом, я получаю выгоду с практической точки зрения. Язык гарантирует, что если базовый класс не содержит нестатических членов-данных, то память будет аллоцироваться (выделяться) для него в объекте производного класса (§ iso.1.8). Этот дизайн также стилистически близок к реальной библиотеке бинарных деревьев [Austern,2003]. Мы можем использовать эти классы следующим образом:

```
struct Red_black_balance {
    // данные и операции, нужные для реализации red-black деревьев
};

template<typename T>
using Rbnode = Search_node<T, Red_black_balance>; // синонимы типов для red-black деревьев
```

```
Rbnode<double> my_root;     // red-black дерево из doubles

using My_node = Rb_node<double>;

void user(const vector<double>& v)
{
     for (auto x : v)
          root.insert(*new My_node{x});
}
```

Раскладка узла компактна и мы можем легко осуществить встраивание всех критически важных для производительности функций. За счет слегка большей работы мы достигли безопасности по типам и получили простую возможность композиции. Также при этом достигается повышенная эффективность программы по сравнению с любым вариантом, использующим **void*** в структурах данных или в интерфейсе функций (использование **void*** отрицает полезные технологии оптимизации по типам).

Мы передаем балансировщик в виде отдельного аргумента шаблона:

```
template<typename N, typename Balance>
struct Node_base : Balance {
     // ...
};

template<typename Val, typename Balance>
struct Search_node
     : public Node_base<Search_node<Val, Balance>, Balance>
{
     // ...
};
```

Некоторые люди считают такое решение ясным, явным и общим; другие считают его туманным и избыточным. Альтернативно можно сделать балансировщик неявным аргументом в форме ассоциированного типа (типом члена у **Search_node**):

```
template<typename N>
struct Node_base : N::balance_type {// используем N::balance_type
     // ...
};

template<typename Val, typename Balance>
struct Search_node
     : public Node_base<Search_node<Val, Balance>>
{
     using balance_type = Balance;
     // ...
};
```

Эта техника интенсивно используется в стандартной библиотеке для минимизации явных аргументов шаблона.

Техника наследования от базового класса весьма древняя. Она упоминалась в ARM (1989) и в математическом программном обеспечении [Barton,1994].

27.4.2. Линеаризация классовых иерархий

Пример из §27.4.1 использует шаблон для сжатия представления и ради того, чтобы избежать использования **void***. Рассмотренная техника полезна и общепризнана. В частности, многие программы, использующие деревья, опираются на нее для достижения без-

опасности по типам и ради высокой производительности. Подход IPR (Internal Program Representation — Внутреннее Представление Программ) [DosReis,2011] является общим и систематическим представлением C++ кода в виде типизированных абстрактных синтаксических деревьев. Он интенсивно использует шаблонные параметры в качестве базовых классов как для помощи в реализации, так и для представления абстрактных интерфейсов в классическом объектно-ориентированном стиле. Этот подход пытается ответить на многие проблемы проектирования, включая достижение компактности узлов (ведь возможен миллион узлов), оптимизации управления памятью, времени доступа, безопасности по типам, полиморфных интерфейсов и общности.

Пользователи видят иерархию абстрактных классов, обеспечивающих отличную инкапсуляцию, и ясные интерфейсы функций, представляющие семантику программы. Например, переменная есть объявление, является оператором, участвует в выражении, которое есть узел:

```
Var -> Decl -> Stmt -> Expr -> Node
```

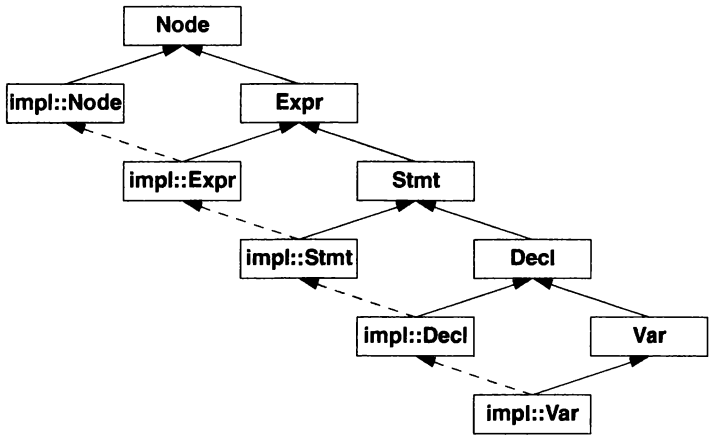
Понятно, что в дизайне IPR сделаны некоторые обобщения, ибо в языке ISO C++ операторы не могут использоваться как выражения.

Кроме того, имеется параллельная иерархия конкретных классов, предоставляющая компактные и эффективные реализации классов в иерархии интерфейсов:

```
impl::Var -> impl::Decl -> impl::Stmt -> impl::Expr -> impl::Node
```

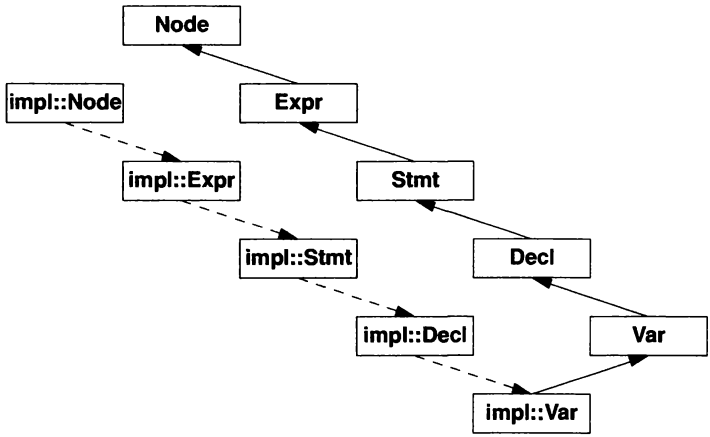
Всего имеется 80 классов листьев дерева (таких как **Var**, **If_stmt** и **Multiply**) и около 20 обобщений (таких как **Decl**, **Unary** и **impl:: Stmt**).

Первая попытка дизайна в этом направлении привела к классической объектно-ориентированной ромбовидной иерархии (сплошные стрелки отражают наследование интерфейса, а пунктирные — наследование реализации):



Это работало, но приводило к избыточному расходу памяти: узлы были слишком большими из-за данных, необходимых для навигации по виртуальным базовым классам. Дополнительно, программы заметно тормозили из-за многочисленных косвенных обращений для доступа ко многим виртуальным функциям в каждом объекте (§21.3.5).

Решением стала линеаризация двойной иерархии, так что никаких виртуальных базовых классов не стало:



Для полного набора классов цепочка наследования выглядит так:

```
impl::Var ->
    impl::Decl<impl::Var> ->
        impl::Stmt<impl::Var> ->
            impl::Expr<impl::Var> ->
                impl::Node<impl::Var> ->
                    ipr::Var ->
                        ipr::Decl ->
                            ipr::Stmt ->
                                ipr::Expr ->
                                    ipr::Node
```

Это представляется компактным объектом без внутренних данных управления, за исключением единственного **vptr** (§3.2.3, §20.3.2).

Я сейчас покажу, как это делается. Интерфейсная иерархия, определенная в пространстве имен **ipr**, описывается первой. Стартуя снизу, узлы типа **Node** содержат данные, необходимые для оптимизации траверса, идентификации типа узла (**code_category**) и для помощи в хранении IPR-графов в файлах (**node_id**). Имеются детали реализации, спрятанные от пользователей. Что пользователь знает, так это то, что каждый узел в IPR-графе имеет уникальный базовый класс типа **Node** и что это можно использовать для реализации операций, нужных для «паттерна посетитель» [Gamma,1994] (§22.3):

```
struct ipr::Node {
    const int node_id;
    const Category_code category;

    virtual void accept(Visitor&) const = 0;        // hook для классов посетителей
protected:
    Node(Category_code);
};
```

Предполагается, что **Node** используется только как базовый класс, так что его конструктор маркирован как **protected**. Он также располагает чисто виртуальными функциями, так что его можно использовать исключительно в качестве базового класса.

Для выражения (**Expr**) используется **Node** с типом:

```
struct ipr::Expr : Node {
    virtual const Type& type() const = 0;
protected:
    Expr(Category_code c) : Node(c) {}
};
```

Очевидно, что это является обобщением языка C++, потому что предполагает, что даже операторы и типы обладают типами: цель IPR состояла в том, чтобы представить C++ без реализации всех его иррегулярностей и ограничений.

Оператор (**Stmp**) есть **Expr**, располагающий определенным файлом-источником, и он может аннотироваться различной информацией:

```
struct ipr::Stmt : Expr {
    virtual const Unit_location& unit_location() const = 0;          // строка в файле
    virtual const Source_location& source_location() const = 0;      // файл

    virtual const Sequence<Annotation>& annotation() const = 0;
protected:
    Stmt(Category_code c) :  Expr(c) {}
};
```

Объявление (**Decl**) есть **Stmt**, который вводит имя;

```
struct ipr::Decl : Stmt {
    enum Specifier {/* класс хранения, virtual, контроль доступа, и.д. */};

    virtual Specifier specifiers() const = 0;
    virtual const Linkage& lang_linkage() const = 0;

    virtual const Name& name() const = 0;

    virtual const Region& home_region() const = 0;
    virtual const Region& lexical_region() const = 0;

    virtual bool has_initializer() const = 0;
    virtual const Expr& initializer() const = 0;

    // ...
protected:
    Decl(Category_code c) : Stmt(c) {}
};
```

Вполне ожидаемо, что **Decl** является одним из центральных понятий, когда речь идет о представлении C++ кода. Здесь вы находите информацию об области видимости, классе хранения, спецификаторах доступа, инициализаторах и т.д.

Наконец, можно определить класс для представления переменной (**Var**) как лист (большинство производных классов) иерархии интерфейсов:

```
struct ipr::Var : Category<var_cat, Decl> {
};
```

По сути, **Category** помогает в нотации с эффектом наследования **Var** от **Decl** и с **Category_code** для оптимизации типовой идентификации:

```
template<Category_code Cat, typename T = Expr>
struct Category : T {
protected:
    Category() : T(Cat) {}
};
```

Каждый член данных есть **Var**. Это включает глобальные переменные, переменные из пространств имен, локальные переменные и классические статические переменные и константы.

По сравнению с тем, что можно найти в реальных компиляторах, этот интерфейс, конечно же, крошечный. За исключением некоторых оптимизационных данных в **Node**, это

просто набор классов с чисто виртуальными функциями. Это линейная иерархия без каких-либо виртуальных базовых классов. Это прямолинейный объектно-ориентированный дизайн. Правда, не так-то просто это реализовать эффективно и удобным для дальнейшей поддержки образом, и реализация IPR не является образцом для опытного объектно-ориентированного программиста.

Для каждого IPR интерфейсного класса (в **ipr**) имеется соответствующий класс реализации (в **impl**). К примеру:

```
template<typename T>
struct impl::Node : T {
    using Interface = T;      // делаем тип шаблонного аргумента доступным пользователям
    void accept(ipr::Visitor& v) const override {v.visit(*this);}
};
```

Здесь «трюк» состоит в том, чтобы установить соответствие между **ipr**-узлами и **impl**-узлами. В частности, **impl**-узлы должны предоставлять необходимые члены-данных и замещать абстрактные виртуальные функции в **ipr**-узлах. Для **impl::Node** видно, что если **T** имеет тип **ipr::Node** или тип класса, производного от **ipr::Node**, то функция **accept()** замещается надлежащим образом.

Теперь мы можем приступить к классам реализации для оставшейся части интерфейсных классов **ipr**:

```
template<typename Interface>
struct impl::Expr : impl::Node<Interface> {
    const ipr::Type* constraint;   // ограничение является типом выражения

    Expr() : constraint(0) {}

    const ipr::Type& type() const override {return *util::check(constraint);}
};
```

Если аргумент у **Interface** есть **ipr::Expr** (или производный от него класс), тогда **impl::Expr** есть реализация для **ipr::Node**. Поскольку **ipr::Expr** наследуется от **ipr::Node**, то в результате **impl::Node** получает необходимый ему базовый класс **ipr::Node**.

Другими словами, нам удалось предоставить реализации для двух (разных) интерфейсных классов. Можно продолжить в той же манере:

```
template<typename S>
struct impl::Stmt : S {
    ipr::Unit_location unit_locus;      // логическая позиция в единице трансляции
    ipr::Source_location src_locus;     // исходный файл, строка в исходнике, и позиция
    ref_sequence<ipr::Annotation> notes;

    const ipr::Unit_location& unit_location() const override {return unit_locus;}
    const ipr::Source_location& source_location() const override {return src_locus;}
    const ipr::Sequence<ipr::Annotation>& annotation() const override {return notes;}
};
```

Таким образом, **impl::Stmt** предоставляет три члена данных, нужных для реализации интерфейса **ipr::Stmt**, и замещает три виртуальные функции **ipr::Stmt**, необходимые для этого.

В самом общем плане все **impl**-классы следуют паттерну **Stmt**:

```
template<typename D>
struct impl::Decl : Stmt<Node<D> > {
    basic_decl_data<D> decl_data;
    ipr::Named_map* pat;
    val_sequence<ipr::Substitution> args;
    Decl() : decl_data(0), pat(0) {}
```

```
    const ipr::Sequence<ipr::Substitution>& substitutions() const {return args;}
    const ipr::Named_map& generating_map() const override {return *util::check(pat);}
    const ipr::Linkage& lang_linkage() const override;
    const ipr::Region& home_region() const override;
};
```

Наконец, мы можем определить лист дерева — класс **impl::Var**:

```
struct Var : impl::Decl<ipr::Var> {
    const ipr::Expr* init;
    const ipr::Region* lexreg;

    Var();

    bool has_initializer() const override;
    const ipr::Expr& initializer() const override;
    const ipr::Region& lexical_region() const override;
};
```

Заметьте, что **Var** не является шаблоном; это пользовательская абстракция нашего приложения. Реализация **Var** служит примером обобщенного программирования, но его использование выполняется в классическом стиле объектно-ориентированного программирования.

Комбинация наследования и параметризации является мощнейшим средством. Однако, столь мощное средство может обескуражить новичков и даже опытных программистов из другой области программирования. Но преимущества здесь налицо: безопасность по типам, производительность и минимальный размер исходного кода.

27.5. Советы

[1] Когда нужно выразить общую идею в коде, рассмотрите альтернативу — представить ее в виде шаблона или же классовой иерархией; §27.1.

[2] Шаблон обычно предоставляет общий код для различных аргументов; §27.1.

[3] Абстрактный класс может полностью скрыть детали реализации от пользователей; §27.1.

[4] Нерегулярные реализации лучше представлять производными классами; §27.2.

[5] Если явное использование свободной памяти нежелательно, то шаблоны имеют преимущество перед классовыми иерархиями; §27.2.

[6] Шаблоны имеют преимущество перед абстрактными классами тогда, когда важно встраивание; §27.2.

[7] Шаблонные интерфейсы легко выражаются в терминах типов аргументов шаблона; §27.2.

[8] Если нужно разрешение на этапе выполнения, тогда необходимы классовые иерархии; §27.2.

[9] Комбинация шаблонов и классовых иерархий бывает просто замечательной; §27.2.

[10] Представляйте себе шаблоны как генераторы типов (и генераторы функций); §27.2.1.

[11] Нет никакой умолчательной связи между двумя классами, сгенерированными из одного и того же шаблона; §27.2.1.

[12] Не смешивайте классовые иерархии и массивы; §27.2.1.

[13] Не следует наивно параметризовать большие классовые иерархии; §27.3.

[14] Шаблон может использоваться, чтобы предоставлять интерфейс, безопасный по типам, к одной (слабо типизированной) реализации; §27.3.1.

[15] Шаблоны можно использовать для композиции структур данных, компактных и безопасных по типам; §27.4.1.

[16] Шаблоны можно использовать для линеаризации классовой иерархии (минимизируя при этом объем памяти и время доступа); §27.4.2.

28

Метапрограммирование

Путешествие в незнакомую страну нужно совершить дважды;
первое ради ошибок, второе — для их исправления.
— Джон Стейнбек

- Введение
- Функции типа
  Синонимы типов; Предикаты типа; Выбор функции
- Управляющие структуры
  Выбор; Итерация и рекурсия; Когда использовать метапрограммирование
- Условное определение: **Enable_if**
  Реализация **Enable_if**; **Enable_if** и концепции; Дополнительные примеры с **Enable_if**
- Список времени компиляции: **Tuple**
  Простая функция вывода; Доступ к элементам; Функция **make_tuple**
- Шаблоны с переменным числом параметров
  Безопасная по типам функция **printf()**; Технические детали; Перенаправление; Шаблон **tuple** из стандартной библиотеки
- Библиотека для работы с единицами измерений
  Тип **Unit**; Тип **Quantity**; Литералы для **Unit**; Вспомогательные функции
- Советы

28.1. Введение

Программирование, имеющее дело с манипулированием программными сущностями, такими как классы и функции, часто называют *метапрограммированием*. Я считаю полезным представлять себе шаблоны как генераторы: они используются для генерации классов и функций. Это ведет к понятию метапрограммирования, как к упражнению в написании программ, выполняющих вычисления во время компиляции и генерирующих программы. Иногда применяют уточняющий термин: *шаблонное метапрограммирование*.

Существуют две причины для применения технологии метапрограммирования:

- *Улучшенная безопасность типов*: Можно в результате вычислений на этапе компиляции получить точные типы, нужные для алгоритмов и структур данных, так что нам самим не приходится выполнять низкоуровневую манипуляцию структурами данных (например, можно устранить большинство случаев применения явных приведений типов).
- *Улучшенная производительность на этапе выполнения*: На этапе компиляции можно вычислить значения и выбрать функции, которые затем будут вызываться во вре-

мя выполнения программы. Таким образом мы уменьшаем количество вычислений на этапе выполнения (можно вместо полиморфных вызовов функций выполнять прямые вызовы). В особенности, опираясь на достоинства системы типов, можно резко улучшить производительность за счет встраивания. Кроме того, за счет использования компактных структур данных (возможно сгенерированных; §27.4.2, §28.5) улучшается использование памяти с положительным влиянием на ее размеры и на время доступа к ней.

Шаблоны конструировались в очень общем ключе и со способностью генерировать оптимальный код [Stroustrup,1994]. Они допускают арифметику, выбор и рекурсию. Фактически, они формируют полноценный язык функционального программирования времени компиляции [Veldhuizen, 2003]. В качестве демонстрации Eisenecker и Czarnecki написали интерпретатор языка Lisp всего лишь на двух страницах, используя шаблоны [Czarnecki,2000]. Механизмы языка C++, работающие на этапе компиляции, составляют чисто функциональный язык. Можно создавать значения любых типов (но они не есть переменные), можно выполнять присваивания, операции инкремента и т.д. Бесконечные рекурсии устраняются слежением за определенными ресурсами компиляции, вроде количества рекурсивных вызовов **constexpr**, числа вложенных классов или числа рекурсивно вложенных конкретизаций шаблонов.

Где провести черту между обобщенным программированием и шаблонным метапрограммированием? Крайние позиции таковы:

- Все есть метапрограммирование: в конце концов, любое использование параметризации на этапе компиляции предполагает конкретизацию, которая генерирует «обычный код».
- Все есть обобщенное программирование: в конце концов, мы просто определяем и используем генерируемые типы и алгоритмы.

Обе этих позиции бессмысленны, потому что они определяют обобщенное программирование и шаблонное метапрограммирование как синонимы. Но я считаю, что есть некоторое отличие. Ведь к каждой проблеме можно подходить с альтернативных позиций и фокусироваться на том, что важно именно для данной проблемы. Когда я программирую обобщенный тип или алгоритм, я не думаю, что занимаюсь написанием программы для исполнения на этапе компиляции, и не использую свои навыки в этом направлении. Вместо этого я фокусируюсь на написании требований к аргументам (§24.3). Обобщенное программирование по своей сути является философией дизайна программы — программной парадигмой (§1.2.1).

Метапрограммирование есть просто программирование. Фокус направлен на вычисления, часто с возможностью выбора и некоторых форм итерации. Метапрограммирование в своей основе является набором некоторых приемов на означенную тему. Я выделяю четыре уровня сложностей в реализации:

1. Вычисления отсутствуют (просто передача типов и значений в виде аргументов)
2. Простые вычисления (на типах или значениях), не использующие проверки во время компиляции или итерации, например выполнение операции **&&** над логическими значениями (§24.4) или простые сложения единиц (§28.7.1)
3. Вычисления с явными проверками на этапе компиляции, например **if** времени компиляции (§28.3)
4. Вычисления с итерациями на этапе компиляции (в форме рекурсии; §28.3.2)

Порядок пунктов отражает нарастание сложности (в разработке, отлаживании и вероятности ошибок).

Итак, метапрограммирование есть комбинация «мета» и программирования: *метапрограмма* сводится к вычислениям на этапе компиляции с тем, чтобы предоставить типы и функции, которые будут далее использоваться во время выполнения программы. Заметьте, что я не сказал «шаблонное метапрограммирование», ибо вычисления могут выполняться при помощи **constexpr**-функций. Заметьте также, что можно самому не выполнять метапрограммирование, а положиться на средства, разработанные другими людьми: на вызовы **constexpr**-функций, скрывающих метапрограммирование (§28.2.2), или на извлечение типа из шаблонных функций типа (§28.2.4).

Обобщенное программирование обычно попадает в первую категорию, то есть в ту, что не использует вычисления, но возможна поддержка обобщенного программирования со стороны метапрограммирования. В таком случае нужно внимательно следить за тем, чтобы наши интерфейсы были точно определены и корректно реализованы. При использовании метапрограммирования как части интерфейса возрастает возможность ошибок.

Обобщенное программирование фокусируется на спецификациях интерфейса, а метапрограммирование есть программирование, обычно с типами вместо величин.

Избыточное применение метапрограммирования усложняет отладку и увеличивает время компиляции. Как всегда, следует руководствоваться здравым смыслом. Есть несложные приемы метапрограммирования, приводящие к более качественному коду без особо сильного увеличения времени компиляции. Многие компоненты стандартной библиотеки, такие как **function** (§33.5.3), **thread** (§5.3.1, §42.2.2) и **tuple** (§34.2.4.2), служат примерами относительно простых способов применения метапрограммирования.

В настоящем разделе книги мы исследуем лишь основные приемы метапрограммирования и представляем основные строительные блоки метапрограмм. Глава 29 содержит расширенные примеры.

28.2. Функции типа

Функция типа есть функция, принимающая по крайней мере один аргумент-тип, или производит по крайней мере один тип в качестве результата. Например, **sizeof(T)** является встроенной функцией типа, получающей на вход аргумент-тип **T**, и возвращающая размер соответствующих объектов (измеряемый в размере **char**; §6.2.8).

Функции типа не обязаны выглядеть так же, как обычные функции. Фактически, большинство и не выглядят. Например, в стандартной библиотеке **is_polymorphic<T>** принимает аргумент как аргумент шаблона и возвращает результат как член с именем **value**:

```
if (is_polymorphic<int>::value) cout << "Big surprise!";
```

Член **value** типа **is_polymorphic** есть или **true**, или **false**. Аналогично, стандартная библиотека следует соглашению, что функция типа возвращает тип в виде члена **type**. Например:

```
enum class Axis : char { x, y, z };
enum flags { off, x=1, y=x<<1, z=x<<2, t=x<<3 };

typename std::underlying_type<Axis>::type x;     // x имеет тип char
typename std::underlying_type<Axis>::type y;     // y – вероятно int (§ 8.4.2)
```

Функция типа может принимать более одного аргумента и возвращать несколько результатов. Например:

```
template<typename T, int N>
struct Array_type {
    using type = T;
    static const int dim = N;
    // ...
};
```

Array_type не является функцией стандартной библиотеки (и вообще не является слишком полезной). Я просто использовал ее для иллюстрации того, как писать функцию типа с несколькими аргументами и несколькими возвратами. Ее можно использовать следующим образом:

```
using Array = Array_type<int,3>;

Array::type x;                  // x это int
constexpr int s = Array::dim;   // s есть 3
```

Функции типа являются функциями времени компиляции. То есть, они могут принимать лишь аргументы (типы и значения), известные на этапе компиляции, и производить результаты (типы и значения), которые могут использоваться во время компиляции.

Большинство функций типа принимают по крайней мере один аргумент, но есть и полезные примеры без аргументов. Например, вот функция типа, которая возвращает целый тип с подходящим числом байтов:

```
template<int N>
struct Integer {
    using Error = void;
    using type = Select<N, Error, signed char, short, Error, int, Error, Error, Error, long>;
};

typename Integer<4>::type i4 = 8;       // 4-byte целое
typename Integer<1>::type i1 = 9;       // 1-byte целое
```

Использованная здесь **Select** определяется в §28.3.1.3. Конечно, можно писать шаблоны, которые принимают только значения и производят лишь значения. Я не рассматриваю такие шаблоны как функции типа. Я считаю, что **constexpr**-функции (§12.1.6) представляют более подходящий способ вычислений над величинами на этапе компиляции. Я могу вычислить корень квадратный во время компиляции, используя шаблоны, но зачем это делать, если лучше выразить алгоритм с помощью **constexpr**-функций (§2.2.3, §10.4, §28.3.2)?

Итак, функции типа в языке C++ преимущественно являются шаблонами. Они могут выполнять очень общие вычисления, используя типы и величины. Они служат становым хребтом метапрограммирования. Например, нам может потребоваться аллоцировать объект в стеке, при условии, что он небольшой, и в свободной памяти — в противном случае:

```
constexpr int on_stack_max = sizeof(std::string); // максимальный размер объекта на стеке

template<typename T>
struct Obj_holder {
    using type = typename std::conditional<(sizeof(T)<=on_stack_max),
                                           Scoped<T>,       // первая альтернатива
                                           On_heap<T>       // вторая альтернатива
                                          >::type;
};
```

Шаблон **conditional** из стандартной библиотеки является селектором времени компиляции между двумя альтернативами. Если его первый аргумент есть **true**, то результат

(представлянемый как член **type**) является вторым аргументом; в противном случае результат есть третий аргумент. В §28.3.1.1 показывается, как **conditional** реализован. В этом случае **type** у **Obj_holder<X>** определяется как **Scoped<X>**, если объект типа **X** небольшой, и как **On_heap<X>**, если большой. Можно использовать **Obj_holder** следующим образом:

```
void f()
{
    typename Obj_holder<double>::type v1;              // double – в стек
    typename Obj_holder<array<double,200>>::type v2; // array – в свободную память
    // ...
    *v1 = 7.7;      // Scoped предоставляет доступ в стиле указателя (* и [])
    v2[77] = 9.9;   // On_heap предоставляет доступ в стиле указателя (* и [])
    // ...
}
```

Пример с **Obj_holder** вовсе не гипотетический. К примеру, в стандарте C++ имеется комментарий к типу **function** (§33.5.3) для аллокации сущностей «a'la функция»: «Реализациям рекомендуется избегать динамически выделяемой памяти для небольших вызываемых объектов» (§ iso.20.8.11.2.1). Будет трудно следовать этому совету без чего-нибудь вроде **Obj_holder**.

Как реализуются **Scoped** и **On_heap**? Их реализации тривиальны и не используют никакого метапрограммирвания:

```
template<typename T>
struct On_heap {
    On_heap(): p(new T) { }      // аллоцируем
    ~On_heap() { delete p; }     // деаллоцируем

    T& operator*() { return *p; }
    T* operator->() { return p; }

    On_heap(const On_heap&) = delete;          // предотвращение копирования
    On_heap operator=(const On_heap&) = delete;
private:
    T* p;     // указатель на объект в свободной памяти
};

template<typename T>
struct Scoped {
    T& operator*() { return x; }
    T* operator->() { return &x; }

    Scoped(const Scoped&) = delete;            // предотвращаем копирование
    Scoped operator=(const Scoped&) = delete;
private:
    T x;      // объект
};
```

Примеры с **On_heap** и **Scoped** являются хорошими примерами того, как обобщенное программирование и шаблонное метапрограммирование требуют от нас разработки однородных интерфейсов к разным реализациям общей идеи (у нас здесь — это идея аллоцирования объекта).

Как **On_heap**, так и **Scoped**, могут использоваться как члены или локальные переменные. В то время, как **On_heap** всегда помещает объект в свободную память, **Scoped** содержит свой объект.

В §28.6 рассказывается, как мы можем реализовать версии **On_heap** и **Scoped** для типов с конструкторами, принимающими аргументы.

28.2.1. Синонимы типов

Заметим, что реализационные детали у **Obj_holder** просвечивают наружу, когда мы используем **typename** и **::type** для извлечения типа члена. Это следствие того, как мы используем спецификации языка, и это задает направление, в котором шаблонное метапрограммирование развивалось в течение последних 15 лет, что также проникло в стандарт C++11. Это напоминает мне старые времена языка C, где каждое появление пользовательского типа должно снабжаться префиксом с ключевым словом **struct**. Введением синонима шаблона (§23.6) мы можем спрятать детали реализации с **::type** и позволить функциям типа выглядеть более похожими на функции, возвращающими тип. Например:

```
template<typename T>
using Holder = typename Obj_holder<T>::type;

void f2()
{
    Holder<double> v1;
        // double – в стек
    Holder<array<double,200>> v2;    // array – в свободную память
    // ...
    *v1 = 7.7;      // Scoped предоставляет доступ в стиле указателя (* и [])
    v2[77] = 9.9;   // On_heap предоставляет доступ в стиле указателя (* и [])
    // ...
}
```

Кроме редких случаев, я систематически использую синонимы типов. Когда стандарт определяет функцию типа, такую как **conditional**, я тут же определяю соответствующий синоним типа (§35.4.1):

```
template<typename C, typename T, typename F>
using Conditional = typename std::conditional<C, T, F>::type;
```

К сожалению, эти синонимы не являются частью стандарта.

28.2.1.1. Когда использовать синонимы не следует

Есть один случай, когда важно использовать **::type** напрямую вместо синонима. Если лишь одна альтернатива полагается быть правильным типом, то не следует использовать синоним. Рассмотрим простую аналогию:

```
if (p) {
    p->f(7);
    // ...
}
```

Важно, что мы не входим в блок, если **p** есть **nullptr**. Мы используем проверку для того, чтобы узнать, является ли значение указателя **p** корректным. Аналогично, можно выполнять проверку для того, чтобы выяснить корректность типа. Например:

```
conditional<
    is_integral<T>::value,
    make_unsigned<T>,
    Error<T>
>::type
```

Здесь мы проверяем, является ли **T** интегральным типом (используя предикат типа **std::is_integral**), и приходим к беззнаковому варианту типа, если он есть (используя функцию типа **std::make_unsigned**). Если это проходит, то мы имеем беззнаковый тип; в противном случае нам придется иметь дело с индикатором **Error**.

Если бы мы написали **Make_unsigned<T>**, означающим

```
typename make_unsigned<T>::type
```

и пытались бы использовать это для неинтегрального типа, скажем **std::string**, то мы пытались бы создать несуществующий тип (**make_unsigned<std::string>::type**). В результате была бы ошибка компиляции.

В редких случаях, где мы не можем использовать синонимы согласованно для сокрытия **::type**, остается вернуться к более явному стилю **::type**. В качестве альтернативы можно ввести функцию типа **Delay** для того, чтобы отложить выполнение функции типа до того момента, когда ее нужно использовать:

```
Conditional<
    is_integral<T>::value,
    Delay<Make_unsigned, T>,
    Error<T>
>
```

Реализация **Delay** нетривиальна, но в большинстве случаев подойдет следующее:

```
template<template<typename...> class F, typename ... Args>
using Delay = F<Args...>;
```

Здесь используются шаблонный аргумент шаблона (§25.2.4) и шаблоны с переменным числом параметров (§28.6).

Независимо от варианта, который мы выберем для того, чтобы избежать нежелательной конкретизации шаблона, все это относится к территории экспертов, о которой я здесь сообщил с некоторым трепетом.

28.2.2. Предикаты типа

Предикат является функцией, возвращающей логическое значение. Если вы хотите написать функцию, которая принимает типы в качестве аргументов, то представляется очевидным, что вам нравится задавать вопросы о типах, например: «Это знаковый тип?», «Этот тип полиморфный (то есть имеет хотя бы одну виртуальную функцию)» или «Этот тип производный от другого типа?».

Ответы на многие из таких вопросов известны компилятору и представляются программисту в виде стандартных предикатов типа (§35.4.1). Например:

```
template<typename T>
void copy(T* p, const T* q, int n)
{
    if (std::is_pod<T>::value)
        memcpy(p, q, n);          // оптимизированное копирование памяти
    else
        for (int i=0; i!=n; ++i)
            p[i] = q[i];          // копируем индивидуальные значения
}
```

Здесь мы пытаемся оптимизировать копирование, используя функцию **memcpy()** из стандартной библиотеки тогда, когда мы можем трактовать объекты как «простые старые данные» (POD; §8.2.6). В противном случае, мы копируем объекты один за одним, используя их копирующие конструкторы. Мы определяем, является ли аргумент шаблона типом POD при помощи стандартного предиката типа **is_pod**. Результат отражается в члене **value**. Это соглашение стандартной библиотеки аналогично тому, как функции типа отражают свой результат в члене **type**.

Предикат **std::is_pod** является одним из многих предикатов, предоставляемых стандартной библиотекой (§35.4.1). Так как правила для типа, чтобы он был POD, довольно замысловатые, то **is_pod** скорее является глубоко запрятанным в компилятор, а не реализованным на языке C++.

Как и для соглашения с **::type**, **::value** является отклонением от стандартной нотации: функция с возвратом **bool** должна вызываться посредством **()**:

```
template<typename T>
void copy(T* p, const T* q, int n)
{
    if (is_pod<T>())
    // ...
}
```

По-счастью, стандарт поддерживает это для всех предикатов типа из стандартной библиотеки. К сожалению, по техническим причинам это недоступно в контексте аргументов шаблона. Например:

```
template<typename T>
void do_something()
{
    Conditional<is_pod<T>(), On_heap<T>, Scoped<Y>) x; // ошибка: is_pod<T>() это тип
    // ..
}
```

В частности, **is_pod<T>** интерпретируется как тип функции, не принимающей аргументов и возвращающей **is_pod<T>** (§ iso.14.3[2]).

Мое решение состоит в том, чтобы добавить функции, которые обеспечивают традиционную нотацию во всех контекстах:

```
template<typename T>
constexpr bool Is_pod()
{
    return std::is_pod<T>::value;
}
```

Я применяю заглавные буквы для этих функций типа во избежание путаницы со стандартными версиями. Кроме того, я держу их в отдельном пространстве имен (**Estd**).

Мы можем определить свои собственные предикаты типа. Например:

```
template<typename T>
constexpr bool Is_big()
{
    return 100<sizeof(T);
}
```

Мы можем использовать это (довольно грубое) понятие «большой» следующим образом:

```
template<typename T>
using Obj_holder = Conditional<(Is_big<T>()), Scoped<T>, On_heap<T>>;
```

На практике редко когда приходится определять предикаты, которые отражают базовые свойства типов, потому что стандартная библиотека предоставляет многие из них. Примерами служат **is_integral**, **is_pointer**, **is_empty**, **is_polymorphic** и **is_move_assignable** (§35.4.1). Когда же требуется определить подобного рода предикаты, у нас имеются мощные техники для выполнения этой задачи. Например, можно определить функцию типа

для того, чтобы определить, имеет ли класс член с заданным именем и соответствующего типа (§28.4.4).

Естественно, предикаты типа с более чем одним аргументом тоже полезны. В особенности, это имеет место, когда мы представляем отношение между двумя типами, такими как **is_same**, **is_base_of** и **is_convertible**. Они также из стандартной библиотеки.

Я использую функции **Is_*constexpr** для поддержки обычного синтаксиса вызова с **()**.

28.2.3. Выбор функции

Функциональный объект является объектом некоторого типа, так что приемы для выбора типов и значений могут использоваться для выбора функции. Например:

```
struct X { // пишем X
    void operator()(int x) { cout <<"X" << x << "!"; }
    // ...
};

struct Y { // пишем Y
    void operator()(int y) { cout <<"Y" << y << "!"; }
    // ...
};

void f()
{
    Conditional<(sizeof(int)>4), X, Y>{}(7);    // создаем X или Y и вызываем

    using Z = Conditional<(Is_polymorphic<X>()), X, Y>;
    Z zz;       // создаем X или Y
    zz(7);      // вызываем X или Y
}
```

Как здесь показано, выбранный тип функционального объекта можно использовать немедленно или «запомнить» для последующего использования.

Классы с функциями-членами, вычисляющими некоторое значение, составляют самый общий и гибкий механизм для вычислений в шаблонном метапрограммировании.

Conditional является механизмом программирования на этапе компиляции. В частности, это означает, что условие должно быть константным выражением. Обратите внимание на круглые скобки вокруг **sizeof(int)>4**; без них будет получено сообщение о синтаксической ошибке, поскольку компилятор будет интерпретировать **>** как конец списка аргументов шаблона. По этой и другим причинам я предпочитаю использовать **<**, а не **>**. А иногда я использую круглые скобки вокруг условий для повышения читаемости кода.

28.2.4. Шаблоны _traits из стандартной библиотеки

Стандартная библиотека серьезно полагается на шаблоны **_traits**. Они используются для ассоциации свойств с типом. Например, свойства итераторов определяются посредством шаблона **iterator_traits** (§33.1.3):

```
template<typename Iterator>
struct iterator_traits {
    using difference_type = typename Iterator::difference_type;
    using value_type = typename Iterator::value_type;
    using pointer = typename Iterator::pointer;
    using reference = typename Iterator::reference;
    using iterator_category = typename Iterator::iterator_category;
};
```

Можно трактовать это как функцию типа со многими результатами, или как связку функций типа.

В стандартной библиотеке имеются **allocator_traits** (§34.4.2), **char_traits** (§36.2.2), **iterator_traits** (§33.1.3), **regex_traits** (§37.5), **pointer_traits** (§34.4.3). Кроме того, имеются еще и **time_traits** (§35.2.4) и **type_traits** (§35.4.1), которые являются просто функциями типа.

Имея **iterator_traits** для указателя, мы можем говорить о **value_type** и **difference_type** для указателя, хотя указатели и не имеют членов:

```
template<typename Iter>
Iter search(Iter p, Iter q, typename iterator_traits<Iter>::value_type val)
{
    typename iterator_traits<Iter>::difference_type m = q-p;
    // ...
}
```

Это мощная техника, но:

- Она избыточна.
- Она часто увязывает друг с другом слабо связанные функции типа.
- Она экспонирует детали реализации пользователям.

Также люди иногда «ударяются» в синонимы типов «просто так на случай», что ведет к ненужному усложнению. Как следствие, я предпочитаю использовать простые функции типа:

```
template<typename T>
    using Value_type = typename std::iterator_trait<T>::value_type;

template<typename T>
    using Difference_type = typename std::iterator_trait<T>::difference_type;

template<typename T>
    using Iterator_category = typename std::iterator_trait<T>::iterator_category;
```

Сразу все проясняется:

```
template<typename Iter>
Iter search(Iter p, Iter q, Value_type<Iter> val)
{
    Difference_type<Iter> m = q-p;
    // ...
}
```

Я полагаю, что шаблоны **_traits** на практике применяются излишне часто. Рассмотрим, как можно использовать предыдущий пример без какого-либо упоминания про **_traits** и без других функций типа:

```
template<typename Iter, typename Val>
Iter search(Iter p, Iter q, Val val)
{
    auto x = *p;                          // если не хотим именовать тип *p
    auto m = q-p;                         // если не хотим именовать тип q-p

    using value_type = decltype(*p);      // если хотим именовать тип *p
    using difference_type = decltype(q-p); // если хотим именовать тип q-p

    // ...
}
```

Конечно, **decltype()** есть функция типа, так что мне осталось лишь устранить пользовательские и стандартные функции типов. Также, **auto** и **decltype** появились лишь в C++11, и старый код не мог их использовать.

Нам требуются **_traits** (или эквивалент **decltype()**) для того, чтобы ассоциировать один тип с другим, например **value_type** с **T***. Эти средства нужны для неинтрузивного добавления новых имен, что необходимо в обобщенном программировании или в метапрограммировании. Когда же они используются просто для того, чтобы предоставить имя для чего-либо, что уже имеет совершенно корректное имя, вроде **pointer** для **value_type*** и **reference** для **value_type&**, то это приводит лишь к потенциальным недоразумениям.

28.3. Управляющие структуры

Для выполнения развитых вычислений на этапе компиляции нам требуются возможности выбора и рекурсии.

28.3.1. Выбор

В дополнение к тому, что тривиально выполняется посредством константных выражений (§10.4), я использую:

- **Conditional**: как способ выбора между двумя типами (синоним для **std::conditional**)
- **Select**: как способ выбора из нескольких типов (определен в §28.3.1.3)

Эти функции типа возвращают типы. Если нужен выбор из значений, операции?: достаточно; **Conditional** и **Select** — это для выбора типов. Они не являются для времени компиляции эквивалентами **if** и **switch**, хотя и кажутся таковыми, когда используются для выбора между функциональными объектами (§3.4.3, §19.2.2).

28.3.1.1. Выбор из двух типов

Удивительно просто реализовать **Conditional**, как он используется в §28.2. Шаблон **conditional** является частью стандартной библиотеки (находится в файле **<type_traits>**), так что нам не нужно его реализовывать, но он иллюстрирует хорошую технику:

```
template<bool C, typename T, typename F>        // общий шаблон
struct conditional {
    using type = T;
};

template<typename T, typename F>                // специализация для false
struct conditional<false, T, F> {
    using type = F;
};
```

Первичный шаблон (§25.3.1.1) просто определяет свой **type** равным **T** (первый аргумент шаблона после условия). Если условие не **true**, то выбирается специализация для **false**, и **type** определяется равным **F**. Например:

```
typename conditional<(std::is_polymorphic<T>::value), X, Y>::type z;
```

Очевидно, что синтаксис оставляет желать лучшего (§28.2.2), но идея чудесная.

Специализации используют для того, чтобы отделить общий случай от некоторого количества специальных случаев (§25.3). В нашем примере первичный шаблон заботится о примерно половине функциональности, варьирующейся от почти нулевой (все безошибочные случаи обрабатываются специализацией; §25.3.1.1) до практически полной

за исключением одного лишь финального случая (§28.5). Такой выбор целиком относится ко времени компиляции и не стоит ни одного байта или цикла во время выполнения.

Для улучшения синтаксиса я ввожу синоним типа:

```
template<bool B, typename T, typename F>
using Conditional = typename std::conditional<B, T, F>::type;
```

После этого мы можем написать:

```
Conditional<(Is_polymorphic<T>()), X, Y> z;
```

Я считаю это существенным улучшением.

28.3.1.2. Этап компиляции или этап выполнения

Когда в первый раз мы сталкиваемся с чем-либо вроде

```
Conditional<(std::is_polymorphic<T>::value), X, Y> z;
```

то сразу же возникает вопрос, «Почему бы не написать нормальный **if**?». Рассмотрим выбор из двух альтернатив: **Square** и **Cube**:

```
struct Square {
    constexpr int operator()(int i) { return i*i; }
};

struct Cube {
    constexpr int operator()(int i) { return i*i*i; }
};
```

Мы могли бы попробовать стандартный оператор **if**:

```
if (My_cond<T>())
    using Type = Square;        // ошибка: объявление в ветви оператора if
else
    using Type = Cube;          // ошибка: объявление в ветви оператора if

Type x;                         // ошибка: Type вне области видимости
```

Но объявление не может быть единственным оператором в ветви оператора **if** (§6.3.4, §9.4.1), так что это не будет работать, несмотря на то, что **My_cond<T>()** вычисляется во время компиляции. Таким образом, обычный оператор **if** полезен для обычных выражений, но не для выбора типа.

Попробуем пример, не использующий определение переменной:

```
Conditional<My_cond<T>(), Square, Cube>{}(99);  // вызов Square{}(99) или Cube{}(99)
```

то есть выбрать тип, сконструировать умолчательный объект этого типа и вызвать его. Это работает. При использовании стандартных структур управления это выглядело бы следующим образом:

```
((My_cond<T>())?Square:Cube){}(99);
```

Этот пример не работает, поскольку **Square{}(99)** и **Cube{}(99)** не возвращают типы, а скорее значения типов, подходящих для условных выражений (§11.1.3). Можно еще попробовать

```
(My_cond<T>()?Square{}:Cube{})(99);     // ошибка: несовместимые аргументы для ?:
```

К сожалению, этот пример также страдает от того, что **Square{}** и **Cube{}** не относятся к типам, допустимым в качестве альтернатив в выражении **?:**. Ограничение совместимыми

типами часто неприемлемо в метапрограммировании, потому что нам нужен выбор из типов, которые не связаны явным образом.

Наконец, вот это работает:

```
My_cond<T>()?Square{}(99):Cube{}(99);
```

Но оно не более читаемо, чем

```
Conditional<My_cond<T>(), Square, Cube>{}(99);
```

28.3.1.3. Выбор из нескольких типов

Выбор из N альтернатив почти аналогичен выбору из двух альтернатив. Вот функция типа, возвращающая тип своего N-го аргумента:

```
class Nil {};

template<int I, typename T1 =Nil, typename T2 =Nil, typename T3 =Nil, typename T4 =Nil>
struct select;

template<int I, typename T1 =Nil, typename T2 =Nil, typename T3 =Nil, typename T4 =Nil>
using Select = typename select<I, T1, T2, T3, T4>::type;

// Специализации для 0–3:

template<typename T1, typename T2, typename T3, typename T4>
struct select<0, T1, T2, T3, T4> { using type = T1; }; // специализация для N==0

template<typename T1, typename T2, typename T3, typename T4>
struct select<1, T1, T2, T3, T4> { using type = T2; }; // специализация для N==1

template<typename T1, typename T2, typename T3, typename T4>
struct select<2, T1, T2, T3, T4> { using type = T3; }; // специализация для N==2

template<typename T1, typename T2, typename T3, typename T4>
struct select<3, T1, T2, T3, T4> { using type = T4; }; // специализация для N==3
```

Так как общая версия **select** никогда не будет использоваться, то я ее и не определил. Я выбрал отсчет с нуля, чтобы соответствовать остальному C++-коду. Наша техника очень общая: такие специализации могут представить любой аспект аргументов шаблона. Нам не нужно максимальное число альтернатив (здесь, четыре), но эту задачу можно решить с помощью шаблонов с переменным числом параметров (§28.6). В результате выбора несуществующей альтернативы используется первичный (общий) шаблон. Например:

```
Select<5, int, double, char> x;
```

В этом случае получаем немедленную ошибку компиляции, так как общий **Select** не определен.

В реальном случае нужно выбрать тип для функции, возвращающей N-ый элемент множества:

```
template<int N, typename T1, typename T2, typename T3, typename T4>
Select<N, T1, T2, T3, T4> get(Tuple<T1, T2, T3, T4>& t);    // см. §28.5.2

auto x = get<2>(t); // положим, что t есть Tuple
```

Здесь тип **x** будет тем же, чем **T3** является для **t** (типа **Tuple**). Индексирование выполняется с нуля.

Используя шаблоны с переменным числом параметров (§28.6), мы можем получить более простой и более общий **select**:

```
template<unsigned N, typename... Cases>          // общий случай; не конкретизируется
struct select;

template<unsigned N, typename T, typename ... Cases>
struct select<N, T, Cases...>: select<N-1, Cases...> {
};

template<typename T, typename ... Cases>         // последний случай: N==0
struct select<0, T, Cases...> {
    using type = T;
};

template<unsigned N, typename... Cases>
using Select = typename select<N, Cases...>::type;
```

28.3.2. Итерация и рекурсия

Базовый прием вычисления значений на этапе компиляции можно проиллюстрировать на примере шаблона для вычисления факториала:

```
template<int N>
constexpr int fac()
{
    return N*fac<N-1>();
}}

template<>
constexpr int fac<1>()
{
    return 1;
}

constexpr int x5 = fac<5>();
```

Факториал реализуется посредством рекурсии, а не при помощи цикла. Поскольку у нас нет переменных во время компиляции (§10.4), то в этом есть смысл. В самом общем случае, если нам нужно пройти набор элементов во время компиляции, то нужно использовать рекурсию.

Обратите внимание на отсутствие условия: нет никаких проверок **N==1** или **N<2**. Вместо этого окончание рекурсии имеет место при вызове **fac()**, который выбирает специализацию для **N==1**. В шаблонном метапрограммировании (как и в функциональном программировании) для прохода по последовательности значений используется рекурсия до тех пор, пока не будет достигнута терминирующая специализация.

В нашем случае можно было бы выполнить вычисления и более традиционным способом:

```
constexpr int fac(int i)
{
    return (i<2)?1: fac(i-1);
}

constexpr int x6 = fac(6);
```

Мне кажется это более ясным решением, чем реализация идеи посредством шаблона функции, но о вкусах не спорят и есть алгоритмы, лучше выражаемые через отделение терминирующего случая от общей канвы.

Версия с **constexpr** может использоваться как во время компиляции, так и для вычислений во время выполнения. Шаблонная версия (метапрограммирование) подходит лишь для исполнения на этапе компиляции.

28.3.2.1. Рекурсия с классами

Рекурсия с более сложными состояними или с более сложной параметризацией может использовать классы. Например, факториальная программа принимает вид:

```
template<int N>
struct Fac {
    static const int value = N*Fac<N-1>::value;
};

template<>
struct Fac<1> {
    static const int value = 1;
};

constexpr int x7 = Fac<7>::value;
```

Более реалистичный пример представлен в §28.5.2.

28.3.3. Когда использовать метапрограммирование

Применяя описанные конструкции управления, можно выполнить любые вычисления на этапе компиляции. Остается вопрос: «Захотим ли мы этого?». Применение этих техник оправдано в том случае, если мы сочтем код более чистым и ясным, с лучшей производительностью и более простым в поддержке. Очевидным препятствием к метапрограммированию служит применение усложненных вариантов использования шаблонов, которые тяжело читать и отлаживать. Также при этом возрастает время компиляции.

Шаблонное метапрограммирование привлекает к себе умных людей:

- Частично потому, что метапрограммирование позволяет выразить такие вещи, которые иначе не реализуются на таком уровне безопасности по типам и высокой производительности во время выполнения.
- Частично потому, что метапрограммирование позволяет проявить нашу смекалку и квалификацию. Но это совсем не обязательно делать.

Как далеко следует заходить в метапрограммировании? Одним настораживающим признаком может послужить необходимость в макросах (§12.6) для упрятывания деталей, слишком безобразно выглядящих (чтобы явно показывать их). Рассмотрим следующее:

```
#define IF(c, x, y) typename std::conditional<(c), x, y>::type
```

Зашли слишком далеко? Но это позволяет нам писать

```
IF(cond, Cube, Square) z;
```

вместо

```
typename std::conditional<(cond), Cube, Square>::type z;
```

Сами решайте, что лучше: использовать короткое имя **IF** или длинное **std::conditional**.

Я вынужден писать **typename** и **::type** ради стандартной терминологии. Это проявляет технику реализации шаблонов. При желании скрыть это я обращаюсь к макросам. В случае больших проектов и множества участвующих в них людей немного излишеств предпочтительнее, чем расхождение в нотации.

Серьезным аргументом против **IF**-макроса служит то, что его имя дезориентирует — это не замена традиционного **if**, так как **conditional** осуществляет выбор из типов и не является вариацией управляющей структуры кода. Макрос **IF** серьезным образом скрывает функциональность. Часто это общее возражение против «чувствительных» макросов: они именуются в рамках конкретной идеи их использования и не отражают фундаментальную функциональность.

Проблемы избыточности, проявления деталей реализации и плохого именования решаются с помощью синонимов типа (**Conditional**; §28.2.1). В общем случае следует упорно работать над тем, чтобы синтаксис, обращенный к пользователям, не содержал бы «частного языка». Предпочитайте систематичность, основанную на специализациях и использовании синонимов, а не «хакерство» с макросами. Для выполнения вычислений на этапе компиляции отдавайте предпочтение **constexpr** перед шаблонами, и где возможно прячьте реализацию шаблонного метапрограммирования в **constexpr**-функциях (§28.2.2).

Можно посмотреть и с другой стороны на фундаментальные сложности в коде:

1. Нужны ли нам явные проверки?
2. Требуется ли рекурсия?
3. Можем ли мы написать концепции (§24.3) для аргументов шаблона?

Если ответ на вопросы [1] и [2] «Да» или ответ на вопрос 3 «Нет», то следует задуматься о том, не будут ли проблемы с поддержкой кода. Может нужна какая-то форма инкапсуляции. Помните, что сложности в реализации шаблонов становятся видимыми пользователям, когда конкретизация терпит неудачу. Кроме того, многие программисты заглядывают в заголовочные файлы, в которых все детали метапрограммирования проявляются явным образом.

28.4. Условное определение: Enable_if

Когда мы пишем шаблон, нам может потребоваться операция для некоторых аргументов шаблона, но не для всех аргументов. Например:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();      // возврат ссылки на весь объект
    T* operator->();     // выбор члена (только для классов)
    // ...
}
```

Если **T** это класс, то мы должны предоставить **operator->()**, но если **T** является встроенным типом, то это сделать невозможно (в обычной семантике). Поэтому нам нужен механизм языка, позволяющий сказать: «Если у этого типа есть такое-то вот свойство, то тогда определяем ...». Можно попробовать очевидное:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();                          // возврат ссылки на целый объект
    if (Is_class<T>()) T* operator->();      // синтаксическая ошибка
    // ...
}
```

К сожалению, это не работает. В языке отсутствует **if**, который может выполнить выбор среди определений, базирующихся на общем условии. Но, как и в случае с **Conditional**

и **Select** (§28.3.1), выход есть. Мы можем написать любопытную функцию типа, чтобы сделать определение **operator->()** условным. Стандартная библиотека (в файле **<type_traits>**) с этой целью предоставляет **enable_if**. Пример для **Smart_pointer** принимает вид:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();                                  // возврат ссылки на весь объект
    Enable_if<Is_class<T>(), T>* operator->();       // выбор члена (только для классов)
    // ...
}
```

Как обычно, я использовал синонимы типов и **constexpr**-функции ради упрощения нотации:

```
template<bool B, typename T>
using Enable_if = typename std::enable_if<B, T>::type;

template<typename T> bool Is_class()
{
    return std::is_class<T>::value;
}
```

Если условие в **Enable_if** возвращает **true**, то результатом будет второй аргумент (здесь **T**). Если же это условие дает **false**, то все объявление функции, частью которой оно является, полностью игнорируется. В таком случае, если **T** это класс, мы получаем определение **operator->()** с возвратом **T***, а в противном случае нет никакого объявления вообще.

Имея определение **Smart_pointer**, использующее **Enable_if**, получаем:

```
void f(Smart_pointer<double> p, Smart_pointer<complex<double>> q)
{
    auto d0 = *p; // OK
    auto c0 = *q; // OK
    auto d1 = q->real();      // OK
    auto d2 = p->real();      // ошибка: p не указывает на объект класса
    // ...
}
```

Вы можете счесть **Smart_pointer** и **operator->()** экзотикой, но условное предоставление операций достаточно распространено на практике. Стандартная библиотека содержит много примеров условных определений, таких как **Alloc::size_type** (§34.4.2) и **pair**, которое может быть перемещаемым, если оба ее элемента таковы (§34.2.4.1). Язык сам по себе определяет **->** только для указателей на объекты классов (§8.2).

В этом случае вариант **operator->()** с **Enable_if** просто поменяет код ошибки, который породят примеры, такие как **p->real()**:

- Если мы в безусловном порядке объявим **operator->()**, то мы получим сообщение об ошибке, что « **->** применяется к неклассовому указателю» в момент конкретизации для определения **Smart_pointer<double>::operator->()**.
- Если мы в условном порядке объявим **operator->()** с **Enable_if**, то если мы используем **->** для **smart_ptr<double>**, то мы получим сообщение об ошибке «**Smart_ptr<double::operator->()** не определен» в точке использования **Smart_pointer<double>::operator->()**.

В остальных случаях мы не получаем сообщения об ошибке, если только мы не используем **->** для **smart_ptr<T>**, где **T** не является классом.

Мы переместили обнаружение ошибки и сообщение о ней из реализации **smart_pointer<T>::operator->()** в объявление. Это сильно меняет ситуацию в зависимости от того, насколько глубоко в сетке вложенных конкретизаций ошибка возникает. Вообще-то, предпочтительнее точно детектировать ошибку пораньше, а не полагаться на то, что будут перехвачены ошибочные конкретизации. В таком разрезе мы можем трактовать **Enable_if** как некий вариант идеи концепций (§24.3): в этом случае мы можем точнее специфицировать требования к шаблону.

28.4.1. Использование Enable_if

Для многих случаев функциональности **enable_if** вполне достаточно. Однако, нотация при этом несколько неуклюжая. Рассмотрим следующее:

```
Enable_if<Is_class<T>(), T>* operator->();
```

Здесь реализация выступает слишком явно. В то же время то, что здесь отражается, весьма близко к минимальному идеалу:

```
declare_if (Is_class<T>()) T* operator->();      // не C++
```

К сожалению язык C++ не располает конструкцией **declare_if** для выбора объявлений.

Применение **Enable_if** для декорирования типов возврата помещает их туда, где они видны вам и где они с логической точки зрения должны быть, поскольку при этом затрагивается все объявление (а не один лишь тип возврата). Однако ж, не все объявления имеют тип возврата. Рассмотрим два конструктора у **vector**:

```
template<typename T>
class vector<T> {
public:
    vector(size_t n, const T& val);      // n элементов типа T со значением val

    template<typename Iter>
        vector(Iter b, Iter e);          // инициализируем из [b:e)
    // ...
};
```

Все это выглядит невинно, но конструктор, принимающий число элементов, терпит неудачу. Вот пример:

```
vector<int> v(10,20);
```

Это что: **10** элементов со значением **20**, или попытка инициализации из интервала [**10:20**]? Стандарт требует первого, но в нашем коде будет выбран второй вариант, поскольку **int-в-size_t** конверсия требуется для первого конструктора, в то время как пара значений **int** является точным совпадением для шаблонного конструктора. Проблема в том, что я «забыл» сказать компилятору, что тип **Iter** должен быть итератором. Однако, это можно сделать:

```
template<typename T>
class vector<T> {
public:
    vector(size_t n, const T& val);      // n элементов типа T со значением val

    template<typename Iter, typename =Enable_if<Input_iterator<Iter>(), Iter>>
        vector(Iter b, Iter e);          // инициализируем из [b:e)
    // ...
};
```

Этот (неиспользуемый) умолчательный аргумент шаблона будет конкретизироваться, потому что мы действительно не можем вывести этот неиспользуемый параметр шаблона. Из этого вытекает, что объявление **vector(Iter, Iter)** потерпит неудачу, если только **Iter** не будет **Input_iterator** (§24.4.4).

Я представил **Enable_if** в качестве умолчательного шаблонного аргумента, поскольку это наиболее общее решение. Его можно использовать для шаблонов без аргументов и/или не имеющих возвращаемых типов. Но в этом случае мы могли бы в качестве альтернативы применить его к типу аргумента конструктора:

```
template<typename T>
class vector<T> {
public:
    vector(size_t n, const T& val);      // n элементов типа T со значением val

    template<typename Iter>
    vector(Enable_if<Input_iterator<Iter>(), Iter>> b, Iter e);      // инициализируем из [b:e)
    // ...
};
```

Приемы работы с **Enable_if** работают только в случае шаблонов функций (включая функции-члены шаблонов классов и специализаций). Реализация и использование **Enable_if** опираются на детали правил перегрузки шаблонов функций (§23.5.3.2). Как следствие, его нельзя использовать для контроля объявлений классов, переменных и нешаблонных функций. Например:

```
Enable_if<(version2_2_3<config), M_struct>* make_default() // ошибка: не шаблон
{
    return new Mystruct{};
}

template<typename T>
void f(const T& x)
{
    Enable_if<(20<sizeof<T>), T> tmp = x;                  // ошибка: tmp не функция
    Enable_if<!(20<sizeof<T>), T&> tmp = *new T{x};        // ошибка: tmp не функция
    // ...
}
```

Для **tmp** использование **Holder** (§28.2) будет несомненно более чистым решением: если вам удалось сконструировать этот объект в свободной памяти, то как вы будете удалять (**delete**) его?

28.4.2. Реализация Enable_if

Реализация **Enable_if** практически тривиальная:

```
template<bool B, typename T = void>
struct std::enable_if {
    typedef T type;
};

template<typename T>
struct std::enable_if<false, T> {};  // no::type если B==false

template<bool B, typename T = void>
using Enable_if = typename std::enable_if<B, T>::type;
```

Заметьте, что мы можем опустить аргумент типа и получить **void** по умолчанию.

Объяснение с точки зрения технических подробностей языка того факта, что такие простые объявления становятся полезными в роли фундаментальных конструкций, дано в §23.5.3.2.

28.4.3. Enable_if и концепции

Мы можем использовать **Enable_if** для разных предикатов, включая проверки свойств типов (§28.3.1.1). Концепции являются одними из самых общих и полезных предикатов, имеющихся в нашем распоряжении. В идеале, хотелось бы осуществлять перегрузку на основе концепций, но из-за языковых ограничений в поддержке концепций самое большее, что мы можем сделать, это использовать **Enable_if** для выбора ограничений. Например:

```
template<typename T>
Enable_if<Ordered<T>()> fct(T*, T*);      // оптимизированная реализация

template<typename T>
Enable_if<!Ordered<T>()> fct(T*, T*);     // неоптимизированная реализация
```

Так как по умолчанию **Enable_if** дает **void**, то **fct()** является **void**-функцией. Я не уверен, что использование умолчательного значения повышает читаемость, но можно использовать **fct()** следующим образом:

```
void f(vector<int>& vi, vector<complex<int>>& vc)
{
    if (vi.size()==0 || vc.size()==0) throw runtime_error("bad fct arg");
    fct(&vi.front(),&vi.back());      // оптимизированный вызов
    fct(&vc.front(),&vc.back());      // неоптимизированный вызов
}
```

Эти вызовы разрешаются так, как описано выше, потому что мы можем использовать **<** для **int**, но не для **complex<int>**. **Ebable_if** разрешается к **void**, если мы не предоставим аргумент типа.

28.4.4. Дополнительные примеры с Enable_if

Используя **Enable_if**, рано или поздно мы спросим, имеет ли класс член с заданным именем и подходящим типом. Для многих стандартных операций, таких как конструкторы и присваивания, стандартная библиотека предоставляет предикат свойств типа, такой как **is_copy_assignable** и **is_default_constructible** (§35.4.1). Но мы можем создавать и свои собственные предикаты. Рассмотрим такую постановку вопроса: «Можно ли вызвать **f(x)**, если **x** имеет тип **X**?». Для ответа мы создадим **has_f**, что даст нам возможность продемонстрировать некоторые приемы и непростой код, внутренне реализованный во многих библиотеках метапрограммирования (включая отдельные части стандартной библиотеки). Сначала определим обычный класс плюс специализацию для представления альтернативы:

```
struct substitution_failure { };      // представляет failure объявить что-нибудь

template<typename T>
struct substitution_succeeded : std::true_type
{ };

template<>
struct substitution_succeeded<substitution_failure> : std::false_type
{ };
```

Здесь **substitution_failure** используется для представления ситуации с неудачей подстановки (§23.5.3.2). Мы выполняем наследование от **std::true_type**, если только аргумент не имеет тип **substitution_failure**. Очевидно, что **std::true_type** и **std::false_type** являются типами, которые представляют значения **true** и **false**, соответственно:

```
std::true_type::value == true
std::false_type::value == false
```

Мы используем **substitution_failure** для того, чтобы определять функции типа, которые нам нужны. Например, нам может потребоваться функция **f**, которую можно вызывать как **f(x)**. Для этого мы можем определить **has_f**:

```
template<typename T>
struct has_f
    : substitution_succeeded<typename get_f_result<T>::type>
{ };
```

Итак, если **get_f_result<T>** выдает надлежащий тип (предположительно тип возврата при вызове **f**), **has_f::value** дает **true_type::value**, что есть **true**. Если же **get_f_result<T>** не компилируется, то возвращается **substitution_failure** и **has_f::value** равное **false**.

Пока что все хорошо, но как мы получим **get_f_result<T>** с возвратом **substitution_failure**, если **f(x)** не компилируется для значения **x** типа **X**? Определение, приводящее к такой ситуации, выглядит достаточно невинно:

```
template<typename T>
struct get_f_result {
private:
    template<typename X>
        static auto check(X const& x) -> decltype(f(x));   // может вызвать f(x)
    static substitution_failure check(...);                // не может вызвать f(x)
public:
    using type = decltype(check(std::declval<T>()));
};
```

Мы просто объявляем функцию **check** так, что **check(x)** имеет тот же тип возврата, что и **f(x)**. Очевидно, что это не компилируется, если только мы не можем вызывать **f(x)**. Итак, это объявление для **check** терпит неудачу, если мы не можем вызывать **f(x)**. В таком случае, поскольку неудача с подстановкой не есть ошибка (SFINAE; §23.5.3.2), мы приходим ко второму определению **check()**, у которого **substitution_failure** является типом возврата. И, да, этот трюковый код терпит неудачу, если наша функция **f** объявляется с типом возврата **substitution_failure**.

Отметим, что **decltype()** не оценивает свой операнд.

Нам удалось повернуть то, что выглядит как ошибка типа, в сторону значения **false**. Было бы проще, если бы язык предоставлял примитивную (встроенную) операцию для выполнения такой конверсии; например:

```
is_valid());      // can f(x) be compiled?
```

К сожалению, язык не в состоянии обеспечить все на свете как собственный примитив. Имея зубодробительный код, мы должны обеспечить стандартный синтаксис:

```
template<typename T>
constexpr bool Has_f()
{
    return has_f<T>::value;
}
```

Теперь мы можем написать:

```
template<typename T>
class X {
    // ...
    Enable_if<Has_f<T>()> use_f(const T&)
    {
        // ...
        f(t);
        // ...
    }
    // ...
};
```

X<T> имеет член **use_f()** тогда и только тогда, когда **f(t)** может вызываться для **t** типа **T**. Заметим, что мы не можем просто написать следующее:

```
if (Has_f<decltype(t)>()) f(t);
```

Вызов **f(t)** будет проверен на тип (и это не пройдет), даже если **Has_f<decltype(t)>()** возвращает **false**.

Исходя из техники определения **Has_f**, мы можем определить **Has_foo** для любой операции или члена **foo**, какие только мы можем представить. Все это займет 14 строк кода для каждого **foo**. При повторениях это утомительно, но не сложно.

Все это предполагает, что **Enable_if<>** позволяет выбрать среди перегруженных шаблонов на основе логических критериев для типов аргументов. Например, мы можем определить функцию типа **Has_not_equals()** для проверки того, доступна ли операция **!=**, и использовать ее следующим образом:

```
template<typename Iter, typename Val>
Enable_if<Has_not_equals<Iter>(), Iter> find(Iter first, Iter last, Val v)
{
    while (first!=last &&!(*first==v))
        ++first;
    return first;
}

template<typename Iter, typename Val>
Enable_if<! Has_not_equals<Iter>(), Iter> find(Iter first, Iter last, Val v)
{
    while (!(first==last) &&!(*first==v))
        ++first;
    return first;
}
```

Такая перегрузка по факту легко становится запутанной и сложной в поддержке. К примеру, попробуйте добавить версии, использующие **!=** для сравнения значений (то есть ***first!=v**, а не **!(*first==v)**), когда это возможно. Как следствие, я рекомендую полагаться на более структурированый стандарт перегрузки (§12.3.1) и правила специализаций (§25.3), когда есть выбор. Например:

```
template<typename T>
auto operator!=(const T& a, const T& b) -> decltype(!(a==b))
{
    return!(a==b);
}
```

Правила гарантируют, что если специфический вариант **!=** уже определен для типа **T** (как шаблон или нешаблонный вариант функции), то это определение не будет конкрети-

зироваться. Я использую **decltype()** частично для того, чтобы показать, как в общем случае выводить тип возврата из ранее определенный операции, и частично для того, чтобы обрабатывать редкие случаи, когда != возвращает нечто, отличное от **bool**.

Аналогично, мы можем условно определить **>**, **<=**, **>=** и т.д., исходя из **<**.

28.5. Список времени компиляции: Tuple

Здесь я продемонстрирую базовую технику метапрограммирования на простом, но реалистичном примере. Я определю **Tuple** с ассоциированной операцией доступа и операцией вывода. Такое определение широко используется более чем десятилетие. Более общий и более элегантный вариант **std::tuple** представлен в §28.6.4 и в §34.2.4.2.

Идея заключается в том, чтобы сделать возможным следующий код:

```
Tuple<double, int, char> x {1.1, 42, 'a'};
cout << x << "\n";
cout << get<1>(x) << "\n";
```

Вот вывод этого кода:

```
{1.1, 42, 'a'};
42
```

Определение **Tuple** фундаментально простое:

```
template<typename T1=Nil, typename T2=Nil, typename T3=Nil, typename T4=Nil>
struct Tuple : Tuple<T2, T3, T4> {      // раскладка: {T2, T3, T4} перед T1
    T1 x;

    using Base = Tuple<T2, T3, T4>;

    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1, const T2& t2, const T3& t3, const T4& t4): Base{t2, t3, t4}, x{t1}
{ }
};
```

Таким образом, **Tuple** с четырьмя элементами есть **Tuple** с тремя элементами с последующим четвертым элементом.

Мы сконструировали **Tuple** конструктором, который принимает четыре значения (потенциально четырех различных типов). Он использует последние три элемента (хвост) для инициализации своей базы из трех элементов и первый элемент (голова) для инициализации члена **x**.

Манипуляции с хвостом **Tuple** — то есть манипуляции с базовым классом **Tuple** — важны и часто используются в реализациях **Tuple**. Как следствие, я предоставил синоним **Base** и пару функций-членов **base()** для упрощения манипуляций с базой/хвостом.

Очевидно, что данное определение работает тогда, когда элементов действительно четыре. Случаи, когда элементов меньше четырех, определяются как специализации:

```
template<>
struct Tuple<> { Tuple() {} }; // 0-tuple

template<typename T1>
struct Tuple<T1> : Tuple<> { // 1-tuple
    T1 x;
```

```
    using Base = Tuple<>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1): Base{}, x{t1} { }
};

template<typename T1, typename T2>
struct Tuple<T1, T2> :  Tuple<T2> {              // 2-tuple, раскладка: T2 перед T1
    T1 x;

    using Base = Tuple<T2>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1, const T2& t2): Base{t2}, x{t1} { }
};

template<typename T1, typename T2, typename T3>
struct Tuple<T1, T2, T3> : Tuple<T2, T3> {  // 3-tuple, раскладка: {T2, T3} перед T1
    T1 x;

    using Base = Tuple<T2, T3>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

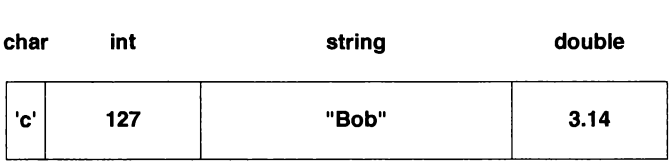
    Tuple(const T1& t1, const T2& t2, const T3& t3): Base{t2, t3}, x{t1} { }
};
```

Это повторяющиеся определения, следующие схеме первого **Tuple** (из четырех элементов). То определение является первичным шаблоном и предоставляет интерфейс к разноэлементным вариантам (0, 1, 2, 3 и 4-элементным вариантам **Tuple**). Поэтому я вынужден был предоставить **Nil**-умолчательные аргументы шаблона. Фактически они никогда не будут использоваться. Специализации выберут какой-либо вариант вместо использования **Nil**.

Способ, которым я определил **Tuple** в виде «стека» производных классов, достаточно традиционный (например, **std::tuple** определяется аналогично; §28.5). Любопытно, что при этом первый элемент **Tuple** получает (при обычной технике реализации) наивысший адрес, а адрес последнего элемента совпадает с адресом самого **Tuple**. Пример:

```
tuple<double, string, int, char>{3.14, string{"Bob"},127,'c'}
```

можно графически отобразить следующим образом:



Это открывает интересные возможности для оптимизации:

```
class FO { /* объект-функция без членов данных */ };

typedef Tuple<int*, int*> T0;
typedef Tuple<int*, FO> T1;
typedef Tuple<int*, FO, FO> T2;
```

На своей системе я получил, что **sizeof(T0)==8**, **sizeof(T1)==4** и **sizeof(T2)==4**, так как компилятор выкинул (в качестве оптимизации) пустые базовые классы. Это носит название *оптимизации пустых базовых классов* и гарантируется языком (§27.4.1).

28.5.1. Простая функция вывода

Определение **Tuple** имеет регулярную рекурсивную структуру, которую мы можем использовать для определения функции, демонстрирующей список элементов. Пример:

```
template<typename T1, typename T2, typename T3, typename T4>
void print_elements(ostream& os, const Tuple<T1, T2, T3, T4>& t)
{
    os << t.x << ", ";   // x y t
    print_elements(os,*t.base());
}

template<typename T1, typename T2, typename T3>
void print_elements(ostream& os, const Tuple<T1, T2, T3>& t)
{
    os << t.x << ", ";
    print_elements(os,*t.base());
}

template<typename T1, typename T2>
void print_elements(ostream& os, const Tuple<T1, T2>& t)
{
    os << t.x << ", ";
    print_elements(os,*t.base());
}

template<typename T1>
void print_elements(ostream& os, const Tuple<T1>& t)
{
    os << t.x;
}

template<>
void print_elements(ostream& os, const Tuple<>& t)
{
    os << " ";
}
```

Похожесть вариантов с разным числом элементов намекает на более оптимальное решение (§28.6.4), но пока что я использую **print_elements()** в определении **<<** для **Tuple**:

```
template<typename T1, typename T2, typename T3, typename T4>
ostream& operator<<(ostream& os, const Tuple<T1, T2, T3, T4>& t)
{
    os << "{";
    print_elements(os, t);
    os << " }";
    return os;
}
```

Теперь мы можем написать:

```
Tuple<double, int, char> x {1.1, 42, 'a'};
cout << x << "\n";
```

```
cout << Tuple<double, int, int, int>{1.2,3,5,7} << "\n";
cout << Tuple<double, int, int>{1.2,3,5} << "\n";
cout << Tuple<double, int>{1.2,3} << "\n";
cout << Tuple<double>{1.2} << "\n";
cout << Tuple<>{} << "\n";
```

Удивительно, но вывод будет таков:

```
{1.1 42, a }
{1.2,3,5,7 }
{1.2,3,5 }
{1.2,3 }
{1.2 }
{ }
```

28.5.2. Доступ к элементам

Как мы определили **Tuple**, он имеет переменное число элементов потенциально разных типов. Хотелось бы иметь эффективный доступ к элементам без нарушения системы типов (без использования явного приведения типов). Можно рассматривать различные схемы, включая именование элементов, их нумерирование, рекурсивный просмотр вплоть до нахождения нужного элемента. Наконец, то что мы хотим реализовать — это индексировние элементов. К сожалению, я не смогу реализовать надлежащий **operator[]**, так что я использую шаблонную функцию **get()**:

```
Tuple<double, int, char> x {1.1, 42, 'a'};

cout << "{"
     << get<0>(x) << ", "
     << get<1>(x) << ", "
     << get<2>(x) << " }\n";  // пишет { 1.1, 42, a }

auto xx = get<0>(x);          // xx есть double
```

Идея состоит в том, чтобы индексировать элементы с нуля таким образом, чтобы выбор элемента выполнялся на этапе компиляции и сохранялась вся информация о типе.

Функция **get()** конструирует объект типа **getNth<T, int>**, работа которого состоит в том, чтобы возвращать ссылку на **N**-ый элемент, который предположительно имеет тип **X**. Отталкиваясь от такого помошника, мы определяем функцию **get()**:

```
template<int N, typename T1, typename T2, typename T3, typename T4>
Select<N, T1, T2, T3, T4>& get(Tuple<T1, T2, T3, T4>& t)
{
     return getNth<Select<N, T1, T2, T3, T4>, N>::get(t);
}
```

Определение **getNth** является вариантом обычной рекурсии от **N** вниз к специализации для **0**:

```
template<typename Ret, int N>
struct getNth {                  // getNth() помнит тип (Ret) N-го элемента
     template<typename T>
     static Ret& get(T& t)       // получить значение элемента N из базы t
     {
          return getNth<Ret, N-1>::get(*t.base());
     }
};
```

```
template<typename Ret>
struct getNth<Ret,0> {
    template<typename T>
    static Ret& get(T& t)
    {
        return t.x;
    }
};
```

С фундаментальной точки зрения **getNth** является специализированным **for**-циклом, рекурсирующим **N-1** раз. Функции-члены статические, поскольку нам на самом деле не нужны объекты класса **getNth**. Этот класс используется лишь для того, чтобы размещать **Ret** и **N** так, чтобы компилятор мог их использовать.

Все немного замысловато, но результирующий код безопасен по типам и эффективен. Когда я говорю эффективен, я имею ввиду применение достаточно разумного компилятора, который не допустит никаких дополнительных накладных расходов для доступа к элементам **Tuple**.

Почему мы должны писать **get<2>(x)**, а не **x[2]**? Можно попробовать следующее:

```
template<typename T>
constexpr auto operator[](T t, int N)
{
    return get<N>(t);
}
```

К сожалению, это не работает:

- **operator[]()** должен быть членом, но с этим можно справиться, определяя его внутри **Tuple**.
- Внутри **operator[]** аргумент **N** не представляется константным выражением.
- Я «забыл», что только лямбды могут выводить тип своего возврата из операторов **return** (§11.4.4), но с этим можно справиться, добавляя **->dectype(get<N>(t))**.

Все это предполагает некоторое узаконивание в языке, но пока что приходится иметь дело с **get<2>(x)**.

28.5.2.1 Константный Tuple

В приведенном определении **get()** работает с неконстантными элементами **Tuple** и может использоваться в левой части операций присваивания. Например:

```
Tuple<double, int, char> x {1.1, 42, 'a'};

get<2>(x) = 'b';      // OK
```

Однако, это можно использовать и для констант:

```
const Tuple<double, int, char> xx {1.1, 42, 'a'};

get<2>(xx) = 'b';          // ошибка: xx константа
char cc = get<2>(xx);      // ошибка: xx константа (сюрприз?)
```

Проблема лишь в том, что **get()** принимает свой аргумент по неконстантной ссылке. Но **xx** константа, так что это неприемлемый аргумент.

Естественно, хотелось бы иметь и константный **Tuple**. Пример

```
const Tuple<double, int, char> xx {1.1, 422, 'a'};
char cc = get<2>(xx);      // OK: чтение из const
cout << "xx: " << xx << "\n";
get<2>(xx) = 'x';          // ошибка: xx константа
```

Чтобы работать с константными **Tuple**, мы должны добавить **const**-версии для **get()** и для **get()** из **getNth**. Пример:

```
template<typename Ret, int N>
struct getNth {                          // getNth() помнит тип (Ret) N-го элемента
template<typename T>
     static Ret& get(T& t)               // получить величину элемента N из базы (base) у t
     {
          return getNth<Ret, N-1>::get(*t.base());
     }

     template<typename T>
     static const Ret& get(const T& t) // получить величину элемента N из базы t
     {
          return getNth<Ret, N-1>::get(*t.base());
     }
};

template<typename Ret>
struct getNth<Ret,0> {
     template<typename T> static Ret& get(T& t) { return t.x; }
     template<typename T> static const Ret& get(const T& t) { return t.x; }
};

template<int N, typename T1, typename T2, typename T3, typename T4>
Select<N, T1, T2, T3, T4>& get(Tuple<T1, T2, T3, T4>& t)
{
     return getNth<Select<N, T1, T2, T3, T4>, N>::get(t);
}

template<int N, typename T1, typename T2, typename T3>
const Select<N, T1, T2, T3>& get(const Tuple<T1, T2, T3>& t)
{
     return getNth<Select<N, T1, T2, T3>, N>::get(t);
}
```

Теперь мы можем работать как с константными, так и с неконстантными аргументами.

28.5.3. Функция make_tuple

Классовый шаблон не может выводить свои аргументы, а функциональный шаблон — может — из своих функциональных аргументов. Отсюда следует, что мы можем неявно задавать тип **Tuple** в коде, имеющим функцию для его конструирования:

```
template<typename T1, typename T2, typename T3, typename T4>
Tuple<T1, T2, T3, T4> make_tuple(const T1& t1, const T2& t2, const T3& t3, const T4& t4)
{
     return Tuple<T1, T2, T3, T4>{t1, t2, t3, t4};
}

// ... и другие четыре make_Tuples ...
```

Располагая функцией **make_tuple**, мы можем писать:

```
auto xxx = make_Tuple(1.2,3,'x',1223);
cout << "xxx: " << xxx << "\n";
```

Другие полезные функции, такие как **head()** и **tail()**, легко могут быть реализованы. В стандартном **tuple** реализованы несколько таких функций (§28.6.4).

28.6. Шаблоны с переменным числом параметров

Работа с неизвестным числом элементов является общей проблемой. Например, функция сообщения об ошибках принимает от нулевого количества аргументов до десяти аргументов, матрицам не возбраняется иметь от одного до десяти измерений и т.д. И элементы могут даже иметь разный тип в некоторых случаях. И чаще всего нам не хотелось бы порознь работать с этими случаями. И мой выбор ограничения в десять элементов — это произвольный выбор, и хорошо бы вообще отказаться от фиксированного верхнего предела.

За многие годы было придумано много решений. К примеру, умолчательные аргументы (§12.2.5) можно использовать для того, чтобы разрешить единственной функции принимать переменное число аргументов, и перегрузка функций (§12.3) используется для того, чтобы сопоставить функции тому или иному набору аргументов. Передача единственного списка элементов (§11.3) может составить альтернативу перечисленным методам за счет переменного числа элементов (одного и того же типа). Но для элегантного решения проблемы переменного числа аргументов потенциально разных типов нужна поддержка со стороны языка. Соответствующая черта языка называется *шаблонами с переменным числом параметров*.

28.6.1. Безопасная по типам функция printf()

Рассмотрим сверхтипичную функцию, которая нуждается в переменном числе аргументов разных типов: **printf()**. Как она представлена в стандартных библиотеках языка C и языка C++, эта функция гибкая и работает чудесно (§43.3). Однако, она не расширяемая на пользовательские типы и небезопасна в отношении типов вообще, так что она является популярной целью для хакеров.

Первым аргументом этой функции служит строка в C-стиле, трактуемая как строка форматирования. Остальные аргументы используются так, как того требует строка форматирования. Форматирующие спецификаторы, вроде **%g** для чисел с плавающей запятой и **%s** для нуль-терминированныхъ строк символов, контролируют интерпретацию дополнительных аргументов. Например:

```
printf("The value of %s is %g\n","x",3.14);

string name = "target";
printf("The value of %s is %P\n", name, Point{34,200});

printf("The value of %s is %g\n",7);
```

Первый вызов **printf()** работает как надо, но второй вызов имеет две проблемы: форматирующий спецификатор **%s** требует C-строку, и **printf()** не будет интерпретировать аргумент **std::string** корректным образом. Далее, формата **%P** не существует, а также нет способа работы с пользовательскими типами вроде **Point**. В случае третьего вызова я предоставил аргумент типа **int** там, где указан формат **%s**, а еще я «забыл» предоставить аргумент для спецификатора **%g**. В общем случае, компилятор не в состоянии сопоставить число и типы аргументов, требуемые форматирующей строкой, и реально предоставленные аргументы при вызове функции. Вывод последнего примера вызова функции **printf()** вряд ли устроит программиста.

Применяя шаблоны с переменным числом параметров, мы можем обеспечить расширяемую и безопасную по типам версию функции **printf()**. Что типично для программирования этапа компиляции, реализация состоит из двух частей:

1. Обработать случай с одним аргументом (форматирующей строкой).
2. Обработать случай, когда имеется по крайней мере один дополнительный аргумент, который будучи надлежащим образом форматированным, требует вывода в соответствующей точке, указуемой форматирующей строкой.

Вот простейший случай:

```
void printf(const char* s)
{
    if (s==nullptr) return;

    while (*s) {
        if (*s=='%' && *++s!='%')       // убедитесь в отсутствии ожидание других арг-тов
                                        // %% представляет простой % в формате строки
            throw runtime_error("invalid format: missing arguments");
        std::cout << *s++;
    }
}
```

Здесь печатается форматирующая строка. Если имеется спецификатор формата, то **printf()** генерирует исключение, поскольку нет аргумента для форматирования. Форматирующий спецификатор определяется по знаку **%**, за которым не следует другой знак **%** (**%%** — это нотация для **%**, не являющегося спецификатором). Отметим, что ***++s** не вызывает переполнения даже если **%** является последним символом в строке. В таком случае ***++s** ссылается на терминирующий нуль.

Далее нам нужно обработать случай с дополнительными аргументами. Именно здесь на сцену выходят шаблоны, и в особенности шаблоны с переменным числом параметров:

```
template<typename T, typename ... Args> // арг-ты шаблона с перем. числом пар-в: один и более арг-тов
void printf(const char* s, T value, Args... args) // список арг-в функции: два и более арг-тов
{
    while (s && *s) {
        if (*s=='%' && *++s!='%') {     // спецификатор формата
            std::cout << value;         // используем первый неформатный аргумент
            return printf(++s, args...); // рекурсивный вызов с хвостом списка аргум-ов
    }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

Функция **printf()** находит и печатает первый неформатирующий аргумент, «очищает его от кожуры», а затем вызывает саму себя рекурсивно. Когда заканчиваются неформатирующие аргументы, вызывается первый (простой) вариант функции **printf()**. Простые символы (то есть не спецификаторы) просто печатаются.

Перегрузка операции **<<** устраняет потенциально опасные намеки из форматирующей строки. Если аргумент имеет тип, для которого операция **<<** определена, то он печатается; в противном случае программа не запускается. Форматирующий символ после знака **%** не используется. Я могу представить себе варианты безопасного по типам использования этих символов, но в настоящий момент моей целью является не разработка совершенного дизайна, а иллюстрация применения шаблонов с переменным числом параметров.

Выражение **Args...** определяет то, что называется *пачкой параметров*. Это последовательность пар (type/value — тип/значение), из которых вы можете «вычистить» аргументы, стартуя с первого из них.

Когда **printf()** вызывается с двумя и более аргументами

```
void printf(const char* s, T value, Args... args);
```

то первый аргумент есть **s**, второй — **value**, а остальные (если имеются) упаковываются в «пачку аргументов» **args** для дальнейшего использования. При вызове **printf(++s, args...)** «пачка параметров» **args** расширяется так, что выбирается первый элемент из **args** как **value**, и **args** становится на один элемент короче, чем в предыдущем вызове. Это продолжается до тех пор, пока **args** не станет пустым, так что мы сможем вызвать

```
void printf(const char*);
```

При желании мы могли бы проверять в **printf()** форматирующие директивы, такие как%s. Например:

```
template<typename T, typename... Args> // арг-ты шаблона с перем. числом параметров: один и более арг-тов
void printf(const char* s, T value, Args... args) // список арг-ов функции: два и более арг-тов
{
    while (s && *s) {
        if (*s=='%') {   // форматный спецификатор или %%
            switch (*++s) {
            case '%': // неформатный спецификатор
                break;
            case 's':
                if (!Is_C_style_string<T>() &&! Is_string<T>())
                    throw runtime_error("Bad printf() format");
                break;
            case 'd':
                if (!Is_integral<T>()) throw runtime_error("Bad printf() format");
                break;
            case 'g':
                if (!Is_floating_point<T>()) throw runtime_error("Bad printf() format");
                break;
            }
            std::cout << value; // используем первый неформатный аргумент
            return printf(++s, args...); // рекурсивный вызов с хвостом списка аргументов
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

В стандартной библиотеке имеются **std::is_integral** и **std::is_floating_point**, но вам самим придется разработать **is_C_style_string**.

28.6.2. Технические детали

Если вы знакомы с функциональным программированием, то вы найдете пример с **printf()** (§28.6) как нетрадиционную нотацию для довольно стандартной техники. Если нет, то вот минимальное количество технических по сути примеров, которые могут помочь. Первое, мы можем объявить и использовать простую шаблонную функцию с переменным числом параметров:

```
template<typename... Types>
void f(Types... args);     // шаблонная функция с переменным числом параметров
```

То есть, **f()** является функцией, которую можно вызвать с произвольным числом аргументов произвольных типов:

```
f();       // OK: args не содержит аргументов
f(1);      // OK: args содержит один аргумент: int
```

```
f(2, 1.0);          // OK: args содержит два аргумента: int и double
f(2, 1.0, "Hello"); // OK: args содержит три аргумента: int, double, и const char*
```

Шаблон с переменным числом параметров определяется с помощью ... нотации:

```
template<typename... Types>
void f(Types... args);    // шаблонная функция с переменным числом параметров
```

Обозначение **typename...** в объявлении специфицирует **Types** как упакованный набор шаблонных параметров. Многоточие в типе для **args** специфицирует **args** как упакованный набор параметров функции. Тип каждого аргумента функции соответствует шаблонному аргументу в **Types**. Вместо **typename...** можно использовать **class...** с тем же смыслом. Так как многоточие является отдельной лексемой, то можно ставить пробел до или после него. Многоточие может появляться в разных контекстах грамматики, но всегда имеет смысл «ноль или более актов появления чего-либо». Об упакованном наборе параметров можно мыслить как о последовательности значений, тип которых компилятор запомнил. Например, можно графически проиллюстрировать упакованный набор параметров для **{'c',127, string{«Bob»},3.14}**:

char	int	string	double
'c'	127	"Bob"	3.14

При этом раскладка памяти не специфицируется стандартом C++. Например, реальная раскладка может оказаться обратной тому, что показано у нас на рисунке (последний элемент по наименьшему адресу; §28.5). Но в любом случае это плотная структура, а не связь через указатели. Чтобы добраться до значения, мы должны стартовать с самого начала и продираться вдоль структуры вплоть до того, что нам нужно. Реализация **Tuple** демонстрирует эту технику (§28.5). Можно найти тип первого элемента и реализовать доступ к нему на этой основе, затем мы можем (рекурсивно) перебраться к следующему аргументу. При желании мы можем придать индексируемому доступу вид вроде **get<N>** для **Tuple** (как и для **std::tuple**; §28.5), но прямой поддержки языка в этом вопросе нет.

Если у вас есть упакованный набор параметров, то его раскрытие в последовательность элементов отмечается путем простановки ... после этого набора. Например:

```
template<typename T, typename... Args>
void printf(const char* s, T value, Args... args)
{
    // ...
    return printf(++s, args...); // делаем рекурсивный вызов с эл-ми args как аргументами
    // ...
}
```

Раскрытие упакованного набора параметров в сторону его элементов не ограничивается лишь случаем вызова функции. Пример:

```
template<typename... Bases>
class X : public Bases... {
public:
    X(const Bases&... b) :  Bases(b)... { }
};

X<> x0;
X<Bx> x1(1);
X<Bx, By> x2(2,3);
X<Bx, By, Bz> x3(2,3,4);
```

Здесь **Bases...** означает, что **X** имеет нуль или более элементов. Когда приходит момент инициализировать **X**, конструктор запрашивает нуль или более значений типов, специфицированных в переменном числе аргументов у **Bases**. Один за одним эти значения передаются соответствующему инициализатору.

Мы можем использовать многоточие, означающее «нуль или более элементов чего-либо» в большинстве мест программы, где требуется список элементов (§ iso.14.5.3):

- Список аргументов шаблона
- Список аргументов функции
- Список инициализаторов
- Список базовых классов
- Список инициализаторов для базовых классов или членов
- Выражение с **sizeof...**

Выражение **sizeof...** используется для того, чтобы получить число элементов упакованного набора параметров. Например, мы можем определить конструктор для **tuple**, имея пару (**pair**), при условии, что число элементов в **tuple** равно двум:

```
template<typename... Types>
class tuple {
    // ...
    template<typename T, typename U, typename = Enable_if<sizeof...(Types)==2>
        tuple(const pair<T, U>>&);
};
```

28.6.3. Перенаправление

Одним из наиболее частых примеров использования шаблонов с переменным числом параметров является случай перенаправления от одной функции к другой. Рассмотрим вопрос о том, как написать функцию, которая принимает в качестве аргументов что-нибудь, что можно вызвать, и возможно пустой список аргументов, чтобы передать этому что-нибудь в виде аргументов:

```
template<typename F, typename... T>
void call(F&& f, T&&... t)
{
    f(forward<T>(t)...);
}
```

Это достаточно простой и вовсе не гипотетический пример. В стандартной библиотеке **thread** имеет конструкторы, использующие эту технику (§5.3.1, §42.2.2). Я использую передачу по rvalue-ссылке с типом выведенного аргумента шаблона, чтобы можно было корректно различать rvalue и lvalue (§23.5.2.1) и **std::forward()**, извлекающую из этого выгоду (§35.5.1). Многоточие в **T&&...** читается как «получить нуль или более **&&** аргументов, каждый типа, соответствующего **T**». Многоточие в **forward<T>(t)...** читается как «перенаправить нуль или более аргументов из **t**».

Я использовал шаблонный аргумент для типа этого «что-либо», подлежащего вызову, так что **call()** может принять функции, указатели на функции, функциональные объекты и лямбды:

```
void g0()
{
    cout << "g0()\n";
}
```

```
template<typename T>
void g1(const T& t)
{
    cout << "g1(): " << t << '\n';
}

void g1d(double t)
{
    cout << "g1d(): " << t << '\n';
}

template<typename T, typename T2>
void g2(const T& t, T2&& t2)
{
    cout << "g2(): " << t << ' ' << t2 << '\n';
}

void test()
{
    call(g0);
    call(g1);                          // ошибка: слишком мало аргументов
    call(g1<int>,1);
    call(g1<const char*>,"hello");
    call(g1<double>,1.2);
    call(g1d,1.2);
    call(g1d,"No way!");               // ошибка: неправильный тип аргумента для g1d()
    call(g1d,1.2,"I can't count");     // ошибка: слишком много аргум-ов для g1d()
    call(g2<double, string>,1,"world!");

    int i = 99;                        // тестируем с lvalues
    const char* p = "Trying";
    call(g2<double, string>, i, p);

    call([](){cout <<"l1()\n"; });
    call([](int i){cout <<"l0(): " << i << "\n";},17);
    call([i](){cout <<"l1(): " << i << "\n"; });
}
```

Я должен быть точен в вопросе о том, какую из специализаций шаблонной функции передавать, поскольку **call()** не может решить (вывести), какую именно использовать, исходя из типов остальных аргументов.

28.6.4. Шаблон tuple из стандартной библиотеки

Наш простой **Tuple** из §28.5 имеет очевидную слабость: он может работать максимум с четырьмя элементами. В настоящем разделе мы рассмотрим определение **tuple** из стандартной библиотеки (из файла **<tuple>**; §34.2.4.2), а также обсудим приемы реализации. Ключевое отличие **std::tuple** от нашего простого **Tuple** состоит в том, что первый использует шаблоны с переменным числом параметров ради устранения ограничения на число элементов. Вот соответствующие ключевые определения:

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
    : private tuple<Tail...> { // здесь рекурсия
/*
    tuple хранит head (first (type, value) pairs)
    и наследует tuple от своего tail (оставш. (type/value) пары).
    Тип закодирован в type, а не хранится как data
*/
    typedef tuple<Tail...> inherited;
```

```
public:
    constexpr tuple() { } // по умолчанию: пустой tuple

    // Конструируем tuple из отдельных аргументов:
    tuple(Add_const_reference<Head> v, Add_const_reference<Tail>... vtail)
        : m_head(v), inherited(vtail...) { }

    // Конструируем tuple из другого tuple:
    template<typename... VValues>
    tuple(const tuple<VValues...>& other)
    : m_head(other.head()), inherited(other.tail()) { }

    template<typename... VValues>
    tuple& operator=(const tuple<VValues...>& other) // присваивание
    {
        m_head = other.head();
        tail() = other.tail();
        return *this;
    }
    // ...

protected:
    Head m_head;
private:
    Add_reference<Head> head() { return m_head; }
    Add_const_reference<const Head> head() const { return m_head; }

    inherited& tail() { return *this; }
    const inherited& tail() const { return *this; }
};
```

Нет гарантии того, что **std::tuple** реализуется так, как это кажется наиболее вероятным, исходя из приведенного определения. На самом деле, имеется несколько популярных реализаций, основанных на промежуточном вспомогательном классе (также использующим шаблон с переменным числом параметров), чтобы иметь раскладку элементов в памяти такую же, как у **struct** с аналогичными типами членов.

Функции с именами, имеющими в своем составе **Add_reference**, добавляют ссылку к типу, если он не является к этому моменту ссылкой. Это используется для того, чтобы избежать копирования (§35.4.1).

Любопытно, что **std::tuple** не предоставляет функций-членов **head()** и **tail()**, так что я их обозначил как **private**. Более того, **tuple** не предоставляет никаких функций-членов для доступа к элементам, и вы должны явно (или неявно) вызывать функцию, которая расщепляет их на значение и .... Если желательно иметь функции **head()** и **tail()** для **std::tuple**, то их можно написать:

```
template<typename Head, typename... Tail>
Head head(tuple<Head, Tail...>& t)
{
    return std::get<0>(t);      // получаем первый элемент t (§ 34.2.4.2)
}

template<typename Head, typename... Tail>
tuple<T&...> tail(tuple<Head, Tail...>& t)
{
    return /* детали */;
}
```

Детали реализации функции **tail()** сложны и некрасивы. Если бы разработчики **tuple()** предполагали бы, что нам нужна функция **tail()**, то они реализовали бы ее в виде функции-члена.

Можно использовать **tuple** следующим образом:

```
tuple<string, vector, double> tt("hello",{1,2,3,4},1.2);
string h = head(tt.head);                    // "hello"
tuple<vector<int>, double> t2 = tail(tt.tail);   // {{1,2,3,4},1.2};
```

Может оказаться утомительным упоминать все эти типы. Вместо этого мы можем выводить их из типов аргументов, например, используя **make_tuple()** из стандартной библиотеки:

```
template<typename... Types>
tuple<Types...> make_tuple(Types&&... t)     // упрощено (§ iso.20.4.2.4)
{
    return tuple<Types...>(t...);
}

string s = "Hello";
vector<int> v = {1,22,3,4,5};
auto x = make_tuple(s, v,1.2);
```

У стандартного **tuple()** имеется много других функций, кроме тех, что перечислены выше. Кроме того, стандарт определяет несколько вспомогательных функций. Например, **get()** предоставляется ради доступа к элементам (вроде **get()** из §28.5.2), так что мы можем писать

```
auto t = make_tuple("Hello tuple", 43, 3.15);
double d = get<2>(t);                    // d становится 3.15
```

Таким образом, **std::get()** предоставляет индексирование **std::tuple** на этапе компиляции.

Каждый член **std::tuple** бывает полезен в каких-то случаях, а некоторые члены чрезвычайно полезны, но нам они не помогут в изучении шаблонов с переменным числом параметров, так что я не буду вдаваться в большие подробности.

Можно с сожалением констатировать, что в стандартной библиотеке не предоставляются операции >> и << для **tuple**. Что еще хуже, написать << для **std::tuple** довольно трудно, поскольку нет простого и общего способа итерирования по элементам. При этом нам потребуется вспомогательная конструкция; это **struct** с двумя функциями **printf()**. Одна из этих функций рекурсивно обходит элементы, печатая их, а другая останавливает рекурсию тогда, когда больше нет элементов для печати:

```
template<size_t N> // печатаем элемент N и последующие элементы
struct print_tuple {
    template<typename... T>
    typename enable_if<(N<sizeof...(T))>::type
    print(ostream& os, const tuple<T...>& t) const    // непустой tuple
    {
        os << ", " << get<N>(t);        // печатаем элемент
        print_tuple<N+1>()(os, t);      // печатаем оставшиеся элементы
    }

    template<typename... T>
    typename enable_if<!(N<sizeof...(T))>::type    // пустой tuple
    print(ostream&, const tuple<T...>&) const
    {
    }
};
```

Здесь терминирование рекурсии основывается на перегрузке (как в **printf()** из §28.6.1). Отметим, что очень затратно позволять **get<N>()** считать от **0** до **N**.

Вот теперь мы напишем << для **tuple**:

```
std::ostream& operator << (ostream& os, const tuple<>&)  // пустой tuple
{
    return os << "{}";
}

template<typename T0, typename ...T>
ostream& operator<<(ostream& os, const tuple<T0, T...>& t)      // непустой tuple
{
    os << '{' << std::get<0>(t);     // печатаем первый элемент
    print_tuple<1>::print(os, t);    // печатаем оставшиеся элементы
    return os << '}';
}
```

Наконец, можно написать код для печати:

```
void user()
{
    cout << make_tuple() << '\n';
    cout << make_tuple("One meatball!") << '\n';
    cout << make_tuple(1,1.2,"Tail!") << '\n';
}
```

28.7. Библиотека для работы с единицами измерений

Применяя **constexpr** и шаблоны, мы можем вычислять много чего на этапе компиляции. Несмотря на то, что предоставить входные данные для таких вычислений не слишком просто, мы всегда можем вставить данные непосредственно в текст программы посредством директивы **#include**. Я всегда предпочитаю более простые примеры, поскольку они полезнее. Здесь я покажу пример, компромиссный в плане реализации и поддержки. Накладные расходы во время компиляции минимальные, а во время выполнения — вообще никакие. В примере мы реализуем небольшую библиотеку для вычислений с единицами измерений, такими как метры, килограммы и секунды. Она является частью международного стандарта, повсеместно используемого в научных вычислениях. Пример иллюстрирует совместное использование простейшего метапрограммирования и иных возможностей языка C++.

Нам нужно ассоциировать единицы измерений с нашими значениями, чтобы избежать бессмысленных рассчетов. Пример:

```
auto distance = 10_m;          // 10 метров
auto time = 20_s;              // 20 секунд
auto speed = distance/time;    // 5 m/s (метры в секунду)

if (speed == 20)               // ошибка: 20 безразмерное
// ...
if (speed == distance)         // ошибка: не могу сравнить m с m/s
// ...
if (speed == 10_m/20_s)        // OK: единицы соответствуют
// ...
Quantity<MpS2> acceleration = distance/square(time);  // MpS2 значит m/(s*s)

cout << "speed==" << speed << " acceleration==" << acceleration << "\n";
```

Библиотека объектов типа **Unit** предоставит систему типов для физических значений. Как мы показали, можно использовать **auto** для сокрытия типов тогда, когда мы этого хо-

тим (§2.2.2), можно применять пользовательские литералы для введения типизированных значений (§19.2.6) и тип **Quantity**, когда мы хотим явно ссылаться на библиотеку **Unit**.

28.7.1. Тип Unit

Сначала определим тип **Unit**:

```
template<int M, int K, int S>
struct Unit {
    enum { m=M, kg=K, s=S };
};
```

Здесь имеются компоненты, представляющие три системы единиц измерения, которыми мы интересуемся:

- Метры для измерения длин
- Килограммы для измерения массы
- Секунды для измерения времени

Отметим, что значения для единиц измерения кодируются в самом типе. Предполагается, что **Unit** используется во время компиляции.

Можно предоставить более традиционную нотацию для самых общих единиц:

```
using M = Unit<1,0,0>;        // метры
using Kg = Unit<0,1,0>;       // килограммы
using S = Unit<0,0,1>;        // секунды
using MpS = Unit<1,0,-1>;     // метры в секунду (m/s)
using MpS2 = Unit<1,0,-2>;    // метры на секунду в квадрате (m/(s*s))
```

Отрицательные значения единиц означают деление на величину с этой единицей измерения. Такое трехзначное представление для единиц очень гибкое: можно предоставить единицы для вычислений сразу для длины (расстояния), массы и времени. **Unit<0,0,0>** индицирует безразмерную сущность, то есть величину без единиц измерения.

Полезно определить сложение для **Unit**:

```
template<typename U1, typename U2>
struct Uplus {
    using type = Unit<U1::m+U2::m, U1::kg+U2::kg, U1::s+U2::s>;
};

template<typename U1, U2>
using Unit_plus = typename Uplus<U1, U2>::type;
```

Аналогично можно опредлить вычитание:

```
template<typename U1, typename U2>
struct Uminus {
    using type = Unit<U1::m-U2::m, U1::kg-U2::kg, U1::s-U2::s>;
};

template<typename U1, U2>
using Unit_minus = typename Uminus<U1, U2>::type;
```

28.7.2. Тип Quantity

Quantity представляет значение с ассоциированным **Unit**:

```
template<typename U>
struct Quantity {
    double val;
    explicit Quantity(double d) : val{d} {}
};
```

Дальнейшее уточнение могло бы сделать тип, использованный для представления значения, параметром шаблона, возможно с умолчательным типом **double**. Можно определять **Quantity** с разнообразными единицами:

```
Quantity<M> x {10.5};    // x есть 10.5 метров
Quantity<S> y {2};       // y есть 2 секунды
```

Я сделал конструктор у **Quantity** со спецификатором **explicit**, чтобы снизить вероятность неявных конверсий из безразмерных типов вроде простых литералов с плавающей запятой:

```
Quantity<MpS> s = 7;          // ошибка: попытка конвертировать int в метры на секунду

Quantity<M> comp(Quantity<M>);
// ...
Quantity<M> n = comp(7);      // ошибка: comp() требует расстояния
```

Теперь можно начать думать о вычислениях. Что делать с физическими измерениями? Я не собираюсь цитировать целый учебник по физике, но нам очевидно понадобятся сложение, вычитание, умножение и деление. Можно складывать и вычитать значения с одинаковыми единицами измерения:

```
template<typename U>
Quantity<U> operator+(Quantity<U> x, Quantity<U> y) // та же размерность
{
    return Quantity<U>{x.val+y.val};
}

template<typename U>
Quantity<U> operator-(Quantity<U> x, Quantity<U> y) // та же размерность
{
    return Quantity<U>{x.val-y.val};
}
```

Так как у **Quantity** конструктор **explicit**, то нужно конвертировать результат типа **double** обратно в тип **Quantity**.

Умножение для **Quantity** требует сложения для **Unit**. Аналогично, деление для **Quantity** требует вычитания для **Unit**. Пример:

```
template<typename U1, typename U2>
Quantity<Unit_plus<U1, U2>> operator*(Quantity<U1> x, Quantity<U2> y)
{
    return Quantity<Unit_plus<U1, U2>>{x.val*y.val};
}

template<typename U1, typename U2>
Quantity<Unit_minus<U1, U2>> operator/(Quantity<U1> x, Quantity<U2> y)
{
    return Quantity<Unit_minus<U1, U2>>{x.val/y.val};
}
```

Исходя из представленных арифметических операций мы можем выразить большинство необходимых вычислений. Но в реальном мире потребуется еще и масштабирование с некоторым числом операций, то есть умножений и делений на безразмерные величины. Можно использовать **Quantity<Unit<0,0,0>>**, но это утомительно:

```
Quantity<MpS> speed {10};
auto double_speed = Quantity<Unit<0,0,0>>{2}*speed;
```

Чтобы устранить такую избыточность, мы можем либо предоставить неявную конверсию из **double** в **Quantity<Unit<0,0,0>>**, либо добавить пару вариантов в арифметические операции. Я выбираю последнее:

```
template<typename U>
Quantity<U> operator*(Quantity<U> x, double y)
{
    return Quantity<U>{x.val*y};
}

template<typename U>
Quantity<U> operator*(double x, Quantity<U> y)
{
    return Quantity<U>{x*y.val};
}
```

Теперь можно написать следующее:

```
Quantity<MpS> speed {10};
auto double_speed = 2*speed;
```

Главной причины того, что я не определил неявную конверсию из **double** в в **Quantity<Unit<0,0,0>>**, стало то, что я не хотел такой конверсии для сложения или вычитания:

```
Quantity<MpS> speed {10};
auto increased_speed = 2.3+speed; // ошибка: нельзя прибавить безразмерный скаляр к скорости
```

Хорошо иметь такие детальные требования к коду, диктуемые областью его применения.

28.7.3. Литералы для Unit

Благодаря синонимам типов для большинства самых общих единиц измерения мы можем написать следующее:

```
auto distance = Quantity<M>{10};      // 10 метров
auto time = Quantity<S>{20};          // 20 секунд
auto speed = distance/time;           // .5 m/s (метры в секунду)
```

Неплохо, но все еще избыточно по сравнению с кодом, который просто удаляет единицы из головы программиста:

```
auto distance = 10.0;                 // 10 метров
double time = 20;                     // 20 секунд
auto speed = distance/time;           // .5 m/s (метры в секунду)
```

Нужен **.0** или явное указание **double**, чтобы гарантировать тип **double** (и получить корректный результат для деления).

Код, генерируемый для двух примеров, должен быть идентичным. Мы можем ввести пользовательские литералы (§19.2.6) для типа **Quantity**:

```
constexpr Quantity<M> operator"" _m(double d) { return Quantity<M>{d}; }
constexpr Quantity<Kg> operator"" _kg(double d) { return Quantity<Kg>{d}; }
constexpr Quantity<S> operator"" _s(double d) { return Quantity<S>{d}; }
```

Это предоставляет нам литералы из нашего исходного примера:

```
auto distance = 10_m;                 // 10 метров
auto time = 20_s;                     // 20 секунд
auto speed = distance/time;           // .5 m/s (метры в секунду)
```

```
if (speed == 20)                    // ошибка: 20 безразмерное
// ...
if (speed == distance)              // ошибка: нельзя сравнить m с m/s
// ...
if (speed == 10_m/20_s)             // OK: единицы соответствуют
```

Я определил * и / для комбинаций **Quantity** и безразмерных величин, так что мы можем масштабировать единицы, используя умножение или деление. Однако, можно предоставить больше традиционных единиц как пользовательские литералы:

```
constexpr Quantity<M> operator"" _km(double d) { return 1000*d; }
constexpr Quantity<Kg> operator"" _g(double d) { return d/1000; }
constexpr Quantity<Kg> operator"" _mg(double d) { return d/10000000; } // милиграмм
constexpr Quantity<S> operator"" _ms(double d) { return d/1000; }      // милисекунды
constexpr Quantity<S> operator"" _us(double d) { return d/1000; }      // микросекунды
constexpr Quantity<S> operator"" _ns(double d) { return d/1000000000; } // наносекунды
// ...
```

Очевидно, что тут легко потерять контроль из-за обилия нестандартных суффиксов.

Я мог бы предоставить различные величины как иные типы (сделано для **std::ratio**; §35.3), но посчитал более простым сохранять тип **Unit** простым и сфокусированным на выполнение своей основной задачи.

28.7.4. Вспомогательные функции

Для завершения работы нам нужны вспомогательная функция **square()**, операция сравнения на равенство и операция вывода. Определение **square()** тривиально:

```
template<typename U>
Quantity<Unit_plus<U, U>> square(Quantity<U> x)
{
    return Quantity<Unit_plus<U, U>>(x.val*x.val);
}
```

Этот пример показывает в самом общем плане, как писать произвольные вычислительные функции. Я мог бы сконструировать **Unit** здесь прямо в определении возвращаемого значения, но использование существующей функции типа проще.

Операция == определяется для значений одинаковых **Unit**:

```
template<typename U>
bool operator==(Quantity<U> x, Quantity<U> y)
{
    return x.val==y.val;
}

template<typename U>
bool operator!=(Quantity<U> x, Quantity<U> y)
{
    return x.val!=y.val;
}
```

Заметьте, что я передаю **Quantity** по значению. На этапе выполнения они представляются как **double**.

Функции вывода просто выполняют работу по традиционному манипулированию символами:

```
string suffix(int u, const char* x) // вспомогательная функция
{
    string suf;
    if (u) {
```

```
            suf += x;
            if (1<u) suf += '0'+u;

            if (u<0) {
                suf += '-';
                suf += '0'-u;
            }
        }
        return suf;
}

template<typename U>
ostream& operator<<(ostream& os, Quantity<U> v)
{
        return os << v.val << suffix(U::m,"m") << suffix(U::kg,"kg") << suffix(U::s,"s");
}
```

Наконец, мы можем написать следующее:

```
auto distance = 10_m;              // 10 метров
auto time = 20_s;                  // 20 секунд
auto speed = distance/time;        // .5 m/s (метры в секунду)

if (speed == 20)                   // ошибка: 20 безразмерное
// ...
if (speed == distance)             // ошибка: нельзя сравнивать m с m/s
// ...
if (speed == 10_m/20_s)            // OK: единицы соответствуют
// ...

Quantity<MpS2> acceleration = distance/square(time);  // MpS2 значит m/(s*s)

cout << "speed==" << speed << " acceleration==" << acceleration << "\n";
```

Этот код, при наличии разумного компилятора, генерирует тот же код, что и в случае прямого использования типа **double**. Но он демонстрирует проверку типов (во время компиляции) согласно правилам для физических единиц.

28.8. Советы

[1] Используйте метапрограммирование для улучшения безопасности типов; §28.1.

[2] Используйте метапрограммирование для улучшения производительности путем переноса вычислений на этап компиляции; §28.1.

[3] Не применяйте метапрограммирование тогда, когда это приводит к серьезному замедлению компиляции; §28.1.

[4] Мыслите в понятиях вычислений на этапе компиляции и функций типов; §28.2.

[5] Используйте синонимы шаблонов как интерфейсы к функциям типов, возвращающим типы; §28.2.1.

[6] Используйте **constexpr**-функции как интерфейсы к функциям типов, возвращающим значения (не типы); §28.2.2.

[7] Используйте **_traits** из стандартной библиотеки для неинтрузивной ассоциации свойств с типами; §28.2.4.

[8] Используйте **Conditional** для выбора между двумя типами; §28.3.1.1.

[9] Используйте **Select** для выбора между несколькими альтернативными типами; §28.3.1.3.

[10] Используйте рекурсию для выражения итерации на этапе компиляции; §28.3.2.

[11] Используйте метапрограммирование для задач, которые не очень хорошо реализуются на этапе выполнения; §28.3.3.
[12] Используйте **Enable_if** для селективного объявления шаблонов функций; §28.4.
[13] Концепции являются одними из самых полезных предикатов, используемых совместно с **Enable_if**; §28.4.3.
[14] Используйте шаблоны с переменным числом параметров, когда вам нужна функция, принимающая переменное число аргументов разнообразных типов; §28.6.
[15] Не используйте шаблоны с переменным числом параметров для однородных списков аргументов (предпочитайте списки инициализаторов в таком случае); §28.6.
[16] Используйте шаблоны с переменным числом параметров и **std::move()** там, где нужно перенаправление; §28.6.3.
[17] Используйте простое метапрограммирование для реализации эффективных и элегантных систем единиц; §28.7.
[18] Используйте пользовательские литералы для упрощения использования единиц измерений; §28.7.

29

Проект для работы с матрицами

Никогда не выражайтесь яснее того,
что вы в состоянии осмыслить.
— Нильс Бор

- Введение
  Использование матриц; Требования к матрицам
- Шаблон **Matrix**
  Конструирование и присваивания; Индексирование и срезы
- Арифметические операции с **Matrix**
  Скалярные операции; Сложение; Умножение
- Реализация **Matrix**
  Matrix_slice; **Matrix_ref**; Инициализация **Matrix** списком инициализаторов; Доступ к **Matrix**; Матрицы нулевой размерности
- Решение линейных уравнений
  Классический алгоритм Гаусса; Более общее решение (выделение главного элемента); Тестирование
- Советы

29.1. Введение

Изучение языка программирования в отрыве от практических задач может быть скучным и бесполезным. В данной главе мы продемонстрируем, как язык может использоваться в комбинации с серьезной практической задачей: разработкой проекта для N-мерных матриц.

Мне никогда не встречались проекты с совершенным классом матриц. Более того, исходя из самых разных требований и способов использования матриц, я вообще сомневаюсь, что это возможно. В настоящей главе я буду разрабатывать проект и его программное воплощение для N-мерных матриц, использование которого будет проще, а код не менее компактен, чем все то, что программист может соорудить, опираясь на класс **vector** или встроенные массивы. Кроме того, все наработанное для матриц, может быть применено и в иных проектах.

29.1.1. Использование матриц

Полагаем, что **Matrix<T, N>** представляет **N**-мерную матрицу со значениями типа **T**. Ее можно использовать следующим образом:

```
Matrix<double,0> m0 {1};          // нулевые измерения: скаляр
Matrix<double,1> m1 {1,2,3,4};    // одно измерение: вектор (4 элемента)
Matrix<double,2> m2 {             // два измерения (4*3 элементов)
    {00,01,02,03},      // строка 0
    {10,11,12,13},      // строка 1
    {20,21,22,23}       // строка 2
};
Matrix<double,3> m3(4,7,9);       // 3 измерения (4*7*9 элементов), 0-инициализир.
Matrix<complex<double>,17> m17; // 17 измерений (пока без элементов)
```

Типом элемента может быть все, что мы можем хранить. Мы не будем требовать для типов элементов всех свойств, которые типичны для чисел с плавающей запятой. Например:

```
Matrix<double,2> md;    // OK
Matrix<string,2> ms;    // OK: не пробуйте арифметические операции

Matrix<Matrix<int,2>,2> mm { // 3-на-2 матрица из 2-на-2 матриц
    { // ряд 0
        {{1, 2}, {3, 4}}, // столбец 0
        {{4, 5}, {6, 7}}, // столбец 1
    },
    { // ряд 1
        {{8, 9}, {0, 1}}, // столбец 0
        {{2, 3}, {4, 5}}, // столбец 1
    },
    { // ряд 2
        {{1, 2}, {3, 4}}, // столбец 0
        {{4, 5}, {6, 7}}, // столбец 1
    }
};
```

Арифметика матриц не обладает всеми математическими особенностями, свойственными арифметике целых чисел или чисел с плавающей запятой (например, перемножение матриц некоммутативно), что следует всегда помнить, используя матрицы.

Как и в случае векторов, используем () для специфицирования размеров и {} для специфицирования значений элементов (§17.3.2.1, §17.3.4.1). Число рядов должно соответствовать числу измерений (размерности) и числу элементов в каждом измерении. Например:

```
Matrix<char,2> mc1(2,3,4);      // ошибка: слишком много размерностей
Matrix<char,2> mc2 {
    {'1','2','3'}               // ошибка: отсутствует инициализатор для второго измерений
};
Matrix<char,2> mc2 {
    {'1','2','3'},
    {'4','5'}                   // ошибка: отсутствует элемент для третьего столбца
};
```

Для **Matrix<T, N>** число измерений (**order()**) задается аргументом шаблона (здесь это N). В каждом измерении имеется число элементов (**extent()**), которое выводится из списка инициализаторов, или же специфицируется аргументом конструктора с применением ()-нотации. Полное число элементов является возвратом функции **size()**. Пример:

```
Matrix<double,1> m1(100);       // одно измерение: vector (100 эл-ов)
Matrix<double,2> m2(50,6000);   // два измерения: 50*6000 эл-ов

auto d1 = m1.order();           // 1
auto d2 = m2.order();           // 2
auto e1 = m1.extent(0);         // 100
```

```
auto e1 = m1.extent(1);          // ошибка: m1 одноразмерный

auto e2 = m2.extent(0);          // 50
auto e2 = m2.extent(1);          // 6000

auto s1 = m1.size();             // 100
auto s2 = m2.size();             // 50*6000
```

К элементам матрицы можно обращаться несколькими видами индексирования. Например:

```
Matrix<double,2> m {    // два измерения (4*3 elements)
    {00,01,02,03},  // строка 0
    {10,11,12,13},  // строка 1
    {20,21,22,23}   // строка 2
};

double d1 = m(1,2);              // d==12
double d2 = m[1][2];             // d==12
Matrix<double,1> m1 = m[1]; // строка 1: {10,11,12,13}
double d3 = m1[2];               // d==12
```

Можно определить операцию вывода (печати) для использования в целях отладки:

```
template<typename M>
    Enable_if<Matrix_type<M>(), ostream&>
operator<<(ostream& os, const M& m)
{
    os << '{';
    for (size_t i = 0; i!=rows(m); ++i) {
        os << m[i];
        if (i+1!=rows(m)) os << ',';
    }
    return os << '}';
}
```

Здесь **Matrix_type** является концепцией (§24.3). **Enable_if** является синонимом для **enable_if** (§28.4), так что **operator<<()** возвращает **ostream&**.

Исходя из представленного кода для вызова **cout<<m** получается следующий вывод: **{{0,1,2,3},{10,11,12,13},{20,21,22,23}}**.

29.1.2. Требования к матрицам

Перед тем, как приступить к реализации, рассмотрим набор свойств, которые нам могут пригодиться:

- **N** измерений, где **N** является параметром, который может варьироваться от **0** до больших величин (без специализированного кода для каждого измерения).
- **N**-мерное хранилище полезно в самом общем случае, так что типом элементов может быть что угодно, что можно хранить (как в случае элементов у **vector**).
- Математическиме операции должны применяться к любому типу, который может разумно описываться как число, включая **Matrix**.
- Индексирование в стиле языка Fortran, использующее один индекс на каждое измерение, например **m(1,2,3)** для трехмерных **Matrix**.
- Индексирование в стиле языка C, например **m[7]**, возвращающее строку.
- Желательно, чтобы индексирование было быстрым и с проверкой диапазона.

- Перемещающее присваивание и перемещающий конструктор, чтобы гарантировать эффективную передачу **Matrix** (устраняющее дорогостоящие временные объекты).
- Некоторый набор математических операций над матрицами, вроде + и *=.
- Возможность читать, писать и передавать ссылки на субматрицы, **Matrix_ref**, как для чтения элементов, так и для их записи.
- Отсутствие утечек памяти в форме базовой гарантии (§13.2).
- Критически важные операции, вроде **m*v+v2**, в виде единственного функционального вызова.

Этот список довольно длинный и достаточно амбициозный, но его нельзя назвать требованием «всего для всех». Например, я не стал требовать:

- Более обширного списка математических операций над матрицами.
- Специализированных матриц (например, диагональных или треугольных матриц).
- Поддержки разреженных матриц.
- Поддержки параллельного выполнения матричных операций.

Эти свойства, как бы полезны они не были, выходят за рамки того, что нужно для представления основ программирования.

Для реализации заявленного я использую комбинацию нескольких механизмов языка и несколько техник программирования:

- Классы (конечно же).
- Параметризацию числами и типами.
- Перемещающие конструкторы и присваивания (ради минимизации копирования).
- Опора на конструкторы и деструкторы.
- Шаблоны с переменным числом параметров (для специфицирования пределов и для индексации).
- Списки инициализаторов.
- Перегрузка операций (для поддержки традиционных нотаций).
- Функциональные объекты (для хранения информации об индексации).
- Простое метапрограммирование (для проверки списков инициализаторов и для отличения чтения от записи для **Matrix_ref**).
- Наследование реализации ради минимизации реплицирования кода.

Очевидно, что подобного рода **Matrix** мог бы быть встроенным типом (как во многих других языках), но в языке C++ такой тип не является встроенным. Вместо этого в C++ имеются все средства, чтобы можно было создать свой собственный тип такого рода.

29.2. Шаблон Matrix

Вот объявление **Matrix** с наиболее интересными операциями:

```
template<typename T, size_t N>
class Matrix {
public:
  static constexpr size_t order = N;
  using value_type = T;
  using iterator = typename std::vector<T>::iterator;
  using const_iterator = typename std::vector<T>::const_iterator;

  Matrix() = default;
  Matrix(Matrix&&) = default;                    // перемещение
  Matrix& operator=(Matrix&&) = default;
```

```
    Matrix(Matrix const&) = default;              // копирование
    Matrix& operator=(Matrix const&) = default;
    ~Matrix() = default;

    template<typename U>
      Matrix(const Matrix_ref<U, N>&);            // конструируем из Matrix_ref
    template<typename U>
      Matrix& operator=(const Matrix_ref<U, N>&); // присваивание из Matrix_ref

    template<typename...Exts>                     // специфицируем пределы (размах)
      explicit Matrix(Exts... exts);

    Matrix(Matrix_initializer<T, N>);             // инициализация из списка
    Matrix& operator=(Matrix_initializer<T, N>);  // присваивание из списка

    template<typename U>
      Matrix(initializer_list<U>) =delete;        // не используйте {} кроме как для эл-тов
    template<typename U>
      Matrix& operator=(initializer_list<U>) = delete;

    static constexpr size_t order() { return N; }          // число измерений
    size_t extent(size_t n) const { return desc.extents[n]; } // #эл-ов в n-ом измерении
    size_t size() const { return elems.size(); }           // полное число элементов
    const Matrix_slice<N>& descriptor() const { return desc; } // срез, определяющий индексиров.

    T* data() { return elems.data(); }                     // "плоский" доступ к элементу
    const T* data() const { return elems.data(); }

    // ...

  private:
    Matrix_slice<N> desc; // срез, определяющий размахи по N измерениям
    vector<T> elems;      // элементы
};
```

Применение **vector<T>** для хранения элементов избавляет нас от беспокойств за управление памятью и безопасности исключений. Здесь **Matrix_slice** хранит размеры, необходимые для доступа к элементам **N**-мерных матриц (§29.4.2). Его можно трактовать как **gslice** (§40.5.6), специализированный для **Matrix**.

Matrix_ref (§29.4.3) ведет себя как **Matrix** за исключением того, что он ссылается на **Matrix**, в типичном случае — на подматрицу вроде строки или столбца, и не владеет своими элементами. Можете его рассматривать как ссылку на подматрицу.

Matrix_initializer<T, N> — это удобно вложенный список инициализаторов для **Matrix<T, N>** (§29.4.4).

29.2.1. Конструирование и присваивания

Умолчательные копирующие и перемещающие операции имеют правильную семантику: почленное копирование или перемещение **desc** (описатель срезов, определяющий индексирование) и **elements**. Заметьте, что для управления хранилищем элементов **Matrix** получает все преимущества от **vector**. Умолчательные конструктор и деструктор также имеют правильную семантику.

Конструктор, принимающий пределы (количество элементов по измерениям), является довольно простым примером шаблона с переменным числом параметров (§28.6):

```
template<typename T, size_t N>
    template<typename... Exts>
```

```
Matrix<T, N>::Matrix(Exts... exts)
    : desc{exts...},      // копируем размахи
    elems(desc.size) // аллоцируем desc.size эл-ов и инициализируем их по умолчанию
{ }
```

Конструктор, который принимает список инициализаторов, требует совсем немного работы:

```
template<typename T, size_t N>
Matrix<T, N>::Matrix(Matrix_initializer<T, N> init)
{
  Matrix_impl::derive_extents(init, desc.extents); // выводим размахи (пределы) из инициал-ра list (§ 29.4.4)
  elems.reserve(desc.size);                // резервируем пространство для срезов
  Matrix_impl::insert_flat(init, elems);   // инициализация из списка инициализаторов (§ 29.4.4)
  assert(elems.size() == desc.size);
}
```

Здесь **Matrix_initializer** — это удобным образом вложенный **initializer_list** (§29.4.4). Пределы выводятся конструктором **Matrix_slice**, проверяются и сохраняютсмя в **desc**. Затем элементы сохраняются в **elems** функцией **insert_flat()** из пространства имен **Matrix_impl**.

Чтобы гарантировать, что {}-инициализация используется только для списков элементов, я применил **=delete** для простого конструктора с **initializer_list**. Это сделано, чтобы навязать **()**-инициализацию для пределов. Например:

```
enum class Piece { none, cross, naught };

Matrix<Piece,2> board1 {
    {Piece::none, Piece::none, Piece::none},
    {Piece::none, Piece::none, Piece::none},
    {Piece::none, Piece::none, Piece::cross}
};
Matrix<Piece,2> board2(3,3);      // OK
Matrix<Piece,2> board3 {3,3};     // ошибка: конструктор из initializer_list<int> удален
```

Без **=delete** последнее определение принималось бы.

Наконец, мы должны иметь возможность конструирования из **Matrix_ref**, то есть из ссылки на **Matrix** или на часть **Matrix** (подматрицу):

```
template<typename T, size_t N>
    template<typename U>
    Matrix<T, N>::Matrix(const Matrix_ref<U, N>& x)
        : desc{x.desc}, elems{x.begin(), x.end()}      // копируем desc и элементы
    {
        static_assert(Convertible<U, T>(),"Matrix constructor: incompatible element types");
    }
```

Применение шаблона позволяет конструировать из **Matrix** с совместимым типом элементов.

Как всегда, присваивания напоминают конструкторы. Например:

```
template<typename T, size_t N>
    template<typename U>
    Matrix<T, N>& Matrix<T, N>::operator=(const Matrix_ref<U, N>& x)
    {
        static_assert(Convertible<U, T>(),"Matrix =: incompatible element types");

        desc = x.desc;
        elems.assign(x.begin(), x.end());
        return *this;
    }
```

Таким образом мы копируем члены **Matrix**.

29.2.2. Индексирование и срезы

Обращение к **Matrix** возможно посредством индексирования (к элементам или строкам), через строки и столбцы, или посредством срезов (части строк или столбцов).

Доступ к Matrix<T, N>	
m.row(i)	Строка i из m; Matrix_ref<T, N-1>
m.column(i)	Столбец i из m; Matrix_ref<T, N-1>
m[i]	Индексирование в С-стиле: m.row(i)
m(i, j)	Доступ к эл-ту в стиле Fortran: m[i][j]; а T&; число индексов N
m(slice(i, n), slice(j))	Доступ к подматрице со срезами: Matrix_ref<T, N>; slice(i, n) есть эл-ты [i: i+n) размерности индексирования; slice(j) есть эл-ты [i: max) размерности индексирования; max размах по измерению; число индексов N

А вот все функции-члены:

```
template<typename T, size_t N>
class Matrix {
public:
    // ...

    template<typename... Args>                  // m(i, j, k) индексирование целыми
        Enable_if<Matrix_impl::Requesting_element<Args...>(), T&>
        operator()(Args... args);
    template<typename... Args>
        Enable_if<Matrix_impl::Requesting_element<Args...>(), const T&>
        operator()(Args... args) const;
    template<typename... Args>                  // m(s1, s2, s3) индексирование срезами
        Enable_if<Matrix_impl::Requesting_slice<Args...>(), Matrix_ref<T, N>>
        operator()(const Args&... args);
    template<typename... Args>
        Enable_if<Matrix_impl::Requesting_slice<Args...>(), Matrix_ref<const T, N>>
        operator()(const Args&... args) const;

    Matrix_ref<T, N-1> operator[](size_t i) { return row(i); }      // доступ к строке m[i]
    Matrix_ref<const T, N-1> operator[](size_t i) const { return row(i); }

    Matrix_ref<T, N-1> row(size_t n);                               // доступ к строке
    Matrix_ref<const T, N-1> row(size_t n) const;

    Matrix_ref<T, N-1> col(size_t n);                               // доступ к столбцу
    Matrix_ref<const T, N-1> col(size_t n) const;

    // ...
};
```

Индексирование в С-стиле выполняется посредством **m[i]**, что выделяет и возвращает i-ую строку:

```
template<typename T, size_t N>
Matrix_ref<T, N-1> Matrix<T, N>::operator[](size_t n)
{
    return row(n); // §29.4.5
}
```

Про **Matrix_ref** (§29.4.3) можно мыслить как о ссылке на подматрицу.

Matrix_ref<T,0> специализируется так, чтобы ссылаться на единственный элемент (§29.4.6).

Индексирование в стиле языка Fortran выполняется посредством перечисления индексов для каждого измерения, например **m(i, j, k)**, что дает в результате скаляр:

```
Matrix<int,2> m2 {
    {01,02,03},
    {11,12,13}
};

m(1,2) = 99;          // переписывание эл-та из строки 1 столбца; то есть 13
auto d1 = m(1);       // ошибка: слишком мало индексов
auto d2 = m(1,2,3);   // ошибка: слишком много индексов
```

В дополнение к индексированию целыми мы можем индексировать срезами. Срез описывает подмножество элементов какого-либо измерения (§40.5.4). В частности, срез **slice{i, n}** ссылается на элементы **[i, i+n)** измерения, к которому он относится. Пример:

```
Matrix<int> m2 {
    {01,02,03},
    {11,12,13},
    {21,22,23}
};

auto m22 = m(slice{1,2}, slice{0,3});
```

Теперь **m22** это **Matrix<int,2>** со значением

```
{
    {11,12,13},
    {21,22,23}
}
```

Первый (строковый) индекс — **slice{1,2}**, выбирает две последних строки, а второй индекс — **slice{0,3}**, выбирает все элементы в столбцах.

Тип возврата для **()** с индексами типа **slice** — это **Matrix_ref**, так что мы можем использовать его в качестве цели присваивания. Например:

```
m(slice{1,2}, slice{0,3}) = {
    {111,112,113},
    {121,122,123}
}
```

Теперь **m** имеет значение

```
{
    {01,02,03},
    {111,112,113},
    {121,122,123}
}
```

Выбор всех элементов, начиная с некоторого, столь популярен, что для него есть сокращенная запись: **slice{i}**, что означает **slice{i, max}**, где **max** больше, чем наибольший индекс в измерении. Таким образом, мы можем упростить запись **m(slice{1,2}, slice{0,3})** до эквивалента **m(slice{1,2}, slice{0})**.

Другим простым примером является выбор всех элементов строки или столбца, так что просто индекс **i** в наборе индексов у **slice** интерпретируется как **slice{i,1}**. Пример:

```
Matrix<int> m3 {
    {01,02,03},
    {11,12,13},
    {21,22,23}
};

auto m31 = m(slice{1,2},1);   // m31 становится {{12},{22}}
auto m32 = m(slice{1,2},0);   // m33 становится {{11},{21}}
auto x = m(1,2);              // x == 13
```

Понятие срезающих индексов распространено во всех языках, используемых для численного программирования, так что оно более-менее известно.

Реализации функций **row()**, **column()** и **operator()()** представлены в §29.4.5. Реализация константных версий этих функций примерно такая же, как и у неконстантных версий. Но константные версии возвращают результат с **const**-элементами.

29.3. Арифметические операции с Matrix

Итак, мы можем создавать объекты типа **Matrix**, копировать их, осуществлять доступ к их элементам и рядам. Но нам нужны также и математиченские операции, которые помогут нам в алгоритмах действовать не только в терминах доступа к отдельным элементам (скалярам). Например:

```
Matrix<int,2> mi {{1,2,3}, {4,5,6 }};   // 2-на-3
Matrix<int,2> m2 {mi};                  // копирование
mi*=2;                                  // масштабирование: {{2,4,6},{8,10,12}}
Matrix<int,2> m3 = mi+m2;               // сложение: {{3,6,9},{12,15,18}}
Matrix<int,2> m4 {{1,2}, {3,4}, {5,6}}; // 3-на-2
Matrix<int,1> v = mi*m4;                // умножение: {{18,24,30},{38,52,66},{58,80,102}}
```

Математические операции определяются следующим образом:

```
template<typename T, size_t N>
class Matrix {
    // ...

    template<typename F>
        Matrix& apply(F f);                     // f(x) для каждого элемента x

    template<typename M, typename F>
        Matrix& apply(const M& m, F f);         // f(x, mx) для соответствующих элементов

    Matrix& operator=(const T& value);          // присваивание скаляра

    Matrix& operator+=(const T& value);         // сложение со скаляром
    Matrix& operator-=(const T& value);         // вычитание скаляра
    Matrix& operator*=(const T& value);         // умножение на скаляр
    Matrix& operator/=(const T& value);         // деление на скаляр
    Matrix& operator%=(const T& value);         // деление на скаляр с остатком

    template<typename M>                        // сложение матриц
        Matrix& operator+=(const M& x);
    template<typename M>                        // вычитание матриц
        Matrix& operator-=(const M& x);

    // ...
};

// Бинарные +,-, * реализованы как глобальные функции (не члены)
```

29.3.1. Скалярные операции

Скалярная арифметическая операция просто применяет действие (операцию) и свой правый операнд к каждому элементу. Пример:

```
template<typename T, size_t N>
Matrix<T, N>& Matrix<T, N>::operator+=(const T& val)
{
    return apply([&](T& a) { a+=val; });          // используем лямбду (§11.4)
}
```

Функция **apply()** применяет функцию (или функциональный объект) к каждому элементу **Matrix**:

```
template<typename T, size_t N>
    template<typename F>
    Matrix<T, N>& Matrix<T, N>::apply(F f)
    {
        for (auto& x : elems) f(x);          // этот цикл использует операцию шага
        return *this;
    }
```

Как всегда, возврат ***this** обеспечивает возможность цепочек вызовов, например

```
m.apply(abs).apply(sqrt);      // m[i] = sqrt(abs(m[i])) для всех i
```

Как всегда (§3.2.1.1, §18.3) можно определить «простые операции», такие как **+**, вне класса, используя операции присваивания, такие как **+=**. Пример:

```
template<typename T, size_t N>
Matrix<T, N> operator+(const Matrix<T, N>& m, const T& val)
{
    Matrix<T, N> res = m;
    res+=val;
    return res;
}
```

Без перемещающего конструктора такой тип возврата был бы проблемой для производительности.

29.3.2. Сложение

Сложение для **Matrix** аналогично скалярным версиям:

```
template<typename T, size_t N>
    template<typename M>
    Enable_if<Matrix_type<M>(), Matrix<T, N>&> Matrix<T, N>::operator+=(const M& m)
    {
        static_assert(m.order()==N,"+=: mismatched Matrix dimensions");
        assert(same_extents(desc, m.descriptor())); // убедитесь в соответствии размеров

        return apply(m, [](T& a, Value_type<M>&b) { a+=b; });
    }
```

Matrix::apply(m, f) является двухаргументной версией **Matrix::apply(f)**. Тут **f** прилагается к двум **Matrix** (**m** и ***this**).

```
template<typename T, size_t N>
    template<typename M, typename F>
    Enable_if<Matrix_type<M>(), Matrix<T, N>&> Matrix<T, N>::apply(M& m, F f)
    {
        assert(same_extents(desc, m.descriptor())); // убедитесь в соответствии размеров
```

```
        for (auto i = begin(), j = m.begin(); i!=end(); ++i, ++j)
            f(*i,*j);
        return *this;
    }
```

Теперь **operator+()** определить легко:

```
template<typename T, size_t N>
Matrix<T, N> operator+(const Matrix<T, N>& a, const Matrix<T, N>& b)
{
    Matrix<T, N> res = a;
    res+=b;
    return res;
}
```

Здесь + определяется для двух матриц одного и того же типа, и результат имеет тот же тип. Можно сделать обобщение:

```
template<typename T, typename T2, size_t N,
    typename RT = Matrix<Common_type<Value_type<T>, Value_type<T2>>, N>
Matrix<RT, N> operator+(const Matrix<T, N>& a, const Matrix<T2, N>& b)
{
    Matrix<RT, N> res = a;
    res+=b;
    return res;
}
```

Если **T** и **T2** одного типа, то тот же тип и у **Common_type**. Тип **Common_type** наследуется от **std::common_type** (§35.4.2). Для встроенных типов принимается тип, наилучшим образом сохраняющий значения для арифметических операций. Если **Common_type** не определен для пары типов, которые мы хотим использовать в комбинации, мы можем определить его. Например:

```
template<>
struct common_type<Quad, long double> {
    using type = Quad;
};
```

Теперь **Common_type<Quad, long double>** есть **Quad**.

Нам также нужны операции с **Matrix_ref** (§29.4.3). Например:

```
template<typename T, size_t N>
Matrix<T, N> operator+(const Matrix_ref<T, N>& x, const T& n)
{
    Matrix<T, N> res = x;
    res+=n;
    return res;
}
```

Такие операции выглядят точно так же, как эквиваленые операции для **Matrix**. Нет разницы в доступе к элементам для **Matrix** и **Matrix_ref**. Разница между **Matrix** и **Matrix_ref** состоит лишь в инициализации и владении элементами.

Вычитание, перемножение и т.д. со скалярами и обращение с **Matrix_ref** просто повторяет технику, примененную для сложения.

29.3.3. Умножение

Перемножение матриц не столь просто, как сложение: результат умножения N-на-M матрицы на M-на-P матрицу будет матрицей размерности N-на-P. Для **M==1** имеем, что

результат перемножения двух векторов есть матрица. Для **P==1** мы имеем результат умножения матрицы и вектора в виде вектора. Можно обобщить умножение на большие размерности, но для этого придется ввести понятие тензора [Kolecki,2002], и мне не хотелось бы отклоняться от программирования в сторону вопросов физики и инженерии. Так что я ограничусь одномерным и двумерным случаями.

Трактуя матрицу **Matrix<T,1>** как матрицу N-на-1, а другую — как 1-на-M матрицу, получаем:

```
template<typename T>
Matrix<T,2> operator*(const Matrix<T,1>& u, const Matrix<T,1>& v)
{
    const size_t n = u.extent(0);
    const size_t m = v.extent(0);
    Matrix<T,2> res(n, m);              // матрица n-на-m
    for (size_t i = 0; i!=n; ++i)
        for (size_t j = 0; j!=m; ++j)
            res(i, j) = u[i]*v[j];
    return res;
}
```

Это простейший случай: элемент матрицы **res(i, j)** есть **u[i]*v[j]**. Я не пытаюсь обобщать на случай, когда типы элементов векторов различны. При необходимости можно использовать ту же технику, что ранее рассматривалась для сложения.

Заметьте, что запись в каждый элмент **res** выполнялась дважды: один раз для инициализации **T{}**, а другой раз для присваивания **u[i]*v[j]**. Это удваивает цену для умножения. В принципе, можно избежать этих накладных расходов. Попробуйте это сделать, если вас волнуют вопросы эффективности.

Затем мы можем перемножить N-на-M матрицу на вектор, который трактуется как M-на-1 матрица. Результатом будет N-на-1 матрица:

```
template<typename T>
Matrix<T,1> operator*(const Matrix<T,2>& m, const Matrix<T,1>& v)
{
    assert(m.extent(1)==v.extent(0));

    const size_t n = m.extent(0);
    Matrix<T,1> res(n);
    for (size_t i = 0; i!=n; ++i)
        for (size_t j = 0; j!=n; ++j)
            res(i) += m(i, j)*v(j);
    return res;
}
```

Заметьте, что объявление **res** инициализирует его элементы в **T{}**, что является нулем для числовых типов, так что операция **+=** стартует с нуля.

Обработка умножения N-на-M матрицы на M-на-P матрицу выполняется аналогично:

```
template<typename T>
Matrix<T,2> operator*(const Matrix<T,2>& m1, const Matrix<T,2>& m2)
{
    const size_t n = m1.extent(0);
    const size_t m = m1.extent(1);
    assert(m==m2.extent(0));        // столбцы должны соответствовать строкам

    const size_t p = m2.extent(1);
    Matrix<T,2> res(n, p);
    for (size_t i = 0; i!=n; ++i)
```

```
            for (size_t j = 0; j!=m; ++j)
                for (size_t k = 0; k!=p; ++k)
                    res(i, j) = m1(i, k)*m2(k, j);
        return res;
    }
```

Имеется масса возможностей для оптимизации этой важной операции.

Самый внутренний цикл можно реализовать более элегантно:

```
res(i, j) = dot_product(m1[i], m2.column(j))
```

Здесь **dot_product()** это просто интерфейс к функции **inner_product()** (§40.6.2) из стандартной библиотеки:

```
template<typename T>
T dot_product(const Matrix_ref<T,1>& a, const Matrix_ref<T,1>& b)
{
    return inner_product(a.begin(), a.end(), b.begin(),0.0);
}
```

29.4. Реализация Matrix

До сих пор я откладывал представление наиболее сложных «механических» частей реализации типа **Matrix**. Например: что есть **Matrix_ref**? И что есть **Matrix_slice**? Как инициализируется **Matrix** из списков типа **initializer_list** и при этом получаются разумные измерения матрицы? Как гарантировать, что **Matrix** не сможет быть инициализирован неподходящим типом элементов?

Простейшим способом представления этого кода будет помещение его целиком в заголовочный файл. В этом случае следует добавить **inline** к каждому определению функции, не являющейся функцией-членом.

Определения функций, которые не являются членами **Matrix**, **Matrix_ref**, **Matrix_slice** или частью общего интерфейса, помещаются в пространство имен **Matrix_impl**.

29.4.1. slice()

Простейший тип **slice**, как он используется при индексировании, описывает соответствие между целым (индексом) и местоположением элемента в терминах трех значений:

```
struct slice {
    slice(): start(-1), length(-1), stride(1) { }
    explicit slice(size_t s): start(s), length(-1), stride(1) { }
    slice(size_t s, size_t l, size_t n = 1): start(s), length(l), stride(n) { }

    size_t operator()(size_t i) const { return start+i*stride; }

    static slice all;

    size_t start;    // первый индекс
    size_t length;   // число индексов (может использоваться для проверки диапазона)
    size_t stride;   // расстояние между элементами в полседовательности
};
```

Это стандартная версия **slice**; смотри §40.5.4 для более подробного рассмотрения. Данная версия предоставляет традиционную нотацию (например, умолчательные значения, предоставляемые конструкторами).

29.4.2. Matrix_slice

Matrix_slice является частью реализации **Matrix**, которая сопоставляет набор индексов местоположению элемента. Используется общая идея обобщенных срезов (§40.5.6):

```
template<size_t N>
struct Matrix_slice {
  Matrix_slice() = default;      // пустая матрица: нет элементов

  Matrix_slice(size_t s, initializer_list<size_t> exts); // размахи
  Matrix_slice(size_t s, initializer_list<size_t> exts, initializer_list<size_t> strs); // размахи и шаги

  template<typename...Dims>      // N размахов
    Matrix_slice(Dims... dims);

  template<typename... Dims,
        typename = Enable_if<All(Conver tible<Dims, size_t>()...)>>
    size_t operator()(Dims... dims) const; // вычисляем индекс из набора нижних номеров

  size_t size;                   // полное число элементов
  size_t start;                  // начальное смещение
  array<size_t, N> extents;      // число элементов по каждому измерению
  array<size_t, N> strides;      // смещение между элементами по каждому измерению
};
```

Другими словами, **Matrix_slice** описывает то, что рассматривается как строки и столбцы в области памяти. Как обычно для раскладки матрицы, элементы строк располагаются непрерывно (смежным образом), а элементы столбцов отделяются фиксированным числом элементов (stride — шаг строки). **Matrix_slice** является функциональным объектом, и его **operator()()** выполняет вычисление шага строки (§40.5.6):

```
template<size_t N>
    template<typename... Dims>
    size_t Matrix_slice<N>::operator()(Dims... dims) const
    {
        static_assert(sizeof...(Dims) == N, "");

        size_t args[N] { size_t(dims)... };  // копируем аргументы в массив

        return inner_product(args, args+N, strides.begin(), size_t(0));
}
```

Индексирование должно быть эффективным. У нас представлен упрощенный алгоритм, который следует оптимизировать. Можно использовать специализацию для устранения упрощенного копирования индексов из упакованных параметров шаблона с переменным числом параметров. Например:

```
template<>
struct Matrix_slice<1> {

    // ...

    size_t operator()(size_t i) const
    {
        return i;
    }
}

template<>
struct Matrix_slice<2> {
```

```
    // ...

    size_t operator()(size_t i, size_t j) const
    {
        return i*stides[0]+j;
    }
};
```

Роль **Matrix_slice** важна для определения формы **Matrix** (размеров) и для реализации N-мерного индексирования. К тому же, имеется польза и для определения подматриц.

29.4.3. Matrix_ref

Matrix_ref является клоном класса **Matrix**, используемым для представления подматриц. При этом **Matrix_ref** не владеет своими элементами. Он конструируется из **Matrix_slice** и указателя на элементы:

```
template<typename T, size_t N>
class Matrix_ref {
public:
    Matrix_ref(const Matrix_slice<N>& s, T* p): desc{s}, ptr{p} {}
    // ... остальное подобно Matrix ...

private:
    Matrix_slice<N> desc;    // форма матрицы
    T*ptr;                   // первый элемент матрицы
};
```

Matrix_ref просто указывает на элементы «своей» **Matrix**. Очевидно, что **Matrix_ref** не должно покидать свой **Matrix**. Пример:

```
Matrix_ref<double,1> user()
{
    Matrix<double,2> m = {{1,2}, {3,4}, {5,6}};
    return m.row(1);
}

auto mr = user();   // беда
```

Значительное сходство между **Matrix** и **Matrix_ref** приводит к дублированию. Если это составляет проблему, то можно наследовать их от общего базового класса:

```
template<typename T, size_t N>
class Matrix_base {
    // ... общий материал ...
};

template<typename T, size_t N>
class Matrix : public Matrix_base<T, N> {
    // ... что-то специфическое для Matrix ...
private:
    Matrix_slice<N> desc;    // форма матрицы
    vector<T> elements;
};

template<typename T, size_t N>
class Matrix_ref : public Matrix_base<T, N> {
    // ... что-то специфическое для Matrix_ref ...
private:
    Matrix_slice<N> desc;    // форма матрицы
    T* ptr;
};
```

29.4.4. Инициализация Matrix списком инициализаторов

У конструктора **Matrix**, который конструирует из **initializer_list**, в качестве типа аргумента используется синоним **Matrix_initializer**:

```
template<typename T, size_t N>
using Matrix_initializer = typename Matrix_impl::Matrix_init<T, N>::type;
```

Matrix_init описывает структуру вложенных **initializer_list**.

```
template<typename T, size_t N>
struct Matrix_init {
    using type = initializer_list<typename Matrix_init<T, N-1>::type>;
};
```

Случай **N==1** является особым. Именно здесь мы подбираемся к наиболее вложенному **initializer_list<T>**:

```
template<typename T>
struct Matrix_init<T,1> {
    using type = initializer_list<T>;
};
```

Во избежание сюрпризов мы определяем **N=0** как ошибку:

```
template<typename T>
struct Matrix_init<T,0>;   // намеренно не определено
```

Теперь мы можем завершить конструктор для **Matrix**, который принимает **Matrix_initializer**:

```
template<typename T, size_t N>
Matrix<T, N>::Matrix(Matrix_initializer<T, N> init)
{
  Matrix_impl::derive_extents(init, desc.extents); // выводим размахи из списка инициал-ов (§29.4.4)
  elems.reserve(desc.size);                        // резервируем пространство для срезов
  Matrix_impl::insert_flat(init, elems);           // инициализировать из списка инициал-ов (§29.4.4)
  assert(elems.size() == desc.size);
}
```

Для этого нам нужны были две операции, которые выполняют рекурсию вниз по дереву списков типа **initializer_list** для **Matrix<T, N>**:

- **derive_extents()** определяет форму **Matrix**:
  - Проверяет, что дерево действительно имеет глубину **N**
  - Проверяет, что каждая строка имеет одно и то же число элементов
  - Устанавливает протяженность по каждой строке
- **insert_flat()** копирует элементы дерева **initializer_list<T>** в член **elems** матрицы.

Функция **derived_extents()**, которая вызывается из конструктора **Matrix** для инициализации **desc**, выглядит следующим образом:

```
template<size_t N, typename List>
array<size_t, N> derive_extents(const List& list)
{
    array<size_t, N> a;
    auto f = a.begin();
    add_extents<N>(f, list);  // помещаем размахи из списка в f[]
    return a;
}
```

Вы предоставляете этой функции **initializer_list** и в результате получаете вектор (**array**) протяженностей по измерениям.

Рекурсия выполняется от **N** до **1**, где **initializer_list** есть **initializer_list<T>**.

```
template<size_t N, typename I, typename List>
Enable_if<(N>1), void> add_extents(I& first, const List& list)
{
    assert(check_non_jagged(list));
    *first = list.size();
    add_extents<N-1>(++first,*list.begin());
}

template<size_t N, typename I, typename List>
Enable_if<(N==1), void> add_extents(I& first, const List& list)
{
    *first++ = list.size();        // достигли самого глубокого уровня вложения
}
```

Функция **check_non_jagged()** проверяет, что все строки имеют одинаковое число элементов:

```
template<typename List>
bool check_non_jagged(const List& list)
{
    auto i = list.begin();
    for (auto j = i+1; j!=list.end(); ++j)
        if (i->size()!=j->size())
            return false;
    return true;
}
```

Нам нужна **insert_flat()**, чтобы принимать потенциально вложенный список инициализаторов и предоставлять его элементы для **Matrix<T>** как **vector<T>**. Эта функция берет **initializer_list**, который дается **Matrix** как **Matrix_initializer**, и имеет **elements** в качестве цели:

```
template<typename T, typename Vec>
void insert_flat(initializer_list<T> list, Vec& vec)
{
    add_list(list.begin(), list.end(), vec);
}
```

К сожалению, мы не можем положиться на то, что элементы располагаются в памяти смежным образом, и поэтому мы должны строить вектор посредством набора рекурсивных вызовов. Если у нас есть набор **initializer_list**, то мы рекурсивно работаем с каждым из них:

```
template<typename T, typename Vec>        // вложенные initializer_lists
void add_list(const initializer_list<T>* first, const initializer_list<T>* last, Vec& vec)
{
    for (; first!=last;++first)
        add_list(first->begin(), first->end(), vec);
}
```

Когда мы достигаем списка с элементами, не входящимив **initializer_list**, то мы внедряем эти элементы в наш вектор (**vector**):

```
template<typename T, typename Vec>
void add_list(const T* first, const T* last, Vec& vec)
{
    vec.insert(vec.end(), first, last);
}
```

Я использую **vec.insert(vec.end(), first, last)**, потому что здесь нет функции **push_back()**, принимающей последовательность аргументов.

29.4.5. Доступ к Matrix

Matrix предоставляет доступ к строке, столбцу, срезу (§29.4.1) и к элементу (§29.4.3). Функции **row()** или **column()** возвращают **Matrix_ref<T, N-1>**, операция **()** с целыми индексами возвращает **T&**, и операция **()** со **slice**-аргументами возвращает **Matrix<T, N>**.

Строка в **Matrix<T, N>** есть **Matrix_ref<T, N-1>**, пока **1<N**:

```
template<typename T, size_t N>
Matrix_ref<T, N-1> Matrix<T, N>::row(size_t n)
{
    assert(n<rows());
    Matrix_slice<N-1> row;
    Matrix_impl::slice_dim<0>(n, desc, row);
    return {row, data()};
}
```

Нам нужны специализации для **N==1** и **N==0**:

```
template<typename T>
T& Matrix<T,1>::row(size_t i)
{
    return &elems[i];
}

template<typename T>
T& Matrix<T,0>::row(size_t n) = delete;
```

Вызов **column()** по существу такой же, как и вызов **row()**. Разница состоит лишь в конструкции **Matrix_slice**:

```
template<typename T, size_t N>
Matrix_ref<T, N-1> Matrix<T, N>::column(size_t n)
{
    assert(n<cols());
    Matrix_slice<N-1> col;
    Matrix_impl::slice_dim<1>(n, desc, col);
    return {col, data()};
}
```

Константные версии эквивалентны.

Requesting_element() и **Requesting_slice()** являются концепциями, используемыми для индексирования набором целых чисел и индексирования с помощью slice, соответственно (§29.4.5). Они проверяют последовательность аргументов функций доступа на предмет, имеют ли они подходящие типы.

Индексирование целыми определяется следующим образом:

```
template<typename T, size_t N>    // индексирование целыми
    template<typename... Args>
    Enable_if<Matrix_impl::Requesting_element<Args...>(), T&>
    Matrix<T, N>::operator()(Args... args)
    {
        assert(Matrix_impl::check_bounds(desc, args...));
        return *(data() + desc(args...));
    }
```

Предикат **check_bounds()** проверяет то, что число индексов равно числу измерений и что индексы находятся в корректных пределах:

```
template<size_t N, typename... Dims>
bool check_bounds(const Matrix_slice<N>& slice, Dims... dims)
{
    size_t indexes[N] {size_t(dims)...};
    return equal(indexes, indexes+N, slice.extents, less<size_t> {});
}
```

Действительное расположение элемента в **Matrix** определяется посредством активизации у **Matrix_slice** обобщенных вычислений, представленных как функциональный объект: **desc(args...)**. Результат этих вычислений нужно добавить к началу данных (**data**) и мы получаем искомое расположение:

```
return *(data() + desc(args...));
```

Спецификация типа возврата у **operator()()** выглядит следующим образом:

```
Enable_if<Matrix_impl::Requesting_element<Args...>(), T&>
```

Итак, тип возврата есть **T&**, при условии, что

```
Matrix_impl::Requesting_element<Args...>()
```

есть **true** (§28.4). Этот предикат проверяет, может ли быть конвертирован каждый индекс к требуемому **size_t**, используя предикат **is_convertible** (§35.4.1) из стандартной библиотеки:

```
template<typename... Args>
constexpr bool Requesting_element()
{
    return All(Convertible<Args, size_t>()...);
}
```

All() просто применяет предикат к каждому элементу шаблона с переменным числом параметров:

```
constexpr bool All() { return true; }

template<typename... Args>
constexpr bool All(bool b, Args... args)
{
    return b && All(args...);
}
```

Причина для использования предиката (**Requesting_element**) и **Enable_if()**, спрятанная внутри **Request**, состоит в том, чтобы выбрать между элементом и операциями **slice**-индексирования. Предикат, который используется **slice**-версией, выглядит так:

```
template<typename... Args>
constexpr bool Requesting_slice()
{
    return All((Convertible<Argssize_t>() || Same<Args, slice>())...)
        && Some(Same<Args, slice>()...);
}
```

То есть, если имеется хотя бы один **slice**-аргумент и если все аргументы конвертируются в **slice** или **size_t**, то мы располагаем тем, что можно использовать для описания **Matrix<T, N>**:

```
template<typename T, size_t N>    // индексирование срезами
    template<typename... Args>
        Enable_if<Matrix_impl::Requesting_slice<Args...>(), Matrix_ref<T, N>>
```

```
Matrix<T, N>::operator()(const Args&... args)
{
    matrix_slice<N> d;
    d.start = matrix_impl::do_slice(desc, d, args...);
    return {d, data()};
}
```

Все **slice**, представленные как протяженности строк или как шаги строк в **Matrix_slice**, и используемые для **slice**-индексирования, вычисляется следующим образом:

```
template<size_t N, typename T, typename... Args>
size_t do_slice(const Matrix_slice<N>& os, Matrix_slice<N>& ns, const T& s, const
Args&... args)
{
    size_t m = do_slice_dim<sizeof...(Args)+1>(os, ns, s);
    size_t n = do_slice(os, ns, args...);
    return m+n;
}
```

Как всегда, рекурсия терминируется простой функцией:

```
template<size_t N>
size_t do_slice(const Matrix_slice<N>& os, Matrix_slice<N>& ns)
{
    return 0;
}
```

Функция **do_slice_dim()** довольно непростая, но она не иллюстрирует никаких новых техник программирования.

29.4.6. Матрицы нулевой размерности

Код **Matrix** содержит много мест с **N-1**, где **N** является числом измерений. Таким образом, **N==0** становится неприятным моментом (как для программирования, так и для математики). Мы решаем эту проблему, вводя специализацию:

```
template<typename T>
class Matrix<T,0> {
public:
    static constexpr size_t order = 0;
    using value_type = T;

    Matrix(const T& x) : elem(x) { }
    Matrix& operator=(const T& value) { elem = value; return *this; }

    T& operator()() { return elem; }
    const T& operator()() const { return elem; }

    operator T&() { return elem; }
    operator const T&() { return elem; }
private:
    T elem;
};
```

Matrix<T,0> реально не является матрицей. Здесь просто хранится единственный элемент типа **T** и имеется возможность конвертирования в ссылку на этот тип.

29.5. Решение линейных уравнений

Код для численных расчетов имеет смысл только в том случае, если вы понимаете решаемую проблему, а также математические основания, на которых строится решение, а в противном случае все это превращается просто в головную боль. Рассматриваемый ниже пример тривиален в том случае, если вы изучали основы линейной алгебры.

Выбранный пример демонстрирует разумно реалистичный и важный способ использования типа **Matrix**. Мы будем решать набор линейных уравнений следующего вида:

$$a_{1,1}x_1 + \cdots + a_{1,n}x_n = b_1$$
$$\cdots$$
$$a_{n,1}x_1 + \cdots + a_{n,n}x_n = b_n$$

Здесь разные **x** обозначают **n** неизвестных; в то же время все **a** и **b** являются заданными константами. Предположим, что они представимы числами с плавающей запятой. Надо найти значения неизвестных, при которых удовлетворяются все **n** уравнений. Эти уравнения можно записать в компактной форме в терминах матрицы и двух векторов:

Ax =b

Здесь **A** это квадрантная **n**-на-**n** матрица, определяемая коэффициентами:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,n} \\ \cdots & \cdots \\ a_{n,1} & a_{n,n} \end{bmatrix}$$

Векторы **x** и **b** являются векторами неизвестных и констант, соответственно:

$$x = \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix}, \text{ и } b = \begin{bmatrix} b_1 \\ \cdots \\ b_n \end{bmatrix}$$

Система уравнений может иметь нуль, одно или бесконечное число решений в зависимости от коэффициентов в матрице **A** и от вектора **b**. Существуют разные методы решения линейных уравнений. Мы будем использовать классическую схему, называемую Гауссовой [Freeman,1992], [Stewart,1998], [Wood,1999]. Сначала мы трансформируем **A** и **b** так, чтобы **A** стала верхне-треугольной матрицей. Под верхне-треугольной мы понимаем матрицу, все коэффициенты которой ниже диагонали равны нулю. Другими словами, система выглядит так:

$$\begin{bmatrix} a_{1,1} & & a_{1,n} \\ 0 & \cdots & \cdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \cdots \\ b_n \end{bmatrix}$$

Это сделать нетрудно. Нуль для **a(i, j)** получается умножением уравнения для строки **i** на константу так, что величина **a(i, j)** становится равной значению элемента матрицы в столбце **j**, скажем **a(k, j)**. После этого мы вычитаем два уравнения и получаем **a(i, j)==0** а другие значения в строке **i** изменяются соответствующим образом.

Если при этом получается, что все диагональные элементы не равны нулю, то система имеет единственное решение, которое находится обратной подстановкой. Последнее уравнение легко решается:

$$a_{n,n}x_n = b_n$$

Очевидно, что **x[n]** это **b[n]/a(n, n)**. После этого устраняем ряд **n** из системы уравнений и приступаем к нахождению величины **x[n-1]**, и так далее, пока не будет найдено значение **x[1]**. Для каждого **n** мы делим на **a(n, n)**, чтобы диагональные элементы были ненулевыми. Если это не получается, то обратная подстановка не проходит, и система имеет нуль или бесконечное число решений.

29.5.1. Классический алгоритм Гаусса

Теперь рассмотрим код C++ для выражения рассмотренной схемы решения. Первое, мы упрощаем нотацию для двух типов **Matrix**, которые мы намереваемся использовать:

```
using Mat2d = Matrix<double,2>;
using Vec = Matrix<double,1>;
```

Затем выразим наши вычисления следующим образом:

```
Vec classical_gaussian_elimination(Mat2d A, Vec b)
{
    classical_elimination(A, b);
    return back_substitution(A, b);
}
```

Мы делаем копии входной информации, то есть **A** и **b**, вызываем функцию для решения системы уравнений и возврата результата путем обратной подстановки. Для завершения решения задачи нам нужно реализовать функции **classical_elimination()** и **back_substitution()**. Вот решение:

```
void classical_elimination(Mat2d& A, Vec& b)
{
    const size_t n = A.dim1();

    // траверс с 1-го столбца-до-следующего за последним, заполняя нулями эл-ты ниже диагонали:
    for (size_t j = 0; j!=n-1; ++j) {
        const double pivot = A(j, j);
        if (pivot==0) throw Elim_failure(j);
        // зануляя элементы ниже диагонали в i-ой строке:
        for (size_t i = j+1; i!=n; ++i) {
            const double mult = A(i, j) / pivot;
            A[i](slice(j)) = scale_and_add(A[j](slice(j)), -mult, A[i](slice(j)));
            b(i) -= mult*b(j); // с соответ. изменениями в b
        }
    }
}
```

Здесь **pivot** это диагональный элемент из текущей строки, с которой мы имеем дело. Он должен быть ненулевым, потому что на него мы будем делить далее; если он нуль, то мы генерируем исключение:

```
Vec back_substitution(const Mat2d& A, const Vec& b)
{
    const size_t n = A.dim1();
    Vec x(n);
```

```
    for (size_t i = n-1; i>=0; --i) {
        double s = b(i)-dot_product(A[i](slice(i+1)), x(slice(i+1)));
        if (double m = A(i, i))
            x(i) = s/m;
        else
            throw Back_subst_failure(i);
    }
    return x;
}
```

29.5.2. Более общее решение (выделение главного элемента)

Можно избежать проблему деления на нуль и получить более общее решение, пересортируя строки для получения нулей и устранения малых величин с диагонали матрицы. Более общее решение означает, что оно менее чувствительно к погрешностям (неизбежным) вычислений. Так как величины меняются по мере того, как мы помещаем нули под диагональю, то нам снова приходится все переупорядочивать ради устранения малых величин из диагонали:

```
void elim_with_partial_pivot(Mat2d& A, Vec& b)
{
    const size_t n = A.dim1();

    for (size_t j = 0; j!=n; ++j) {
        size_t pivot_row = j;

    // ищем подходящую ось:
        for (size_t k = j+1; k!=n; ++k)
            if (abs(A(k, j)) > abs(A(pivot_row, j)))
                pivot_row = k;

        // обмениваемся строками, если находится более подходящая ось:
        if (pivot_row!=j) {
            A.swap_rows(j, pivot_row);
            std::swap(b(j), b(pivot_row));
        }

        // удаление:
        for (size_t i = j+1; i!=n; ++i) {
            const double pivot = A(j, j);
            if (pivot==0) error("can't solve: pivot==0");
            const double mult = A(i, j)/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult*b(j);
        }
    }
}
```

Мы используем **swap_rows()** и **scale_and_multiply()**, чтобы сделать код более традиционным и чтобы избежать написания явного цикла.

29.5.3. Тестирование

Естественно, что предложенный нами код следует простестировать. Вот несложный вариант тестирования:

```
void solve_random_system(size_t n)
{
    Mat2d A = random_matrix(n);      // генерируем случайную Mat2d
    Vec b = random_vector(n);        // генерируем случайный Vec
```

```
        cout << "A = " << A << endl;
        cout << "b = " << b << endl;

        try {
            Vec x = classical_gaussian_elimination(A, b);
            cout << "classical elim solution is x = " << x << endl;
                Vec v = A * x;
            cout << " A * x = " << v << endl;
        }
        catch(const exception& e) {
        cerr << e.what() << endl;
        }
}
```

В **catch**-ветвь можно попасть тремя способами:

- Ошибкой в коде (оптимисты всегда считают, что ошибок нет)
- Входными данными, на которых спотыкается **classical_elimination()** (применение **elim_with_partial_pivot()** минимизирует вероятность этого)
- Из-за ошибок округления

Наш тест, однако, не столь реалистичный, как нам хотелось бы, поскольку истинно случайные матрицы чаще всего не вызывают проблем с **classical_elimination()**.

Для проверки решения мы печатаем **A*x**, что должно совпадать с **b** (или иметь близкие величины с учетом ошибок округления). Вероятность ошибок округления служит причиной того, что мы не только выполняем следующий вывод:

```
if (A*x!=b) error("substitution failed");
```

В общем случае использование операций сравнения == и != к результату вычислений с плавающей запятой лучше всего избегать. Если все же нужны проверки машинной точноти, то я бы определил функцию **equal()**, которая определяла бы степень приемлемых отклонений, после чего можно осуществить вывод:

```
if (equal(A*x, b)) error("substitution failed");
```

Использование **random_matrix()** и **random_vector()** демонстрируют простейший пример использования случайных чисел и оставлены читателям в качестве несложного упражнения.

29.5.4. Оптимизация кода

Кроме предоставления эффективных базовых операций общий класс матриц должен справляться с тремя проблемами, имеющими отношение к производительности:

1. Минимизация использования временных объектов.
2. Минимизация копирования матриц.
3. Множественные циклы около одних и тех же данных в составных операциях должны минимизироваться.

Рассмотрим выражение **U=M*V+W**, где **U**, **V** и **W** есть векторы (**Matrix<T,1>**), и **M** есть **Matrix<T,2>**. Наивная реализация создаст временный объект для **M*V** и **M*V+W**, и скопирует вычисленные значения **M*V** и **M*V+W**. Умная реализация вызовет функцию **mul_add_and_assign(&U,&M,&V,&W)**, которая не вводит временных объектов, не копирует векторы и обращается к каждому элементу матриц минимально необходимое число раз.

Помогают перемещающие конструкторы: временный объект для **M*V** используется для **(M*V)+W**. Если написать

```
Matrix<double,1> U=M*V+W;
```

то будут устранены копирования элементов: элементы, размещенные в локальных переменных у **M*V**, те же самые, что окажутся в **U**.

Это решает задачу устранения множественных циклов. Такая степень оптимизации редко когда бывает действительно нужна; простое решение проблемы эффективности состоит в предоставлении функции **mul_add_and_assign()**, и пусть пользователь вызывает ее, когда ему нужно. Можно, в принципе, еще многое потребовать от класса матриц. Например, можно было бы трактовать **U=M*V+W** как единую операцию с четырьмя операндами. Такая техника демонстрируется для манипуляторов класса **ostream** (§38.4.5.2). Для некоторых компьютерных систем движение в этом направлении позволяет выполнить тридцатикратное ускорение. Для простоты, давайте ограничим себя двумерными матрицами чисел с плавающей запятой двойной точности:

```
using Mat2d = Matrix<double,2>;
using Vec = Matrix<double,1>;
```

Определяем результат перемножения **Mat2d** на **Vec**:

```
struct MVmul {
    const Mat2d& m;
    const Vec& v;

    MVmul(const Mat2d& mm, const Vec &vv): m{mm}, v{vv} { }

    operator Vec(); // вычислить и вернуть результат
};

inline MVmul operator*(const Mat2d& mm, const Vec& vv)
{
    return MVmul(mm, vv);
}
```

Таким умножением следует заменить ее форму из §29.3, и оно не делает ничего, кроме хранения ссылок на свои операнды; вычисление **M*V** откладывается. Объект, производимый операцией * близок к тому, что часто называют замыканием (closure) во многих технических областях. Аналогично, нам нужно справляться с тем, что произойдет при сложении с **Vec**:

```
struct MVmulVadd {
    const Mat2d& m;
    const Vec& v;
    const Vec& v2;

    MVmulVadd(const MVmul& mv, const Vec& vv): m(mv.m), v(mv.v), v2(vv) { }

    operator Vec(); // вычисление и возврат результата
};

inline MVmulVadd operator+(const MVmul& mv, const Vec& vv)
{
    return MVmulVadd(mv, vv);
}
```

Это откладывает «на потом» вычисление **M*V+W**. Нужно еще удостовериться, что в конце концов все вычисляется с помощью хорошего алгоритма, когда идет присваивание **Vec**:

```
template<>
class Matrix<double,1> {      // специализация (для данного примера)
    // ...
```

```
public:
    Matrix(const MVmulVadd& m) // инициализация результатом m
    {
        // аллоцируем элементы, и т. п.
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
    }

    Matrix& operator=(const MVmulVadd& m) // приписываем результат m к *this
    {
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
        return *this;
    }
    // ...
};
```

Теперь **U=M*V+W** автоматически раскрывается в

```
U.operator=(MVmulVadd(MVmul(M, V), W))
```

что в результате встраивания разрешается к простому вызову

```
mul_add_and_assign(&U,&M,&V,&W)
```

Ясно, что при этом устраняются все временные объекты и копирование. Даже если мы напишем **mul_add_and_assign()** без ультраоптимизационных методик, то все равно остается множество возможностей для оптимизации.

Рассмотренная методика базируется на идее использования анализа этапа компиляции и замыкающих объектов для переноса вычислений субвыражений в объект, представляющий собой композитную операцию. Методика может быть перенесена на множество проблем, в которых некоторое число отдельных кусков информации должны быть собраны в одну функцию перед тем, как вычисление будет выполняться. Я ссылаюсь на такие объекты, как на композиционные замыкающие объекты.

Если такая техника применяется для откладывания выполнения всех операций, то ее именуют как шаблон выражения [Vandevoorde,2002], [Veldhuizen,1995]. Шаблоны выражений систематически используют функциональные объекты для представления выражений как абстрактных синтаксических деревьев.

29.6. Советы

[1] Рассмотрите разные варианты использования матриц; §29.1.1.

[2] Всегда определяйте операции ввода и вывода для упрощения проведения простых тестов; §29.1.1.

[3] Тщательно перечислите свойства программы, класса, библиотеки, которые должны быть в идеале; §29.1.2.

[4] Перечислите свойства программы, класса, библиотеки, которые находятся вне контуров текущего проекта; §29.1.2.

[5] Проектируя дизайн шаблона контейнера, тщательно рассмотрите требования к типу элементов; §29.1.2.

[6] Подумайте, как ваш проект может использовать проверки на этапе выполнения (например, для отладки); §29.1.2.

[7] По возможности проектируйте класс с учетом мнемонических особенностей профессиональной нотации и семантики; §29.1.2.

[8] Удостоверьтесь, что ваш проект не допускает утечек ресурсов; §29.2.

[9] Рассмотрите вопрос о конструировании и копировании класса; §29.1.1.

[10] Предоставляйте полный, гибкий, эффективный и семантически осмысленный доступ к элементам; §29.2.2, §29.3.

[11] Помещайте детали реализации в их собственное пространство имен _impl; §29.4.

[12] Реализуйте общие операции, не требующие прямого доступа к представлению класса, в виде вспомогательных функций; §29.3.2, §29.3.3.

[13] Для быстрого доступа храните данные компактно и используйте объекты доступа для предоставления нетривиальных операций доступа; §29.4.1, §29.4.2, §29.4.3.

[14] Структура данных часто может отражаться вложенными списками инициализаторов; §29.4.4.

[15] Имея дело с числами, не забывайте рассмотреть крайние случаи вроде «нуль» или «много»; §29.4.6.

[16] Помимо тестирования частей проекта и проверки соответствия требованиям тестируйте проект посредством примеров реальных случаев использования; §29.5.

[17] Рассмотрите важный вопрос о том, как ваш проект может учесть особые требования к производительности; §29.5.4.

Часть IV

Обзор стандартной библиотеки

В этой части описана стандартная библиотека. Цель главы — дать понимание о том, как использовать библиотеку, показать полезные техники разработки и проектирования и как расширять библиотеку так, как и задумано.

Главы

30. Обзор стандартной библиотеки
31. STL контейнеры
32. STL Алгоритмы
33. STL Итераторы
34. Память и ресурсы
35. Прочие компоненты библиотеки
36. Строки
37. Регулярные выражения
38. Потоки ввода/вывода
39. Локализация
40. Численные рассчеты
41. Параллелизм
42. Потоки и задачи
43. Стандартная библиотека C
44. Совместимость

«... Я только сейчас начинаю испытывать всю сложность реализации идей на бумаге. Пока речь идет об описании чего-либо, все просто; но как только в игру вступают рассуждения, так сразу взаимосвязи, ясность и беглость изложения становятся для меня весьма нелегкой задачей, о чем ранее я и понятия не имел.»

— Чарльз Дарвин

30

Обзор стандартной библиотеки

Многие загадки и секреты природы
воспринимаются незнающими как чудеса.
— Роджер Бэкон

- Введение
  Средства стандартной библиотеки; Ограничения дизайна; Стиль описания
- Заголовочные файлы
- Поддержка языка
  Поддержка **initializer_list**; Поддержка range-**for**
- Обработка ошибок
  Исключения; Утверждения; **system_error**
- Советы

30.1. Введение

Стандартная библиотека состоит из компонентов, входящих в стандарт ISO C++ и поставляемых с каждой реализацией C++, причем поведение этих компонентов в любой реализации идентично. Я настоятельно рекомендую применять эти компоненты всюду, где это возможно, ради достижения переносимости кода и его долговременной актуальности. Даже если вы можете предоставить более эффективную альтернативу, стоит учесть следующее:

- Смогут ли программисты, которые в дальнейшем будут поддерживать вашу программу, получить доступ к документации по вашему уникальному решению?
- Будет ли ваше альтернативное решение поддерживаться на неизвестной ныне платформе лет этак через десять?
- Насколько ваша альтернатива будет полезна для иных приложений?
- Насколько хорошо ваше альтернативное решение будет взаимодействовать с кодом, построенным на стандартной библиотеке?
- Сможете ли вы потратить столько же времени и усилий на тестирование вашей альтернативы, сколько совокупно было потрачено на тестирование стандартной библиотеки?

И, наконец, вы сами или ваша организация будут навсегда ответственны за поддержку и дальнейшее развитие альтернативного решения. Мораль: не нужно изобретать велосипед.

Описание стандартной библиотеки весьма велико. Оно занимает в рамках ISO C++ стандарта 785 страниц. И это без описания стандартной ISO C библиотеки, которая входит

в описание ISO C++ стандарта (еще 139 страниц). Для сравнения, описание собственно языка C++ занимает всего лишь 389 страниц. Здесь я резюмирую информацию, опираясь преимущественно на таблицы, а также привожу немногочисленные примеры. Подробные сведения можно найти в иных местах, как-то: печатная версия ISO C++ стандарта, его же online-версия, подробные варианты документации к различным реализациям, исходный код открытых реализаций, наконец.

Главы, посвященные стандартной библиотеке, не обязательно читать строго в порядке их изложения. Каждая глава и почти каждый ее подраздел могут рассматриваться изолированно. Обратитесь к оглавлению и индексу в случае, если вы вдруг натолкнетесь на что-то вам неизвестное.

30.1.1. Средства стандартной библиотеки

Что должно входить в стандартную библиотеку языка C++? Для программиста было бы идеальным найти в библиотеке любые интересные и значимые классы, функции, шаблоны и т.д. Но вопрос задан по отношению именно к стандартной библиотеке, а не по отношению к некоторой библиотеке вообще, так что ответ «Все!» здесь неуместен. Стандартная библиотека должна включать в себя только то, что каждая реализация языка могла бы предоставить надежнейшим образом и на что каждый программист мог бы всегда опереться.

Средства стандартной библиотеки языка C++ спроектированы так, чтобы они предоставляли:

- Поддержку средств языка, таких как как управление выделением памяти (§11.2), оператор цикла range-**for** (§9.5.1) и информацию о типах на этапе выполнения программы (§22.2).
- Информацию о зависимых от реализации деталях языка, таких как, например, наибольшее конечное значение для типа **float** (§40.2).
- Примитивные операции, которые нелегко или неэффективно реализовывать непосредственно в языке, например, **is_polymorphic**, **is_scalar** и **is_nothrow_constructible** (§35.4.1).
- Низкоуровневые средства ("lock-free") для параллельного программирования (§41.3).
- Поддержку параллельного выполнения на базе потоков (§5.3, §42.2).
- Минимальную поддержку параллельного выполнения задач (вроде **future** и **async()**) (§42.4).
- Функции, которые большинство программистов не смогут реализовать эффективно и переносимо, например, **uninitialized_fill()** (§32.5) и **memmove()** (§43.5).
- Минимальную поддержку для отзыва неиспользуемой памяти (сборка мусора), вроде **declare_reachable()** (§34.5).
- Нетривиальные фундаментальные средства, на которые программист мог бы опереться в целях переносимости кода, например, **list** (§31.4), **map** (§31.4.3), **sort()** (§32.6) и потоки ввода/вывода (Глава 38).
- Архитектуру расширения предоставляемых библиотекой средств, включающую соглашения и средства поддержки для реализации ввода/вывода пользовательских типов в стиле ввода/вывода встроенных типов языка (Глава 38) и библиотеки STL (Глава 31).

Некоторые средства предоставляются стандартной библиотекой просто потому, что это удобно и полезно с практической точки зрения. Например, это стандартные математи-

ческие функции, вроде **sqrt()** (§40.3), генераторы случайных чисел (§40.7), арифметика для типа **complex** (§40.4) и регулярные выражения (Глава 37).

Также стандартная библиотека стремится быть базовым фундаментом для других библиотек. В частности, комбинация ее средств позволяет стандартной библиотеке играть роли:

- Основы для переносимости
- Набора компактных и эффективных компонентов, которые могут использоваться как фундамент библиотек и приложений, чувствительных к производительности
- Набора компонентов, способствующих межбиблиотечному взаимодействию

Архитектура стандартной библиотеки подчинена преимущественно этим трем ролям. Эти три роли тесно связаны между собой. К примеру, переносимость часто является важным критерием проекта специализированной библиотеки, и общераспространенные типы контейнеров, такие как списки, векторы и т.д., полезны для успешного взаимодействия раздельно разрабатываемых библиотек.

Последняя роль особо полезна с точки зрения перспектив дизайна, поскольку она ограничивает ареал стандартной библиотеки и накладывает разумные ограничения на ее средства. Например, средства для работы со строками и списками входят в стандартную библиотеку. Если бы этого не было, то разные библиотеки могли бы опираться лишь на общие для них встроенные типы языка C++. Но лишних компонентов в стандартной библиотеке нет, например отсутствуют средства линейной алгебры и средства для работы с графикой. Конечно, эти средства тоже очень полезны, но они не входят в каналы взаимодействия между библиотеками. Подобного рода средства можно оставить для реализации в иных библиотеках. Если же сторонняя библиотека на практике докажет свою полезность для многих операционных сред и разных предметных областей, то она станет кандидатом на включение в состав стандартной библиотеки. Таким примером является библиотека регулярных выражений (Глава 37).

Сокращенные версии стандартной библиотеки встречаются в составе свободных реализаций для запуска на системах с минимальной операционной системой или вообще без таковой (§6.1.1).

30.1.2. Ограничения дизайна

Роль стандартной библиотеки накладывает определенные ограничения на ее дизайн. Средства стандартной библиотеки спроектированы и реализованы для того, чтобы:

- Быть важными и доступными для каждого студента и профессионального программиста, включая разработчиков других библиотек.
- Использовались прямо или косвенно любым программистом для решения любой задачи, относящейся к области стандартной библиотеки.
- Были достаточно эффективными, чтобы при реализации иных библиотек эти средства могли составить естественную альтернативу функциям, классам и шаблонам, программируемым вручную.
- Были независимыми от политик использования или позволяли задавать эти политики в виде аргументов.
- Были примитивными в математическом смысле. То есть, компоненты стандартной библиотеки должны выполнять ровно одну роль.
- Быть удобными в использовании, эффективными и достаточно безопасными.
- Полными в своей области действия, чтобы пользователям из-за недостатка функциональности не пришлось заменять их иными средствами.

- Удобными в использовании со встроенными типами и операциями.
- Безопасными по типам и допускающими проверки на этапе выполнения.
- Были способными поддерживать общепринятые стили программирования.
- Расширяемыми и способными работать с пользовательскими типами в манере, аналогичной работе со встроенными типами.

Рассмотрим к примеру функцию сортировки: встраивание в ее код критерия сортировки неприемлемо, так как те же самые данные могут потребовать сортировки в соответствие с иными критериями. Поэтому функция **qsort()** из стандартной библиотеки языка С принимает критерий сортировки (функцию сортировки) в качестве аргумента, а не опирается на что-то заданное, например на операцию < (§12.5). В то же время, вызов функции влечет за собой дополнительные накладные расходы, так что функция **qsort()** приемлема не во всех случаях, например она плохо подходит в качестве строительного блока других библиотек. Почти для всех типов данных можно выполнить сравнение, не прибегая к функциональному вызову.

Естественно, что расходы на вызов функции не всегда критичны, однако в случае разработки некоторых алгоритмов это может привести к неприемлемому ухудшению производительности. Технология предоставления критерия сортировки посредством аргумента шаблона (§25.2.3) решает проблему эффективности для функции **sort()** и многих других алгоритмов стандартной библиотеки языка С++. Стандартная библиотека реализует свои средства на столь высоком уровне эффективности, что пользователи не должны искать альтернативы по этому поводу.

Требования примитивности и удобства использования могут конфликтовать друг с другом. Дело в том, что строгое следование принципу примитивности (ортогональности), может осложнить жизнь новичкам. Поэтому в стандартную библиотеку в качестве дополнительных средств добавлены пусть и непримитивные, но удобные операции.

30.1.3. Стиль описания

Полное описание даже простой операции из стандартной библиотеки, такой как конструктор или алгоритм, может занять много страниц. Как следствие, я использую чрезвычайно сокращенный стиль описания. Наборы связанных операций я помещаю в типичном случае в таблицы:

Некоторые операции	
p=op(b,e,x)	**op** делает что-то с диапазоном [**b**:**e**) и с **x**, возвращая **p**
foo(x)	**foo** делает что-то с **x** но не возвращает результат
bar(b,e,x)	Имеет ли **x** какое-то отношение к [**b**:**e**)?

Я стараюсь следовать стандартной мнемонике при выборе идентификаторов, так что **b** и **e** будут итераторами, специфицирующими диапазон, **p** будет указателем или итератором, а **x** будет относиться к какому-либо значению, но все это будет зависеть от контекста. Здесь только комментарии помогут отличить отрицательный ответ от результата логического типа. Обычно в конце описаний с возвратом логического типа я ставлю знак вопроса. Если же алгоритм следует стандартной схеме возврата конца входной последовательности для указания на результат «не найдено» и т.д. (§4.5.1, §33.1.1), то я не помечаю это явным образом.

Чаще всего, столь сокращенная презентация материала сопровождается ссылкой на стандарт ISO C++, дальнейшие пояснения и примеры кода.

30.2. Заголовочные файлы

Средства стандартной библиотеки определены в пространстве имен **std** и расположены в некотором наборе заголовочных файлов, реализующих большую часть этих средств. Перечисление этих заголовочных файлов дает представление о стандартной библиотеке и поясняет направление ее рассмотрения в настоящей и последующих главах книги.

Ниже в данном разделе мы приводим список заголовочных файлов стандартной библиотеки, сгруппированный по функциональности, и сопровождаемый краткими пояснениями и ссылками на разделы книги, в которых они рассматриваются.

Стандартный заголовочный файл, начинающийся на букву **c**, эквивалентен соответствующему заголовочному файлу стандартной библиотеки языка С. Для каждого файла **<X.h>**, определяющего часть стандартной библиотеки языка С в глобальном пространстве имен и в пространстве имен **std**, имеется заголовочный файл **<cX>**, определяющий те же имена. В идеале, имена из заголовочного файла **<cX>** не должны загрязнять глобальное пространство имен (§15.2.4), но к сожалению, по причинам сложностей в поддержке многоязыковых сред разных операционных систем полностью избежать этого не удается.

Контейнеры		
<vector>	Одномерный массив с изменяемым размером	§31.4.2
<deque>	Очередь с двумя концами	§31.4.2
<forward_list>	Односвязанный список	§31.4.2
<list>	Двусвязанный список	§31.4.2
<map>	Ассоциативный массив	§31.4.3
<set>	Множество	§31.4.3
<unordered_map>	Хэшированный ассоциативный массив	§31.4.3.2
<unordered_set>	Хэшированное множество	§31.4.3.2
<queue>	Очередь	§31.5.2
<stack>	Стек	§31.5.1
<array>	Одномерный массив с фиксир. размером	§34.2.1
<bitset>	Массив элементов **bool**	§34.2.2

Ассоциативные контейнеры **multimap** и **multiset** находятся в **<map>** и **<set>**, соответственно. Контейнер **priority_queue** (§31.5.3) объявляется в **<queue>**.

Общие утилиты		
<utility>	Операции и pairs	§35.5, §34.2.4.1
<tuple>	Объекты типа Tuple	§34.2.4.2
<type_traits>	Типовые traits	§35.4.1
<typeindex>	Применение type_info как ключ или код хэширования	§35.5.4
<functional>	Объекты-функции	§33.4
<memory>	Указатели управления ресурсами	§34.3
<scoped_allocator>	Аллокаторы зоне видимости	§34.4.4
<ratio>	Рациональная арифметика времени компиляции	§35.3
<chrono>	Утилиты для работы со временем	§35.2
<ctime>	Дата и время в С-стиле	§43.6
<iterator>	Итераторы и их поддержка	§33.1

Итераторы предоставляют механизм, позволяющий выполнить стандартные алгоритмы в обобщенном стиле (§3.4.2, §33.1.4).

Алгоритмы		
<algorithm>	Общие алгоритмы	§32.2
<cstdlib>	**bsearch(), qsort()**	§43.7

Типичный обобщенный алгоритм может применяться к любому интервалу (последовательности) (§3.4.2, §32.2) элементов любого типа. Функции **bsearch()** и **qsort()** из стандартной библиотеки языка С применимы ко встроенным массивам элементов типов, не имеющих пользовательских копирующих конструкторов и деструкторов (§12.5).

Диагностика		
<exception>	Класс исключений	§30.4.1.1
<stdexcept>	Стандартные исключения	§30.4.1.1
<cassert>	Макросы Assert	§30.4.2
<cerrno>	Обработка ошибок в стиле С	§13.1.2
<system_error>	Системные ошибки	§30.4.3

Проверка диагностических утверждений с генерацией исключений описана в §13.4.

Строки и символы		
<string>	Строка элементов **T**	Глава 36
<cctype>	Классификация символов	§36.2.1
<cwctype>	Классификация «широких» символов	§36.2.1
<cstring>	Функции над строками в стиле С	§43.4
<cwchar>	«Широкие» строковые ф-ии в стиле С	§36.2.1
<cstdlib>	Функции выделения в стиле С	§43.5
<cuchar>	Многобайтовые символы в стиле С	
<regex>	Регулярные выражения	Глава 37

Заголовочный файл **<cstring>** объявляет семейство функций **strlen()**, **strcpy()** и т.д. Заголовочный файл **<cstdlib>** объявляет функции **atof()** и **atoi()**, конвертирующие С-строки в числовые значения.

Ввод/вывод		
<iosfwd>	«Опережающие» объявления средств ввода/вывода	§38.1
<iostream>	Стандартные объекты **iostream** и операции	§38.1
<ios>	Базовые классы **iostream**	§38.4.4
<streambuf>	Буферы потоков	§38.6
<istream>	Шаблон потока ввода	§38.4.1
<ostream>	Шаблон потока вывода	§38.4.2
<iomanip>	Манипуляторы	§38.4.5.2
<sstream>	Потоки ввода/вывода в строки и из строк	§38.2.2
<cctype>	Функции для работы с символами	§36.2.1
<fstream>	Потоки ввода/вывода в файлы и из файлов	§38.2.1
<cstdio>	Семейство функций **printf()**	§43.3
<cwchar>	Ввод/вывод «широких» символов в стиле **printf()**	§43.3

Манипуляторы — это объекты, предназначенные для управления (манипулирования) состоянием потока (§38.4.5.2).

Локализация		
<locale>	Представляет локальные особенности	Глава 39
<clocale>	Представляет локальные особенности в стиле C	
<codecvt>	Фасеты конверсии кодов символов	§39.4.6

Заголовочный файл **<locale>** посвящен локализации отличий в формате дат, символах обозначения валют и критериях сравнения строк, свойственных разным естественным языкам и культурам.

Поддержка языка		
<limits>	Числовые пределы	§40.2
<climits>	Макросы числовых скалярных пределов в стиле C	§40.2
<cfloat>	Макросы пределов чисел с плавающей запятой в стиле C	§40.2
<cstdint>	Стандартные имена целочисленных типов	§43.7
<new>	Динамическое распределение памяти	§11.2.3
<typeinfo>	Поддержка идентификации типов на этапе исполнения	§22.5
<exception>	Поддержка обработки исключений	§30.4.1.1
<initializer_list>	**initializer_list**	§30.3.1
<cstddef>	Языковая поддержка библиотеки C	§10.3.1
<cstdarg>	Поддержка функций с переменным числом аргументов	§12.2.4
<csetjmp>	Раскрутка стека в стиле C	
<cstdlib>	Завершение программ	§15.4.3
<ctime>	Системные часы	§43.6
<csignal>	Обработка сигналов в стиле C	

Заголовочный файл **<cstddef>** определяет тип значений, возвращаемых **sizeof()**, **size_t**, тип результата вычитания указателей и индексов массива, **ptrdiff_t** (§10.3.1) и пользующийся дурной репутацией макрос **NULL** (§7.2.2).

Раскрутка стека в стиле языка C (с использованием **setjmp** и **longjmp** из **<csetjmp>**) не совместима с использованием деструкторов и с обработкой исключений (§30.4, Глава 13) и ее лучше избегать. В нашей книге раскрутка стека в стиле языка C и сигналы не рассматриваются.

Номера		
<complex>	Комплексные числа и операции с ними	§40.4
<valarray>	Векторы чисел и операции с ними	§40.5
<numeric>	Обобщенные числовые операции	§40.6
<cmath>	Стандартные математические функции	§40.3
<cstdlib>	Случайные числа в стиле C	§40.7
<random>	Генераторы случайных чисел	§40.7

По историческим причинам **abs()** и **div()** находятся в **<cstdlib>**, а не вместе с остальными математическими функциями в **<cmath>** (§40.3).

Параллелизм		
<atomic>	Атомарные типы и операции	§41.3
<condition_variable>	Ожидание события	§42.3.4
<future>	Асинхронные задачи	§42.4.4
<mutex>	Классы мьютексов	§42.3.1
<thread>	Потоки	§42.2

Язык С предоставляет средства стандартной библиотеки разной степени пригодности для программистов на C++. Стандарт языка C++ (стандартная библиотека) обеспечивает доступ ко всем таким средствам:

Совместимость с С		
<cinttypes>	Псевдонимы обычных целочисленных типов	§43.7
<cstdbool>	С **bool**	
<ccomplex>	**<complex>**	
<cfenv>	Среда вычислений с плавающей запятой	
<cstdalign>	Выравнивание С	
<ctgmath>	С "type generic math": **<complex>** и **<cmath>**	

Файл **<cstdbool>** не определяет макросов **bool**, **true** или **false**. Файл **<cstdalign>** не определяет макрос **alignas**. Эквиваленты с **.h** для **<cstdbool>**, **<ccomplex>**, **<calign>** и **<ctgmath>** аппроксимируют средства языка C++ под язык С. Вообще-то, лучше избегать их.

Файл **<cfenv>** предоставляет типы (такие как **fenv_t** и **fexcept_t**), флаги статуса для чисел с плавающей запятой и управляющие средства, описывающие реализацию среды для чисел с плавающей запятой.

Пользователю или реализатору библиотеки запрещено добавлять или изымать объявления из стандартных заголовочных файлов, а также запрещено изменять смысл объявлений в этом файле (§15.2.3). Любые игры в этом направлении приведут к непереносимости кода.

Чтобы использовать некоторое средство из стандартной библиотеки, в текст программы нужно включить соответствующий заголовочный файл. Не является стандартной попытка изымать часть объявлений и включать их отдельно в текст программы. Дело в том, что многие реализации явным образом реагируют на включение стандартных заголовочных файлов и на этой основе срабатывают механизмы оптимизации, а также иные действия, которые программист предвидеть не может.

Программист может специализировать вспомогательные шаблоны из стандартной библиотеки, например **swap()** (§35.5.2), под конкретные нужды нестандартных библиотек и пользовательских типов.

30.3. Поддержка языка

Небольшая часть стандартной библиотеки осуществляет непосредственную поддержку языковых средств, без которой программы не смогут запуститься и работать.

Поддержка библиотекой свойств языка		
<new>	**new** и **delete**	§11.2
<typeinfo>	**typeid()** и **type_info**	§22.5
<iterator>	Range-**for**	§30.3.2
<initializer_list>	**initializer_list**	§30.3.1

30.3.1. Поддержка initializer_list

{}-список конвертируется в объект типа **std::initializer_list<X>** согласно правилам, описанным в §11.3. В файле **<initializer_list>** мы и находим **initializer_list**:

```
template<typename T>
class initializer_list { // § iso.18.9
public:
    using value_type = T;
    using reference = const T&; // внимание const: элементы initializer_list неизменяемы
    using const_reference = const T&;
    using size_type = size_t;
    using iterator = const T*;
    using const_iterator = const T*;

    initializer_list() noexcept;

    size_t size() const noexcept;       // число элементов
    const T* begin() const noexcept;    // первый элемент
    const T* end() const noexcept;      // "элемент за-последним"
};

template<typename T>
    const T* begin(initializer_list<T> lst) noexcept {return lst.begin();}
template<typename T>
    const T* end(initializer_list<T> lst) noexcept {return lst.end();}
```

К сожалению, **initializer_list** не предоставляет операции индексирования. Если вам хочется использовать [] вместо *, индексируйте указатель:

```
void f(initializer_list<int> lst)
{
    for(int i=0; i<lst.size(); ++i)
        cout << lst[i] << '\n';     // ошибка

    const int* p = lst.begin();
    for(int i=0; i<lst.size(); ++i)
        cout << p[i] << '\n';       // OK
}
```

Естественно, **initializer_list** можно использовать с range-**for**. Пример:

```
void f2(initializer_list<int> lst)
{
    for (auto x : lst)
        cout << x << '\n';
}
```

30.3.2. Поддержка range-for

Оператор range-**for** сопоставляется с оператором **for** на основе итераторов, как это описано в §9.5.1.

В файле **<iterator>** стандартная библиотека определяет функции **std::begin()** и **std::end()** для встроенных массивов и для каждого типа, который предоставляет члены **begin()** и **end()**; см. §33.3.

Все контейнеры стандартной библиотеки (например, **vector** и **unordered_map**) и строки поддерживают итерирование в операторе range-**for**, но контейнерные адаптеры (такие как **stack** и **priority_queue**) этого не делают. Заголовочные файлы контейнеров, такие как **<vector>**, включают **<initializer_list>**, так что пользователям редко когда приходится делать это непосредственным образом.

30.4. Обработка ошибок

В стандартную библиотеку входят компоненты, которые разрабатывались в течение последних сорока лет. В результате их стиль и подходы к обработке ошибок не являются согласованными:

- С-библиотеки состоят из функций, многие из которых устанавливают переменную **errno** для индикации факта ошибки; см. §13.1.2 и §40.3.
- Многие алгоритмы, оперирующие на последовательности элементов, возвращают итератор на позицию «за последним элементом» для индикации ошибок; см. §33.1.1.
- Библиотека потокового ввода/вывода полагается на состояние потока для индикации ошибок и может генерировать исключение с этой целью; см. §38.3.
- Некоторые компоненты стандартной библиотеки, такие как **vector**, **string** и **bitset**, в случае ошибок генерируют исключения.

В стандартной библиотеке все ее средства должны подчиняться «базовой гарантии» (§13.2); это означает, что если сгенерировано исключение, то никаких утечек ресурсов не происходит и не нарушаются инварианты для классов библиотеки.

30.4.1. Исключения

Ряд средств стандартной библиотеки для индикации ошибок генерируют исключения:

Исключения стандартной библиотеки	
bitset	Генерирует **invalid_argument**, **out_of_range**, **overflow_error**
iostream	Генерирует **ios_base::failure**, если исключения разрешены
regex	Генерирует **regex_error**
string	Генерирует **length_error**, **out_of_range**
vector	Генерирует **out_of_range**
new T	Генерирует **bad_alloc**, если не может выделить память для **T**
dynamic_cast<T>(r)	Генерирует **bad_cast**, если не может преобразовать ссылку **r** в **T**
typeid()	Генерирует **bad_typeid**, если не может предоставить **type_info**
thread	Генерирует **system_error**
call_once()	Генерирует **system_error**
mutex	Генерирует **system_error**
condition_variable	Генерирует **system_error**
async()	Генерирует **system_error**
packaged_task	Генерирует **system_error**
future и **promise**	Генерирует **future_error**

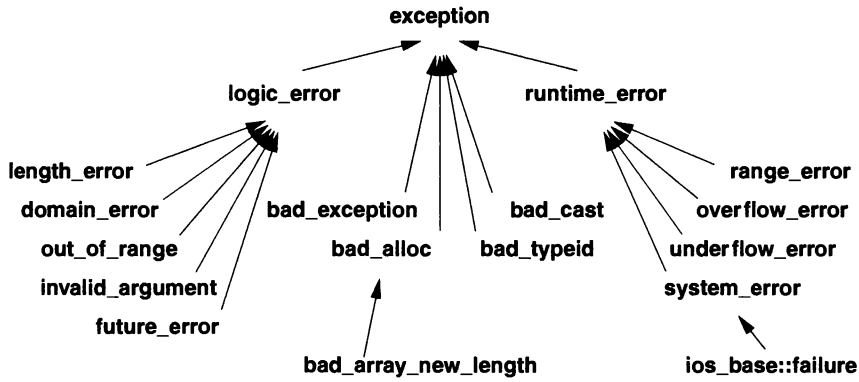
Исключения могут учитываться в любом коде, который использует эти средства стандартной библиотеки. Кроме того, любая операция, манипулирующая объектом, который может генерировать исключения, также должна считаться генерирующей это исключение, если только не приняты меры во избежание этого. Например, **packaged_task** будет генерировать исключение, если функция, необходимая для работы, генерирует исключение.

В любом случае, даже если вам кажется, что используемые вами средства не должны генерировать исключение, то будет неплохой идеей всегда перехватывать какой-либо из базовых классов иерархии исключений стандартной библиотеки (такой как **exception**) или любое исключение (...) (§13.5.2.3), например в функции **main()**.

30.4.1.1. Стандартная иерархия exception

Не нужно генерировать исключения встроенных типов, вроде **int** или строк в С-стиле. Вместо этого генерируйте исключения с объектами пользовательских типов, специально спроектированных так, чтобы они использовались для исключений.

Иерархия классов стандартных исключений отражает классификацию исключений:



Эта иерархия пытается основать рамочную конструкцию для исключений, выходящих за рамки определенных в стандартной библиотеке. Логические ошибки — это ошибки, которые в принципе могут обнаруживаться до того, как программа начнет выполняться, или же они выявляются тестированием аргументов функций и конструкторов. Ошибки этапа выполнения — это все остальные ошибки. Например, **system_error** описывается в §30.4.3.3.

Корнем иерархии исключений стандартной библиотеки является класс **exception**:

```
class exception {
public:
    exception();
    exception(const exception&);
    exception& operator=(const exception&);
    virtual ~exception();
    virtual const char* what() const;
};
```

Функция **what()** может использоваться для того, чтобы получить строку, которая предположительно сможет разъяснить что-нибудь по поводу ошибки, вызвавшей исключение.

Программист может определять исключения посредством наследования от стандартного исключения, например вот так:

```
struct My_error : runtime_error {
    My_error(int x): runtime_error{"My_error"}, interesting_value{x} {}
    int interesting_value;
};
```

Не все исключения являются частью **exception**-иерархии стандартной библиотеки. Однако, все исключения, генерируемые средствами стандартной библиотеки, являются означенной частью.

Даже если вам кажется, что исключения у вас в программе генерироваться не должны, хорошей идеей будет все же попытка перехвата любых исключений. Например:

```
int main()
try {
    // ...
}
catch (My_error& me) {          // произошло My_error
    // можем использовать me.interesting_value и me.what()
}
catch (runtime_error& re) {     // произошла runtine_error
    // можно использовать re.what()
}
catch (exception& e) {          // произошло исключение стандартной библиотеки
    // можно использовать e.what()
}
catch (...) {                   // произошло неопределенное исключение
    // делаем локальную очистку
}
```

Как и в случае аргументов функций, мы используем ссылки во избежание срезки (§17.5.1.4).

30.4.1.2. Продвижение (распространение) исключений

В файле **<exception>** стандартная библиотека предоставляет средства, которые дают программисту возможность продвижения исключений:

Распространение исключений (§ iso.18.8.5)	
exception_ptr	Бестиповое указание на исключение
ep=current_exception() ep	это **exception_ptr** на текущее исключение, или на не исключение если нет текущего исключения; noexcept
rethrow_exception(ep)	Повторная генерация исключения, указуемая **ep**; **ep** – внутренний указатель не должен быть **nullptr**; noreturn (§ 12.1.7)
ep=make_exception_ptr(e)	**ep** есть **exception_ptr** на **exception e**; noexcept

exception_ptr может указывать на любое исключение, а не только на исключение из **exception**-иерархии. Можно думать об **exception_ptr** как об умном указателе (вроде **shared_ptr**), который хранит свое исключение до тех пор, пока **exception_ptr** указывает на него. Таким образом, мы можем передавать **exception_ptr** исключению вне функции, которая перехватывает его и снова генерирует где-либо в ином месте. В частности, можно использовать **exception_ptr** для реализации повторной генерации исключения в другом потоке (отличном от того потока, где оно было перехвачено). На это, к примеру, полагаются **promise** и **future** (§42.4). Применение **rethrow_exception()** на **exception_ptr** (из разных потоков) не приводит к «гонке за данными».

Функция **make_exception_ptr()** может реализовываться следующим образом:

```
template<typename E>
exception_ptr make_exception_ptr(E e) noexcept;
try {
    throw e;
}
catch(...) {
    return current_exception();
}
```

Класс **nested_exception** это класс, который хранит **exception_ptr**, полученный из вызова **current_exception()**:

Вложенные исключения (§ iso.18.8.6)	
`nested_exception ne {};`	Умолчательный конструктор: **ne** содержит **exception_ptr** на **current_exception()**; noexcept
`nested_exception ne {ne2};`	Копирующий конструктор: как **ne** так и **ne2** содержат **exception_ptr** на хранимое **exception**
`ne2=ne`	Копирующее присваивание: как **ne** так и **ne2** содержат **exception_ptr** на хранимое exception
`ne.~nested_exception()`	Деструктор; виртуальный
`ne.rethrow_nested()`	Повторная генерация хранимого в **ne** исключения; **terminate()** Если нет в **ne** хранимого исключения; noreturn
`ep=ne.nested_ptr()`	**ep** есть **exception_ptr** указующий на хранящееся в **ne** исключение; noexcept
`throw_with_nested(e)`	Генерация исключения типа, производного от **nested_exception** и типа **e**; **e** не должно быть производным от **nested_exception**; noreturn
`rethrow_if_nested(e)`	**dynamic_cast<const nested_exception&>(e).rethrow_nested();** тип **e** должен быть производным от **nested_exception**

Предполагается применять **nested_exception** в качестве базового класса для класса, который используется обработчиком исключения для передачи некоторой информации о локальном контексте ошибки вместе с указателем **exception_ptr** на само исключение. Например:

```
struct My_error : runtime_error {
    My_error(const string&);
    // ...
};

void my_code()
{
    try {
        // ...
    }
    catch (...) {
        My_error err {"something went wrong in my_code()"};
        // ...
        throw_with_nested(err);
    }
}
```

Теперь информация из **My_error** распространяется (продвигается, генерируется повторно) вместе с **nested_exception**, содержащим **exception_ptr** на перехваченное исключение.

Где-то выше по цепи вызовов мы можем захотеть взглянуть на вложенное исключение:

```
void user()
{
    try {
        my_code();
    }
    catch(My_error& err) {

    // ... очистка My_error проблем ...
```

```
            try {
                rethrow_if_nested(err);    // повтор. генерация исключения
            }
            catch (Some_error& err2) {
                // ... очистка Some_error проблем ...
            }
        }
    }
```

Это предполагает, что мы знаем, что **some_error** вложено в **My_error**.

Исключение не может быть продвинуто (распространено) вне функции **noexcept** (§13.5.1.1).

30.4.1.3. terminate()

В файле **<exception>** стандартная библиотека реализует средства, имеющие дело с неожиданными исключениями:

terminate (§ iso.18.8.3, § iso.18.8.4)	
h=get_terminate()	**h** – текущий обработчик завершения; noexcept
h2=set_terminate(h)	**h** становится текущим обработчиком завершения; **h2** есть предыдущий обработчик; noexcept
terminate()	Завершить программу; noreturn; noexcept
uncaught_exception()	Было ли в текущем потоке сгенерировано и еще не перехвачено исключение? noexcept

Старайтесь не использовать эти функции, разве что **set_terminate()** и **terminate()** время от времени. Вызов **terminate()** завершает программу вызовом обработчика завершения, установленного вызовом **set_terminate()**. По умолчанию (что корректно почти всегда) программа сразу завершается. По фундаментальным причинам, относящимся к операционной системе, от нее зависит факт вызова деструкторов для локальных объектов при вызове **terminate()**. Если **terminate()** вызывается вследствие нарушения **noexcept**, то системе разрешается оптимизация, означающая, что стек может быть частично раскручен (§ iso.15.5.1).

Иногда утверждается, что **uncaught_exception()** может быть полезна для написания деструкторов, которые будут вести себя по-разному в зависимости от того, завершилась ли функция нормально или же по исключению. Однако, **uncaught_exception()** также верна во время раскрутки стека (§13.5.1) после того, как было перехвачено начальное исключение. Я полагаю, что **uncaught_exception()** — это слишком тонкая материя для практического применения.

30.4.2. Утверждения

Стандарт предоставляет следующее:

Утверждения (§ iso.7)	
static_assert(e, s)	Проверить **e** во время компиляции; выдать сообщение об ошибке **s**, если **!e**
assert(e)	Если **NDBUG** не определено, проверить **e** во время исполнения и, если **!e**, вывести сообщ-е в **cerr** и вызвать **abort()**; если **NDBUG** определено, ничего не делать

Пример:

```
template<typename T>
void draw_all(vector<T*>& v)
{
    static_assert(Is_base_of<Shape, T>(),"non-Shape type for draw_all()");

    for (auto p : v) {
        assert(p!=nullptr);
        // ...
    }
}
```

Здесь **assert()** это макрос, находящийся в файле **<cassert>**. Сообщение об ошибке из **assert()** зависит от реализации, но оно должно содержать имя исходного файла (**__FILE__**) и номер строки в исходном коде (**__LINE__**), содержащей вызов **assert()**.

В реальном производственном коде утверждений всегда намного больше, чем в иллюстративных примерах из учебников.

Имя функции (**__func__**) также может содержаться в сообщении. Утверждения проверяются не всегда. Серьезной ошибкой будет полагать, что если **assert(p!=nullptr)** ловит ошибку во время отладки, то это же будет иметь место и в окончательном (неотладочном) коде.

Насчет способов работы с утверждениями см. §13.4.

30.4.3. system_error

В файле **<system_error>** стандартная библиотека реализует каркас кода для сообщений об ошибках от операционной системы и других низкоуровневых компонентов системы. Например, можно написать функцию для проверки имени файла и его последующем открытии:

```
ostream& open_file(const string& path)
{
    auto dn = split_into_directory_and_name(path);    // расщепить на {path, name}

    error_code err {does_directory_exist(dn.first)};  // спросить систему про path
    if (err) {  // err!=0 означает ошибку

        // ... смотрим, что еще можно сделать ...

        if (cannot_handle_err)
            throw system_error(err);
    }

    // ...
    return ofstream{path};
}
```

В **<system_error>** стандартная библиотека реализует средства классификации кодов ошибок, для отображения специфичных для операционной системы кодов в более переносимые варианты и для отображения кодов ошибок в исключения:

Типы системных ошибок	
error_code	Содержит значение, указывающее ошибку и категорию этой ошибки; системно-зависимо (§30.4.3.1)
error_category	Базовый класс для типов, указывающих источник и значение кодов конкретной категории ошибок (§30.4.3.2)
system_error	Исключение **runtime_error**, содержащее **error_code** (§30.4.3.3)

Продолжение

Типы системных ошибок	
error_condition	Содержит значение, указывающее ошибку и категорию этой ошибки; потенциально переносимо (§30.4.3.4)
errc	**enum class**, перечисляющий коды ошибок из **<cerrno>** (§40.3); практически, коды ошибок POSIX
future_errc	**enum class**, перечисляющий коды ошибок из **<future>** (§42.4.4)
io_errc	**enum class**, перечисляющий коды ошибок из **<ios>** (§38.4.4)

30.4.3.1. Коды ошибок

По мере того, как ошибка «всплывает» наверх от более низкого уровня, мы должны ее обработать или перевести ее в исключение. Но сначала ее надо классифицировать: разные операционные системы имеют разные кода ошибок для одних и тех же проблемных ситуаций, и еще разные системы выявляют разные виды ошибок.

error_code (§ iso.19.5.2)	
error_code ec {};	Умолчательный конструктор: **ec={0,&generic_category}**; noexcept
error_code	**ec {n, cat}**; **ec={n, cat}**; **cat** есть **error_category** and **n** есть **int** содержащий код ошибки из **cat**; noexcept
error_code ec {n};	**ec={n,&g eneric_category}**; **n** есть код ошибки; **n** – значение типа EE, для которого **is_error_code_enum<EE>::value==true**; noexcept
ec.assign(n, cat)	**ec={n, cat}**; **cat** есть **error_category**; **n** есть код ошибки; **n** – значение типа EE, для которого **is_error_code_ enum<EE>::value==true**; noexcept
ec=n	**ec={n,&generic_category}**: **ec=make_error_code(n)**; **n** есть код ошибки; **n** – значение типа EE, для которого **is_error_code_ enum<EE>::value==true**; noexcept
ec.clear()	**ec={0,&generic_category()}**; noexcept
n=ec.value()	**n** есть значение, хранимое в **ec**; noexcept
cat=ec.category()	**cat** – ссылка на категорию, хранимую в **ec**; noexcept
s=ec.message()	**s** есть **string**, представляющий потенциальное сообщение об ошибке **ec**: **ec.category().message(ec.value())**
bool b {ec};	Преобразовать **ec** в **bool**; **b** есть **true**, если **ec** представляет ошибку; то есть, **b==false** значит "нет ошибки"; explicit
ec==ec2	**ec** и/или **ec2** могут быть **error_code**; чтобы **ec** и **ec2** считались равными, у них должны быть эквивалентные **category()** и эквивалентные **value()**; если **ec** и **ec2** одного типа, **by** эквивалентность определяется с помощью **==**; в противном случае, с помощью **category().equivalent()**.
ec!=ec2	**!(ec==ec2)**
ec<ec2	В порядке **ec.category()<ec2.category()** **|| (ec.category()==ec2.categor y() && ec.value()<ec2.value())**
e=ec.default_error_condition()	**e** есть ссылка на **error_condition**: **e=ec.category().default_error_ condition(ec.value())**
os<<ec	Вывести **ec.name()** в **ostream os**
ec=make_error_code(e)	**e** есть **errc**; **ec=error_code(static_cast<int>(e),&g eneric_category())**

Для типа, отражающего простейшую идею о кодах ошибок, **error_code** имеет неожиданно много членов. В своей основе, это простое отображение от целого значения к указателю на **error_category**:

```
class error_code {
public:
    // представление: {value, category} типа {int, const error_category*}
};
```

error_category является интерфейсом к объекту класса, производного от **error_category**. Поэтому **error_category** передается по ссылке и хранится как указатель. Каждый отдельный **error_category** представляется уникальным объектом.

Рассмотрим снова пример с **open_file()**:

```
ostream& open_file(const string& path)
{
    auto dn = split_into_directory_and_name(path);    // расщепить на {path, name}

    if (error_code err {does_directory_exist(dn.first)}) {// спросить систему про path
        if (err==errc::permission_denied) {
            // ...
        }
        else if (err==errc::not_a_directory) {
            // ...
        }
        throw system_error(err); // ничего не могу сделать локально
    }

    // ...
    return ofstream{path};
}
```

Коды ошибок **errc** описываются в §30.4.3.6. Обратите внимание на то, что я использовал оператор **if**, а не **switch**, поскольку операция == определена в терминах эквивалентности, что учитывает как **category()** для ошибки, так и **value()** для ошибки.

В зависимости от операционной системы **error_codes** могут отображаться в **error_condition** (§30.4.3.4), используя механизмы, рассмотренные в §30.4.3.5. **error_condition** извлекается из **error_code**, используя **default_error_condition()**. В типичном случае **error_condition** содержит меньше информации, чем **error_code**, так что будет хорошей идеей сохранять **error_code**, и извлекать **error_condition** по-потребности.

Манипулирование **error_code** не влияет на величину **errno** (§13.1.2, §40.3). Стандартная библиотека не влияет на значения кодов ошибок, предоставляемых другими библиотеками.

30.4.3.2. Категории ошибок

error_category представляет классификацию ошибок. Специфические ошибки представлены классом, наследующим от класса **error_category**:

```
class error_category {
public:
    // ...интерфейс к конкретным категорим, производным от error_category ...
};
```

error_category (§ iso.19.5.1.1)	
cat.~error_category()	Деструктор; virtual; noexcept
s=cat.name()	**s** есть имя **cat**; **s** – строка в стиле C; virtual; noexcept

Продолжение

error_category (§ iso.19.5.1.1)	
ec=cat.default_error_condition(n)	ec есть error_condition для n в cat; virtual; noexcept
cat.equivalent(n, ec)	Верно ли ec.category()==cat и ec.value()==n? ec есть error_condition; virtual; noexcept
cat.equivalent(ec, n)	Верно ли ec.category()==cat и ec.value()==n? ec есть error_code; virtual; noexcept
s=cat.message(n)	s есть string, описывающий ошибку n в cat; virtual
cat==cat2	Обозначает ли cat ту же категорию, что и cat2? noexcept
cat!=cat2	!(cat==cat2); noexcept
cat<cat2	cat<cat2 в порядке адресов error_category: std::less<const error_category*>()(cat, cat2)? noexcept

Поскольку error_category разрабатывался как базовый класс, никаких копирующих и перемещающих операций в нем нет. Осуществляйте доступ к error_category по ссылкам или указателям.

Имеется четыре именованные категории:

Категории ошибок стандартной библиотеки (§ iso.19.5.1.1)	
ec=generic_category()	ec.name()=="generic"; ec есть ссылка на error_category
ec=system_category()	ec.name()=="system" ec есть ссылка на error_category; представляет системные ошибки: если ec соответствует ошибке POSIX, то ec.value() равно знач-ю errno для той ошибки
ec=future_category()	ec.name()=="future"; есть ссылка на error_category; обозначает ошибки из <future>
iostream_category()	ec.name()=="iostream"; ec есть ссылка на error_category; обозначает ошибки из библиотеки iostream

Эти категории полезны, поскольку простые целочисленные коды ошибок имеют разный смысл в разных контекстах. Например, 1 означает «операция не разрешена» (EPERM) в рамках стандарта POSIX, является общим кодом (state) для всех ошибок в iostream, и означает future_already_retrieved в случае ошибки в future.

30.4.3.3. Исключение system_error

system_error используется для сообщения об ошибках, которые в конечном счете возникают в той или иной части стандартной библиотеки, и которые имеют дело с операционной системой. Также выдается error_code и опционально строка с сообщением об ошибке:

```
class system_error : public runtime_error {
public:
    // ...
};
```

Класс исключения system_error (§ iso.19.5.6)	
system_error	se {ec, s}; se содержит {ec, s}; ec есть error_code; s есть string или строка в стиле C, используемая как часть сообщения об ошибке
system_error se {ec};	se содержит {ec}; ec есть error_code
system_error se {n, cat, s};	se содержит {error_code{n, cat}, s}; cat есть error_category, а n есть int, содержащий код ошибки из cat; s есть string или строка в стиле C, используемая как часть сообщения об ошибке

Продолжение

Класс исключения system_error (§ iso.19.5.6)	
system_error se {n, cat};	se содержит error_code{n, cat}; cat есть error_category, а n есть int содержащий код ошибки из cat
ec=se.code()	ec есть ссылка на error_code из se; noexcept
p=se.what()	p есть сообщение об ошибке из se в виде строки в стиле C; noexcept

Код, перехватывающий **system_error**, имеет также и **error_code**. Пример:

```
try {
    // что-нибудь
}
catch (system_error& err) {
    cout << "caught system_error " << err.what() <<'\n';    // сообщение об ошибке

    auto ec = err.code();
    cout << "category: " << ec.category().what() <<'\n';
    cout << "value: " << ec.value() <<'\n';
    cout << "message: " << ec.message() <<'\n';
}
```

Естественно, **system_error** может использоваться кодом, не являющимся частью стандартной библиотеки. Передается системно зависимый **error_code**, а не более переносимый **error_condition** (§30.4.3.4). Чтобы получить **error_condition** из **error_code**, используйте **default_error_condition()** (§30.4.3.1).

30.4.3.4. Потенциально переносимый error_condition

Потенциально переносимые коды ошибок (**error_condition**) представлены практически идентично системно зависимым **error_code**:

```
class error_condition {  // потенциально переносимо (§ iso.19.5.3)
public:
    // вроде error_code, но
    // нет операции вывода (<<) и
    // нет default_error_condition()
};
```

Общая идея формулируется так, что каждая система имеет набор специфических кодов, которые отображаются в потенциально переносимые ради удобства программистов (библиотек) и в случае разработок для разных платформ.

30.4.3.5. Отображение кодов ошибок

Начинаем с перечисления с желаемыми значениями **error_code**. Пример:

```
enum class future_errc {
    broken_promise = 1,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
};
```

Смысл этих значений зависит от категории. Целые значения в этом перечислении определяются реализациями.

Категория ошибок **future** является частью стандарта, так что ее можно найти в стандартной библиотеке. Детали могут отличаться от описанного мной.

Затем мы должны определить подходящую категорию для наших кодов ошибок:

```
class future_cat : error_category {      // подлежит выводу из future_category()
public:
    const char* name() const noexcept override {return "future";}

    string message(int ec) const override;
};

const error_category& future_category() noexcept
{
    static future_cat obj;
    return &obj;
}
```

Отображение от целых значений к строкам сообщений об ошибках довольно утомительно. Нам нужно изобрести набор сообщений, которые вероятно окажутся осмысленными для программиста. Здесь я не буду пытаться быть слишком умным:

```
string future_error::message(int ec) const
{
  switch (ec) {
  default:                                  return "bad future_error code";
  future_errc::broken_promise:              return "future_error: broken promise";
  future_errc::future_already_retrieved:    return "future_error: future already retrieved";
  future_errc::promise_already_satisfied:   return "future_error: promise already satisfied";
  future_errc::no_state:                    return "future_error: no state";
  }
}
```

Теперь можно сделать **error_code** из **future_errc**:

```
error_code make_error_code(future_errc e) noexcept
{
    return error_code{int(e), future_category()};
}
```

Для конструктора **error_code** и присваивания, которые принимают единственное значение для ошибки, требуется, чтобы аргумент имел подходящий тип для **error_category**. Например, аргумент, который претендует быть **value()** для **error_code** у **future_category()**, должен быть типа **future_errc**. В частности, мы не можем применить просто тип **int**. Пример:

```
error_code ec1 {7};                                 // ошибка
error_code ec2 {future_errc::no_state};             // OK

ec1 = 9;  // ошибка
ec2 = future_errc::promise_already_satisfied;       // OK
ec2 = errc::broken_pipe;                            // ошибка: неправильная категория ошибки
```

Для помощи реализаторам мы предоставляем специализацию **is_error_code_enum** для нашего перечисления:

```
template<>
struct is_error_code_enum<future_errc> : public true_type {};
```

Стандартная библиотека располагает общим шаблоном:

```
template<typename>
struct is_error_code_enum : public false_type {};
```

Чтобы **error_condition** работал для нашей категории, мы должны повторить то, что мы делали для **error_code**. Например:

```
error_condition make_error_condition(future_errc e) noexcept;

template<>
struct is_error_condition_enum<future_errc> : public true_type {};
```

Естественно, что помимо представленного варианта возможны и другие решения, например разные перечисления для **error_condition**, и отображение с помощью **make_error_condition()** из **future_errc** в эти перечисления.

30.4.3.6. Коды ошибок errc

Стандартные **error_code** для **system_category()** определяются посредством **enum class errc** со значениями, эквивалентными POSIX-совместимым значениям из **<cerrno>**:

Значения enum class errc (§ iso.19.5)	
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG
broken_pipe	EPIPE
connection_aborted	ECONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR
message_size	EMSGSIZE
network_down	ENETDOWN
network_reset	ENETRESET

Продолжение

Значения enum class errc (§ iso.19.5)	
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_child_process	ECHILD
no_link	ENOLINK
no_lock_available	ENOLCK
no_message	ENOMSG
no_message_available	ENODATA
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device	ENODEV
no_such_device_or_address	ENXIO
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT
too_many_files_open	EMFILE
too_many_files_open_in_system	ENFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

Эти коды справедливы для категории "system": system_category(). Для систем с поддержкой POSIX-средств они также справедливы и для категории "generic": generic_category().

POSIX-макросы есть суть целые, в то время как энумераторы errc имеют тип errc. Например:

```
void problem(errc e)
{
    if (e==EPIPE) {                 // ошибка: нет конверсии из errc в int
        // ...
    }

    if (e==broken_pipe) {           // ошибка: broken_pipe не в области видимости
        // ...
    }

    if (e==errc::broken_pipe) {     // OK
        // ...
    }
}
```

30.4.3.7. Коды ошибок future_errc

Стандартные **error_code** для **future_category()** определены посредством **enum class future_errc**:

Энумераторы enum class future_errc (§ iso.30.6.1)	
broken_promise	1
future_already_retrieved	2
promise_already_satisfied	3
no_state	4

Эти коды справедливы для категории "**future**": **future_category()**.

30.4.3.8. Коды ошибок io_errc

Стандартные **error_code** для **iostream_category()** определяются посредством **enum class io_errc**:

Энумератор enum class io_errc (§ iso.27.5.1)	
stream	1

Этот код справедлив для категории "**iostream**": **iostream_category()**.

30.5. Советы

[1] Используйте средства стандартной библиотеки ради переносимости; §30.1, §30.1.1.

[2] Используйте средства стандартной библиотеки для минимизации стоимости поддержки кода; §30.1.

[3] Используйте средства стандартной библиотеки в качестве базы для более изощренных и специализированных библиотек; §30.1.1.

[4] Используйте средства стандартной библиотеки как модель для построения гибкого и широко используемого кода; §30.1.1.

[5] Средства стандартной библиотеки определены в пространстве имен **std** и находятся в заголовочных файлах стандартной библиотеки; §30.2.

[6] Стандартный заголовочный файл С-библиотеки, имеющий вид **X.h**, представлен в стандартной библиотеке языка C++ в виде **<cX>**; §30.2.

[7] Не пытайтесь использовать средства стандартной библиотеки без явного включения (**#include**) в ваш код соответствующего заголовочного файла; §30.2.

[8] Для использования оператора range-**for** на встроенном массиве, нужно вставить в код директиву препроцессора **#include<iterator>**; §30.3.2.

[9] Предпочитайте обработку ошибок на базе исключений вместо системы на базе анализа кодов возврата; §30.4.

[10] Всегда перехватывайте **exception&** и ...; §30.4.1.

[11] Иерархия **exception** стандартной библиотеки может использоваться и для пользовательских исключений; §30.4.1.1.

[12] Вызывайте **terminate()** в крайних случаях серьезных ошибок; §30.4.1.3.

[13] Широко используйте **static_assert()** и **assert()**; §30.4.2.

[14] Не предполагайте, что **assert()** отрабатывает всегда; §30.4.2.

[15] Если вы не можете использовать исключения, рассмотрите **<system_error>**; §30.4.3.

31

STL контейнеры

Это было новым.
Это было особенным.
Это было простым.
Это должно было сработать.
— Г. Нельсон

- Введение
- Обзор контейнеров
  Внутреннее представление; Требования к элементам
- Обзор операций
  Типы, связанные с контейнерами; Конструкторы, деструкторы и присваивания; Размер и емкость; Итераторы; Доступ к элементам; Стековые операции; Операции, типичные для списков; Прочие операции
- Контейнеры
  vector; Списки; Ассоциативные контейнеры
- Адаптеры контейнеров
  stack; **queue**; **priority_queue**
- Советы

31.1. Введение

К STL относятся части стандартной библиотеки, касающиеся итераторов, контейнеров, алгоритмов и объектов-функций. STL рассматривается в настоящей книге еще и в Главе 32 и Главе 33.

31.2. Обзор контейнеров

Контейнеры содержат последовательность объектов. В данном разделе перечисляются типы контейнеров и даются их краткие характеристики. Операции с контейнерами перечисляются в разделе §31.3.

В стандартной библиотеке определяются два вида контейнеров:

- Последовательные контейнеры (*sequence containers*), открывающие доступ к последовательности элементов.
- Ассоциативные контейнеры (*associative containers*), обеспечивающие доступ к своим элементам на основе ключа.

Кроме того, в стандартной библиотеке имеются типы объектов, содержащих элементы, но не предоставляющих всех средств, которые предоставляют последовательные или ассоциативные контейнеры:

- Адаптеры контейнеров, реализующие специализированный доступ к нижележащим контейнерам.
- «Почти контейнеры» реализуют почти все (но все же не все) средства интерфейсов стандартных контейнеров.

STL контейнеры (последовательные и ассоциативные) соответствуют концепции дескрипторов ресурсов с операциями копирования и перемещения (§3.3.1). Операции с контейнерами предоставляют базовую гарантию (§13.2) для корректного взаимодействия с механизмами обработки исключений.

Последовательные контейнеры	
vector<T, A>	Непрерывно расположенная в памяти последовательность эл-тов **T**; контейнер по умолчанию
list<T, A>	Двусвязанный список эл-тов **T**; используйте, когда нужно вставлять и удалять без перемещения существующих эл-тов
forward_list<T, A>	Односвязанный список эл-тов **T**; идеален для пустых или очень коротких списков
deque<T, A>	Двусторонняя очередь эл-тов **T**; гибрид между вектором и списком; медленнее и того и другого для большинства задач

Шаблонный параметр **A** — это аллокатор, который используется контейнером для выделения и освобождения памяти (§13.6.1, §34.4). Например:

```
template<typename T, typename A = allocator<T>>
class vector {
    // ...
};
```

Здесь параметру **A** в качестве умолчательного значения назначен аллокатор **std::allocator<T>** (§34.4.1), который использует **operator new()** и **operator delete()** для выделения и освобождения памяти под элементы.

Последовательные контейнеры определяются в файлах **<vector>**, **<list>** и **<deque>**. Они являются непрерывными списками (например, **vector**) или связанными списками (например, **forward_list**) элементов типа **value_type** (выше обозначено как **T**). Контейнер **deque** — это смешанное решение из связанных списков и непрерывного распределения памяти.

Если нет конкретных причин против, используйте **vector**. Он предоставляет операции вставки и удаления элементов, что позволяет ему расширяться или сужаться по мере необходимости. Для последовательностей небольших элементов контейнер **vector** — это идеальное представление для структур данных, требующих операций над списками.

При вставке и удалении элементы в контейнере **vector** могут перемещаться. В контейнере **list** или в ассоциативных контейнерах элементы не подвергаются перемещениям при вставке новых или удалении иных элементов.

Контейнер **forward_list** (односвязанный список) — это список, оптимизированный для пустых или очень коротких списков. Пустой **forward_list** занимает лишь одно машинное слово. Как ни странно, но существует масса применений для списков, из которых большинство пустые (а остальные очень короткие).

Упорядоченные ассоциативные контейнеры (§ iso.23.4.2) C есть тип операции сравнения; A – тип аллокатора	
map<K,V,C,A>	Ассоциативный массив K и V; последовательность пар (K,V)
multimap<K,V,C,A>	Ассоциативный массив K и V; допускает одинаковые ключи
set<K,C,A>	Упорядоченный набор K
multiset<K,C,A>	Упорядоченный набор K; допускает одинаковые ключи

Эти контейнеры обычно реализуются в виде сбалансированных бинарных деревьев (как правило, это «красно-черные» деревья).

По умолчанию критерий сортировки ключей K есть std::less<K> (§33.4).

Как и для последовательных контейнеров, здесь шаблонный параметр A — это аллокатор, используемый контейнером для выделения и освобождения памяти (§13.6.1, §34.4). По умолчанию, это std::allocator<std::pair<const K, T>> (§31.4.3) для контейнера map и std::allocator<K> для контейнера set.

Неупорядоченные ассоциативные контейнеры (§ iso.23.5.2) H есть тип хэш-функции; E – функция сравнения; A – тип аллокатора	
unordered_map<K,V,H,E,A>	Неупорядоченный map K и V
unordered_multimap<K,V,H,E,A>	Неупорядоченный map K и V; допускает одинаковые ключи
unordered_set<K,H,E,A>	Неупорядоченный набор K
unordered_multiset<K,H,E,A>	Неупорядоченный набор K; допускает одинаковые ключи

Эти контейнеры реализованы как хэш-таблицы с дополнительным связыванием (with linked overflow). По умолчанию, хэш-функция H для типа K есть std::hash<K> (§31.4.3.2). Умолчательная функция E (сравнение на равенство) для типа K есть std::equal_to<K> (§33.4); эта функция используется для того, чтобы ответить на вопрос: равны ли друг другу два объекта, имеющих одинаковый хэш-код.

Ассоциативные контейнеры являются связанными структурами (деревьями) с узлами типа value_type (в использованной выше нотации это pair<const K, V> для map и K для set). Последовательности для set, map или multimap упорядочены по значению ключа K. Последовательности для неупорядоченных контейнеров не имеют определенного порядка. Отличие multimap от map состоит в том, что значение ключа не уникально (может встретиться не один раз).

Контейнерные адаптеры — это контейнеры, предоставляющие специализированные интерфейсы для других контейнеров:

Контейнерные адаптеры C есть тип контейнера	
priority_queue<T, C, Cmp>	Очередь T с приоритетом; Cmp – ф-ия приоритетизации
queue<T, C>	Очередь T с push() и pop()
stack<T, C>	Стек T с push() и pop()

Для priority_queue умолчательной функцией приоритетизации Cmp служит std::less<T>. Для queue умолчательным типом контейнера C служит std::deque<T>, а для stack и priority_queue это std::vector<T>. См. §31.5.

Некоторые типы данных предоставляют большую часть того, что требуется для стандартных контейнеров (но не все, что требуется). В этом случае мы называем их почти контейнерами. Вот наиболее интересные из них:

«Почти контейнеры»	
T[N]	Встроенный массив фиксированного размера: **N** смежных эл-тов типа **T**; нет **size()** или других функций-членов
array<T, N>	Массив фиксированного размера из **N** смежных эл-тов типа **T**; как встроенный массив, но решает многие проблемы
basic_string<C,Tr,A>	Смежным образом выделенная последовательность символов типа **C** с операциями обработки текста, напр., конкатенации (**+** и **+=**); **basic_string** обычно оптимизирована так, что не нуждается в отдельной памяти для коротких строк (§19.3.3)
string	**basic_string<char>**
u16string	**basic_string<char16_t>**
u32string	**basic_string<char32_t>**
wstring	**basic_string<wchar_t>**
valarray<T>	Числовой вектор с векторными операциями, оптимизированный для интенсивных вычислений; используйте только, если используете много векторной арифметики
bitset<N>	Набор **N** бит с операциями установки, вроде **&** и **|**
vector<bool>	Специализация **vector<T>** с компактно расположенными битами

Для **basic_string**, **A** это аллокатор (§34.4) и **Tr** — это характеризующая символы строки специализация шаблона **char_traits** (§36.2.2).

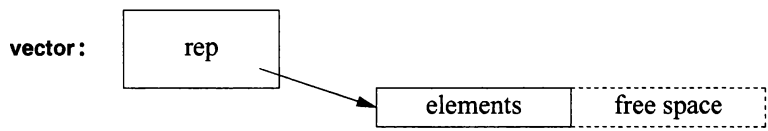
Когда есть такая возможность, используйте вместо встроенных массивов контейнеры **vector**, **string** или **array**, так как неявное автоматическое приведение к типу указателя и необходимость самостоятельно отслеживать размер массива служат источником частых ошибок (см., например, §27.2.1).

Отдавайте предпочтение стандартным строкам (тип **string**) перед любыми другими типами строк и особенно перед строками в С-стиле. Семантика указателей С-строк порождает неуклюжие конструкции и требует от программиста дополнительной работы, а также служит мощным источником ошибок (таких как, например, утечки памяти) (§36.3.1).

31.2.1. Внутреннее представление

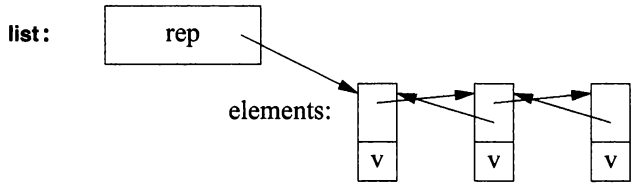
Стандарт не определяет для контейнеров никаких внутренних представлений (representation, сокращенно rep). Вместо этого, он специфицирует интерфейсы контейнеров и некоторые требования к сложности. Чтобы удовлетворить стандарту, разработчики применяют подходящие и тщательно оптимизированные варианты реализаций. Внутренний дескриптор ресурсов (rep), управляемых контейнером, использует аллокатор для выделения (и освобождения) памяти под элементы (§34.4).

Для контейнера **vector** структура данных под элементы больше всего похожа на встроенные массивы:

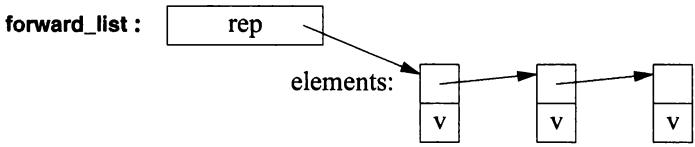


Реализации контейнера **vector** могут содержат указатель на массив элементов, число элементов, а также емкость (число выделенных, но неиспользуемых позиций под элементы), или нечто иное, но вполне эквивалентное (§13.6).

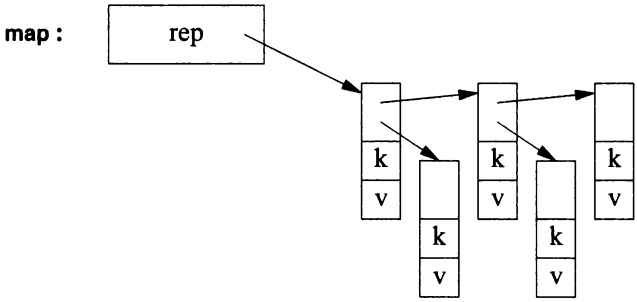
Контейнер **list** наиболее вероятно представим последовательностью связей, ссылающихся на элементы, и числом элементов:



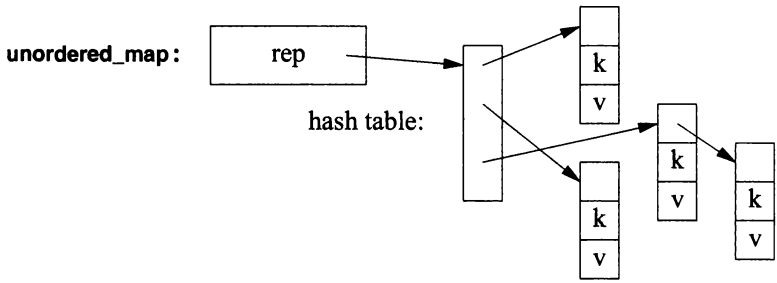
Контейнер **forward_list** вполне представим последовательностью связей, указывающих на элементы:



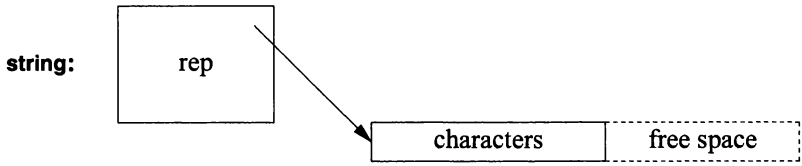
Контейнер **map** вероятнее всего представим сбалансированным деревом узлов, указывающих на (key, value)-пары:



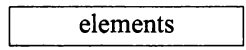
Контейнер **unordered_map** скорее всего реализуется с помощью хэш-таблицы:



Строки типа **string** могли бы реализовываться так, как описано выше в §19.3 и §23.2; то есть для коротких строк символы сохраняются непосредственно в дескрипторе, а для более длинных строк символы располагаются смежным образом в свободной памяти (как элементы контейнера **vector**). Контейнер **string** может расширяться (как это делает контейнер **vector**) в свободной памяти, избегая при этом лишних «перевыделений памяти» (reallocations):



Контейнер **array**, как и встроенный массив (§7.3), есть просто последовательность элементов (без лишнего дескриптора):



Это подразумевает, что локальный **array** вообще не использует свободную память (если только его самого там не расположили), и что его классовые элементы не предполагают операций со свободной памятью.

31.2.2. Требования к элементам

Чтобы объект мог быть элементом контейнера, нужно, чтобы его тип позволял внутреннему представлению контейнера копировать и перемещать объект, а также обменивать элементы местами. Если контейнер копирует элементы, используя копирующий конструктор или операцию присваивания, результат должен быть эквивалентным объектом. Это означает, что любое сравнение на равенство, применяемое к значениям элементов, должно констатировать равенство копии и оригинала. Другими словами, копирование элементов должно работать как обычное копирование для встроенного типа **int**. Аналогично, перемещающий конструктор (move constructor) и операция присваивания с семантикой перемещения (move assignment) должны стандартно определяться и иметь стандартную семантику перемещения (§17.5.1). Наконец, нужно иметь возможность стандартным образом обменивать элементы местами. Если тип допускает копирование и перемещение, функция **swap()** стандартной библиотеки будет работать.

Детали всех требований к элементам разбросаны по всему стандарту и это затрудняет прочтение такой информации (§ iso.23.2.3, § iso.23.2.1, §17.6.3.2), но в общем можно сказать, что контейнеры могут содержать элементы типов со стандартными операциями копирования или перемещения. Многие базовые алгоритмы, такие как **copy()**, **find()** и **sort()** будут работать до тех пор, пока выполняются требования к контейнерным элементам плюс дополнительные требования, специфичные для алгоритмов (например, чтобы элементы были отсортированы; §31.2.2.1).

Часть отклонений от требований стандартных контейнеров может обнаруживаться компилятором, но другая часть отклонений им не обнаруживается и может приводить к непредсказуемому поведению. Например, если копирующая операция может порождать исключение, то она может оставить элемент в частично скопированном состоянии. Это признак плохого дизайна (§13.6.1), порождающего нарушение базовых гарантий (§13.2). Элемент, оставленный в ненадлежащем состоянии, впоследствии может вызвать серьезные проблемы.

В тех случаях, когда копирование объектов не имеет смысла, в контейнер можно поместить указатели на объекты (вместо самих объектов). Типичным примером здесь служат полиморфные объекты (§3.2.2, §20.3.2). Например, мы используем **vector<unique_ptr<Shape>>** или **vector<Shape*>** вместо **vector<Shape>**, чтобы сохранить полиморфное поведение.

31.2.2.1. Сравнения

Ассоциативные контейнеры требуют, чтобы их элементы могли быть упорядочены. То же относится и ко многим операциям, которые применяются к контейнерам (например, **sort()** или **merge()**). По умолчанию, операция **<** используется для формирования порядка. Если же эта операция не подходит, программист должен предоставить альтернативу (§31.4.3, §33.4). Критерий упорядочения должен обеспечивать так называемое *строгое слабое упорядочение* (*strict weak ordering*). Неформально это означает, что «меньше» и «равно» должны быть транзитивными. То есть, для критерия упорядочения **cmp** (можно представлять себе его как «меньше чем») требуются:

1. Нерефлексивность: **cmp(x, x)** есть **false**.
2. Антисимметрия: **cmp(x, y)** предполагает **!cmp(y, x)**.
3. Транзитивность: если **cmp(x, y)** и **cmp(y, z)**, то **cmp(x, z)**.
4. Транзитивность эквивалентности: Определим **equiv(x, y)** как **!(cmp(x, y)||cmp(y, x))**. Тогда, если **equiv(x, y)** и **equiv(y, z)**, то **equiv(x, z)**.

Последнее правило позволяет определить равенство (**x==y**) как **!(cmp(x, y)||cmp(y, x))** если нужна операция **==**.

Операции стандартной библиотеки, требующие сравнения, имеются в двух версиях. Например:

```
template<typename Ran>
    void sort(Ran first, Ran last);              // используем < для сравнения
template<typename Ran, typename Cmp>
    void sort(Ran first, Ran last, Cmp cmp);     // используем cmp
```

Первая версия использует операцию **<**, а вторая применяет пользовательскую функцию сравнения **cmp**.

Например, мы можем решить сортировать тип **fruit** (то есть **фрукт**) при помощи операции сравнения, нечувствительной к регистру. Мы определяем для этого функциональный объект (§3.4.3, §19.2.2), который выполняет сравнение, будучи вызванным для пары строк типа **string**:

```
class Nocase {      // сравнение строк без учета регистра
public:
    bool operator()(const string&, const string&) const;
};

bool Nocase::operator()(const string& x, const string& y) const
    // вернуть true если x лексикографически меньше чем y (без учета регистра)
{
    auto p = x.begin();
    auto q = y.begin();

    while (p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q)) {
        ++p;
        ++q;
    }
    if (p == x.end()) return q!= y.end();
    if (q == y.end()) return false;
    return toupper(*p) < toupper(*q);
}
```

Мы можем вызывать **sort()**, используя этот критерий сравнения. Пусть имеется набор фруктов:

```
fruit:
    apple pear Apple Pear lemon
```

Сортировка вызовом **sort(fruit.begin(), fruit.end(), Nocase())** переупорядочит этот набор следующим образом:

```
fruit:
    Apple apple lemon Pear pear
```

Если в символьном наборе символы верхнего регистра предшествуют символам нижнего регистра, то простой вызов **sort(fruit.begin(), fruit.end())** даст такой результат:

```
fruit:
    Apple Pear apple lemon pear
```

Имейте в виду, что операция < для С-строк (то есть для типа **const char***) сравнивает значения указателей (§7.4). Из-за этого ассоциативные контейнеры работают не так, как ожидают многие люди в случаях, когда С-строки используются в качестве ключей. Чтобы заставить их работать надлежащим образом, нужно использовать операцию «меньше-чем», выполняющую сравнение на базе лексикографического порядка:

```
struct Cstring_less {
    bool operator()(const char* p, const char* q) const {return strcmp(p, q)<0;}
};

map<char*, int, Cstring_less> m;  // map использует strcmp() для срав. const char* ключей
```

31.2.2.2. Другие операции сравнения

По умолчанию, контейнеры и алгоритмы используют операцию <, когда им необходимо выполнить операцию сравнения «меньше-чем». Когда это не подходит, программист может определить свой собственный критерий. Однако, для сравнения на равенство отсуствует механизм передачи соответствующего критерия. Вместо этого, когда программист определяет свой критерий **cmp**, равенство проверяется с помощью двух сравнений. Пример:

```
if (x == y)                    // не выполняется там, где пользов. предоставил сравнение

if (!cmp(x, y) &&!cmp(y, x))   // выполняется, если пользов. предоставил cmp для сравнения
```

Это избавляет программиста от необходимости предоставлять сравнение на равенство для всех типов, используемых в качестве value-типов в ассоциативных контейнерах или в алгоритмах, использующих проверки на равенство. В то же время, это может показаться расточительным, но стандартная библиотека использует проверки на равенство не часто, к тому же в 50% таких случаев требуется всего лишь одно обращение к **cmp()**, да и компилятор может устранить двойную проверку.

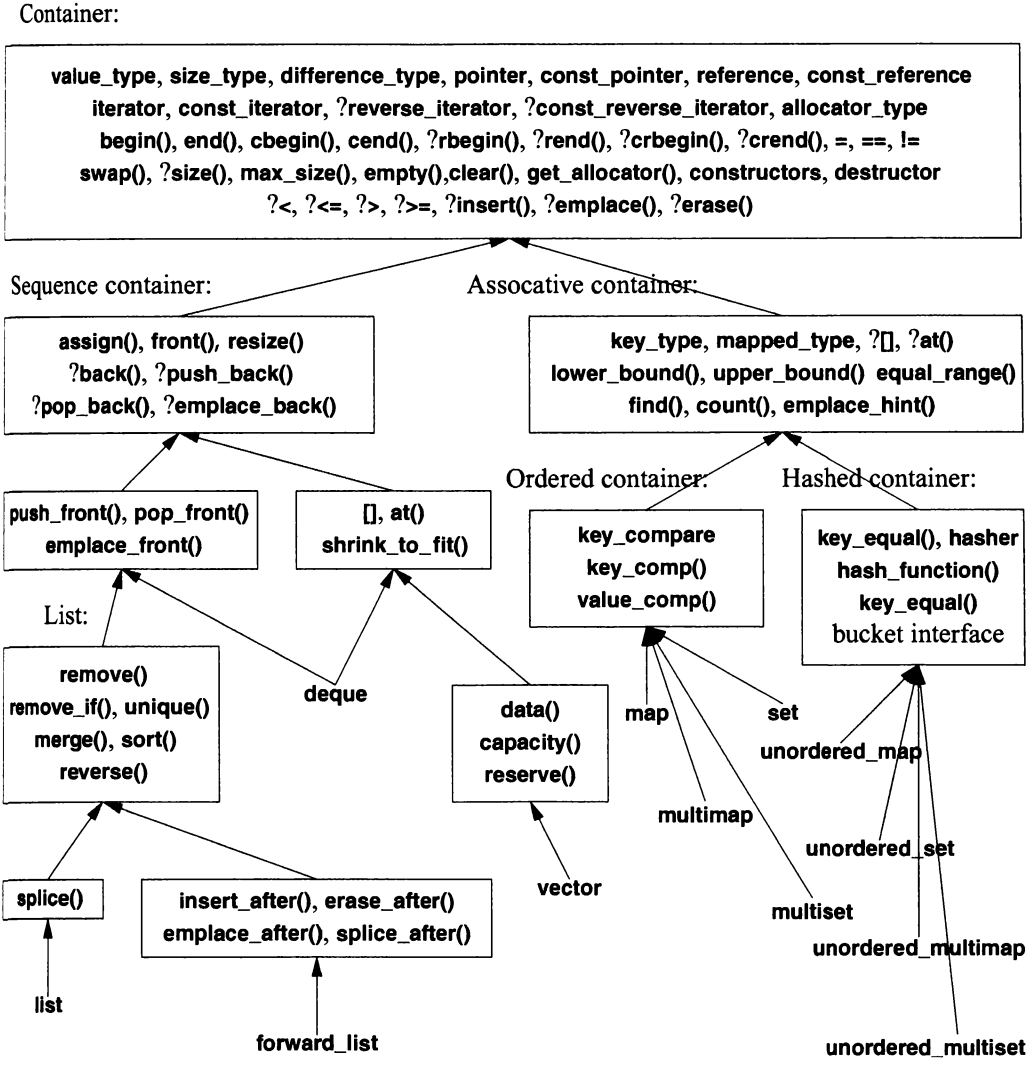
Помимо сравнения на равенство (по умолчанию операция ==) применяется также и проверка на эквивалентность на основе операции сравнения «меньше-чем» (операция < по умолчанию). Например, ассоциативные контейнеры (§31.4.3) сравнивают ключи, используя тест эквивалентности **!(cmp(x, y)||cmp(y, x))**. Из этого вытекает, что эквивалентные ключи не обязаны быть равными. Например, **multimap** (§31.4.3), который в качестве критерия сравнения применяет сравнение, нечувствительное к регистру, будет считать строки **Last**, **last**, **lAst** и **lasT** эквивалентными, в то время как операция == сочтет их отличающимися. Все это позволяет нам игнорировать различия, несущественные для сортировки.

Если тест на равенство (по умолчанию это операция ==) и тест на эквивалентность !(cmp(x, y)||(cmp(y, x)) (по умолчанию cmp() это <) всегда дают один и тот же результат, то говорят, что имеет место полное упорядочение (*total order*).

Имея < и ==, мы можем легко сконструировать остальные операции сравнения. Стандартная библиотека определяет их в пространстве имен **std::rel_ops** и размещает в файле <utility> (§35.5.3).

31.3. Обзор операций

Можно следующим образом суммировать операции и типы для стандартных контейнеров:



Стрелка показывает, какие операции доступны для контейнера (не имеет отношения к наследованию). Знак вопроса (?) означает, что я включил операции, доступные лишь некоторой части контейнеров. В частности:

- Ассоциативный контейнер с префиксом **multi**, а также **set** не имеют операций [] и **at()**.
- Контейнер **forward_list** не предоставляет **insert()**, **erase()** и **emplace()**; вместо них он предоставляет операции с суффиксом **_after**.
- Контейнер **forward_list** не обеспечивает **back()**, **push_back()**, **pop_back()** и **emplace_back()**.
- Контейнер **forward_list** не предоставляет **reverse_iterator**, **const_reverse_iterator**, **rbegin()**, **rend()**, **crbegin()**, **crend()** и **size()**.
- Ассоциативный контейнер с префиксом **unordered_** не обеспечивает **<**, **<=**, **>** и **>=**.

Операции [] и **at()** реплицируются просто для уменьшения количества стрелок.

«Интерфейс корзин (buckets)» описывается в §31.4.3.2.

Где это имеет смысл, операция доступа существует в двух версиях: одна для **const**-объектов, а другая для не-**const**-объектов.

Операции стандартной библиотеки имеют гарантии сложности:

Сложность операций стандартных контейнеров

	[]	List	Front	Back	Iterators
	§31.2.2	§31.3.7	§31.4.2	§31.3.6	§33.1.2
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
встроенный массив	const				Ran
valarray	const				Ran
bitset	const				

Front-операции подразумевают вставку/удалении перед первым элементом. Аналогично, Back-операции подразумевают вставку/удаление после последнего элемента; List-операции действуют не обязательно на концах контейнера.

В колонке *Итераторы* «Ran» означает «random-access iterator» (итератор с произвольным доступом); «For» означает «forward iterator» (прямой итератор) и «Bi» означает «bidirectional iterator» (двунаправленный итератор) (§33.1.4).

Остальные обозначения относятся к эффективности операций. Операции с отметкой **const** занимают время, не зависящее от количества элементов в контейнере; другим стандартным обозначением для этого случая служит **O(1)**. Соответственно, **O(n)** означает,

что операция занимает время, пропорциональное количеству задействованных элементов. Суффикс + означает, что в отдельных случаях цена операции значительно возрастает. Например, цена вставки элемента в **list** фиксирована (она помечена как **const**), в то время как такая же операция для **vector** требует перемещения элементов после точки вставки (так что она помечена как **O(n)**). В отдельных случаях приходится перемещать все элементы контейнера **vector** (добавлен +). Нотация с большой буквой **O** традиционна. Плюс же я добавляю для программистов, которым важна предсказуемость вдобавок к оценке средней производительности. Часто обозначение **O(n)+** озвучивают как «линейное время с амортизацией».

Естественно, если константа велика, она вполне может превысить небольшую цену, пропорциональную количеству элементов. И, тем не менее, для больших структур данных **const** можно трактовать как «дешево», **O(n)** — как «дорого», **O(log(n))** — как «более-менее дешево». Для умеренно больших значений **n** величина **O(log(n))**, где **log** означает логарифм по основанию 2, намного ближе к **const**, чем к **O(n)**. Например:

Примеры логарифмов					
n	16	128	1,024	16,384	1,048,576
log(n)	4	7	10	14	20
n*n	256	802,816	1,048,576	268,435,456	1.1e+12

Тем, кому цена операции важна, стоит приглядеться к этим цифрам повнимательнее. В частности, им нужно точно понять, какие элементы берутся для подсчета релевантного значения **n**. Ну а простой посыл таков: не связывайтесь с квадратичными алгоритмами для больших значений **n**.

Оценки для сложности и цены — это оценки сверху. Эти оценки служат для того, чтобы дать представление о том, что можно ожидать от реализации операций. Естественно, ничто не мешает тем, кто реализует операции, попробовать улучшить эти оценки в отдельных важных случаях.

Отметим, что оценка с «большим **O**» асимптотическая; то есть может потребоваться большое количество элементов для того, чтобы оценка сложности проявила себя в полной мере. Другие факторы, такие как цена отдельной операции над элементом, могут оказаться превалирующими. Например, проход по элементам контейнеров **vector** и **list** имеют одинковую оценку сложности **O(n)**. Однако ж для современных компьютерных архитектур переход к следующему элементу по отдельной ссылке на элемент (как в контейнере **list**) может оказаться намного более дорогостоящим нежели переход к следующему элементу контейнера **vector** (элементы которого располагаются в памяти смежным образом). Аналогично, линейный алгоритм может возрастать большими или меньшими темпами при росте элементов контейнера в зависимости от деталей реализации архитектуры процессора и памяти. Не доверяйте своей интуиции и общим представлениям о цене и сложности, а проводите прямые измерения. К счастью, интерфейсы контейнеров столь похожи, что закодировать сравнительные замеры несложно.

Операция **size()** имеет константную оценку для всех контейнеров. Отметим, что **forward_list** не имеет операции **size()**, так что если вам нужно узнать число элементов, придется самим сосчитать их (по цене **O(n)**). Контейнер **forward_list** оптимизирован по занимаемому объему памяти и не хранит своего размера или указателя на последний элемент.

Оценки в случае **string** актуальны для длинных строк. Оптимизация для коротких строк (§19.3.3) приводит к константной оценке для любых операций над короткими строками (к примеру, менее 14 символов).

Оценки для **stack** и **queue** отражают тот факт, что по умолчанию они реализуются с помощью **deque** в качестве нижележащего контейнера (§31.5.1, §31.5.2).

31.3.1. Типы, связанные с контейнерами

В контейнерах определяется набор следующих типов:

Типы (§ iso.23.2, § iso.23.3.6.1)	
value_type	Тип элемента
allocator_type	Тип аллокатора
size_type	Беззнаковый тип индексов, счетчиков, и т.д.
difference_type	Знаковый тип разности итераторов
iterator	Ведет себя подобно **value_type***
const_iterator	Ведет себя подобно **const_value_type***
reverse_iterator	Ведет себя подобно **value_type***
const_reverse_iterator	Ведет себя подобно **const_value_type***
reference	**value_type&**
const_reference	**const_value_type&**
pointer	Ведет себя подобно **value_type***
const_pointer	Ведет себя подобно **const_value_type***
key_type	Тип ключа; для ассоциативных контейнеров
mapped_type	Тип **mapped_value**; для ассоциативных контейнеров
key_compare	Тип критерия сравнения; для упорядоченных контейнеров
hasher	Тип хэш-функции; для неупорядоченных контейнеров
key_equal	Тип ф-ии эквивалентности; для неупорядоченных контейнеров
local_iterator	Тип итератора корзин; для неупорядоченных контейнеров
const_local_iterator	Тип итератора корзин; для неупорядоченных контейнеров

Каждый контейнер и «почти контейнер» определяет большинство из указанных типов. Естественно, что контейнер не определяет бессмысленных (для себя) типов. Например, **array** не имеет **allocator_type**, а **vector** не определяет **key_type**.

31.3.2. Конструкторы, деструкторы и присваивания

Контейнеры предоставляют множество конструкторов и операций присваивания. Для контейнера **C** (например, **vector<double>** или **map<string, int>**) имеем:

Конструкторы, деструкторы и присваивания **C** есть контейнер; а по умолчанию, **C** использует аллокатор **C::allocator_type{}**	
C c {};	Умолчательный конструктор: **c** – пустой контейнер
C c {a};	Сконструировать **c** по умолчанию; использовать аллокатор **a**
C c(n);	**c** инициализирован **n** эл-тами со значением **value_type{}**; не для ассоциативных контейнеров
C c(n,x);	Инициализировать **c n** копиями **x**; не для ассоциативных контейнеров
C c(n,x,a);	Инициализировать **c n** копиями **x**; использовать аллокатор **a**; не для ассоциативных контейнеров

Продолжение

Конструкторы, деструкторы и присваивания C есть контейнер; а по умолчанию, C использует аллокатор C::allocator_type{}	
C c {elem};	Инициализировать c из elem; если C имеет конструктор со списком инициализаторов, использовать его; иначе – другой конструктор
C c {c2};	Копирующий конструктор: копирует эл-ты и аллокатор из c2 в c
C c {move(c2)};	Перемещающий конструктор: перемещает эл-ты и аллокатор из c2 в c
C c {{elem}, a};	Инициализировать c из initializer_list {elem}; использовать аллокатор a
C c {b,e};	Инициализировать c элементами из [b:e)
C c {b,e,a};	Инициализировать c элементами из [b:e); использовать аллокатор a
c.~C()	Деструктор: уничтожить элементы c и освободить все ресурсы
c2=c	Копирующее присваивание: копировать эл-ты c в c2
c2=move(c)	Перемещающее присваивание: переместить эл-ты c в c2
c={elem}	Присваивание c из initializer_list {elem}
c.assign(n,x)	Присвоить n копий x; не для ассоциативных контейнеров
c.assign(b,e)	Присваивание c из [b:e)
c.assign({elem})	Присваивание c из initializer_list {elem}

Дополнительные конструкторы для ассоциативных контейнеров описываются в §31.4.3.

Заметим, что присваивания не копируют (или перемещают) аллокаторы. Целевой контейнер получает новый набор элементов, но сохраняет связь со старым контейнером, который используется для выделения памяти под новые элементы (если они есть). Аллокаторы описываются в §34.4.

Нужно помнить, что конструктор или операция копирования элементов могут генерировать исключения в случае невозможности выполнить возложенную на них задачу.

Потенциальные неоднозначности в инициализаторах обсуждаются в §11.3.3 и §17.3.4.1. Например:

```
void use()
{
    vector<int> vi {1,3,5,7,9};          // vector инициализируется пятью целыми
    vector<string> vs(7);                // vector инициализируется семью пустыми строками

    vector<int> vi2;
    vi2 = {2,4,6,8};                     // присваиваем последовательность из 4-х целых vi2
    vi2.assign(&vi[1],&vi[4]);           // присваиваем последовательность 3,5,7  vi2

    vector<string> vs2;
    vs2 = {"The Eagle", "The Bird and Baby"}; // присваиваем две строки vs2
    vs2.assign("The Bear", "The Bull and Vet"); // ошибка времени выполнения
}
```

Ошибка в указанном присваивании объекту **vs2** состоит в том, что передаются два указателя (а не **initializer_list**), которые не указывают на один и тот же массив. Используйте () для размерных инициализаторов и {} для любых видов итераторов.

Так как контейнеры чаще всего имеют большой размер, мы передаем их почти всегда по ссылке. Поскольку они являются дескрипторами ресурсов (§31.2.1) мы можем возвращать их (неявно используя перемещение) эффективно. Аналогично, мы можем перемещать их как аргументы, если хотим избежать создания синонимов.

Например:

```
void task(vector<int>&& v);

vector<int> user(vector<int>& large)
{
    vector<int> res;
    // ...
    task(move(large));          // перемещаем владение данными в task()
    // ...
    return res;
}
```

31.3.3. Размер и емкость

Размер — это количество элементов в контейнере; емкость — это количество элементов, которое может принять контейнер без дополнительного выделения памяти:

Размер и емкость	
x=c.size()	**x** есть число элементов **c**
c.empty()	Пусто ли **c**?
x=c.max_size()	**x** есть наибольшее возможное число эл-тов **c**
x=c.capacity()	**x** – объем, выделенный под **c**; только для **vector** и **string**
c.reserve(n)	Резервировать место под **n** элементов в **c**; только для **vector** и **string**
c.resize(n)	Изменить размер **c** до **n**; добавленные эл-ты получают знач-е по умолчанию; только для последовательных контейнеров (и **string**)
c.resize(n, v)	Изменить размер **c** до **n**; использовать **v** для добавленных элементов; только для последовательных контейнеров (и **string**)
c.shrink_to_fit()	Сделать **c.capacity()** равным **c.size()**; только для **vector**, **deque**, и **string**
c.clear()	Стереть все элементы **c**

При изменении размера или емкости элементы могут быть перемещены в новые участки памяти. Это подразумевает, что итераторы (а также указатели и ссылки) к элементам могут стать недействительными (будут относиться к старым позициям элементов). К примеру, см. §31.4.1.1.

Итератор элементов ассоциативного контейнера (например, **map**) становится недействительным только тогда, когда адресуемый им элемент удаляется из контейнера (операцией **erase()**; §31.3.7). Наоборот, итератор элементов последовательного контейнера (например, **vector**) становится недействительным, если элементы перемещаются на новое место (например, при операциях **resize()**, **reserve()** или **push_back()**) или если элемент, адресуемый итератором, перемещается внутри исходного блока памяти контейнера (например, посредством **erase()** или операцией **insert()** для элемента с малым индексом).

Может показаться соблазнительным улучшить производительность операцией **reserve()**, но стандартные стратегии роста для **vector** (§31.4.1.1) столь эффективны, что редко это бывает хорошей причиной. Вместо этого, рассматривайте **reserve()** как средство для увеличения предсказуемости производительности и для избежания ситуаций, когда итераторы становятся недействительными.

31.3.4. Итераторы

Контейнеры могут рассматриваться как последовательности, упорядоченные в соответствии с их итератором, или же в обратном к указанному порядке. Для ассо-

циативного контейнера направление упорядочения базируется на критерии сравнения (по умолчанию это <):

Итераторы	
p=c.begin()	p указывает на первый элемент c
p=c.end()	p указывает на эл-т c, следующий за последним
cp=c.cbegin()	p указывает на первый константный элемент c
p=c.cend()	p указывает на константный эл-т c, следующий за последним
p=c.rbegin()	p указывает на первый в обратном порядке элемент c
p=c.rend()	p указывает на эл-т c, следующий за последним в обратном порядке
p=c.crbegin()	p указывает на первый в обратном порядке константный элемент c
p=c.crend()	p указывает на константный эл-т c, следующий за последним в обратном порядке

Наиболее общей формой итерации по элементам является проход контейнера от его начала до самого конца. Проще всего это можно сделать оператором range-**for** (§9.5.1), который неявно использует **begin()** и **end()**. Например:

```
for (auto& x : v)    // неявное использование v.begin() и v.end()
    cout << x << '\n';
```

Когда нам нужно узнать позицию элемента в контейнере или когда нужно сослаться на более одного элемента за один раз, мы должны обратиться к итератором явным образом. В таких случаях использовать **auto** полезно с целью минимизации объема исходного кода и избежании, тем самым, опечаток. Например, пусть имеется итератор произвольного доступа:

```
for (auto p = v.begin(); p!=end(); ++p) {
    if (p!=v.begin() && *(p-1)==*p)
        cout << "duplicate " << *p << '\n';
}
```

Когда нет необходимости в модификации элементов, подходят **cbegin()** и **cend()**. То есть можно писать следующее:

```
for (auto p = v.cbegin(); p!=cend(); ++p) {    // используем const-итераторы
    if (p!=v.cbegin() && *(p-1)==*p)
        cout << "duplicate " << *p << '\n';
}
```

Для большинства контейнеров и большинства их реализаций повторяющееся использование **begin()** и **end()** не является проблемой для производительности, так что нет никакой необходимости переусложнять код вроде следующего:

```
auto beg = v.cbegin();
auto end = v.cend();

for (auto p = beg; p!=end; ++p) {
    if (p!=beg && *(p-1)==*p)
        cout << "duplicate " << *p << '\n';
}
```

31.3.5. Доступ к элементам

Некоторые элементы доступны напрямую:

Доступ к элементам	
c.front()	Ссылка на первый эл-т **c**; не для ассоциативных контейнеров
c.back()	Ссылка на последний элемент **c**; не для **forward_list** или ассоциативных контейнеров
c[i]	Ссылка на **i**-ый эл-т **c**; неограниченный доступ; не для списков или ассоциативных контейнеров
c.at(i)	Ссылка на **i**-ый эл-т **c**; генерирует **out_of_range**, если **i** за границей диапазона; не для списков или ассоциативных контейнеров
c[k]	Ссылка на элемент **c** с ключом **k**; вставить (**k**, **mapped_type{}**), если не найдено; только для **map** и **unordered_map**
c.at(k)	Ссылка на **i**-ый эл-т **c**; генерирует **out_of_range**, если **k** не найдено; только для **map** и **unordered_map**

Отдельные реализации — особенно отладочные версии — всегда выполняют проверку выхода за границы диапазона, но на это нельзя рассчитывать в обязательном порядке, а также нельзя полагаться на отсутствие такой проверки в целях повышения производительности. Когда все это очень важно, нужно детальнее изучить особенности реализации.

У ассоциативных контейнеров **map** и **unordered_map** операции **[]** и **at()** принимают в качестве аргументов ключи, а не позиции элементов (§31.4.3).

31.3.6. Стековые операции

Стандартные контейнеры **vector**, **deque** и **list** (но не **forward_list** или ассоциативные контейнеры) предоставляют эффективные операции на конце последовательности элементов:

Стековые операции	
c.push_back(x)	Добавить **x** в **c** (копируя или перемещая) после последнего эл-та
c.pop_back()	Удалить последний элемент **c**
c.emplace_back(args)	Добавить объект, созданный из **args** в **c** после последнего эл-та

Операция **c.push_back(x)** копирует (или перемещает) **x** в **c**, увеличивая размер **c** на единицу. Если мы выйдем за границы памяти, или же если копирующий конструктор для объекта **x** сгенерирует исключение, то **c.push_back(x)** окончится неудачей. При этом это не отразится на состоянии контейнера: здесь имеет место сильная гарантия (§13.2).

Отметим, что **pop_back()** не имеет возвращаемого значения. Если бы это было не так, генерация исключения копирующим конструктором серьезным образом усложнила бы реализацию.

Контейнеры **list** и **deque** предоставляют эквивалентные операции в начале (front) последовательности элементов (§31.4.2). То же имеет место и для **forward_list**.

Операция **push_back()** предпочтительна для наращивания контейнера без предварительного выделения достаточной памяти и с хорошим шансом избежать переполнения, но **emplace_back()** можно использовать аналогично. Например:

```
vector<complex<double>> vc;
for (double re, im; cin>>re>>im;)  // читаем две переменные типа double
    vc.emplace_back(re, im);       // добавляем complex<double>{re, im} в конец
```

31.3.7. Операции, типичные для списков

Контейнеры предоставляют операции, типичные для списков:

Операции со списками	
q=c.insert(p, x)	Добавить x перед p; применяя копирование или перемещение
q=c.insert(p, n, x)	Добавить n копий x перед p; если c – ассоциативный контейнер, p подсказывает, откуда начать поиск
q=c.insert(p, first, last)	Добавить эл-ты из [first:last) перед p; не для ассоциативных контейнеров
q=c.insert(p,{elem})	Добавить эл-ты из initializer_list {elem} перед p; p подсказывает, откуда начать поиск места для вставки; только для упорядоченных ассоциативных контейнеров
q=c.emplace(p, args)	Добавить элемент, созданный из args перед p; не для ассоциативных контейнеров
q=c.erase(p)	Удалить элемент, указываемый p, из c
q=c.erase(first, last)	Удалить [first:last) из c
c.clear()	Удалить все элементы c

Для функций **insert()**, результат, **q**, указывает на последний из вставленных элементов. Для функций **erase()**, **q** указывает на элемент, следующий за последним из удаленных.

У контейнеров с непрерывным распределением памяти под элементы, такими как **vector** и **deque**, вставка и удаление элемента может вызвать перемещение элементов. Итератор, показывавший на элемент, подвергающийся перемещению, становится недействительным. Элемент перемещается, если его позиция находится позади точки вставки/удаления или же все элементы перемещаются из-за того, что новый размер превышает предыдущую емкость. Например:

```
vector<int> v {4,3,5,1};
auto p = v.begin()+2;      // указывает на v[2], т.е. 5
v.push_back(6);            // p становится недействительным; v == {4,3,5,1,6}
p = v.begin()+2;           // указывает на v[2], т.е., 5
auto p2 = v.begin()+4;     // p2 указывает на v[4], т.е., 6
v.erase(v.begin()+3);      // v == {4,3,5,6}; p действительный; p2 недействительный
```

Любая операция, добавляющая элемент в вектор, может вызвать перемещение для всех элементов (§13.6.4).

Операция **emplace()** применяется тогда, когда выглядит неуклюже или же потенциально неэффективно сначала создавать объект, а затем копировать (перемещать) его в контейнер. Вот пример:

```
void user(list<pair<string, double>>& lst)
{
    auto p = lst.begin();
    while (p!=lst.end()&& p->first!="Denmark")          // ищем точку вставки
        /* ничего не делаем */;
    p=lst.emplace(p,"England",7.5);                     // чудесно
    p=lst.insert(p, make_pair("France",9.8));           // helper-функция
    p=lst.insert(p, pair<string, double>>{"Greece",3.14});  // мощно
}
```

Контейнер **forward_list** не предоставляет операций, таких как **insert()**, которые оперируют перед элементом, указуемым итератором. Такие операции не могут быть реализованы, поскольку отсутствует общий способ нахождения предыдущего элемента в **forward_**

list, располагая лишь итератором. Вместо этого, forward_list предоставляет операции, такие как insert_after(), которые оперируют после элемента, указуемого итератором. Аналогично, неупорядоченные контейнеры используют emplace_hint() для запроса операции (hint), а не для прямолинейного вызова «простого» emplace().

31.3.8. Прочие операции

Контейнеры можно сравнивать и обменивать (местами):

Сравнение и обмен	
c1==c2	Равны ли все соответствующие эл-ты c1 и c2?
c1!=c2	!(c1==c2)
c1<c2	c1 лексикографически меньше c2?
c1<=c2	!(c2<c1)
c1>c2	c2<c1
c1>=c2	!(c1<c2)
c1.swap(c2)	Обменивает значения c1 и c2; noexcept
swap(c1, c2)	c1.swap(c2)

Когда контейнеры сравниваются операцией (например, <=), элементы сравниваются при помощи операции эквивалентности элементов, построенной на основе == или < (то есть сравнение a>b выполняется, используя !(b<a)).

Операции swap() обменивают как элементы, так и аллокаторы.

31.4. Контейнеры

Здесь мы углубимся в детали, касающиеся:

- vector, контейнер по умолчанию (§31.4.1)
- Связанные списки: list и forward_list (§31.4.2)
- Ассоциативные контейнеры, такие как map и unordered_map (§31.4.3)

31.4.1. vector

STL-контейнер vector является контейнером по умолчанию. Используйте именно этот контейнер, если только нет явных причин не делать этого. Подумайте дважды, если вы вместо vector решили применить list или встроенный массив.

В §31.3 операции над vector неявно сравниваются с операциями, предоставляемыми для других контейнеров. Здесь же, констатируя важность контейнера vector, мы сфокусируемся на том, как именно предоставляются операции.

Шаблонный параметр у vector и его внутренние типы определяются следующим образом:

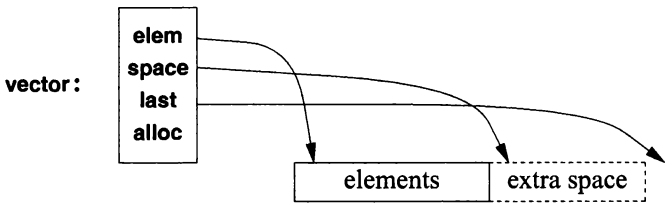
```
template<typename T, typename Allocator = allocator<T>>
class vector {
public:
    using reference = value_type&;
    using const_reference = const value_type&;
    using iterator = /* зависит от рализации */;
    using const_iterator = /* зависит от реализации */;
    using size_type = /* зависит от реализации */;
    using difference_type = /* зависит от реализации */;
    using value_type = T;
```

```
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // ...
};
```

31.4.1.1. vector и его рост

Рассмотрим схему расположения объекта типа **vector** в памяти (§13.6):



Одновременное использование и размера (числа элементов), и емкости (числа имеющихся ячеек под элементы без необходимости перевыделения памяти) делает рост контейнера посредством операции **push_back()** разумно эффективным: нет необходимости в выделении памяти каждый раз при добавлении элемента, а лишь в тех более редких случаях, когда исчерпывается емкость (§13.6). Стандарт не говорит о том, насколько следует увеличивать емкость в случаях ее исчерпания, но типичная практика состоит в добавлении половины от текущей емкости. В начале моего погружения во внутренности **vector** я был озабочен применением **reserve()**. Но в конце концов я был поражен тем, что в большинстве случаев вызов **reserve()** ничего существенно не улучшал в вопросе производительности. Стратегия роста контейнера **vector**, заданная по умолчанию, обеспечивала все те оценки производительности, которые я добивался, использую **reserve()**. В итоге, я перестал пытаться улучшить производительность посредством вызова **reserve()**. Я стал использовать этот вызов лишь для увеличения предсказуемости задержек, связанных с перевыделением памяти, и для предотвращения порчи указателей и итераторов.

Понятие емкости позволяет итераторам в контейнере **vector** оставаться актуальными до тех пор, пока на самом деле не произойдет перевыделение памяти. Рассмотрим пример со считыванием букв в буфер и с отслеживанием границ слов:

```
vector<char> chars;             // входной буфер для символов
constexpr int max = 20000;
chars.reser ve(max);
vector<char*> words;            // указатели на начала слов

bool in_word = false;
for (char c; cin.get(c)) {
    if (isalpha(c)) {
        if (!in_word) {         // находим начало слова
            in_word = true;
            chars.push_back(0);     // конец предыдущего слова
            chars.push_back(c);
            words.push_back(&chars.back());
        }
        else
            chars.push_back(c);
```

```
        }
        else
            in_word = false;
    }
    if (in_word)
        chars.push_back(0);       // завершение предыдущего слова

    if (max<chars.size()) {    // упс: символы типа char растут вне емкости; слова недействительны
        // ...
    }
    chars.shrink_to_fit();       // освобождаем излишнюю емкость
```

Если бы я здесь не использовал **reserve()**, указатели в **words** стали бы недействительными, если вызов **chars.push_back()** привел бы к перевыделению памяти. Под термином «недействительные» я понимаю тот факт, что применение этих указателей могло бы породить непредсказуемое поведение. Они могли бы — а может и не могли — указывать на элемент, но практически наверняка не на тот элемент, на который они указывали до перевыделения памяти.

Возможность наращивать **vector** операцией **push_back()** и другими операциями означает, что применение низкоуровневых С-операций **malloc()** и **realloc()** (§43.5) лишь добавляет ненужные сложности и чревато ошибками.

31.4.1.2. vector и проблема вложенности

Контейнер **vector** (как и другие контейнеры, опирающиеся на структуры данных с непрерывным распределением памяти) имеют три важных преимущества перед иными структурами данных:

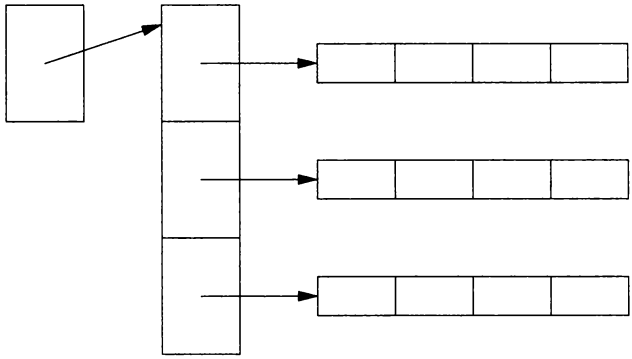
- Элементы у **vector** хранятся компактно: нет поэлементных излишков памяти. Количество памяти, потребляемой объектом **vec** типа **vector<X>**, приблизительно равно **sizeof(vector<X>)+vec.size()*sizeof(X)**. При этом **sizeof(vector<X>)** примерно равно 12 байтам, что абсолютно незначительно для больших контейнеров.
- Проход по элементам контейнера **vector** очень быстрый. Для перехода к следующему элементу не нужно выполнять косвенное обращение по указателю, а современные машинные архитектуры оптимизированы для последовательного прохода по подобным структурам данных. Это делает линейное сканирование элементов у **vector**, таких как в **find()** и **copy()**, близким к оптимальному.
- Для **vector** произвольный доступ прост и эффективен. Это делает эффективными многие алгоритмы для **vector**, такие как **sort()**, и **binary_search()**.

Нельзя недооценить эти преимущества. Например, двусвязанный список, такой как **list**, обычно накладывает поэлементный излишек памяти в четыре слова (four-words overhead; две связи плюс зазор памяти, связанный с динамическим выделением памяти в куче), а проход по его элементам может быть на порядок более дорогим, чем это имеет место у **vector** при эквивалентных данных. Эффект столь впечатляющий и столь поразительный, что я рекомендую вам проверить это самим [Stroustrup, 2012a].

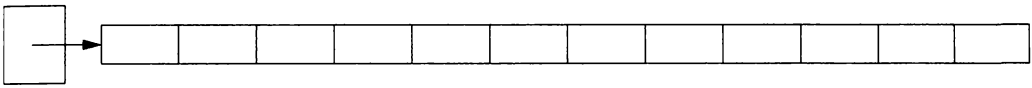
Но преимущества компактности и эффективности можно ненароком и скомпрометировать. Рассмотрим как можно представить двумерную матрицу. Существуют два очевидных решения:

- Строим **vector** с элементами типа **vector**: **vector<vector<double>>** с индексированием **m[i][j]** в **C**-стиле
- Создаем специфический матричный тип: **Matrix<2, double>** (Глава 29), который хранит элементы смежным образом (например, **vector<double>**) и вычисляет местоположения в этом **vector** с помощью пары индексов: **m(i, j)**

Схема памяти для 3x4 **vector<vector<double>>** выглядит следующим образом:



А вот как выглядит схема памяти для **Matrix<2, double>**:



Для конструирования **vector<vector<double>>** требуются четыре вызова конструкторов с четырьмя операциями выделения памяти. Для доступа к элементу нужно дважды косвенно обращаться к памяти.

Для конструирования **Matrix<2, double>** нужен один вызов конструктора с единственным выделением памяти в куче. Доступ к элементу требует единственного косвенного обращения к памяти.

Получив доступ к элементу в строке матрицы, мы не нуждаемся в дополнительном косвенном обращении к памяти, так что доступ для **vector<vector<double>>** не всегда в два раза дороже доступа для **Matrix<2, double>**. Тем не менее, у алгоритмов, которые требуют высокой эффективности, выделение памяти, ее освобождение и цена доступа, имеющие место для **vector<vector<double>>**, могут стать проблемой.

Конечно, тип **vector<vector<double>>** позволяет иметь разные размеры строк, и это может стать преимуществом в некоторых случаях, но чаще это приводит лишь к ошибкам и проблемам при тестировании.

Эти проблемы и дополнительные затраты ресурсов становятся еще большими в случае больших размерностей: сравните затраты для **vector<vector<vector<double>>>** и **Matrix<3, double>**.

Подводим итоги. Я отметил, что важность компактности структуры данных часто недооценивается, или же она может быть скомпрометирована. Преимущества вытекают из ее логики, равно как и эффективность. Добавьте это к тенденции излишнего использования указателей и операций **new**, и вот мы имеем распространенную проблему. К примеру, рассмотрите сложности разработки, дополнительные затраты во время выполнения, нагрузку на память и потенциальные ошибки при релизации двумерных структур, когда строки представляются на куче независимыми объектами типа **vector<vector<double>*>**.

31.4.1.3. vector и массивы

Контейнер **vector** реализуется через дескриптор ресурсов. Это позволяет ему иметь переменный размер и обеспечивает эффективную семантику перемещения. Однако, это же и порождает меньшую его эффективность по сравнению со структурами данных

(такими как встроенные массивы и класс **array**), в которых элементы не хранятся отдельно от дескриптора. Например, хранение последовательности элементов в стеке или в ином объекте может дать выигрыш в производительности (но это может сопровождаться и недостатками).

Контейнер **vector** имеет дело с надежно проинициализированными объектами. Это упрощает нашу работу с такими контейнерами и позволяет рассчитывать на надлежащую очистку ресурсов при уничтожении элементов. Это же приводит и к отдельным недостаткам по сравнению с более простыми структурами данных (встроенные массивы и класс **array**), которые допускают неинициализированные элементы.

К примеру, нет нужды инициализировать элементы массива типа **array** перед началом считывания в них данных (из потока ввода):

```
void read()
{
    array<int, MAX]> a;
    for (auto& x : a)
        cin.get(&x);
}
```

Для **vector** мы могли бы использовать **emplace_back()** для достижения аналогичного эффекта (без необходимости специфицировать **MAX**).

31.4.1.4. vector и string

Тип **vector<char>** представляет собой смежную последовательность элементов типа **char**, с возможностью изменения размера, но таков же и тип **string**. Как сделать выбор между этими типами?

Тип **vector** есть просто общий механизм для хранения значений любых типов. Он не делает предположений о смысле хранимых значений. Для контейнера **vector<char>** текстовая строка **Hello, World!** — это просто 13 элементов типа **char**. Сортировка этой строки приводит к **!,Hdellloor** (с лидирующим пробелом), что является штатным результатом. В то же время, тип **string** предназначен именно для хранения последовательностей символов. Взаимосвязи между символами существенны для типа **string**. К примеру, мы редко выполняем в нем простую сортировку символов, так как это разрушает смысл содержимого. Некоторые операции для типа **string** отражают этот факт (например, **c_str()**, **>>** и **find()** осведомлены о том, что С-строки имеют терминальный нуль). Реализации типа **string** также отвечают предположениям о том, как мы используем объекты типа **string**. Например, оптимизация коротких строк (§19.3.3) была бы излишне пессимистичной, если бы не факт, что в реальных программах используется множество таких строк, так что минимизация обращения к свободной памяти адекватна.

Следует ли в таком случае иметь еще и «оптимизацию коротких векторов»? Я подозреваю, что не следует, но для большей уверенности необходимо иметь обширный массив экспериментальных замеров.

31.4.2. Списки

В рамках STL имеются два типа связанных списков:

- **list**: двухсвязанный список
- **forward_list**: односвязанный список

Контейнерный тип **list** это последовательность, оптимизированная для вставок и удаления элементов. Вставка элемента в **list** или удаление элемента из него не влияет на рас-

положение в памяти других элементов. В частности, при этом не происходит порчи итераторов на не вовлеченные в указанные операции элементы.

По сравнению с **vector** индексирование является чрезвычайно болезненной операцией, так что индексирование для списков даже не реализуется. При необходимости используйте **advance()** и аналогичные операции для навигации по спискам (§33.1.4). Списки можно просматривать при помощи итераторов: **list** предоставляет двунаправленные итераторы (§33.1.2), а **forward_list** предоставляет прямые итераторы.

По умолчанию элементы в **list** расположены в памяти по-отдельности и включают в себя указатели на предшествующий и последующий элементы (§11.2.2). По сравнению с **vector**, контейнеры типа **list** используют больше памяти на каждый элемент (по крайней мере на четыре машинных слова больше), а проход по элементам (итерации) ощутимо более медленные, поскольку используют косвенные обращения к памяти по указателям вместо последовательного доступа.

Контейнер **forward_list** это односвязанный список. Относитесь к нему как к структуре данных, оптимизированной для пустых или же очень коротких списков, которые в типичном случае обходятся в направлении от начала списка. Ради компактности **forward_list** даже не реализует операцию **size()**; пустой **forward_list** занимает в памяти всего лишь одно машинное слово. Если вам нужно узнать число элементов, сосчитайте их сами. В случае, если такой подсчет для не слишком малых контейнеров типа **forward_list** окажется чересчур дорогостоящим, придется отказаться от этого типа и использовать иные контейнеры.

За исключением индексирования, управления емкостью и операцией **size()** для **forward_list** STL-списки предоставляют те же типы и операции, что и **vector** (§31.4). Кроме того, **list** и **forward_list** дополнительно предоставляют специфичные для списков функции-члены:

Операции, общие для list<T> и forward_list<T> (§ iso.23.3.4.5, § iso.23.3.5.4)	
lst.push_front(x)	Добавить **x** в **lst** (копируя или перемещая) перед первым эл-том
lst.pop_front()	Удалить первый элемент из **lst**
lst.emplace_front(args)	Добавить **T{args}** в **lst** перед первым эл-том
lst.remove(v)	Удалить все элементы из **lst** со значением **v**
lst.remove_if(f)	Удалить все эл-ты из **lst**, для которых **f(x)==true**
lst.unique()	Удалить одинаковые соседние эл-ты из **lst**
lst.unique(f)	Удалить одинаковые соседние эл-ты из **lst**, используя **f** для сравнения
lst.merge(lst2)	Объединить упорядоченные списки **lst** и **lst2**, используя **<** для упорядочения; **lst2** вносится в **lst** и в процессе этого очищается
lst.merge(lst2, f)	Объединить упорядоченные списки **lst** и **lst2**, используя **f** для упорядочения; **lst2** вносится в **lst** и в процессе этого очищается
lst.sort()	Сортировать **lst**, используя **<** для упорядочения
lst.sort(f)	Сортировать **lst**, используя **f** для упорядочения
lst.reverse()	Изменить порядок эл-тов **lst** на противоположный; noexcept

В противоположность общим алгоритмам **remove()** и **unique()** (§32.5), алгоритмы списков влияют на их размер. Вот пример:

```
void use()
{
    list<int> lst {2,3,2,3,5};
    lst.remove(3);              // lst теперь {2,2,5}
    lst.unique();               // lst теперь {2,5}
    cout << lst.size() << '\n'; // пишет 2
}
```

Алгоритм **merge()** демонстрирует стабильность, то есть эквивалентные элементы сохраняют свой относительный порядок следования.

Операции list<T> (§ iso.23.3.5.5) p указывает на элемент lst или lst.end()	
lst.splice(p, lst2)	Вставить эл-ты lst2 перед p; lst2 становится пустым
lst.splice(p, lst2, p2)	Вставить эл-т, указываемый p2, в lst2 перед p; эл-т, указываемый p2, удаляется из lst2
lst.splice(p, lst2, b, e)	Вставить эл-ты [b:e) из lst2 перед p; элементы [b:e) удаляются из lst2

Операция **splice()** не копирует значения элементов и не делает итераторы недействительными. Например:

```
list<int> lst1 {1,2,3};
list<int> lst2 {5,6,7};

auto p = lst1.begin();
++p;                    // p указывает на 2

auto q = lst2.begin();
++q;                    // q указывает на 6

lst1.splice(p, lst2);   // lst1 теперь {1,5,6,7,2,3}; lst2 теперь {}
                        // p по-прежнему указывает на 2 и q по-прежнему указывает на 6
```

Контейнер **forward_list** не имеет доступа к элементу, предшествующему тому элементу, который указуется итератором (у него отсутствует явная связь с предшествующим элементом), так что операции **emplace()**, **insert()**, **erase()** и **splice()** оперируют с позициями после позиции итератора:

Операции for forward_list<T> (§ iso.23.3.4.6)	
p2=lst.emplace_after(p, args)	Добавить элемент, созданный из args, после p; p2 указывает на новый элемент
p2=lst.insert_after(p, x)	Вставить x после p; p2 указывает на новый элемент
p2=lst.insert_after(p, n, x)	Вставить n копий x после p; p2 указывает на последний из новых элементов
p2=lst.insert_after(p, b, e)	Вставить [b:e) после p; p2 указывает на последний из новых эл-тов
p2=lst.insert_after(p,{elem})	Вставить {elem} после p; p2 указывает на последний из новых эл-тов; elem есть initializer_list
p2=lst.erase_after(p)	Удалить элемент, находящийся после p; p2 указывает на элемент после p или lst.end()
p2=lst.erase_after(b, e)	Удалить [b:e); p2=e
lst.splice_after(p, lst2)	Переместить lst2 на место после p
lst.splice_after(p, b, e)	Переместить [b:e) на место после p
lst.splice_after(p, lst2, p2)	Переместить p2 на место после p; удалить p2 из lst2
lst.splice_after(p, lst2, b, e)	Переместить [b:e) на место после p; удалить [b:e) из lst2

Эти списковые операции стабильны, то есть они сохраняют относительный порядок следования элементов с эквивалентными значениями.

31.4.3. Ассоциативные контейнеры

Ассоциативные контейнеры обеспечивают поиск элементов по ключу. Существует две разновидности ассоциативных контейнеров:

- *Упорядоченные ассоциативные контейнеры* — выполняют поиск на базе критерия упорядочения, по умолчанию < (операция «меньше чем»). Они реализованы как сбалансированные двоичные деревья, обычно красно-черные деревья.
- *Неупорядоченные ассоциативные контейнеры* — поиск ведется с помощью хэш-функций. Реализуются как хэш-таблицы с дополнительным связыванием (with linked overflow).

Обе эти разновидности представлены как в контейнерах **set** (последовательности пар {ключ, значение}), так и в контейнерах **map** (нет значений, или можно сказать, что ключ он и есть значение).

Наконец, все они, упорядоченные и неупорядоченные, присутствуют в двух вариантах:

- (plain) Без приставки **multi** — уникальный вход (значение) для каждого ключа
- С приставкой **multi** — возможно множество входов (значений) для каждого ключа

Имя ассоциативного контейнера отражает его положение в таком 3-мерном наборе: {set|map, plain|unordered, plain|multi}.

Ассоциативные контейнеры (§ iso.23.4.1, § iso.23.5.1)			
set	**multiset**	**unordered_set**	**unordered_multiset**
map	**multimap**	**unordered_map**	**unordered_multimap**

Их шаблонные аргументы описываются в §31.4.

По своей внутренней организации **map** и **unordered_map** различаются очень сильно. Смотри §31.2.1, где представлена наглядная графическая схема. В частности, **map** использует свой критерий сравнения (операция < в типичном случае) ключей при поиске на сбалансированном дереве (операция с гарантией **O(log(n))**), в то время как **unordered_map** применяет к ключу хэш-функцию для поиска слота в хэш-таблице (операция с гарантией **O(1)** для хорошо подобранной хэш-функции).

31.4.3.1. Упорядоченные ассоциативные контейнеры

Вот шаблонные параметры и типы членов для **map**:

```
template<typename Key,
    typename T,
    typename Compare = less<Key>,
    typename Allocator = allocator<pair<const Key, T>>>
class map {
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = pair<const Key, T>;
    using key_compare = Compare;
    using allocator_type = Allocator;
    using reference = value_type&;
    using const_reference = const value_type&;
    using iterator = /* зависит от реализации */;
    using const_iterator = /* зависит от реализации */;
    using size_type = /* зависит от реализации */;
    using difference_type = /* зависит от реализации */;
```

```
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    class value_compare {/* operator()(k1, k2) выполняет key_compare()(k1, k2) */};
    // ...
};
```

Дополнительно к конструкторам, рассмотренным в §31.3.2, у ассоциативных массивов есть конструкторы, позволяющие программисту предоставить компаратор:

Конструкторы map<K, T, C, A> (§ iso.23.4.4.2)	
`map m {cmp, a};`	Создать m, использующий сравнение **cmp** и аллокатор **a**; explicit
`map m {cmp};`	`map m {cmp, A{}};` explicit
`map m {};`	`map m {C{}};` explicit
`map m {b, e, cmp, a};`	Создать **m**, использующий сравнение **cmp** и аллокатор **a**; проинициализировать эл-тами из **[b:e)**
`map m {b, e, cmp};`	`map m {b, e, cmp, A{}};`
`map m {b, e};`	`map m {b, e, C{}};`
`map m {m2};`	Копирующий и перемещающий конструкторы
`map m {a};`	Создать **map** по умолчанию; использовать аллокатор **a**; explicit
`map m {m2, a};`	Создать, копируя или перемещая **m** из **m2**; использовать аллокатор **a**
`map m {{elem}, cmp, a};`	Создать **m**, использующий сравнение **cmp** и аллокатор **a**; проинициализировать эл-тами из **initializer_list {elem}**
`map m {{elem}, cmp};`	`map m {{elem}, cmp, A{}};`
`map m {{elem}};`	`map m {{elem}, C{}};`

Например:

```
map<string, pair<Coordinate, Coordinate>> locations
    {
        {"Copenhagen",{"55:40N","12:34E"}},
        {"Rome",{"41:54N","12:30E"}},
        {"New York",{"40:40N","73:56W"}
    };
```

Ассоциативные контейнеры предоставляют множество операций вставки и поиска:

Операции с ассоциативными контейнерами (§ iso.23.4.4.1)	
`v=c[k]`	**v** – ссылка на элемент с ключом **k**; если **k** не найден, **{k, mapped_type{}}** вставляется в **c**; только для **map** и **unordered_map**
`v=c.at(k)`	**v** – ссылка на элемент с ключом **k**; если **k** не найден, генерируется **out_of_range**; только для **map** и **unordered_map**
`p=c.find(k)`	**p** указывает на первый эл-т с ключом **k** или **c.end()**
`p=c.lower_bound(k)`	**p** указывает на первый эл-т с ключом **>=k** или **c.end()**; только для упорядоченных контейнеров
`p=c.upper_bound(k)`	**p** указывает на первый эл-т с ключом **>k** или **c.end()**; только для упорядоченных контейнеров
`pair(p1, p2)=c.equal_range(k)`	**p1=c.lower_bound(k); p2=c.upper_bound(k)**

Продолжение

Операции с ассоциативными контейнерами (§ iso.23.4.4.1)	
pair(p, b)=c.insert(x)	x есть value_type или что-то, что может быть скопировано в value_type (напр., двухэлементный tuple); b становится true, если x было добавлено или false, если уже есть вход с ключом для x; p указывает на (возможно, новый) эл-т с ключом для x
p2=c.insert(p, x)	x есть value_type или что-то, что может быть скопировано в value_type (напр., двухэлементный tuple); p подсказывает, где начать поиск эл-та с ключом для x; p2 указывает на (возможно, новый) эл-т с ключом для x
c.insert(b, e)	c.insert(*p) для каждого p в [b:e)
c.insert({args})	Вставить каждый эл-т initializer_list args; элемент имеет тип pair<key_type, mapped_type>
p=c.emplace(args)	p указывает на объект типа value_type из c, созданный из args и вставленный в c
p=c.emplace_hint(h, args)	p указывает на объект типа value_type из c, созданный из args и вставленный в c; h – итератор в c, возможно, подсказывающий, где искать место для нового входа
r=c.key_comp()	r есть копия объекта сравнения ключей; только для упорядоченных контейнеров
r=c.value_comp()	r есть копия объекта сравнения ключей; только для упорядоченных контейнеров
n=c.count(k)	n – число элементов с ключом k

Операции, специфичные для неупорядоченных контейнеров, рассматриваются в §31.4.3.5.

Если ключ k не отыскивается при выполнении операции индексирования, m[k], используется умолчательное значение. Например:

```
map<string, string> dictionary;

dictionary["sea"]="large body of water";      // вставка или присваивание элементу

cout << dictionary["seal"];                   // читаем значение
```

Если seal не находится в dictionary, ничего не выводится; вместо seal подставляется пустая строка и она же возвращается в качестве результата поиска.

Если такое поведение нежелательно, то можно напрямую использовать find() и insert():

```
auto q = dictionary.find("seal");     // ищем; не вставляем

if (q==dictionary.end()) {
    cout << "entry not found";
    dictionary.insert(make_pair("seal","eats fish"));
}
else
    cout q->second;
```

На самом деле, [] это чуть больше, чем просто удобное обозначение для insert(). Результат m[k] эквивалентен результату вычисления выражения (*(m.insert(make_pair(k,V{})).first)).second, где V это тип отображенных значений (mapped_type).

Обозначение insert(make_pair()) несколько многословное. Вместо этого можно использовать emplace():

```
dictionary.emplace("sea cow","extinct");
```

В зависимости от качества оптимизатора это может оказаться и более эффективным.

При попытке вставить значение в **map** в том случае, когда там уже имеется значение с его ключом, ничего не происходит (**map** остается без изменения). Если же вы хотите иметь несколько значений с одним и тем же ключом, применяйте **multimap**.

Первый итератор в паре **pair** (§34.2.4.1), возвращаемой вызовом **equal_range()**, соответствует **lower_bound()**, а второй соответствует **upper_bound()**. Вы можете вывести все значения для ключа **"apple"** в **multimap<string, int>** следующим образом:

```
multimap<string, int> mm {{"apple",2}, {"pear",2}, {"apple",7}, {"orange",2}, {"apple",9}};

const string k {"apple"};
auto pp = mm.equal_range(k);
if (pp.first==pp.second)
    cout << "no element with value '" << k << "'\n";
else {
    cout << "elements with value '" << k << "':\n";
    for (auto p=pp.first; p!=pp.second; ++p)
        cout << p->second << ' ';
}
```

В результате мы получаем **279**.

Вместо показанного выше кода можно использовать и

```
auto pp = make_pair(m.lower_bound(), m.upper_bound());
// ...
```

Но это влечет за собой лишний проход по контейнеру **map**. Операции **equal_range()**, **lower_bound()** и **upper_bound()** применимы также и для отсортированных последовательностей (§32.6).

Я склонен рассматривать **set** как **map**, для которого отсутствует отдельный **value_type**. Для **set** тип **value_type** является и типом **key_type**. Рассмотрим

```
struct Record {
    string label;
    int value;
};
```

Чтобы задать тип **set<Record>**, нужно предоставить функцию сравнения. Например:

```
bool operator<(const Record& a, const Record& b)
{
    return a.label<b.label;
}
```

Тогда можно написать следующий код:

```
set<Record> mr {{"duck",10}, {"pork",12}};

void read_test()
{
    for (auto& r : mr) {
        cout << '{' << r.label << ':' << r.value << '}';
    }
    cout << endl;
}
```

Ключ элемента в ассоциативном контейнере не изменяем (§ iso.23.2.4). Поэтому мы не можем изменять значения в **set**. Мы даже не можем изменять поля элементов, которые не принимают участия в сравнениях. Вот пример:

```
void modify_test()
{
    for (auto& r : mr)
        ++r.value;          // ошибка: элементы множества неизменяемы
}
```

Если нужно изменять элементы, используйте **map**. Не пытайтесь модифицировать ключ: если бы вы в этом преуспели, нижележащий механизм поиска элементов был бы сломан.

31.4.3.2. Неупорядоченные ассоциативные контейнеры

Неупорядоченные ассоциативные контейнеры (**unordered_map**, **unordered_set**, **unordered_multimap**, **unordered_multiset**) являются хэш-таблицами. В простых случаях отличия от упорядоченных контейнеров немногочисленны, а все ассоциативные контейнеры обладают практически идентичным набором операций (§31.4.3.1). Пример:

```
unordered_map<string, int> score1 {
    {"andy", 7}, {"al",9}, {"bill",-3}, {"barbara",12}
};

map<string, int> score2 {
    {"andy", 7}, {"al",9}, {"bill",-3}, {"barbara",12}
};

template<typename X, typename Y>
ostream& operator<<(ostream& os, pair<X, Y>& p)
{
    return os << '{' << p.first << ',' << p.second << '}';
}

void user()
{
    cout <<"unordered: ";
    for (const auto& x : score1)
        cout << x << ", ";

    cout << "\nordered: ";
    for (const auto& x : score2)
        cout << x << ", ";
}
```

Здесь видимое различие состоит в том, что результат итерации по **map** является отсортированным (упорядоченным), а для **unordered_map** — нет:

```
unordered: {andy,7}, {al,9}, {bill,-3}, {barbara,12},
ordered: {al,9}, {andy, 7}, {barbara,12}, {bill,-3},
```

Результат итерации по **unordered_map** зависит от порядка вставки элементов, хэш-функции и даже от фактора нагрузки. В частности, нет гарантии, что элементы будут выведены в том порядке, в которым они вставлялись.

31.4.3.3. Конструирование unordered_map

Шаблон **unordered_map** располагает множеством шаблонных параметров и множеством синонимов для типов:

```
template<typename Key,
         typename T,
         typename Hash = hash<Key>,
```

```
        typename Pred = std::equal_to<Key>,
        typename Allocator = std::allocator<std::pair<const Key, T>>>
class unordered_map {
public:
    using key_type = Key;
    using value_type = std::pair<const Key, T>;
    using mapped_type = T;
    using hasher = Hash;
    using key_equal = Pred;
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer= typename allocator_traits<Allocator>::const_pointer;
    using reference = value_type&;
    using const_reference = const value_type&
    using size_type = /* зависит от реализации */;
    using difference_type = /* зависит от реализации */;
    using iterator = /* зависит от реализации */;
    using const_iterator = /* зависит от реализации */;
    using local_iterator = /* зависит от реализации */;
    using const_local_iterator = /* зависит от реализации */;

    // ...
};
```

По умолчанию **unordered_map<X>** использует **hash<X>** для хэширования и **equal_to<X>** для сравнения ключей.

Умолчательная ралиизация **equal_to<X>** (§33.4) просто сравнивает значения **X**, используя операцию **==**.

Первичный шаблон **hash** не имеет определения. Пользователи типа **X** сами определяют **hash<X>** при необходимости. Для таких распространенных типов, как например **string**, имеются стандартные специализации **hash**, так что пользователям нет нет нужды определять их самим:

Типы с hash<T> (§ iso.20.8.12), предоставляемые стандартной библиотекой			
string	**u16string**	**u32string**	**wstring**
С-строка	**bool**	символы	целые типы
вещественные типы	указатели	**type_index**	**thread::id**
error_code	**bitset<N>**	**unique_ptr<T, D>**	**shared_ptr<T>**

Хэш-функция (то есть специализация **hash** для типа **T** или указатель на функцию) должна обеспечивать вызов с аргументом типа **T** и возвращаемым значением типа **size_t** (§ iso.17.6.3.4). Два вызова хэш-функции для одного и того же значения должны приводить к одинаковому результату, и в идеале эти результаты должны равномерно заполнять интервал значений **size_t**, чтобы минимизировать вероятность получить **h(x)==h(y)** при **x!=y**.

Потенциально возможно немыслимое сочетание шаблонных параметров, конструкторов и умолчательных значений, но к счастью, имеется следующая типовая схема:

Конструкторы unordered_map<K, T, H, E, A> (§ iso.23.5.4)	
unordered_map m {n,hf,eql,a};	Создать **m** с **n** корзинами, хэш-функцией **hf**, функцией сравнения **eql** и аллокатором **a**; explicit
unordered_map m {n,hf,eql};	**unordered_map m{n, hf, eql, allocator_type{}};** explicit
unordered_map m {n,hf};	**unordered_map m {n, hf, key_eql{}};** explicit
unordered_map m {n};	**unordered_map m {n, hasher{}};** explicit
unordered_map m {};	**unordered_map m {N};** число корзин **N** зависит от реализации; explicit

Здесь n представляет число элементов, а при его отсутствии имеем пустой unordered_map.

Конструкторы unordered_map<K, T, H, E, A> (§ iso.23.5.4)	
unordered_map m {b,e,n,hf,eql,a};	Создать m с n корзинами, элементами из [b:e), хэш-функцией hf, функцией сравнения eql и аллокатором a
unordered_map m {b,e,n,hf,eql};	unordered_map m {b,e,n,hf,eql, allocator_type{}};
unordered_map m {b,e,n,hf};	unordered_map m {b,e,n,hf, key_equal{}};
unordered_map m {b,e,n};	unordered_map m {b,e,n, hasher{}};
unordered_map m {b,e};	unordered_map m {b,e,N}; число корзин N зависит от реализации

Здесь начальные элементы получаются из последовательности [b:e). Количество элементов будет равно числу элементов в [b:e), distance(b, e).

Конструкторы unordered_map<K, T, H, E, A> (§ iso.23.5.4)	
unordered_map m {{elem},n,hf,eql,a};	Создать m с эл-тами из initializer_list, используя n корзин, хэш-функцию hf, функцию сравнения eql и аллокатор a
unordered_map m {{elem},n,hf,eql};	unordered_map m {{elem},n,hf,eql,allocator_type{}};
unordered_map m {{elem},n,hf};	unordered_map m {{elem},n,hf,key_equal{}};
unordered_map m {{elem},n};	unordered_map m {{elem},n,hasher{}};
unordered_map m {{elem}};	unordered_map m {{elem},N}; число корзин N зависит от реализации

Здесь мы получаем начальные элементы контейнера согласно последовательности элементов списка инициализации с {}-разделителем. Число элементов контейнера будет равно числу элементов в списке инициализации.

Наконец, unordered_map располагает копирующим и перемещающим конструкторами, а также их эквивалентами, предоставляющими аллокаторы:

Конструкторы unordered_map<K, T, H, E, A> (§ iso.23.5.4)	
unordered_map m {m2};	Копирующий и перемещающий конструкторы: создать m из m2
unordered_map m {a};	Создать m по умолчанию и назначить аллокатор a; explicit
unordered_map m {m2, a};	Создать m из m2 и назначить аллокатор a

Будьте внимательны при конструировании unordered_map с одним или двумя аргументами. Возможно множество комбинаций типов, и несоответствия могут приводить к странным сообщениям об ошибках. Вот пример:

```
map<string, int> m {My_comparator};          // OK
unordered_map<string, int> um {My_hasher};   // ошибка
```

Единственный аргумент конструктора должен быть либо экземпляром unordered_map (для копирующего или перемещающего конструктора), числом корзин (bucket count) или аллокатором. Попробуйте что-нибудь в следующем роде:

```
unordered_map<string, int> um {100, My_hasher}; // OK
```

31.4.3.4. Хэш-функции и функции эквивалентости

Естественно, пользователь может определить хэш-функцию, для чего существует множество способов. Разные способы отвечают разным целям. Я приведу несколько версий, начиная с самых явных, и заканчивая самыми простыми. Рассмотрим простой тип **Record**:

```
struct Record {
    string name;
    int val;
};
```

Для него определяем хэширование и операцию сравнения следующим образом:

```
struct Nocase_hash {
    int d = 1;      // сдвиг на d бит в каждой итерации
    size_t operator()(const Record& r) const
    {
        size_t h = 0;
        for (auto x : r.name) {
            h <<= d;
            h ^= toupper(x);
        }

        return h;
    }
};

struct Nocase_equal {
    bool operator()(const Record& r, const Record& r2) const
    {
        if (r.name.size()!=r2.name.size()) return false;
        for (int i = 0; i<r.name.size(); ++i)
            if (toupper(r.name[i])!=toupper(r2.name[i]))
                return false;
        return true;
    }
};
```

Теперь я могу использовать **unordered_set** для типа **Record**:

```
unordered_set<Record, Nocase_hash, Nocase_equal> m {
    {{"andy", 7}, {"al",9}, {"bill",-3}, {"barbara",12}},
    Nocase_hash{2},
    Nocase_equal{}
};

for (auto r : m)
    cout << "{" << r.name << ',' << r.val << "}\n";
```

Если я, как и в большинстве иных случаев, хочу ограничиться умолчательными хэш-функцией и функцией сравнения, я могу это сделать просто не упомянув их среди аргументов конструктора:

```
unordered_set<Record, Nocase_hash, Nocase_equal> m {
    {"andy", 7}, {"al",9}, {"bill",-3}, {"barbara",12}
    // используем Nocase_hash{} и Nocase_equal{}
};
```

Часто простейшим способом является использование хэш-функций стандартной библиотеки, определенных как специализации для **hash** (§31.4.3.2). Например:

```
size_t hf(const Record& r) {return hash<string>()(r.name)^hash<int>()(r.val);};

bool eq (const Record& r, const Record& r2) {return r.name==r2.name && r.val==r2.val;};
```

Комбинирование хэш-значений при помощи операции «исключительное ИЛИ» (^) помогает распределять их на диапазон значений **size_t** (§3.4.5, §10.3.1).

Имея хэш-фунцию и функцию сравнения, мы можем создать объект **unordered_map**:

```
unordered_set<Record, decltype(&hf), decltype(&eq)> m {
    {{"andy", 7}, {"al",9}, {"bill",-3}, {"barbara",12}},
    hf,
    eq
};

for (auto r : m)
    cout << "{" << r.name << ',' << r.val << "}\n";
}
```

Я применил **decltype**, чтобы избежать явного повторения типов **hf** и **eq**.

Если у нас нет под рукой списка инициализации, можно задать начальный размер:

```
unordered_set<Record, decltype(&hf), decltype(&eq)> m {10, hf, eq};
```

Это также позволяет лучше сфокусироваться на операциях хэширования и сравнения.

А если бы нам захотелось избежать отделения определений функций **hf** и **eq** от их точки использования, можно было бы обратиться к лямбда-нотации:

```
unordered_set<Record,                                    // тип значения
        function<size_t(const Record&)>,                 // тип хэша
        function<bool(const Record&, const Record&)>     // эквивалентный тип
    > m {10,
    [](const Record& r) {return hash<string>{}(r.name)^hash<int>{}(r.val);},
    [](const Record& r, const Record& r2) {return r.name==r2.name && r.val==r2.val;}
};
```

Смысл применения лямбда-нотации (именованной или неименованной) вместо функций состоит в том, что они могут определяться локально в функции, в непосредственной близости к точке использования.

Однако, **function** здесь может повлечь дополнительные расходы ресурсов, что лучше все-таки избежать при интенсивном использовании **unordered_set**. Кроме того, я рассматриваю предыдущий вариант кода несколько неопрятным, так что в итоге предпочитаю поименовать лямбды:

```
auto hf = [](const Record& r) {return hash<string>()(r.name)^hash<int>()(r.val);};
auto eq = [](const Record& r, const Record& r2) {return r.name==r2.name && r.val==r2.
val;};

unordered_set<Record, decltype(hf), decltype(eq)> m {10, hf, eq};
```

Наконец, мы можем однократно определить хэш-функцию и функцию сравнения для всех неупорядоченных контейнеров элементов типа **Record**, специализируя стандартные шаблоны **hash** и **equal_to**:

```
namespace std {
    template<>
    struct hash<Record>{
        size_t operator()(const Record &r) const
        {
            return hash<string>{}(r.name)^hash<int>{}(r.val);
        }
    };
```

```
    template<>
    struct equal_to<Record> {
        bool operator()(const Record& r, const Record& r2) const
        {
            return r.name==r2.name && r.val==r2.val;
        }
    };
}

unordered_set<Record> m1;
unordered_set<Record> m2;
```

Умолчательный **hash**, а также получаемые из него операцией «исключающее ИЛИ», часто оказываются совершенно удовлетворительными. Не кидайтесь применять доморощенные хэш-функции без тщательного экспериментирования.

31.4.3.5. Фактор загрузки и корзины

Значительная часть реализации неупорядоченных контейнеров экспонируется программисту. Про ключи с одинаковым хэш-значением говорят, что это ключи из «одной корзины» (см. §31.2.1). Программист может исследовать и устанавливать размер хэш-таблицы (известной как «число корзин»):

Политика хэширования (§ iso.23.2.5)	
h=c.hash_function()	**h** есть хэш-функция **c**
eq=c.key_eq()	**eq** есть функция сравнения **c**
d=c.load_factor()	**d** – число элементов, поделенное на число корзин: **double(c.size())/c.bucket_count()**; noexcept
d=c.max_load_factor()	**d** – максимальный фактор загрузки для **c**; noexcept
c.max_load_factor(d)	Установить максимальный фактор загрузки **c** равным **d**; если фактор загрузки **c** приблизится к максимальному, **c** изменит размер хэш-таблицы (увеличит число корзин)
c.rehash(n)	Установить для **c** число корзин >= **n**
c.reserve(n)	Создать место для **n** входов (принимая в расчет фактор загрузки): **c.rehash(ceil(n/c.max_load_factor()))**

Фактор загрузки (*load factor*) для неупорядоченного ассоциативного контейнера есть просто доля емкости, которая используется. Например, если **capacity()** равно 100 элементам, а **size()** дает 30, то **load_factor()** есть 0.3.

Заметьте, что установка **max_load_factor** вызовом **rehash()** или вызовом **reserve()** может оказаться очень дорогой операцией (в худшем случае это **O(n*n)**), поскольку при этом может (и в реальных случаях будет) выполняться повторное хэширование всех элементов. Эти функции используются для того, чтобы убедиться в том, что повторное хэширование при выполнении программы имеет место в относительно подходящие моменты времени. Пример:

```
unordered_set<Record,[](const Record& r) {return hash(r.name);}> people;
// ...
constexpr int expected = 1000000;       // ожидаемый максимум числа элементов
people.max_load_factor(0.7);            // максимум 70% заполнения
people.reserve(expected);               // около 1,430,000 корзин
```

Нужно поэкспериментировать, чтобы подобрать подходящий фактор загрузки для заданного набора элементов и специфической хэш-функции, но значение в 70% (**0.7**) часто является достаточным.

Интерфейс корзин (§ iso.23.2.5)	
n=c.bucket_count()	n есть число корзин в c (размер хэш-таблицы); noexcept
n=c.max_bucket_count()	n наибольшее возможное число эл-тов в корзине; noexcept
m=c.bucket_size(n)	m – число элементов в n-ой корзине
i=c.bucket(k)	Элемент с ключом k попадает в i-ую корзину
p=c.begin(n)	p указывает на первый элемент в корзине n
p=c.end(n)	p указывает на элемент, следующий за последним в корзине n
p=c.cbegin(n)	p указывает на первый элемент в корзине n; p есть константный итератор
p =c.cend(n)	p указывает на элемент, следующий за последним в корзине n; p есть константный итератор

Применение n, для которого c.max_bucket_count()<=n, в качестве индекса в корзине приводит к неопределенному результату (и вероятно катастрофичному).

Примером использования интерфейса корзин является экспериментирование с хэш-функциями: плохая хэш-функция приводит к большим bucket_counts() для некоторых значений ключей. То есть, она будет давать одно и то же хэш-значение для многих ключей.

31.5. Адаптеры контейнеров

Адаптеры контейнеров предоставляют иной (часто ограниченный) интерфейс к контейнеру. Адаптеры контейнеров предназначены для использования только через их специализированные интерфейсы. В частности, адаптеры контейнеров STL не предоставляют прямого доступа к их нижележащим контейнерам. Они не предоставляют итераторов или индексирования.

Технологии, используемые для создания адаптеров контейнеров, в общем случае полезны для неинтрузивного адаптирования интерфейса класса к нуждам пользователей.

31.5.1. stack

Контейнерный адаптер **stack** определяется в файле **<stack>**. Его можно описать при помощи частичной реализации:

```
template<typename T, typename C = deque<T>>
class stack {  // § iso.23.6.5.2
public:
    using value_type = typename C::value_type;
    using reference = typename C::reference;
    using const_reference = typename C::const_reference;
    using size_type = typename C::size_type;
    using container_type = C;
public:
    explicit stack(const C&);       // копируем из контейнера
    explicit stack(C&& = C{});      // перемещаем из контейнера

    // умолчательные копирование, перемещение, присваивание, деструктор

    template<typename A>
        explicit stack(const A& a);          // умолчательный контейнер, аллокатор a
    template<typename A>
        stack(const C& c, const A& a);       // элементы из c, аллокатор a
```

```
        template<typename A>
            stack(C&&, const A&);
        template<typename A>
            stack(const stack&, const A&);
        template<typename A>
            stack(stack&&, const A&);

        bool empty() const {return c.empty();}
        size_type size() const {return c.size();}
        reference top() {return c.back();}
        const_reference top() const {return c.back();}
        void push(const value_type& x) {c.push_back(x);}
        void push(value_type&& x) {c.push_back(std::move(x));}
        void pop() {c.pop_back();}      // pop последний элемент

        template<typename... Args>
        void emplace(Args&&... args)
        {
            c.emplace_back(std::forward<Args>(args)...);
        }

        void swap(stack& s) noexcept(noexcept(swap(c, s.c)))
        {
            using std::swap;    // используйте стандартный swap()
            swap(c, s.c);
        }
    protected:
        C c;
    };
```

Таким образом, **stack** является интерфейсом к контейнеру, тип которого передан в качестве параметра шаблона. По сути, **stack** удаляет из интерфейса нестековые операции контейнера и предоставляет традиционные имена: **top()**, **push()** и **pop()**.

Кроме того, **stack** предоставляет обычные операции сравнения (**==**, **<** и т.д.) и функцию **swap()**.

По умолчанию, **stack** использует **deque** для хранения элементов, но можно с этой же целью использовать любую последовательность, предоставляющую операции **back()**, **push_back()** и **pop_back()**. Например:

```
stack<char> s1;                 // используем deque<char> для хранения элементов
stack<int, vector<int>> s2;     // используем vector<int> для хранения элементов
```

Часто **vector** обеспечивает большую производительность и использует меньше памяти.

Элементы добавляются в **stack** с помощью операции **push_back()** нижележащего контейнера. Отсюда следует, что стек не может переполниться до тех пор, пока на машине хватает памяти для выделения нижележащему контейнеру. С другой стороны, **stack** может подвергнуться «переисчерпанию» (underflow):

```
void f()
{
    stack<int> s;
    s.push(2);
    if (s.empty()) { // предотвращение underflow
                     // не надо pop
    }
    else {           // но невозможно
        s.pop();     // чудесно: s.size() становится 0
        s.pop();     // эффект неопределен, возможно плохой
    }
}
```

Для использования элемента не нужно вызывать **pop()**. Вместо этого применяют **top()**, а **pop()** вызывают тогда, когда элемент больше не нужен. Это не слишком неудобно, может повысить эффективность, когда в **pop()** нет необходимости, а также значительно упрощает реализацию гарантий для исключений. Например:

```
void f(stack<char>& s)
{
    if (s.top()=='c') s.pop();  // опционально устраняет опциональный начальный 'c'
    // ...
}
```

По умолчанию, **stack** полагается на аллокатор нижележащего контейнера. Если это не подходит, есть несколько конструкторов для предоставления иного аллокатора.

31.5.2. queue

Адаптер **queue** (очередь) определен в файле **<queue>** и предоставляет интерфейс к контейнеру, который допускает добавление элементов в «хвост» операцией **back()**, и удаление элементов с головы контейнера операцией **front()**:

```
template<typename T, typename C = deque<T> >
class queue { // § iso.23.6.3.1
    // ... вроде стека ...
    void pop() {c.pop_front();}     // pop первый элемент
};
```

Используя **queue**, можно определить простой сервер для системы, базирующейся на сообщениях:

```
void server(queue<Message>& q, mutex& m)
{
    while (!q.empty()) {
        Message mess;
        {   lock_guard<mutex> lck(m);   // блокируем при извлечении сообщения
            if (q.empty()) return;      // кто-то другой получил сообщение
            mess = q.front();
            q.pop();
        }
        // обслуживаем запрос
    }
}
```

31.5.3. priority_queue

Адаптер **priority_queue** это очередь, в которой каждому элементу приписан приоритет, с которым элементы просматриваются операцией **top()**. Объявление для **priority_queue** похоже на объявление для **queue** с добавлениями, касающимися объекта сравнения и пары конструкторов для инициализации из последовательностей:

```
template<typename T, typename C = vector<T>, typename Cmp = less<typename
C::value_type>>
class priority_queue {   // § iso.23.6.4
protected:
    C c;
    Cmp comp;
public:
    priority_queue(const Cmp& x, const C&);
    explicit priority_queue(const Cmp& x = Cmp{}, C&& = C{});
```

```
        template<typename In>
            priority_queue(In b, In e, const Cmp& x, const C& c); // вставляем [b:e) в c
        // ...
};
```

Это объявление содержится в **<queue>**.

По умолчанию, **priority_queue** просто сравнивает элементы, используя операцию **<**, а **top()** возвращает наибольший элемент:

```
struct Message {
    int priority;
    bool operator<(const Message& x) const {return priority < x.priority;}
    // ...
};

void server(priority_queue<Message>& q, mutex& m)
{
    while (!q.empty()) {
        Message mess;
        {   lock_guard<mutex> lck(m); // удерживаем блок-ку во время извлеч. сообщения
            if (q.empty()) return;    // кто-то другой получил сообщение
            mess = q.top();
            q.pop();
        }
        // обслуживаем запрос наивысшего приоритета
    }
}
```

Это отличается от **queue** (§31.5.2) в том, что **Messages** с большим приоритетом будут обслуживаться первыми. Порядок, в котором элементы с одинаковым приоритетом поступают в голову очереди, не определен. Два элемента считаются имеющими эквивалентный приоритет, если ни один из них не имеет приоритета, более высокого, чем у другого (§31.2.2.1).

Поддержание упорядоченного хранения элементов не дается задаром, но это и не столь дорого. Одним из полезных способов реализации **priority_queue** является применение древовидной структуры для отслеживания относительных позиций элементов. Это дает для операций **push()** и **pop()** оценку **O(log(n))**. Для реализации **priotity_queue** почти наверняка используется **heap** (§32.6.4).

31.6. Советы

[1] STL контейнер определяет последовательность; §31.2.

[2] Используйте **vector** по умолчанию; §31.2, §31.4.

[3] Операции вставки, такие как **insert()** и **push_back()**, часто более эффективны для **vector**, чем для **list**; §31.2, §31.4.1.1.

[4] Используйте **forward_list** для последовательностей, которые обычно являются пустыми; §31.2, §31.4.2.

[5] Когда дело касается производительности, не доверяйте ощущениям: измеряйте; §31.3.

[6] Не доверяйте слепо оценкам сложности; последовательности могут быть короткими, а цена индивидуальных операций может варьироваться весьма значительно; §31.3.

[7] STL-контейнеры есть дескрипторы ресурсов;§31.2.1.

[8] **map** обычно реализуется с помощью красно-черных деревьев; §31.2.1, §31.4.3.

[9] **unordered_map** обычно это хэш-таблица; §31.2.1, §31.4.3.2.

[10] Чтобы тип годился для STL-контейнеров, он должен предоставлять операции копирования или перемещения; §31.2.2.
[11] Используйте контейнеры указателей или «умных указателей», чтобы сохранить полиморфное поведение; §31.2.2.
[12] Операции сравнения должны обеспечивать строгое слабое упорядочение; §31.2.2.1.
[13] Передавайте контейнеры «по ссылке», а возвращайте «по значению»; §31.3.2.
[14] Для контейнеров используйте ()-инициализацию для размеров, и {}-инициализацию для списка элементов; §31.3.2.
[15] Для прохода по контейнеру используйте цикл range-**for** или пару итераторов begin/end; §31.3.4.
[16] Когда не нужно модифицировать элементы контейнера, используйте **const**-итераторы; §31.3.4.
[17] Используйте **auto**, чтобы избежать переусложнений в работе с итераторами и чтобы застраховаться от потенциальных опечаток; §31.3.4.
[18] Применяйте **reserve()** для того, чтобы не делать указатели и итераторы недействительными; §31.3.3, §31.4.1.
[19] Не стоит заранее надеяться на улучшение производительности за счет **reserve()**: измеряйте эффект на практике; §31.3.3.
[20] Используйте **push_back()** или **resize()** для контейнеров, чем **realloc()** для массивов; §31.3.3, §31.4.1.1.
[21] Не используйте итераторы после изменения размеров **vector** или **deque**; §31.3.3.
[22] Когда это необходимо, используйте **reserve()** для улучшения предсказуемости поведения; §31.3.3.
[23] Не предполагайте, что [] выполняет проверку диапазона; §31.2.2.
[24] Используйте **at()**, когда вам нужна гарантия проверки диапазона; §31.2.2.
[25] Применяйте **emplace()** для улучшения стиля записи; §31.3.7.
[26] Предпочитайте компактные и непрерывные структуры данных; §31.4.1.2.
[27] Используйте **emplace()**, чтобы избежать необходимости предварительно инициализировать элементы; §31.4.1.3.
[28] Проход по **list** достаточно дорог; §31.4.2.
[29] Обычно **list** требует на каждый элемент излишек памяти в 4 машинных слова; §31.4.2.
[30] Порядок следования элементов в упорядоченном контейнере определяется его объектом сравнения (по умолчанию **<**); §31.4.3.1.
[31] Последовательность элементов в неупорядоченном контейнере (хэшированном контейнере) не имеет предсказуемого порядка; §31.4.3.2.
[32] Применяйте неупорядоченные контейнеры, если вам нужен быстрый поиск в большом массиве данных; §31.3.
[33] Применяйте неупорядоченные контейнеры для элементов с типами, не имеющими естественного порядка (нет осмысленной операции **<**); §31.4.3.
[34] Применяйте упорядоченные ассоциативные контейнеры (**map** или **set**), если вам нужно итерировать элементы в определенном порядке; §31.4.3.2.
[35] Проверяйте экспериментально приемлемость хэш-функции; §31.4.3.4.
[36] Часто подходит комбинирование стандартных хэш-функций для элементов с помощью операции «Исключительное ИЛИ»; §31.4.3.4.
[37] Число **0.7** является разумным значением для фактора загрузки; §31.4.3.5.
[38] Вы можете предоставлять альтернативные интерфейсы к контейнерам; §31.5.
[39] STL-адаптеры не предоставляют прямого доступа к их нижележащим контейнерам; §31.5.

32

STL алгоритмы

Форма освобождает.
— Инженерная поговорка

- Введение
- Алгоритмы

  Последовательности; Аргументы политик; Сложность
- Немодифицирующие алгоритмы

  for_each(); Предикаты для последовательностей; **count()**; **find()**; **equal()** и **mismatch()**; **search()**
- Модифицирующие алгоритмы

  copy(); **unique()**; **remove()** и **replace()**; **rotate()**, **random_shuffle()** и **partition()**; Перестановки; **fill()**; **swap()**
- Сортировка и поиск

  Бинарный поиск; **merge()**; Алгоритмы для множеств; Кучи; **lexicographical_compare()**
- Минимумы и максимумы
- Советы

32.1. Введение

В этой главе рассматриваются STL алгоритмы. В целом, STL состоит из тех частей стандартной библиотеки, которые относятся к итераторам, контейнерам, алгоритмам и функциональным объектам. Кроме настоящей главы, STL рассматривается в Главе 31 и Главе 33.

32.2. Алгоритмы

Всего около 80 стандартных алгоритмов определены в файле **<algorithm>**. Они действуют на последовательностях, определяемых парой итераторов (для входа) или единственным итератором (для выхода). При копировании, сравнении и т.д. двух последовательностей первая представима парой итераторов, [**b,e**), а вторая — единственным итератором, **b2**, который рассматривается как начало последовательности, содержащей достаточное число элементов для алгоритма, например, такое же число элементов, как в первой последовательности: [**b2:b2+(e-b)**). Некоторые алгоритмы, такие как **sort()**, требуют итераторов с произвольным доступом, в то время как многие, такие как **find()**, читают элементы в одном порядке, так что их устроит и прямой итератор. Много алгоритмы сле-

дуют традиционному соглашению, возвращая конец последовательности для индикации результата «не найдено» (§4.5). Но я не сказал, что это касается каждого алгоритма.

Алгоритмы, как стандартные, так и пользовательские, важны:

- Так как именуют конкретную операцию, документируют интерфейс и специфицируют семантику.
- Поскольку могут широко использоваться и известны многим программистам.

Все это делает их более приемлемыми по сравнению со «случайным» кодом с плохо специфицированными функциями и зависимостями. Если вы обнаружите, что пишете код со многими циклами и локальными переменными, которые плохо связаны друг с другом, со сложными управляющими структурами, то рассмотрите возможность упростить его посредством группировки в некую часть с «говорящим» именем, с точно очерченным предназначением, хорошо определенным интерфейсом и с хорошо определенными зависимостями.

Численные алгоритмы в стиле STL алгоритмов рассматриваются в §40.6.

32.2.1. Последовательности

В идеале стандартный алгоритм должен предоставлять общий и гибкий интерфейс к чему-либо, что может реализовываться оптимальным образом. Интерфейсы на базе итераторов не идеальны, но служат некоторым приближением к этому идеалу (§33.1.1). В частности, интерфейс на базе итераторов не представляет последовательность напрямую, что может вводить в заблуждение и приводить к трудностям в нахождении ошибок, связанных с диапазоном:

```
void user(vector<int>& v1, vector<int>& v2)
{
    copy(v1.begin(), v1.end(), v2.begin());    // возможно переполнение v2
    sort(v1.begin(), v2.end());                // упс!
}
```

Многие такие проблемы устраняются предоставлением контейнерных версий стандартных алгоритмов. Например:

```
template<typename Cont>
void sort(Cont& c)
{
    static_assert(Range<Cont>(), "sort(): Cont argument not a Range");
    static_assert(Sortable<Iterator<Cont>>(), "sort(): Cont argument not Sortable");

    std::sort(begin(c), end(c));
}

template<typename Cont1, typename Cont2>
void copy(const Cont1& source, Cont2& target)
{
    static_assert(Range<Cont1>(), "copy(): Cont1 argument not a Range");
    static_assert(Range<Cont2>(), "copy(): Cont2 argument not a Range");
    if (target.size()<source.size()) throw out_of_range{"copy target too small"};

    std::copy(source.begin(), source.end(), target.begin());
}
```

Это упрощает определение **user()**, делает вторую ошибку невозможной и перехватывает первую ошибку во время выполнения:

```
void user(vector<int>& v1, vector<int>& v2)
{
    copy(v1,v2);        // переполнение будет перехвачено
    sort(v1);
}
```

Но, в то же время, контейнерные версии менее общие, чем те, что используют итераторы напрямую. В частности, вы не можете использовать контейнерную **sort()** для сортировки половины контейнера, и вы не можете использовать контейнерную **copy()** для записи в поток вывода.

Дополняющим подходом будет определение «диапазона» и абстракции «последовательности», что позволит нам определять последовательности по мере необходимости. Я использую концепцию **Range** для идентификации чего-либо с **begin()** и **end()** итераторами (§24.4.4). То есть, в самом STL нет класса **Range**, содержащего данные, также как и нет класса **Iterator** и **Container**. В примерах с контейнерной **sort()** и контейнерной **copy()** я называл именем **Cont** шаблонный аргумент (для «контейнера»), но на самом деле может быть принят любой аргумент в виде последовательности с **begin()** и **end()**, удовлетворяющими остальным требованиям алгоритма.

Большинство стандартных контейнеров возвращают итераторы; они не возвращают контейнеров с результатами (редкое исключение в случае **pair()**). Одной причиной этого является тот факт, что когда создавался STL, не было еще прямой поддержки семантики перемещения. То есть, не было способа очевидного и эффективного возврата большого количества данных из алгоритмов. Некоторые программисты пользовались явной косвенной адресацией (указателем, ссылкой или итератором) или же теми или иными трюками. Сейчас все проще:

```
template<typename Cont, typename Pred>
vector<Value_type<Cont>*>
find_all(Cont& c, Pred p)
{
    static_assert(Range<Cont>(), "find_all(): Cont argument not a Range");
    static_assert(Predicate<Pred>(), "find_all(): Pred argument not a Predicate");

    vector<Value_type<Cont>*> res;
    for (auto& x : c)
        if (p(x)) res.push_back(&x);
    return res;
}
```

В C++98 этот **find_all()** имел бы плохую производительность в случае, когда число совпадений велико. Если выбор стандартного алгоритма кажется слишком ограничивающим или неэффективным, то расширения с новыми версиями STL алгоритмов часто являются превосходными альтернативами всякому «случайному» коду для обхода проблемы.

Обметим, что аргументами для STL алгоритмов служат итераторы (Глава 33), и алгоритм ничего не знает о структурах данных, на которые эти итераторы указывают. Итераторы преимущественно служат для того, чтобы изолировать алгоритмы от этих структур, на которых они оперируют, и наоборот.

32.3. Аргументы политик

Большая часть алгоритмов стандартной библиотеки идут в двух версиях:

- «Простая» версия, выполняющая свои действия при помощи стандартных операций, таких как **<** и **==**
- Версия, которая принимает ключевые операции в виде аргументов

Например:

```
template<class Iter>
void sort(Iter first, Iter last)
{
    // ... сортировка с помощью e1<e2 ...
}

template<class Iter, class Pred>
void sort(Iter first, Iter last, Pred pred)
{
    // ... сортировка с помощью pred(e1, e2) ...
}
```

Это увеличивает гибкость стандартной библиотеки и расширяет область ее применения.

Две версии алгоритма могут реализовываться как два (перегруженных) шаблона функции, или как единственный шаблон функции с умолчательным аргументом. Пример:

```
template<typename Ran, typename Pred = less<Value_type<Ran>>> // исп. умолчательный арг-нт шаб-на
sort(Ran first, Ran last, Pred pred ={})
{
    // ... используем pred(x, y) ...
}
```

Рассматривая многие варианты алгоритмов стандартной библиотеки просто как «версии с умолчательным параметром», мы вдвое уменьшаем число шаблонных функций, которые должны помнить.

В некоторых случаях аргумент может трактоваться как предикат или как значение. Например:

```
bool pred(int);

auto p = find(b,e,pred);  // находим элемент pred или применяем pred() от predicate?
```

В общем случае компилятор не в состоянии разобраться с неоднозначностями в таких примерах, да и программисты могут попасть в затруднительную ситуацию, даже если компилятор справится.

Чтобы упростить задачу программисту, используют суффикс **_if** для индикации того, что алгоритм принимает предикат. Рассмотрим следующее:

```
using Predicate = bool(*)(int);

void f(vector<Predicate>& v1, vector<int>& v2)
{
   auto p1 = find(v1.begin(),v1.end(),pred);  // находим элемент со значением pred
   auto p2 = find_if(v2.begin(),v2.end(),pred); // считаем элементы, для которых pred() дает true
}
```

Некоторые операции, передаваемые как аргументы алгоритмов, полагаются изменяющими значения элементов, к которым они прилагаются (например, некоторые операции, передаваемые алгоритму **for_each()**; §32.4.1), но большинство является предикатами (например, операциями сравнения для **sort()**). Если не оговорено обратное, полагаем, что аргумент-политика не модифицирует элементы. В частности, не нужно пытаться модифицировать элементы посредством предикатов:

```
int n_even(vector<int>& v)     // не делайте так
    // считаем число четных значений в v
```

```
{
    return find_if(v.begin(), v.end(),[](int& x) {++x; return x&1;});
}
```

Модификация элемента посредством предиката затуманивает смысл того, что делается. Если вы уж «слишком умны», то можете даже модифицировать последовательность (например, вставляя или удаляя элемент, используя имя контейнера, по которому ведется итерация), а в результате провалится работа по итерированию. Во избежание таких проблем можно передавать аргументы предикатам посредством **const**-ссылок.

Аналогично, предикат не должен переносить состояние, изменяющее смысл операции. Реализация алгоритма может скопировать предикат, а также мы редко когда намереваемся повторно применять предикат на одном и том же значении для получения разных результатов. Некоторые функциональные объекты, передаваемые алгоритмам, такие как генераторы случайных чисел, переносят меняющееся состояние. Если только вы не уверены в том, что алгоритм не выполняет копирования, то храните меняющееся состояние аргумента-функционального объекта в другом объекте и выполняйте доступ к нему посредством указателя или ссылки.

Операции == и < на элементах-указателях редко когда подходят для STL алгоритмов: они сравнивают машинные адреса, а не значения, на которые указывают. В частности, не сортируйте и не выполняйте поиск в контейнерах строк в С-стиле, используя умолчательные == и < (§32.6).

32.3.1. Сложность

Как и в случае контейнеров (§31.3), сложность алгоритмов специфицируется стандартом. Большинство алгоритмов линейно, **O(n)**, где **n** обычно является длиной входной последовательности.

Сложность алгоритмов (§ iso.25)	
O(1)	**swap(), iter_swap()**
O(log(n))	**lower_bound(), upper_bund(), equal_range(), binary_search(), push_heap(), pop_heap()**
O(n*log(n))	**inplace_merge()** (худший случай), **stable_partition()** (худший случай), **sort(), stable_sort(), partial_sort(), partial_sort_copy(), sort_heap()**
O(n*n)	**find_end(), find_first_of(), search(), search_n()**
O(n)	Все остальные

Как всегда, эти сложности асимптотические, и вы должны знать оценку для **n**, чтобы понимать последствия. Например, если **n<3**, то квадратичный алгоритм может быть наилучшим выбором. Стоимость каждой итерации варьируются радикально. Например, траверс по списку может быть намного более медленным, чем траверс по вектору, хотя сложность в обоих случаях линейна (**O(n)**). Оценки сложности не отменяют здравого смысла и реальных измерений времени работы; они лишь один из инструментов для измерения качества кода.

32.4. Немодифицирующие алгоритмы

Немодифицирующий алгоритм просто читает значения элементов входной последовательности; он не изменяет последовательность элементов и не меняет значений элементов. В типичном случае, предоставляемые пользователем операции, не изменяют значения элементов также; они имеют тенденцию быть предикатами.

32.4.1. for_each()

Простейшим алгоритмом является **for_each()**, который применяет операцию к каждому элементу последовательности:

for_each (§ iso.25.2.4)	
f=for_each(b, e, f)	Выполнить f(x) для всех x в [b:e); вернуть f

По мере возможности предпочитайте более специфичные алгоритмы.

Операция, передаваемая алгоритму **for_each()**, может модифицировать элементы. Пример:

```
void increment_all(vector<int>& v)      // инкрементируем каждый элемент v
{
    for_each(v.begin(), v.end(), [](int& x) {++x;});
}
```

32.4.2. Предикаты для последовательностей

Предикаты для последовательностей (§ iso.25.2.1)	
all_of(b, e, f)	Верно ли f(x) для всех x в [b:e)?
any_of(b, e, f)	Верно ли f(x) для какого-либо x в [b:e)?
none_of(b, e, f)	Ложно ли f(x) для всех x в [b:e)?

Пример:

```
vector<double> scale(const vector<double>& val, const vector<double>& div)
{
    assert(val.size()<div.size());
    assert(all_of(div.begin(), div.end(),[](double x){return 0<x;});

    vector res(val.size());
    for (int i = 0; i<val.size(); ++i)
        res[i] = val[i]/div[i];
    return res;
}
```

Когда один их этих предикатов терпит неудачу, он не сообщает, на каком элементе это произошло.

32.4.3. count()

count (§ iso.25.2.9)	
x=count(b,e,v)	x есть число эл-ов *p в [b:e) таких, что v==*p
x=count_if(b,e,v,f)	x есть число эл-ов *p в [b:e) таких, что f(*p)

Пример:

```
void f(const string& s)
{
    auto n_space = count(s.begin(), s.end(),' ');
    auto n_whitespace = count_if(s.begin(), s.end(), isspace);
    // ...
}
```

Предикат **isspace()** (§36.2) позволяет сосчитать все пробельные символы, а не только лишь пробелы.

32.4.4. find()

Семейство алгоритмов **find()** выполняет линейный поиск некоторого элемента или совпадения по предикату:

Семейство алгоритмов find (§ iso.25.2.5)	
p=find(b,e,v)	p указывает на первый эл-т в [b:e) такой, что *p==v
p=find_if(b,e,f)	p указывает на первый эл-т в [b:e) такой, что f(*p)
p=find_if_not(b,e,f)	p указывает на первый эл-т в [b:e) такой, что !f(*p)
p=find_first_of(b,e,b2,e2)	p указывает на первый эл-т в [b:e) такой, что *p==*q для некоторого q в [b2:e2)
p=find_first_of(b,e,b2,e2,f)	p указывает на первый эл-т в [b:e) такой, что f(*p,*q) для некоторого q в [b2:e2)
p=adjacent_find(b,e)	p указывает на первый эл-т в [b:e) такой, что *p==*(p+1)
p=adjacent_find(b,e,f)	p указывает на первый эл-т в [b:e) такой, что f(*p,*(p+1))
p=find_end(b,e,b2,e2)	p указывает на последний *p в [b:e) такой, что *p==*q для элемента *q в [b2:e2)
p=find_end(b,e,b2,e2,f)	p указывает на последний *p в [b:e) такой, что f(*p,*q) для элемента *q в [b2:e2)

Алгоритмы **find()** и **find_if()** возвращают итератор на первый элемент, который отвечает значению или предикату, соответственно:

```
void f(const string& s)
{
    auto p_space = find(s.begin(), s.end(),' ');
    auto p_whitespace = find_if(s.begin(), s.end(), isspace);
    // ...
}
```

Алгоритмы **find_first_of()** находят первое появление в последовательности элемента из другой последовательности. Пример:

```
array<int> x = {1,3,4};
array<int> y = {0,2,3,4,5};

void f()
{
    auto p = find_first_of(x.begin(), x.end(), y.begin(), y.end());    // p = &x[1]
    auto q = find_first_of(p+1, x.end(), y.begin(), y.end());          // q = &x[2]
}
```

Итератор **p** будет указывать на **x[1]**, поскольку **3** есть первый элемент **x** с соответствием в **y**.

32.4.5. equal() и mismatch()

Алгоритмы **equal()** и **mismatch()** сравнивают пары последовательностей:

equal и mismatch() (§ iso.25.2.11, § iso.25.2.10)	
equal(b,e,b2)	Верно ли v==v2 для всех соответствующих эл-ов из [b:e) и [b2:b2+(e-b))?
equal(b,e,b2,f)	Верно ли f(v, v2) для всех соответствующих эл-ов из [b:e) и [b2:b2+(e-b))?

Продолжение

equal и mismatch() (§ iso.25.2.11, § iso.25.2.10)	
pair(p1,p2)=mismatch(b,e,b2)	p1 указывает на первый эл-т в [b:e) в p2 указывает на первый эл-т в [b2:b2+(e-b)) такой, что !(*p1==*p2) или p1==e
pair(p1,p2)=mismatch(b,e,b2,f)	p1 указывает на первый эл-т в [b:e) и p2 указывает на первый эл-т в [b2:b2+(e-b)) такой, что !f(*p1,*p2) или p1==e

Алгоритм **mismatch()** ищет первую пару элементов двух последовательностей, которые не соответствуют друг другу, и возвращает итераторы на эти элементы. Конец для второй последовательности не специфицирован; то есть, нет **last2**. Вместо этого предполагается, что имеется по крайней мере столько же элементов во второй последовательности, сколько и в первой, и что **first2+(last-first)** используется как **last2**. Эта техника применяется во всей стандартной библиотеке, где пары последовательностей используются для операций над парами элементов. Мы могли бы реализовать **mismatch()** следующим образом:

```
template<class In, class In2, class Pred = equal_to<Value_type<In>>>
pair<In, In2> mismatch(In first, In last, In2 first2, Pred p ={})
{
    while (first != last && p(*first,*first2)) {
        ++first;
        ++first2;
    }
    return {first, first2};
}
```

Я использовал стандартный функциональный объект **equal_to** (§33.4) и функцию типа **Value_type** (§28.2.1).

32.4.6. search()

Алгоритмы **search()** и **search_n()** находят одну последовательность как подпоследовательность другой:

Поиск последовательностей (§ iso.25.2.13)	
p=search(b,e,b2,e2)	p указывает на первый *p в [b:e) такой, что [p:p+(e-b)) равно [b2:e2)
p=search(b,e,b2,e2,f)	p указывает на первый *p в [b:e) такой, что [p:p+(e-b)) равно [b2:e2), используя f для сравнения эл-ов
p=search_n(b,e,n,v)	p указывает на первый эл-т в [b:e) такой, что каждый эл-т [p:p+n) имеет значение v
p=search_n(b,e,n,v,f)	p указывает на первый эл-т в [b:e) такой, что для каждого эл-а *q в [p:p+n) имеем f(*p,v)

Алгоритм **search()** ищет вторую последовательность как подпоследовательность первой. Если эта вторая последовательность находится, то возвращается итератор на первый совпадающий элемент в первой последовательности. Как всегда, конец последовательности используется для индикации результата «не найдено». Например:

```
string quote {"Why waste time learning, when ignorance is instantaneous?"};

bool in_quote(const string& s)
{
    auto p = search(quote.begin(), quote.end(), s.begin(), s.end());    // ищем s в quote
    return p!=quote.end();
}

void g()
{
    bool b1 = in_quote("learning");    // b1 = правда
    bool b2 = in_quote("lemming");     // b2 = ложь
}
```

Таким образом, **search()** является полезным алгоритмом для нахождения целой подстроки. Используйте **find()** или **binary_search()** (§32.6) для поиска единственного элемента.

32.5. Модифицирующие алгоритмы

Модифицирующие алгоритмы могут модифицировать элементы своих аргументов-последовательностей.

transform (§ iso.25.3.4)	
p=transform(b,e,out,f)	Применить ***q=f(*p1)** к каждому ***p1** в [**b**:**e**), вывод в соответствующий ***q** в [**out**:**out+(e-b)**); **p=out+(e-b)**
p=transform(b,e,b2,out,f)	Применить ***q=f(*p1,*p2)** к каждому эл-ту ***p1** в [**b**:**e**) и соответствующему ***p2** в [**b2**:**b2+(e-b)**), вывод в соответствующий ***q** в [**out**:**out+(e-b)**); **p=out+(e-b)**

Как ни станно, **transform()** не обязательно изменяет свой вход. Вместо этого он производит выход, который является трансформацией входа, основанной на операции, предоставленной пользователем. Версию **transform()** с одной входной последовательностью можно определить следующим образом:

```
template<class In, class Out, class Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first!=last)
        *res++ = op(*first++);
    return res;
}
```

Выходная последовательность может быть той же, что и входная:

```
void toupper(string& s)        // случай с удалением
{
    transform(s.begin(), s.end(), s.begin(), toupper);
}
```

Здесь трансформируется (на самом деле) входная последовательность **s**.

32.5.1. copy()

Семейство алгоритмов **copy()** копирует элементы одной последовательности в другую. В следующем разделе перечисляются версии **copy()** в сочетании с другими алгоритмами, такими как **replace_copy()** (§32.5.3).

Семейство алгоритмов copy (§ iso.25.3.1)	
p=copy(b,e,out)	Копировать все эл-ы из [b:e) в [out:p); p=out+(e-b)
p=copy_if(b,e,out,f)	Копировать эл-ы x из [b:e), для к-рых верно f(x), в [out:p)
p=copy_n(b,n,out)	Копировать первые n эл-ов из [b:b+n) в [out:p); p=out+n
p=copy_backward(b,e,out)	Копировать все эл-ы из [b:e) в [out:p), начиная с последнего; p=out+(e-b)
p=move(b,e,out)	Переместить все эл-ы из [b:e) в [out:p); p=out+(e-b)
p=move_backward(b,e,out)	Переместить все эл-ы из [b:e) в [out:p), начиная с последнего; p=out+(e-b)

Цель копирующего алгоритма не обязана быть контейнером. Подойдет все, что может быть описано выходным итератором (§38.5). Например:

```
void f(list<Club>& lc, ostream& os)
{
    copy(lc.begin(), lc.end(), ostream_iterator<Club>(os));
}
```

Чтобы читать последовательнсоть, нам нужны два итератора, описывающих, где начинать и где заканчивать. Для записи же нам нужен единственный итератор, показывающий где начать запись (куда писать). Ясно, что нужно принять меры, чтобы не писать дальше конца последовательности. Одним из способов гарантии является использование возможности (§33.2.2) расширять цель в случае необходимости. Например:

```
void f(const vector<char>& vs, vector<char>& v)
{
    copy(vs.begin(), vs.end(), v.begin());          // может переписать конец v
    copy(vs.begin(), vs.end(), back_inserter(v));   // добавляет элементы из vs в конец v
}
```

Входная последовательность и выходная последовательность могут перекрываться. Мы используем **copy()**, когда последовательности не перекрываются или же когда конец выходной последовательности находится внутри входной последовательности.

Мы используем **copy_if()** для копирования элементов, удовлетворяющих некоторому критерию. Например:

```
void f(list<int>&ld, int n, ostream& os)
{
    copy_if(ld.begin(), ld.end(),
        ostream_iterator<int>(os),
        [](int x) {return x>n);});
}
```

См. также **remove_copy_if()**.

32.5.2. unique()

Алгоритм **unique()** устраняет смежные дублирующие элементы из последовательности:

Семейство алгоритмов unique (§ iso.25.3.9)	
p=unique(b,e)	Переместить эл-ы из [b:e) так, что [b:p) не имеет смеж. дубликатов
p=unique(b,e,f)	Переместить эл-ы из [b:e) так, что [b:p) не имеет смежных дубликатов; f(*p,*(p+1)) определяет "дубликат"
p=unique_copy(b,e,out)	Копировать [b:e) в [out:p); не копировать смежн. дубликаты
p=unique_copy(b,e,out,f)	Копировать [b:e) в [out:p); не копировать смежн. дубликаты; f(*p,*(p+1)) определяет "дубликат"

Алгоритмы **unique()** и **unique_if()** устраняют смежные дублирующие элементы. Например:

```
void f(list<string>& ls, vector<string>& vs)
{
    ls.sort();          // сортировка списка (§31.4.2)
    unique_copy(ls.begin(), ls.end(), back_inserter(vs));
}
```

Здесь **ls** копируется в **vs** с устранением дубликатов в этом процессе. Я использую **sort()** для проставления эквивалентных строк смежным образом.

Как и другие стандартные алгоритмы, **unique()** оперирует на итераторах. Он не знает, на какие контейнеры эти итераторы указывают, так что он не может модифицировать контейнер. Он может лишь модифицировать значения элементов. Это означает, что **unique()** не устраняет дубликаты из входной последовательности, что можно было бы предположить по наивности. Поэтому дубликаты из **vector** здесь не устраняются:

```
void bad(vector<string>& vs)      // предупреждение: не делайте того, что кажется!
{
    sort(vs.begin(), vs.end());   // сортируем vector
    unique(vs.begin(), vs.end()); // устраняет дубликаты (нет)?
}
```

Вместо этого **unique()** передвигает уникальные элементы в направлении головы последовательности и возвращает итератор на конец подпоследовательности уникальных элементов. Пример:

```
int main()
{
    string s ="abbcccde";

    auto p = unique(s.begin(), s.end());
    cout << s << ' ' << p-s.begin() << '\n';
}
```

Вот результат работы этого кода:

```
abcdecde 5
```

То есть **p** указывает на вторую **c**.

Алгоритмы, которые могли бы удалять элементы (но не могут), обычно идут в двух формах: простая версия, которая переупорядочивает элементы способом, аналогичным алгоритму **unique()**, и **_copy()**-версия, что производит новую последовательность способом, аналогичным **unique_copy()**.

Для устранения дубликатов из контейнера мы должны явным образом сжать его:

```
template<class C>
void eliminate_duplicates(C& c)
{
    sort(c.begin(), c.end());                 // сортируем
    auto p = unique(c.begin(), c.end());      // компактизируем
    c.erase(p, c.end());                      // сжимаем
}
```

Я мог бы эквивалентно написать **c.erase(unique(c.begin(), c.end(), c.end()))**, но я не считаю это улучшением читаемости кода.

32.5.3. remove() и replace()

Алгоритм **remove()** «удаляет» элементы в конец последовательности:

remove (§ iso.25.3.8)	
p=remove(b,e,v)	Удалить эл-ы со значениями v из [b:e), так, что [b:p) становится эл-ом, для к-рого !(*q==v)
p=remove_if(b,e,v,f)	Удалить эл-ы *q из [b:e), так, что [b:p) становится эл-ом, для к-рого !f(*q)
p=remove_copy(b,e,out,v)	Копировать эл-ы из [b:e), для к-рых !(*q==v) в [out:p)
p=remove_copy_if(b,e,out,f)	Копировать эл-ы из [b:e), для к-рых !f(*q) в [out:p)
reverse(b,e)	Сменить порядок эл-ов на обратный в [b:e)
p=reverse_copy(b,e,out)	Копировать [b:e) в [out:p) в обратном порядке

Алгоритм **replace()** приписывает новые значения выбранным элементам:

replace (§ iso.25.3.5)	
replace(b,e,v,v2)	Заменить эл-ы *p в [b:e), для к-рых *p==v, на v2
replace_if(b,e,f,v2)	Заменить эл-ы *p в [b:e), для к-рых f(*p), на v2
p=replace_copy(b,e,out,v,v2)	Копировать [b:e) в [out:p), заменяя эл-ы, для к-рых *p==v на v2
p=replace_copy_if(b,e,out,f,v2)	Копировать [b:e) в [out:p), заменяя эл-ы, для к-рых f(*p, v) на v2

Эти алгоритмы не могут изменять размер входной последовательности, так что даже **remove()** оставляет неизменным размер своей входной последовательности. Как и **unique_if()**, он «удаляет» передвижением элементов налево. Например:

```
string s {"*CamelCase*IsUgly*"};
cout << s << '\n';                                  // *CamelCase*IsUgly*
auto p = remove(s.begin(), s.end(),'*');
copy(s.begin(), p, ostream_iterator<char>{cout});   // CamelCaseIsUgly
cout << s << '\n';                                  // CamelCaseIsUglyly*
```

32.5.4. rotate(), random_shuffle() и partition()

Алгоритмы **rotate()**, **random_shuffle()** и **partition()** предоставляют систематические способы передвижения элементов внутри последовательности:

rotate() (§ iso.25.3.11)	
p=rotate(b,m,e)	Вращать эл-ы влево: рассматривает [b:e), как круг с первым эл-ом сразу после последнего; перемещает *(b+i) в *((b+(i+(e-m))%(e-b)); внимание: *b перемещается в *m; p=b+(e-m)
p=rotate_copy(b,m,e,out)	Копировать [b:e) во вращаемую последовательность [out:p)

Передвижение элементов алгоритмом **rotate()** (и двумя другими алгоритмами) выполняется посредством **swap()**.

random_shuffle() (§iso.25.3.12)	
random_shuffle(b,e)	Перетасовать эл-ы [b:e), используя генератор случ. чисел по умолч-ю
random_shuffle(b,e,f)	Перетасовать эл-ы [b:e), используя генератор случ. чисел f
shuffle(b,e,f)	Перетасовать эл-ы [b:e), используя однородный генератор случ. чисел f

Алгоритм **random_shuffle()** тасует свою последовательность примерно так, как мы тасуем колоду карт. То есть, после этого элементы располагаются в случайном порядке, причем случайность определяется генератором случайных чисел.

По умолчанию, **random_shuffle()** тасует последовательность, используя генератор случайных чисел с однородным распределением. То есть, он выбирает перестановку элементов последовательности таким образом, что перестановки имеют равные шансы быть выбранными. Если вам нужно другое распределение или более качественный генератор случайных чисел, вы можете предоставить свои. Для вызова **random_shuffle(b,e,r)** генератор вызывается с числом элементов последовательности в качестве аргумента. Например, для вызова **r(e-b)** генератор должен вернуть значение в диапазоне **[0,e-b)**. Если **My_rand** является таким генератором, мы можем тасовать колоду карт следующим образом:

```
void f(deque<Card>& dc, My_rand& r)
{
    random_shuffle(dc.begin(), dc.end(), r);
    // ...
}
```

Алгоритм разбиения на разделы (partitions) разбивает последовательность на две части, основываясь на критерии разбиения:

partition() (§ iso.25.3.13)	
p=partition(b,e,f)	Поместить эл-ы, для к-рых **f(*p1)**, в **[b:p)**, а остальные эл-ы – в **[p:e)**
p=stable_partition(b,e,f)	Поместить эл-ы, для к-рых **f(*p1)**, в **[b:p)**, а остальные эл-ы – в **[p:e)**; сохранить отн. порядок
pair(p1,p2)=partition_copy(b,e,out1,out2,f)	Копировать эл-ы из **[b:e)**, для к-рых **f(*p)**, в **[out1:p1)**, а эл-ы из **[b:e)**, для к-рых **!f(*p)**, в **[out2:p2)**
p=partition_point(b,e,f)	Для **[b:e)**, **p** есть такая точка, что **all_of(b,p,f)** и **none_of(p,e,f)**
is_partitioned(b,e,f)	Находится ли каждый эл-т **[b:e)**, для к-рого **f(*p)**, раньше каждого эл-та, для к-рого **!f(*p)?**

32.5.5. Перестановки

Алгоритмы перестановок предоставляют систематический способ генерирования перестановок в последовательности.

Permutations (§ iso.25.4.9, § iso.25.2.12) **x** есть **true**, если операция **next_*** успешна, иначе **false**	
x=next_permutation(b,e)	Сделать **[b:e)** следующей перестановкой в лексикографич. порядке
x=next_permutation(b,e,f)	Сделать **[b:e)** следующей перестановкой, используя **f** для сравнения
x=prev_permutation(b,e)	Сделать **[b:e)** предыдущей перестановкой в лексикографич. порядке
x=prev_permutation(b,e,f)	Сделать **[b:e)** предыдущей перестановкой, исполь. **f** для сравнения
is_permutation(b,e,b2)	Имеется ли перестановка **[b2:b2+(e-b))**, к-рая равна **[b,e)**?
is_permutation(b,e,b2,f)	Имеется ли перестановка **[b2:b2+(e-b))** к-рая равна **[b,e)**, используя **f(*p,*q)** для сравнения эл-ов?

Перестановки используются для генерирования сочетаний элементов последовательности. Например, перестановки для **abc** есть **acb**, **bac**, **bca**, **cab** и **cba**.

Алгоритм **next_permutation()** берет последовательность **[b:e)** и трансформирует ее в следующую перестановку. Следующая перестановка находится в предположении, что набор всех перестановок лексикографически упорядочен. Если такая перестановка существует, то **next_permutation()** возвращает **true**; в противном случае последовательность трансформируется в наименьшую перестановку, сортированную в восходящем порядке (**abc** в нашем примере) и возвращает **false**. Итак, мы можем генерировать перестановки **abc** следующим образом:

```
vector<char> v {'a','b','c'};
while(next_permutation(v.begin(), v.end()))
    cout << v[0] << v[1] << v[2] << ' ';
```

Аналогично, возвращаемое значение для **prev_permutation()** есть **false**, если **[b:e)** уже содержит первую перестановку (**abc** в нашем примере); в этом случае возвращается последняя перестановка (**cba** в нашем примере).

32.5.6. fill()

Семейство алгоритмов **fill()** предоставляет способы присваивания и инициализации элементов последовательностей:

Семейство fill (§ iso.25.3.6, § iso.25.3.7, § iso.20.6.12)	
fill(b,e,v)	Присвоить **v** каждому эл-у **[b:e)**
p=fill_n(b,n,v)	Присвоить **v** каждому эл-у **[b:b+n)**; **p=b+n**
generate(b,e,f)	Присвоить **f()** каждому эл-у **[b:e)**
p=generate_n(b,n,f)	Присвоить **f()** каждому эл-у **[b:b+n)**; **p=b+n**
uninitialized_fill(b,e,v)	Инициализировать каждый эл-т в **[b:e)** значением v
p=uninitialized_fill_n(b,n,v)	Инициализировать каждый эл-т в **[b:b+n)** значением **v**; **p=b+n**
p=uninitialized_copy(b,e,out)	Инициализировать каждый эл-т **[out:out+(e-b))**; **p=b+n** знач-ем соответствующего эл-а из **[b:e)**
p=uninitialized_copy_n(b,n,out)	Инициализировать каждый эл-т **[out:out+n)** знач-ем соответствующего эл-а из **[b:b+n)**; **p=b+n**

Алгоритм **fill()** повторяющимся образом присваивает заданное значение, в то время как **generate()** присваивает значения, получаемые повторными вызовами своего аргумента-функции. Для примера используем генераторы случайных чисел **Randint** и **Urand** из §40.7:

```
int v1[900];
array<int,900> v2;
vector v3;

void f()
{
    fill(begin(v1), end(v1),99);                    // устанавливаем элементы v1 в 99
    generate(begin(v2), end(v2), Randint{}); // присваиваем случайные значения (§ 40.7)

    // выводим 200 случайных целых в интервале [0:100):
    generate_n(ostream_iterator<int>{cout},200,Urand{100}); // см. § 40.7

    fill_n(back_inserter{v3},20,99);                // добавляем в v3 20 элем-ов со значением 99
}
```

Функции **generate()** и **fill()** скорее присваивают, чем инициализируют. Если вам нужно манипулировать сырой памятью, скажем, чтобы превратить область памяти в объекты

четко определенных типов и состояний, нужно использовать одну из **uninitialized_** версий (представлены в файле **<memory>**).

Неинициализированные последовательности должны появляться лишь в самых низкоуровневых частях кода, обычно в реализациях контейнеров. Целевые элементы для **uninitialized_fill()** и **uninitialized_copy()** должны быть или встроенных типов, или совсем неинициализированными. Пример:

```
vector<string> vs {"Breugel","El Greco","Delacroix","Constable"};
vector<string> vs2 {"Hals","Goya","Renoir","Turner"};
copy(vs.begin(), vs.end(), vs2.begin());                    // OK
uninitialized_copy(vs.begin(), vs.end(), vs2.begin());      // утечка!
```

Еще ряд средств для работы с неинициализированный памятью представленны в §34.6.

32.5.7. swap()

Алгоритм **swap()** обменивает значения двух объектов:

Семейство алгоритмов swap (§ iso.25.3.3)	
swap(x,y)	Обменять значения **x** и **y**
p=swap_ranges(b,e,b2)	**swap(v,v2)** соответствующих эл-ов в [**b**:**e**) и [**b2**,**b2+(e-b)**)
iter_swap(p,q)	**swap(*p,*q)**

Пример:

```
void use(vector<int>& v, int* p)
{
    swap_ranges(v.begin(), v.end(), p); // обмен значениями
}
```

Указатель **p** должен указывать на массив, у которого по крайней мере **v.size()** элементов.

Алгоритм **swap()**, наверное, самый простой и возможно самый критически важный в стандартной библиотеке. Он используется в реализациях многих наиболее интенсивно используемых алгоритмов. Его реализация рассматривалась в качестве примера в §7.7.2, а версия стандартной библиотеки будет представлена в §35.5.2.

32.6. Сортировка и поиск

Сортировка и поиск в отсортированных последовательностях имеют фундаментальное значение, и нужды программистов в этом вопросе сильно разнятся. Сравнение по умолчанию выполняется операцией **<**, а эквивалентность величин **a** и **b** определяется скорее посредством **!(a<b)&&!(b<a)**, а не операцией **==**.

Семейство алгоритмов swap (§ iso.25.4.1)	
sort(b,e)	Сортировать [**b**:**e**)
sort(b,e,f)	Сортировать [**b**:**e**), используя **f(*p,*q)**, как критерий сортировки

Кроме «простой» версии имеются еще и много иных вариантов:

Семейство алгоритмов sort (§ iso.25.4.1)	
stable_sort(b,e)	Сортировать [b:e), сохраняя порядок равных эл-ов
stable_sort(b,e,f)	Сортировать [b:e), используя f(*p,*q), как критерий сортировки, сохраняя порядок равных эл-ов
partial_sort(b,m,e)	Сортировать достаточно эл-ов [b:e), чтобы расставить эл-ы [b:m) по порядку; [m:e) не требует сортировки
partial_sort(b,m,e,f)	Сортировать достаточно эл-ов [b:e), чтобы расставить эл-ы [b:m) по порядку, используя f(*p,*q), как критерий сортировки; [m:e) не требует сортировки
p=partial_sort_copy(b,e,b2,e2)	Сортировать достаточно эл-ов [b:e), чтобы копировать первые e2-b2 эл-ов в [b2:e2); p есть наименьшее из b2 и b2+(e-b)
p=partial_sort_copy(b,e,b2,e2,f)	Сортировать достаточно эл-ов [b:e), чтобы копировать первые e2-b2 эл-ов в [b2:e2), используя f для сравнения; p есть наименьшее из b2 и b2+(e-b)
is_sorted(b,e)	Отсортирован ли [b:e)?
is_sorted(b,e,f)	Отсортирован ли [b:e), используя f для сравнения?
p=is_sorted_until(b,e)	p указывает на первый эл-т [b:e), который идет не по порядку
p=is_sorted_until(b,e,f)	p указывает на первый эл-т [b:e), который идет не по порядку, используя f для сравнения
nth_element(b,n,e)	*n есть позиция, в к-рой был бы эл-т, если бы [b:e) был отсортирован; эл-ты [b:n) находятся <= *n, а *n <= суть эл-ы [n:e)
nth_element(b,n,e,f)	*n есть позиция, в к-рой был бы эл-т, если бы [b:e) был отсортирован; эл-ты [b:n) находятся <= *n, а *n <= суть эл-ы [n:e), используя f для сравнения

Алгоритмы **sort()** требуют итераторов с произвольным доступом (§33.1.2).

Несмотря на свое имя, **is_sorted_until()** возвращает итератор, а не значение типа **bool**.

Стандартный **list** (§31.3) не предоставляет итераторов с произвольным доступом, так что списки следует сортировать, используя специфические для типа **list** (§31.4.2) операции, или же копированием элементов в **vector** с последующей сортировкой последнего и обратным копированием элементов назад в список:

```
template<typename List>
void sort_list(List& lst)
{
    vector v {lst.begin(), lst.end()};    // инициализируем из lst
    sort(v);                              // используем контейнерную сортировку (§ 32.2)
    copy(v, lst);
}
```

Базовый вариант алгоритма **sort()** весьма эффективен (в среднем **N*log(N)**). Если требуется стабильная сортировка, то следует использовать **stable_sort()**, то есть алгоритм с оценкой **N*log(N)*log(N)**, улучшающейся до **N*log(N)** в случае систем с достаточно большой памятью. Дополнительную память можно запросить с помощью **get_temporary_buffer()** (§34.6). Относительный порядок эквивалентных элементов сохраняется в **stable_sort()**, но не в **sort()**.

Иногда нам нужны лишь первые элементы в отсортированных последовательностях. В таком случае имеет смысл сортировать последовательность в достаточной мере, чтобы получить лишь первую часть, то есть выполнить частичную сортировку. Алгоритм **partial_sort(b, m, e)** расставляет элементы из диапазона [b:m) по порядку. Алгоритм **partial_sort_copy()** производит **N** элементов, где **N** это наименьшее из количества элементов в выходной последовательности и числа элементов во входной последовательности. Нужно специфи-

цировать как начало, так и конец результирующей последовательности, потому что это то, что определяет, как много элементов нам нужны для сортировки. Например:

```
void f(const vector<Book>& sales)      // ищем верхние 10 книг
{
    vector<Book> bestsellers(10);
    partial_sort_copy(sales.begin(), sales.end(),
        bestsellers.begin(), bestsellers.end(),
        [](const Book& b1, const Book& b2) {return b1.copies_sold()>b2.copies_sold();});
    copy(bestsellers.begin(), bestsellers.end(), ostream_iterator<Book>{cout,"\n"});
}
```

Поскольку целью для **partial_sort_copy()** служит итератор с произвольным доступом, то мы не можем напрямую сортировать в **cout**.

Если число желаемых для сортировки посредством **partial_sort()** элементов мало по сравнению с полным числом элементов, то эти алгоритмы могут оказаться существенно более быстрыми, чем выполнение полного **sort()**. В таком случае их оценка приближается к **O(N)** по сравнению с **O(N*log(N))** для **sort()**.

Алгоритм **nth_element()** сортирует ровно до тех пределов, которые достаточны для помещения **N**-го элемента на его надлежащее место при отсутствии меньших элементов, помещаемых после **N**-го элемента в последовательсноти. Например:

```
vector<int> v;
for (int i=0; i<1000; ++i)
    v.push_back(randint(1000)); // §40.7
constexpr int n = 30;
nth_element(v.begin(), v.begin()+n, v.end());
cout << "nth: " << v[n] < '\n';
for (int i=0; i<n; ++i)
    cout << v[i] << ' ';
```

Из этой программы мы получаем на выходе:

```
nth: 24
10815192115876172121881932120181073381111222223
```

Алгоритм **nth_element()** отличается от **partial_sort()** тем, что элементы перед **n** не обязательно отсортированы, хотя и все меньше, чем **n**-ый элемент. Заменяя в нашем примере **nth_element()** на **partial_sort()**, получаем:

```
nth: 995
12333677888891010111115151718181920212121222223
```

Алгоритм **nth_element()** особо полезен экономистам, учителям и всем, кому полезны медианные оценки, персентили и т.д.

Сортировка строк в С-стиле требует явного критерия сортировки. Причиной служит то, что С-строки есть просто указатели с некоторым числом соглашений по их использованию, так что операция **<** на указателях сравнивает скорее машинные адреса, чем символы в последовательностях. Пример:

```
vector<string> vs = {"Helsinki","Copenhagen","Oslo","Stockholm"};
vector<char*> vcs = {"Helsinki","Copenhagen","Oslo","Stockholm"};

void use()
{
    sort(vs);       // я определил версию sort() с проверкой диапазона
    sort(vcs);
    for (auto& x : vs)
```

```
        cout << x << ' '
    cout << '\n';
    for (auto& x : vcs)
        cout << x << ' ';
```

Вот вывод этой программы:

```
Copenhagen Helsinki Stockholm Oslo
Helsinki Copenhagen Oslo Stockholm
```

По наивности мы могли ожидать одинакового результата для обоих векторов. Однако, для сортировки С-строк по содержимому строк, а не по адресам, нам нужен предикат сортировки. Пример:

```
sort(vcs, [](const char* p, const char* q){return strcmp(p, q)<0;});
```

Функция **strcmp()** из стандартной библиотеки описывается в §43.4.

Заметьте, что я не должен был предоставлять операцию == для сортировки С-строк. Для упрощения пользовательского интерфейса стандартная библиотека использует **!(x<y||y<x)**, а не == для сравнения элементов (§31.2.2.2).

32.6.1. Бинарный поиск

Семейство алгоритмов **binary_search()** обеспечивает бинарный поиск в сортированных последовательностях:

Бинарный поиск (§ iso.25.4.3)	
p=lower_bound(b,e,v)	**p** указывает на первый **v** в **[b:e)**
p=lower_bound(b,e,v,f)	**p** указывает на первый **v** в **[b:e)**, используя **f** для сравнения
p=upper_bound(b,e,v)	**p** указывает на первое знач-е, большее **v** в **[b:e)**
p=upper_bound(b,e,v,f)	**p** указывает на первое знач-е, большее **v** в **[b:e)**, используя **f** для сравнения
binary_search(b,e,v)	Имеется ли **v** в отсортированной последовательности **[b:e)**?
binary_search(b,e,v,f)	Имеется ли **v** в отсортированной последовательности **[b:e)**, используя **f** для сравнения?
pair(p1,p2)=equal_range(b,e,v)	**[p1,p2)** есть подмножество **[b:e)** со знач-ем **v**; практически, бинарный поиск **v**
pair(p1,p2)=equal_range(b,e,v,f)	**[p1,p2)** есть подмножество **[b:e)** со знач-ем **v**, используя **f** для сравнения; практически, бинарный поиск **v**

Последовательный поиск вроде **find()** (§32.4) ужасно неэффективен для больших последовательностей, но это лучшее, что мы можем сделать без сортировки и хэширования (§31.4.3.2). Но если последовательность отсортирована, то мы можем применить бинарный поиск для ответа на вопрос, находится ли значение в последовательности. Пример:

```
void f(vector<int>& c)
{
    if (binary_search(c.begin(), c.end(),7)) { // 7 в c?
        // ...
    }
    // ...
}
```

Алгоритм **binary_search()** возвращает **bool**, индицирующий факт присутствия значения в последовательности. Как и в случае с **find()**, мы также часто хотим знать, где элементы

с этим значением находятся в последовательности. Но таких элементов может быть много, и поэтому можно найти первый из них или все такие элементы. Следовательно, существуют алгоритмы для нахождения диапазона эквивалентных элементов, **equal_range()**, и алгоритмы для нахождения **lower_bound()** и **upper_bound()** этого диапазона. Эти алгоритмы соответствуют операциям над **multimap** (§31.4.3). Мы можем думать о **lower_bound()** как о быстром **find()** и **find_if()** для сортированных последовательностей. Например:

```
void g(vector<int>& c)
{
    auto p = find(c.begin(), c.end(),7);          // медленно: O(N); с не надо сортировать
    auto q = lower_bound(c.begin(), c.end(),7);   // быстро: O(log(N)); с нужно сортировать
    // ...
}
```

Если **lower_bound(first, last, k)** не находит **k**, то возвращается итератор на первый элемент с ключом, большим чем **k**, или **last**, если таковой не существует. Этот способ сообщения о неудаче используется также и в **upper_bound()** и **equal_range()**. Это означает, что мы можем использовать эти алгоритмы для выявления места, где можно внедрить элемент в сортированную последовательность так, что последовательность останется сортированной: просто элемент внедряется перед **second** в возвращенном **pair**.

Как ни странно, но алгоритмы бинарного посика не требуют итераторов произвольного доступа: достаточно прямых итераторов.

32.6.2. merge()

Алгоритмы **merge()** комбинируют две упорядоченных последовательности в одну:

Семейство алгоритмов merge (§iso.25.4.4)	
p=merge(b,e,b2,e2,out)	Объединить две отсортированные последовательности [**b2:e2**) и [**b:e**) в [**out:p**)
p=merge(b,e,b2,e2,out,f)	Объединить две отсортированные последовательности [**b2:e2**) и [**b:e**) в [**out, out+p**), используя **f** для сравнения
inplace_merge(b,m,e)	Объединить две отсортированные подпоследовательности [**b:m**) и [**m:e**) в отсортированную послед-ть [**b:e**)
inplace_merge(b,m,e,f)	Объединить две отсортированные подпоследовательности [**b:m**) и [**m:e**) в отсортированную послед-ть [**b:e**), используя **f** для сравнения

Алгоритм **merge()** может принимать разные виды последовательностей и элементы разных типов. Например:

```
vector<int> v {3,1,4,2};
list<double> lst {0.5,1.5,2,2.5};      // lst отсортирован

sort(v.begin(), v.end());              // отсортируйте v

vector<double> v2;
merge(v.begin(), v.end(), lst.begin(), lst.end(), back_inserter(v2)); // сливаем v и lst в v2
for (double x: v2)
    cout << x << ", ";
```

Насчет итераторов вставки см. §33.2.2. А вывод программы такой:

```
0.5, 1, 1.5, 2, 2, 2.5, 3, 4,
```

32.6.3. Алгоритмы для множеств

Эти алгоритмы трактуют последовательность как множество элементов и предоставляют набор базовых операций над соответствующим множеством. Входные последовательности предполагаются отсортированными и выходные последовательности также являются сортированными.

Алгоритмы для множеств (§iso.25.4.5)	
includes(b,e,b2,e2)	Входят ли эл-ы [b:e) также в [b2:e2)?
includes(b,e,b2,e2,f)	Входят ли эл-ы [b:e) также в [b2:e2), используя f для сравнения?
p=set_union(b,e,b2,e2,out)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят в [b:e) или в [b2:e2)
p=set_union(b,e,b2,e2,out,f)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят в [b:e) или в [b2:e2), используя f для сравнения
p=set_intersection(b,e,b2,e2,out)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят и в [b:e), и в [b2:e2)
p=set_intersection(b,e,b2,e2,out,f)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят и в [b:e), и в [b2:e2), используя f для сравнения
p=set_difference(b,e,b2,e2,out)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят в [b:e), но не в [b2:e2)
p=set_difference(b,e,b2,e2,out,f)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят в [b:e), но не в [b2:e2), используя f для сравнения
p=set_symmetric_difference(b,e,b2,e2,out)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят в [b:e) или в [b2:e2), но не в оба
p=set_symmetric_difference(b,e,b2,e2,out,f)	Создать отсортированную последовательность [out:p) из эл-ов, к-рые входят в [b:e) или в [b2:e2), но не в оба, используя f для сравнения

Пример:

```
string s1 = "qwertyasdfgzxcvb";
string s2 = "poiuyasdfg/.,mnb";
sort(s1.begin(), s1.end());              // алг-мы для set требуют отсортированных последовательностей
sort(s2.begin(), s2.end());

string s3(s1.size()+s2.size(),'*');      // отложим достаточно памяти для максим-го результата
cout << s3 << '\n';
auto up = set_union(s1.begin(), s1.end(), s2.begin(), s2.end(), s3.begin());
cout << s3 << '\n';
for (auto p = s3.begin(); p!=up; ++p)
     cout << *p;
cout << '\n';

s3.assign(s1.size()+s2.size(),'+');
up = set_difference(s1.begin(), s1.end(), s2.begin(), s2.end(), s3.begin());
cout << s3 << '\n';
for (auto p = s3.begin(); p!=up; ++p)
     cout << *p;
cout << '\n';
```

Этот тест производит такой вывод:

```
******************************
,./abcdefgimnopqrstuvxyz
ceqrtvwxz++++++++++++++++++++++
ceqrtvwxz
```

32.6.4. Кучи

Куча — это компактная структура данных, которая первым хранит элемент с наибольшим значением. Можно думать о куче как о бинарном дереве. Алгоритмы для кучи позволяют программисту трактовать последовательность с произвольным доступом как кучу:

Операции с кучами (§iso.25.4.6)	
make_heap(b,e)	Подготовить [**b**:**e**) для использ-я **a** качестве кучи
make_heap(b,e,f)	Подготовить [**b**:**e**) для использ-я **a** качестве кучи, используя **f** для сравнения
push_heap(b,e)	Добавить ***(e-1)** к куче [**b**:**e-1**); после чего, [**b**:**e**) страновится кучей
push_heap(b,e,f)	Добавить эл-т к куче [**b**:**e-1**), используя **f** для сравнения
pop_heap(b,e)	Удалить ***(e-1)** из кучи [**b**:**e**); после чего, [**b**:**e-1**) страновится кучей
pop_heap(b,e,f)	Удалить эл-т из кучи [**b**:**e**), используя **f** для сравнения
sort_heap(b,e)	Сортировать кучу [**b**:**e**)
sort_heap(b,e,f)	Сортировать кучу [**b**:**e**), используя **f** для сравнения
is_heap(b,e)	Является ли [**b**:**e**) кучей?
is_heap(b,e,f)	Является ли [**b**:**e**) кучей, используя **f** для сравнения?
p=is_heap_until(b,e)	**p** есть наибольшее **p** такое, что [**b**:**p**) есть куча
p=is_heap_until(b,e,f)	**p** есть наибольшее **p** такое, что [**b**:**p**) есть куча, используя **f** для сравнения

Думайте о конце, **e**, кучи [**b**:**e**) как об указателе, который декрементируется посредством **pop_heap()** и инкрементируется посредством **push_heap()**. Наибольший элемент извлекается чтением через **b** (например, **x=*b**) и затем выполняется **push_heap()**. Например:

```
string s = "herewego";
make_heap(s.begin(), s.end());          // rogheeew
pop_heap(s.begin(), s.end());           // rogheeew
pop_heap(s.begin(), s.end()-1);         // ohgeeerw
pop_heap(s.begin(), s.end()-2);         // hegeeorw

*(s.end()-3)='f';
push_heap(s.begin(), s.end()-2);        // hegeefrw
*(s.end()-2)='x';
push_heap(s.begin(), s.end()-1);        // xeheefge
*(s.end()-1)='y';
push_heap(s.begin(), s.end());          // yxheefge
sort_heap(s.begin(), s.end());          // eeefghxy
reverse(s.begin(), s.end());            // yxhgfeee
```

Способ понять изменения в **s** состоит в том, чтобы читать только **s[0]** и писать только **s[x]**, где **x** это индекс текущего конца кучи. Куча устраняет элемент (всегда **s[0]**) посредством обмена его (swapping) с **s[x]**.

Целью кучи является предоставить быстрое добавление элементов и быстрый доступ к элементу с наибольшим значением. Самым очевидным применением куч является реализация приоритетных очередей.

32.6.5. lexicographical_compare()

Лексикографическое сравнение — это правило, которым мы руководствуется для упорядочения слов в словарях.

Лексикографическое сравнение (§iso.25.4.8)	
lexicographical_compare(b,e,b2,e2)	[b:e)<[b2:e2)?
lexicographical_compare(b,e,b2,e2,f)	[b:e)<[b2:e2), используя f для сравнения?

Мы могли бы реализовать **lexicographical_compare(b, e, b2, e2)** следующим образом:

```
template<class In, class In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2)
{
    for (; first!=last && first2!=last2; ++first,++last) {
        if (*first<*first2)
            return true;                // [first:last)<[first2:last2)
        if (*first2<*first)
            return false;               // [first2:last2)<[first:last)
    }
    return first==last && first2!=last2; // [first:last)<[first2:last2) если [first:last) короче
}
```

То есть, строка сравнивается как последовательность символов. Пример:

```
string n1 {"10000"};
string n2 {"999"};

bool b1 = lexicographical_compare(n1.begin(), n1.end(), n2.begin(), n2.end()); // b1 == true

n1 = "Zebra";
n2 = "Aardvark";
bool b2 = lexicographical_compare(n1.begin(), n1.end(), n2.begin(), n2.end()); // b2 == false
```

32.7. Минимумы и максимумы

Сравнение значений полезно во многих контекстах:

Семейство алгоритмов min и max (§iso.25.4.7)	
x=min(a,b)	x есть меньшее из a и b
x=min(a,b,f)	x есть меньшее из a и b, используя f для сравнения
x=min({elem})	x есть наименьший эл-т в {elem}
x=min({elem}, f)	x есть наименьший эл-т в {elem}, используя f для сравнения эл-ов
x=max(a,b)	x есть большее из a и b
x=max(a,b,f)	x есть большее из a и b, используя f для сравнения
x=max({elem})	x есть наибольший эл-т в {elem}
x=max({elem}, f)	x есть наибольший эл-т в {elem}, используя f для сравнения эл-ов
pair(x, y)=minmax(a,b)	x есть min(a,b), а y есть max(a,b)
pair(x, y)=minmax(a,b,f)	x есть min(a,b,f), а y есть max(a,b,f)
pair(x, y)=minmax({elem})	x есть min({elem}), а y есть max({elem})
pair(x, y)=minmax({elem}, f)	x есть min({elem}, f), а y есть max({elem}, f)

Продолжение

Семейство алгоритмов min и max (§iso.25.4.7)	
p=min_element(b,e)	p указывает на наименьший эл-т [b:e) или e
p=min_element(b,e,f)	p указывает на наименьший эл-т [b:e) или e, используя f для сравнения эл-ов
p=max_element(b,e)	p указывает на наибольший эл-т [b:e) или e
p=max_element(b,e,f)	p указывает на наибольший эл-т [b:e) или e, используя f для сравнения эл-ов
pair(x, y)=minmax_element(b,e)	x есть min_element(b,e), а y есть max_element(b,e)
pair(x, y)=minmax_element(b,e,f)	x есть min_element(b,e,f), а y есть max_element(b,e,f)

Если мы сравниваем два l-value, то получаем ссылку на результат; в противном случае возвращается rvalue. К сожалению, версия, которая принимает lvalue, принимает их как **const** lvalue, так что вы не сможете модифицировать результат одной из этих функций. Например:

```
int x = 7;
int y = 9;

++min(x, y);    // результат min(x, y) имеет тип const int&
++min({x, y}); // ошибка: результат min({x, y}) есть rvalue (initializer_list неизменяем)
```

Функции с **_element** возвращают итераторы; функция **minmax** возвращает тип **pair**, и мы можем написать:

```
string s = "Large_Hadron_Collider";
auto p = minmax_element(s.begin(), s.end(),
                [](char c1, char c2) {return toupper(c1)<toupper(c2);});
cout << "min==" << *(p.first) << ' ' << "max==" << *(p.second) << '\n';
```

При ASCII символах на моей машине последний тест дает следующий вывод:

```
min==a max==_
```

32.8. Советы

[1] STL алгоритм оперирует на одной или более последовательностях; §32.2.

[2] Входная последовательность полуоткрытая и определяется парой итераторов; §32.2.

[3] При поиске алгоритм обычно возвращает конец последовательности для индикации результата «не найдено»; §32.2.

[4] Предпочитайте тщательно специфицированные алгоритмы «случайному» ручному коду; §32.2.

[5] При написании цикла подумайте, нельзя ли заменить его на общий алгоритм; §32.2.

[6] Удостоверьтесь, что пара аргументов-итераторов на самом деле специфицируют последовательность; §32.2.

[7] Когда вам наскучит стиль с парой итераторов начните использовать алгоритмы в стиле контейнер/диапазон; §32.2.

[8] Используйте предикаты и иные функциональные объекты для того, чтобы придать стандартным алгоритмам более широкое смысловое поле; §32.3.

[9] Предикат не должен изменять свой аргумент; §32.3.

[10] Умолчательные операции == и < на указателях редко бывают адекватными для стандартных алгоритмов; §32.3.

[11] Знайте сложности применяемых вами алгоритмов, но при этом не забывайте, что меры сложности являются грубым приближением к истинной производительности; §32.3.1.

[12] Используйте **for_each()** и **transform()** только тогда, когда нет более специфичного алгоритма для вашей задачи; §32.4.1.

[13] Алгоритмы не добавляют и не удаляют элементы напрямую из своих аргументов-последовательностей; §32.5.2, §32.5.3.

[14] Если вам требуется иметь дело с неинициализированными объектами, рассмотрите применение **uninitialized_*** алгоритмов; §32.5.6.

[15] STL алгоритмы используют сравнение на эквивалентность, исходя из сравнения на упорядочение, а не операцию ==; §32.6.

[16] Заметьте, что сортировка и поиск для С-строк требуют от пользователя предоставить операцию сравнения строк; §32.6.

33

STL итераторы

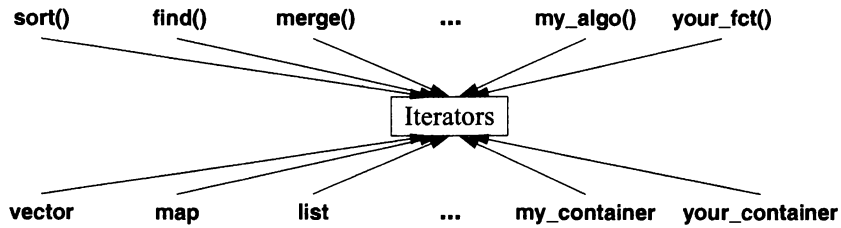
Причина, по которой STL алгоритмы и итераторы работают столь хорошо совместно, состоит в том, что они ничего не знают друг о друге
— Александр Степанов

- Введение
- Модель итераторов
  Категории итераторов; Характеристики итераторов; Операции над итераторами
- Адаптеры итераторов
  Обратные итераторы; Итераторы вставки; Перемещающий итератор
- Функции доступа к диапазону
- Функциональные объекты
- Адаптеры функций
  bind(); mem_fn(); function
- Советы

33.1. Введение

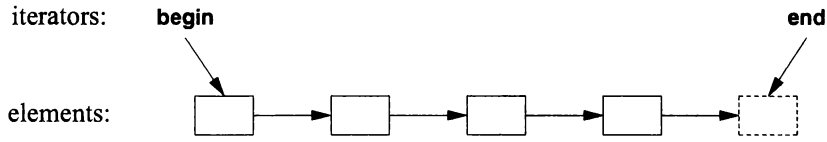
В этой главе рассматриваются STL итераторы и утилиты, а также замечательные функциональные объекты стандартной библиотеки. STL состоит из итераторов, контейнеров, алгоритмов и функциональных объектов. Отдельные части STL были представлены в Главах 31 и 32.

Итераторы служат связующим веществом («клеем») для алгоритмов и их данных. Можно сказать, что итераторы являются механизмом, использующимся для минимизации зависимости алгоритмов от структур данных, на которых они оперируют:



33.1.1. Модель итераторов

Итератор сродни указателю в том отношении, что он обеспечивает возможность выполнения операций при косвенном доступе (как при помощи операции разыменования для указателей) и имеет возможность перемещения для указания нового элемента (вроде операции ++ для перемещения указателя на следующий элемент). Последовательность определяется парой итераторов, определяющих полуоткрытый даиапазон **[begin:end)**:



То есть, **begin** указывает на первый элемент последовательности, а **end** указывает на позицию за последним элементом последовательности. Никогда не выполняйте чтение или запись по ***end**. Заметьте, что для пустой последовательности **begin==end**; то есть **[p:p)** является пустой последовательностью для любого итератора **p**.

Для того, чтобы читать последовательность, алгоритм берет пару итераторов **(b,e)** и выполняет траверс по элементам с помощью операции ++ до тех пор, пока не будет достигнут конец последовательности:

```
while (b!=e) {              // используйте != а не <
    // ... что-то делаем здесь ...
    ++b; // переходим к следующему элементу
}
```

Причина, по которой использовалась операция **!=** вместо **<**, состоит в том, что детекция конца последовательности фиксируется при этом чуть отчетливее с логической точки зрения, а еще потому, что только итераторы произвольного доступа поддерживают операцию **<**.

Алгоритмы, которые разыскивают что-либо в последовательности, обычно возвращают конец последовательности для индикации результата «не найдено»:

```
auto p = find(v.begin(), v.end(), x);       // ищем x в v

if (p!=v.end()) {
    // x найден в p
}
else {
    // x не найден в [v.begin():v.end())
}
```

Алгоритмы, которые пишут в последовательность, часто получают лишь один итератор на первый элемент. В таком случае на программиста ложится ответственность, чтобы не писать в область за концом последовательности. Например:

```
template<typename Iter>
void forward(Iter p, int n)
{
    while (n>0)
        *p++ = --n;
}
```

```
void user()
{
    vector<int> v(10);
    forward(v.begin(), v.size());    // OK
    forward(v.begin(),1000);         // большая проблема
}
```

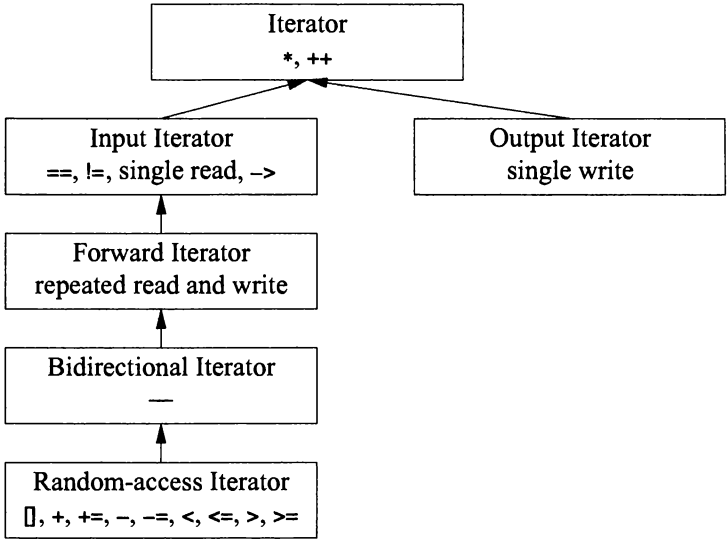
В некоторых реализациях стандартной библиотеки производится проверка диапазона — то есть генерируется исключение для последнего вызова **forward()**, но вы не можете полагаться на это свойство при написании переносимого кода; многие реализации не выполняют генерации исключения. В качестве надежной альтернативы применяйте итераторы вставки (§33.2.2).

33.1.2. Категории итераторов

В стандартной библиотеке имеются пять видов итераторов (пять *категорий итераторов*):

- *Итераторы ввода*: мы можем выполнять прямой траверс, применяя ++, и последовательно читать каждый элемент, используя *. Можно сравнивать прямые итераторы, используя == и !=. Данный вид итератора предлагает **istream** (см. §38.5).
- *Итераторы вывода:* мы можем продвигаться вперед при помощи ++ и прописывать элементы при помощи *. Этот вид итератора предлагает **ostream** (см. §38.5).
- *Прямые итераторы*: мы можем продвигаться вперед при помощи ++ и читать или прописывать элементы (если только они не **const**) при помощи *. Если прямой итератор указывает на объект класса, то можно использовать операцию -> для ссылки на члены. Можно сравнивать прямые итераторы операциями == и !=. Этот вид итератора предлагает **forward_list** (см. §31.4).
- *Двунаправленные итераторы*: можно продвигаться вперед при помощи ++ и назад при помощи — и читать или писать (если только элементы не **const**) при помощи *. Если двунаправленный итератор указывает на объект класса, то можно использовать операцию -> для ссылки на члены. Можно сравнивать двунаправленные итераторы операциями == и !=. Этот вид итератора предлагают контейнеры **list**, **map** и **set** (см. §31.4).
- *Итераторы произвольного доступа*: можно продвигаться вперед (используя ++ или +=) или назад (используя - или -=)и читать или писать элементы (если только они не **const**) при помощи * или []. Если итератор произвольного доступа указывает на объект класса, то можно использовать операцию -> для ссылки на члены. Можно индексировать итераторы произвольного доступа при помощи операции [], можно прибавлять целые значения операцией + и вычитать целые значения посредством операции -. Можно находить расстояние между двумя итераторами произвольного доступа на одну и ту же последовательность, вычитая один из другого. Можно сравнивать итераторы произвольного доступа при помощи ==, !=, <, <=, > и >=. Этот вид итератора предлагает контейнер **vector** (см. §31.4).

С логической точки зрения итераторы организованы в иерархию (§ iso.24.2):



Категории итераторов скорее являются концепциями (§24.3), а не классами, так что эта иерархия не является классовой иерархией, реализуемой посредством механизма наследования. Если вы хотите проделать с категориями итераторов что-нибудь продвинутое, то используйте **iterator_traits** (прямо или косвенно).

33.1.3. Характеристики итераторов

В файле **<iterator>** стандартная библиотека предлагает набор функций типа, которые позволяют нам писать код, специализированный под специфические свойства итераторов:

Характеристики итераторов (§ iso.24.4.1)	
iterator_traits<Iter>	Тип характеристик для не указателя **Iter**
iterator_traits<T*>	Тип характеристик для указателя **T***
iterator<Cat,T,Dist,Ptr,Re>	Простой класс, определяющий основные типы-члены итератора
input_iterator_tag	Категория для итераторов ввода
output_iterator_tag	Категория для итераторов вывода
forward_iterator_tag	Категория для прямых итераторов; производная от **input_iterator_tag**; предназначена для **forward_list**, **unordered_set**, **unordered_multiset**, **unordered_map**, and **unordered_multimap**
bidirectional_iterator_tag	Категория для двунаправленных итераторов; производная от **forward_iterator_tag**; предназначена для **list**, **set**, **multiset**, **map**, **multimap**
random_access_iterator_tag	Категория для итераторов произвольного доступа; производная от **bidirectional_iterator_tag**; предназначена для **vector**, **deque**, **array**, встроенные массивы и, **string**

Тэги и типы итераторов используются для выбора среди алгоритмов, базирующихся на типе итератора. Например, итератор произвольного доступа может непосредственно обращаться к элементу:

```
template<typename Iter>
void advance_helper(Iter p, int n, random_access_iterator_tag)
{
    p+=n;
}
```

С другой стороны, прямой итератор должен переходить к **n**-ому элементу за одно перемещение за раз (например, следуя связям в списке):

```
template<typename Iter>
void advance_helper(Iter p, int n, forward_iterator_tag)
{
    if (0<n)
        while (n--) ++p;
    else if (n<0)
        while (n++) --p;
}
```

Исходя из представленных вспомогательных функций **advance()** может использовать оптимальный алгоритм:

```
template<typename Iter>
void advance(Iter p, int n)              // используем оптимальный алгоритм
{
    advance_helper(p, n, typename iterator_traits<Iter>::iterator_category{});
}
```

В типичном случае **advance()** и/или **advance_helper()** должны быть встраиваемыми, чтобы гарантировать отсутствие дополнительных накладных расходов во время выполнения, что реализуется технологией диспетчеризации на основе тэгов (*tag dispatch*).

Ключевые свойства итераторов описываются синонимами в **iterator_traits**:

```
template<typename Iter>
struct iterator_traits {
    using value_type = typename Iter::value_type;
    using difference_type = typename Iter::difference_type;
    using pointer = typename Iter::pointer;                    // тип указателя
    using reference = typename Iter::reference;                // тип ссылки
    using iterator_category = typename Iter::iterator_category; // (тэг)
};
```

Для итератора, не имеющего типов-членов (например, **int***), мы предоставляем специализацию **iterator_traits**:

```
template<typename T>
struct iterator_traits<T*>{          // специализация для указателей
    using difference_type = ptrdiff_t;
    using value_type = T;
    using pointer = T*;
    using reference = T& reference;
    using iterator_category = random_access_iterator_tag;
};
```

В общем случае мы не можем просто сказать

```
template<typename Iter>
typename Iter::value_type read(Iter p, int n) // нет общности
{
    // ... что-то проверяем ...
    return p[n];
{
```

Вызов **read()** с аргументом-указателем будет ошибкой. Компилятор поймает эту ошибку, но сообщение может оказаться весьма туманным. Вместо этого мы можем написать следующее:

```
template<typename Iter>
typename iterator_traits<Iter>::value_type read(Iter p, int n) // более общее решение
{
    // ... что-то проверяем ...
    return p[n];
{
```

Идея заключается в том, чтобы в случае необходимости найти свойство итератора, смотреть в его **iterator_traits** (§28.2.4). Чтобы напрямую не ссылаться на **iterator_traits**, которые зависят от реализации, можно ввести синоним. Пример:

```
template<typename Iter>
using Category<Iter> = typename std::iterator_traits<Iter>::iterator_category;

template<typename Iter>
using Difference_type<Iter> = typename std::iterator_traits<Iter>::difference_type;
```

Таким образом, если мы захотим узнать тип различий между двумя итераторами, указывающими на одну и ту же последовательность, перед нами несколько возможностей:

```
tempate<typename Iter>
void f(Iter p, Iter q)
{
    Iter::difference_type d1 = distance(p, q);              // отсутствует "typename"

    typename Iter::difference_type d2 = distance(p, q);  // не работает для указ-лей и т.д.

    typename iterator_traits<Iter>::distance_type d3 = distance(p, q);  // OK, но грубо
    Distance_type<Iter> d4 = distance(p, q);                            // OK, много лучше

    auto d5 = distance(p, q);      // OK, если вам не нужно явно ссылаться на тип
    // ...
}
```

Я бы рекомендовал последние две альтернативы.

Шаблон **iterator** просто увязывает ключевые свойства итератора в структуру для удобства реализации итераторов и предоставляет несколько умолчаний:

```
template<typename Cat, typename T, typename Dist = ptrdiff_t, typename Ptr = T*, typename Ref = T&>
struct iterator {
    using value_type = T;
    using difference_type = Dist;       // тип для distance()
    using pointer = Ptr;                // тип указателя
    using reference = Ref;              // тип ссылки
    using iterator_category = Cat;      // категория (тэг)
};
```

33.1.4. Операции над итераторами

В зависимости от категории (§33.1.2) для итераторов предоставляются некоторые или все из следующих операций:

Операции над итераторами (§ iso.24.2.2)	
`++p`	Префиксный инкремент (вперед на один эл-т): `p` указывает на следующий эл-т или на эл-т, находящийся за последним; возвращает уже увеличенное значение
`p++`	Постфиксный инкремент (вперед на один эл-т): `p` указывает на следующий эл-т или на эл-т, находящийся за последним; значение выраж-я есть значение `p` до инкремента
`*p`	Доступ (разыменование): `*p` обозначает эл-т, на к-рый указывает `p`
`--p`	Префиксный декремент (назад на один эл-т): `p` указывает на предыдущий эл-т; значение выраж-я есть уже уменьшенное значение
`p--`	Постфиксный декремент (назад на один эл-т): `p` указывает на предыдущий эл-т; значение выраж-я есть значение `p` до декремента
`p[n]`	Доступ (индексирование): p[n] обозначает эл-т, на к-рый указывает p+n; эквивалентно *(p+n)
`p->n`	Доступ (доступ к члену): эквивалентно `(*p).m`
`p==q`	Равенство: Указывают ли `p` и `q` на один и тот же эл-т или оба они указывают на эл-т, находящийся за последним?
`p!=q`	Неравенство: `!(p==q)`
`p<q`	Указывает ли `p` на эл-т, находящийся перед тем, на к-рый указывает `q`?
`p<=q`	`p<q || p==q`
`p>q`	Указывает ли `p` на эл-т, находящийся после того, на к-рый указывает `q`?
`p>=q`	`p>q || p==q`
`p+=n`	Передвинуть на `n`: пусть `p` указывает на `n`-ый эл-т после того, на к-рый он указывет сейчас
`p-=n`	Передвинуть на `-n`: пусть `p` указывает на `n`-ый эл-т перед тем, на к-рый он указывет сейчас
`q=p+n`	`q` указывает на `n`-ый эл-т после того, на к-рый указывет `p`
`q=p-n`	`q` указывает на `n`-ый эл-т перед тем, на к-рый указывет `p`

Операция `++p` возвращает ссылку на `p`, в то время как `p++` должен возвращать копию `p`, содержащую старое значение. Таким образом, для сложных итераторов `++p` более эффективен, чем `p++`.

Следующие операции работают со всеми итераторами, для которых они могут быть реализованы, но эффективно они работают для итераторов произвольного доступа (см. §33.1.2):

Iterator operations (§ iso.24.4.4)	
`advance(p)`	Подобно `p+=n`; `p` должен как минимум быть итератором ввода
`x=distance(p,q)`	Подобно `x=q-p`; `p` должен как минимум быть итератором ввода
`q=next(p,n)`	Подобно `q=p+n`; `p` должен как минимум быть прямым итератором
`q=next(p)`	`q=next(p,1)`
`q=prev(p,n)`	Подобно `q=p-n`; `p` должен как минимум быть двунаправленным итератором
`q=prev(p)`	`q=prev(p,1)`

Во всех случаях, если `p` не является итератором с произвольным доступом, алгоритм займет `n` шагов.

33.2. Адаптеры итераторов

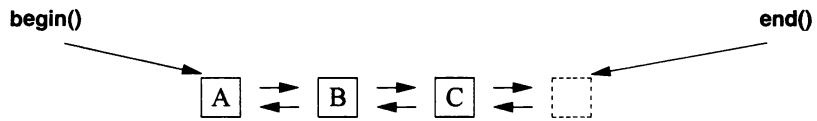
В файле **<iterator>** стандартная библиотека реализует адаптеры для генерирования полезных типов итераторов для данного итераторного типа:

Адаптеры итераторов		
reverse_iterator	Итерирует назад	§33.2.1
back_insert_iterator	Вставляет в конец	§33.2.2
front_insert_iterator	Вставляет в начало	§33.2.2
insert_iterator	Вставляет куда угодно	§33.2.2
move_iterator	Перемещает нежели, чем копирует	§33.2.3
raw_storage_iterator	Пишет в неинициализированную память	§34.6.2.

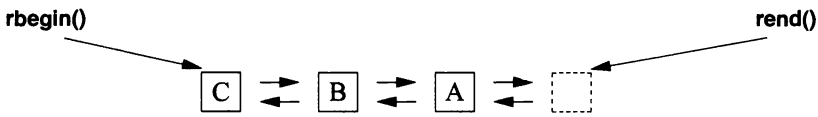
Итераторы для **iostream** описываются в §38.5.

33.2.1. Обратные итераторы

Используя итератор, мы можем выполнить траверс последовательности **[b:e)** в направлении от **b** к **e**. Если последовательность допускает двунаправленный доступ, то мы можем выполнять траверс и в обратном порядке, от **e** к **b**. Итератор, выполняющий обратный траверс, называется обратным итератором (**reverse_iterator**): он выполняет итерирование с конца последовательности, определяемого нижележащим итератором, в направлении начала последовательности. Чтобы получить полуоткрытую последовательность, мы должны рассматривать **b-1** как позицию за концом последовательности, а **e-1** как начало последовательности: **[e-1,b-1)**. Таким образом, фундаментальное отношение между обратным итератором и нижележащим итератором есть **&*(reverse_iterator(p))==&*(p-1)**. В частности, если **v** есть **vector**, то **v.rbegin()** указывает на ее первый элемент, **v[v.size()-1]**. Посмотрите на следующее:



Эта последовательность может рассматриваться следующим образом, используя **reverse_iterator**:



Определение **reverse_iterator** может выглядеть следующим образом:

```
template<typename Iter>
class reverse_iterator
    : public iterator<Iterator_category<Iter>,
                      Value_type<Iter>,
                      Difference_type<Iter>,
                      Pointer<Iter>,
                      Reference<Iter>> {
public:
    using iterator_type = Iter;
```

```
        reverse_iterator(): current{} {}
        explicit reverse_iterator(Iter p): current{p} {}
        template<typename Iter2>
            reverse_iterator(const reverse_iterator<Iter2>& p):current(p.base()) {}

        Iter base() const {return current;} // текущее значение итератора

        reference operator*() const {tmp = current; return *--tmp;}
        pointer operator->() const;
        reference operator[](difference_type n) const;

        reverse_iterator& operator++() {--current; return *this;}       // заметьте: не ++
        reverse_iterator operator++(int) {reverse_iterator t = current; --current; return t;}
        reverse_iterator& operator--() {++current; return *this;}       // заметим: не --
        reverse_iterator operator--(int) {reverse_iterator t = current; ++current; return t;}

        reverse_iterator operator+(difference_type n) const;
        reverse_iterator& operator+=(difference_type n);
        reverse_iterator operator-(difference_type n) const;
        reverse_iterator& operator-=(difference_type n);
        // ...
    protected:
        Iterator current; // current указывает на эл-т после указуемого через *this
    private:
        // ...
        iterator tmp;   // для врем-ых переменных, которым нужно "выжить" вне обл-ти дей-я функции
    };
```

У **reverse_iterator<Iter>** типы членов и операции такие же, как у **Iter**. В частности, если **Iter** является итератором с произвольным доступом, то для него **reverse_iterator<Iter>** имеет операции [], + и <. Например:

```
void f(vector<int>& v, list<char>& lst)
{
    v.rbegin()[3] = 7;              // OK: итератор произвольного доступа
    lst.rbegin()[3] = '4';          // ошибка: двунаправленный итератор не поддерживает []
    *(next(lst.rbegin(),3)) = '4';  // OK!
}
```

Я использую **next()** для продвижения итератора, потому что + не будет работать для двунаправленного итератора, такого как **list<char>::iterator**.

Обратные итераторы позволяют нам использовать алгоритмы способом, который рассматривает последовательности в обратном порядке. Например, чтобы найти последнее вхождение элемента в последовательности, мы можем применить **find()** для обратной последовательности:

```
auto ri = find(v.rbegin(), v.rend(), val);  // последнее вхождение
```

Заметьте, что **C::reverse_iterator** не тот же тип, что и **C::iterator**. Итак, если я захочу написать алгоритм **find_last()**, используя обратный итератор, я должен решить, какой тип итератора возвратить:

```
template<typename C, typename Val>
auto find_last(C& c, Val v) -> decltype(c.begin()) // используем итератор у C в интерфейсе
{
    auto ri = find(c.rbegin(), c.rend(), v);
    if (ri == c.rend()) return c.end();         // используем c.end() для индикации "not found"
    return prev(ri.base());
}
```

Для **reverse_iterator** метод **ri.base()** возвращает **iterator**, указывающий на следующую позицию за той, что указывается через **ri**. Итак, чтобы получить итератор, указывающий на тот же самый элемент, что и обратный итератор **ri**, я должен вернуть **ri.base()-1**. В то же время, мой контейнер может иметь тип **list**, который это не поддерживает — из-за своих итераторов, поэтому я возвращаю **prev()**.

Обратный итератор в своей основе есть просто итератор, так что я мог бы написать цикл явным образом:

```
template<typename C, Val v>
auto find_last(C& c, Val v) -> decltype(c.begin())
{
    for (auto p = c.rbegin(); p!=c.rend(); ++p)    // смотрим послед-сть в обратном порядке
        if (*p==v) return --p.base();
    return c.end();                                // используем c.end() для индикации "not found"
}
```

Эквивалентный код, осуществляющий поиск назад, использующий (прямой) итератор, таков:

```
template<typename C>
auto find_last(C& c, Val v) -> decltype(c.begin())
{
    for (auto p = c.end(); p!=c.begin();)      // обратный поиск (с конца)
        if (*--p==v) return p;
    return c.end();                            // используем c.end() для индикации "not found"
}
```

Как и для более ранних определений **find_last()**, эта версия требует по крайней мере двунаправленного итератора.

33.2.2. Итераторы вставки

Получение вывода в контейнер при помощи итератора предполагает, что элементы, последующие за указуемым итератором, могут быть переписаны. Это допускает возможность переполнения и порчи памяти. Например:

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(),200,7);    // присваиваем vi[0]..[199] значение 7
}
```

Если **vi** имеет менее **200** элементов, то будет проблема.

В файле **<iterator>** стандартная библиотека предоставляет решение в форме итератора вставки; он осуществляет внедрение нового элемента в последовательность, а не переписывание существующего элемента. Например:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi),200,7); // добавляем 200 семерок в конец vi
}
```

При записи в элемент посредством итератора вставки последний вставляет значение, а не переписывает этим значением указуемый элемент. Таким образом, контейнер растет на один элемент каждый раз, когда в него прописывается значение посредством иетратора вставки. Итераторы вставки просты, эффективны и полезны.

Есть три итератора вставки:

- **insert_iterator**: вставляет перед указуемым элементом, используя **insert()**.
- **front_insert_iterator**: вставляет перед первым элементом последовательности, используя **push_front()**.
- **back_insert_iterator**: вставляет после последнего элемента последовательности, используя **push_back()**.

Обычно итератор вставки конструируется при помощи вызова вспомогательной функции:

Конструкторы итераторов вставки (§ iso.24.5.2)	
ii=inserter(c, p)	**ii** есть **insert_iterator**, указывающий на **p** в контейнере **c**
ii=back_inserter(c)	**ii** есть **back_insert_iterator**, указывающий на **back()** в контейнере **c**
ii=front_inserter(c)	**ii** есть **front_insert_iterator**, указывающий на **front()** в контейнере **c**

Итератор, передаваемый **inserter()**, должен указывать на контейнер. Для последовательного контейнера это означает, что он должен быть двунаправленным итератором (чтобы можно было вставить перед ним). Например, вы не можете использовать **inserter()**, чтобы заставить итератор вставлять в **forward_list**. Для ассоциативного контейнера, где итератор используется лишь как намек на место, в которое нужно вставить элемент, прямой итератор (например, предоставляемый в **unordered_set**) приемлем.

Операции над insert_iterator<C> (§ iso.24.5.2) (с продолжением)	
insert_iterator p {c,q};	Итератор вставки для контейнера **c**, указывающий на ***q**; **q** должен указ-ть в **c**
insert_iterator p {q};	Копирующий конструктор: **p** есть копия **q**
p=q	Копирующее присваивание: **p** есть копия **q**
p=move(q)	Перемещающее присваивание: **p** указывает, куда указывал **q**

Итератор вставки является итератором вывода:

Операции над insert_iterator<C> (окончание) (§ iso.24.5.2)	
++p	Пусть **p** указывает на следующий эл-т; значение выраж-я есть новое значение **p**
p++	Пусть **p** указывает на следующий эл-т; значение выраж-я есть старое значение **p**
***p=x**	Вставить **x** перед **p**
***p++=x**	Вставить **x** перед **p**, затем инкрементировать **p**

Отличия во **front_insert_iterator** и **back_insert_iterator** состоят в том, что их конструкторы не требуют итератора. Пример:

```
vector<string> v;
back_inser t_iterator<v> p;
```

Нельзя читать посредством итератора вставки.

33.2.3. Перемещающий итератор

Итератор перемещения есть итератор, для которого чтение указуемого элемента состоит в перемещении, а не в копировании. Обычно мы получаем итератор перемещения из другого итератора, используя вспомогательную функцию:

Функция, конструирующая перемещающий итератор	
mp=make_move_iterator(p)	mp есть move_iterator, указывающий на тот же эл-т, что и p; p должен быть итератором ввода

Итератор перемещения поддерживает те же операции, что и итератор, из которого он был получен. Например, можно выполнять -p, если перемещающий итератор p был получен из двунаправленного итератора. Метод operator*() перемещающего итератора просто возвращает rvalue-ссылку (§7.7.2) на элемент, указуемый std::move(q). Пример:

```
vector<string> read_strings(istream&);
auto vs = read_strings(cin);                    // получаем несколько строк

vector<string> vs2;
copy(vs, back_inserter(vs2));                   // копируем строки из vs в vs2

vector<string> vs3;
copy(vs2, make_move_iterator(back_inserter(vs3)));   // перемещаем строки из vs2 в vs3
```

Предполагается, что определена контейнерная версия std::copy().

33.3. Функции доступа к диапазону

В файле <iterator> стандартная библиотека предоставляет глобальные функции (не члены) begin() и end() для контейнеров:

begin() и end() (§ iso.24.6.5)	
p=begin(c)	p – итератор для первого эл-та c; c есть встроенный массив или имеет c.begin()
p=end(c)	p – итератор для эл-та c, следующего за последним; c есть встроенный массив или имеет c.end()

Эти функции очень просты:

```
template<typename C>
    auto begin(C& c) -> decltype(c.begin());
template<typename C>
    auto begin(const C& c) -> decltype(c.begin());
template<typename C>
    auto end(C& c) -> decltype(c.end());
template<typename C>
    auto end(const C& c) -> decltype(c.end());

template<typename T, size_t N>    // для встроенных массивов
    auto begin(T (&array)[N]) -> T*;
template<typename T, size_t N>
    auto end(T (&array)[N]) -> T*;
```

Эти функции используются оператором range-for (§9.5.1), но они могут применяться пользователями и напрямую. Например:

```
template<typename Cont>
void print(Cont& c)
{
    for(auto p=begin(c); p!=end(c); ++p)
        cout << *p << '\n';
}
```

```
void f()
{
    vector<int> v {1,2,3,4,5};
    print(v);

    int a[] {1,2,3,4,5};
    print(a);
}
```

Если бы я применил **c.begin()** и **c.end()**, то вызов **print(a)** потерпел бы неудачу.

Пользовательский контейнер с членами **begin()** и **end()** автоматически получает глобальные версии, когда директивой **#include** включается файл **<iterator>**. Чтобы к контейнеру **My_container** добавить члены **begin()** и **end()**, я должен написать что-то в следующем роде:

```
template<typename T>
Iterator<My_container<T>> begin(My_container<T>& c)
{
    return Iterator<My_container<T>>{&c[0]};          // итератор на первый элемент
}

template<typename T>
Iterator<My_container<T>> end(My_container<T>& c)
{
    return Iterator<My_container<T>>{&c[0]+c.size()}; // итератор на последний элемент
}
```

Здесь я полагаю, что передача адреса первого элемента есть способ создания итератора на первый элемент контейнера **My_container**, и что контейнер **My_container** имеет член **size()**.

33.4. Функциональные объекты

Многие стандартные алгоритмы принимают функциональные объекты (или же функции) в качестве аргументов для управления способом их работы. Общими примерами являются критерии сравнения, предикаты (функции, возвращающие **bool**) и арифметические операции. В файле **<functional>** стандартная библиотека предоставляет несколько общих функциональных объектов:

Предикаты (§ iso.20.8.5, § iso.20.8.6, § iso.20.8.7)	
p=equal_to<T>(x, y)	p(x, y) означает x==y, когда x и y имеют тип T
p=not_equal_to<T>(x, y)	p(x, y) означает x!=y, когда x и y имеют тип T
p=greater<T>(x, y)	p(x, y) означает x>y, когда x и y имеют тип T
p=less<T>(x, y)	p(x, y) означает x<y, когда x и y имеют тип T
p=greater_equal<T>(x, y)	p(x, y) означает x>=y, когда x и y имеют тип T
p=less_equal<T>(x, y)	p(x, y) означает x<=y, когда x и y имеют тип T
p=logical_and<T>(x, y)	p(x, y) означает x&&y, когда x и y имеют тип T
p=logical_or<T>(x, y)	p(x, y) означает x||y, когда x и y имеют тип T
p=logical_not<T>(x)	p(x) означает !x, когда x имеет тип T
p=bit_and<T>(x, y)	p(x, y) означает x&y, когда x и y имеют тип T
p=bit_or<T>(x, y)	p(x, y) означает x|y, когда x и y имеют тип T
p=bit_xor<T>(x, y)	p(x, y) означает x^y, когда x и y имеют тип T

Например:

```
vector<int> v;
// ...
sort(v.begin(), v.end(), greater<int>{});    // сортируем v в порядке убывания
```

Такие предикаты примерно эквивалентны простым лямбдам. Например:

```
vector<int> v;
// ...
sort(v.begin(), v.end(),[](int a, int b) {return a>b;});    // сортируем v в порядке убывания
```

Заметим, что **logical_end** и **logical_or** всегда оценивают оба аргумента (в отличие от операций **&&** и ||).

Арифметические операции (§ iso.20.8.4)	
f=plus<T>(x, y)	**f(x, y)** означает **x+y**, когда **x** и **y** имеют тип **T**
f=minus<T>(x, y)	**f(x, y)** означает **x-y**, когда **x** и **y** имеют тип **T**
f=multiplies<T>(x, y)	**f(x, y)** означает **x*y**, когда **x** и **y** имеют тип **T**
f=divides<T>(x, y)	**f(x, y)** означает **x/y**, когда **x** и **y** имеют тип **T**
f=modulus<T>(x, y)	**f(x, y)** означает **x%y**, когда **x** и **y** имеют тип **T**
f=negate<T>(x)	**f(x)** означает **-x**, когда **x** имеет тип **T**

33.5. Адаптеры функций

Адаптер функции берет функцию в качестве аргумента и возвращает функциональный объект, который может использоваться для активизации исходной функции.

Адаптеры (§ iso.20.8.9, § iso.20.8.10, § iso.20.8.8)	
g=bind(f, args)	**g(args2)** эквивалентно **f(args3)**, где **args3** получен заменой «местодержателей», таких как **_1**, **_2** и **_3**, в **args** аргументами из **args2**
g=mem_fn(f)	**g(p,args)** означает **p->f(args)** в случае, если **p** – указатель, или **p.mf(args)**, если нет; **args** есть (возможно пустой) список аргументов
g=not1(f)	**g(x)** означает **!f(x)**
g=not2(f)	**g(x, y)** означает **!f(x, y)**

Адаптеры **bind()** и **mem_fn()** выполняют связывание аргумента. Эти связыватели и их устаревшие предшественники (вроде **bind1st()**, **mem_fun()** и **mem_fun_ref()**) ранее использовались интенсивно, но сейчас в большинстве случаев легче применить лямбды (§11.4).

33.5.1. bind()

Получив функцию и набор аргументов, **bind()** производит функциональный объект, который можно вызывать с оставшимися аргументами функции, если они есть. Пример:

```
double cube(double);

auto cube2 = bind(cube,2);
```

Вызов **cube2()** активирует **cube** с аргументом **2**, то есть вызов **cube(2)**. Нам не требуется связывать все аргументы функции. Пример:

```
using namespace placeholders;

void f(int, const string&);
auto g = bind(f,2,_1);      // связываем первый аргумент f() со значением 2
f(2,"hello");
g("hello");                 // также вызов f(2,"hello");
```

Странный аргумент _1 связывателя является «местодержателем», сообщающим **bind()**, куда должны поступать аргументы результирующей функции. В этом случае первый аргумент **g()** используется как второй аргумент **f()**.

«Местодержатели» можно найти в пространстве имен **std::placeholder**, что является частью **<functional>**. Механизм «местодержателей» очень гибок. Рассмотрим следующее:

```
f(2,"hello");
bind(f)(2,"hello");          // также вызов f(2,"hello");
bind(f,_1,_2)(2,"hello");    // также вызов f(2,"hello");
bind(f,_2,_1)("hello",2);    // обратный порядок аргументов: также вызов f(2,"hello");

auto g = [](const string& s, int i) {f(i, s);}   // обратный порядок аргументов
g("hello",2);                                    // также вызов f(2,"hello");
```

Для связывания аргументов перегруженной функции мы должны явным образом указать, какая версия функции подлежит связыванию:

```
int pow(int, int);
double pow(double, double);                                  // pow() перегружена

auto pow2 = bind(pow,_1,2);                                  // ошибка: которая pow()?
auto pow2 = bind((double(*)(double, double))pow,_1,2);       // OK (но грубо)
```

Заметим, что **bind()** принимает обычные выражения как аргументы. Это предполагает, что ссылки разыменовываются перед тем, как их увидит **bind()**. Например:

```
void incr(int& i)
{
    ++i;
}

void user()
{
    int i = 1;
    incr(i);                 // i становится 2
    auto inc = bind(incr,_1);
    inc(i);                  // i остается 2; inc(i) инкрементировала локальную копию i
}
```

Чтобы иметь с этим дело, стандартная библиотека предоставляет еще пару итераторов:

reference_wrapper<T> (§ iso.20.8.3)	
r=ref(t)	r есть **reference_wrapper** для **T&** t; noexcept
r=cref(t)	r есть **reference_wrapper** для **const T&** t; noexcept

Это решает «проблему ссылок» для **bind()**:

```
void user()
{
    int i = 1;
    incr(i);     // i становится 2
    auto inc = bind(incr,_1);
    inc(ref(i));      // i становится 3
}
```

Этому **ref()** требуется передать ссылки на аргументы для **thread**, потому что конструкторы **thread** являются шаблонами с переменным числом параметров (§42.2.2).

До сих пор я использовал результат вызова **bind()** либо немедленно, либо присваивал его переменной, объявленной с **auto**. Это избавило меня от необходимости специфицировать тип возврата **bind()**. Это полезно, поскольку тип возврата **bind()** может варьироваться с типом функции, подлежащей вызову, и со значениями хранимых аргументов. В частности, возвращаемый функциональный объект имеет больший размер, если он вынужден хранить значения связанных параметров. Иногда, тем не менее, нам нужно точно специфицировать типы аргументов и тип возвращаемого результата. В таком случае мы можем специфицировать их для **function** (§33.5.3).

33.5.2. mem_fn()

Функциональный адаптер **mem_fn(mf)** производит функциональный объект, который можно вызвать как глобальную функцию (не как функцию-член). Пример:

```
void user(Shape* p)
{
    p->draw();
    auto draw = mem_fn(&Shape::draw);
    draw(p);
}
```

В основном **mem_fn()** используется тогда, когда алгоритм требует операцию, которая вызывается как глобальная функция. Например:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(), mem_fn(&Shape::draw));
}
```

Можно сказать, что **mem_fn()** осуществляет соответствие между объектно-ориентированным стилем вызова функций и обычным функциональным стилем.

Часто лямбды обеспечивают простую и общую альтернативу связывателям. Например:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(),[](Shape* p) {p->draw();});
}
```

33.5.3. function

В том отношении, что **bind()** можно использовать напрямую или же с помощью **auto**-переменной, этот связыватель напоминает лямбды.

Если мы хотим присвоить результат **bind()** переменной со специфицированным типом, то можно использовать стандартный тип **function**:

function<R(Argtypes...)> (§ iso.20.8.11.2)	
function f {};	f есть пустой function; noexcept
function f {nullptr};	f есть пустой function; noexcept
function f {g};	f есть function, содержащий g; g может быть чем угодно, что может быть вызвано с типами аргументов f
function f {allocator_arg_t,a};	f есть пустой function; использует аллокатор a; noexcept
function f {allocator_arg_t,a,nullptr_t};	f есть пустой function; использует аллокатор a; noexcept
function f {allocator_arg_t,a,g};	f есть function, содержащий g; использует аллокатор a; noexcept

Продолжение

function<R(Argtypes...)> (§ iso.20.8.11.2)	
f2=f	**f2** есть копия **f**
f=nullptr	**f** становится пустым
f.swap(f2)	Обменять содержимое **f** и **f2**; **f** и **f2** должны быть одного и того же типа **function**; noexcept
f.assign(f2, a)	**f** получает копию **f2** и аллокатор **a**
bool b {f};	Преобразование **f** в **bool**; **b** равен **true**, если **f** не пустой; explicit; noexcept
r=f(args)	Вызвать содержащуюся ф-ию с **args**; типы аргументов должны соответствовать таковым от **f**
ti=f.target_type()	**ti** есть **type_info** для **f**; если **f** не содержит что-то вызываемое, то **ti==typid(void)**; noexcept
p=f.target<F>()	Если **f.target_type()==typeid(F)**, **p** указывает на содержащийся объект, иначе, **p==nullptr**; noexcept
f==nullptr	Пуст ли **f**? noexcept
nullptr==f	**f==nullptr**
f!=nullptr	**!(f==nullptr)**
nullptr!=f	**!(f==nullptr)**
swap(f, f2)	**f.swap(f2)**

Например:

```
int f(double);
function<int(double)> fct {f};        // инициализируем в f
int g(int);

void user()
{
    fct = [](double d) {return round(d);};    // приписываем лямбду к fct
    fct = f;                                  // приписываем функцию к fct
    fct = g;                                  // ошибка: некорректный тип возврата
}
```

Целевые функции предназначены для редких случаев, когда кто-либо хочет исследовать **function**, а не просто вызвать ее, как это обычно и происходит.

Стандартный тип **function** может содержать любой объект, который можно вызвать с помощью операции вызова **()** (§2.2.1, §3.4.3, §11.4, §19.2.2). Таким образом, объект типа **function** есть функциональный объект. Например:

```
int round(double x) {return static_cast<double>(floor(x+0.5));}    // стандартное 4/5 округление

function<int(double)> f; // f может содержать нечто, что вызывается с double и возвратом int

enum class Round_style {truncate, round};

struct Round {      // функциональный объект, хранящий состояние
    Round_style s;
    Round(Round_style ss):s(ss) {}
    int operator()(double x) const {return (s==Round_style::round)? (x+0.5) : x;};
};

void t1()
{
    f = round;
    cout << f(7.6) << '\n';     // вызов через f функции round
```

```
    f = Round(Round_style::truncate);
    cout << f(7.6) << '\n';     // вызов функционального объекта

    Round_style style = Round_style::round;
    f = [style] (double x){return (style==Round_style::round)? x+0.5 : x;};

    cout << f(7.6) << '\n';     // вызов лямбды

    vector<double> v {7.6};
    f = Round(Round_style::round);
    std::transform(v.begin(), v.end(), v.begin(), f);       // передаем алгоритму

    cout << v[0] << '\n';                                   // трансформированное лямбдой
}
```

Здесь на выходе получается **8**, **8**, **7** и **8**.

Очевидно, что тип **function** полезен для функций обратного вызова, для передачи операций в качестве аргументов и т.д.

33.6. Советы

[1] Входная последовательность определяется парой итераторов; §33.1.1.

[2] Выходная последовательность определяется единственным итератором; избегайте переполнения; §33.1.1.

[3] Для любого итератора **p**, **[p:p)** есть пустая последовательность; §33.1.1.

[4] Используйте конец последовательности для индикации результата «не найдено»; §33.1.1.

[5] Думайте об итераторах как о более общих и лучше себя ведущих указателях; §33.1.1.

[6] Используйте типы итераторов, такие как **list<char>::iterator**, а не указатели, для ссылки на элементы контейнера; §33.1.1.

[7] Используйте **iterator::traits** для получения информации об итераторах; §33.1.3.

[8] Можно выполнить диспетчеризацию на этапе компиляции с помощью **iterator_traits**; §33.1.3.

[9] Используйте **iterator_traits** для выбора оптимального алгоритма, базируясь на категориях итераторов; §33.1.3.

[10] **iterator_traits** — это деталь реализации; предпочитайте явный выбор; §33.1.3.

[11] Используйте **base()** для извлечения **iterator** из **reverse_iterator**; §33.2.1.

[12] Можно использовать итератор вставки для добавления элементов в контейнер; §33.2.2.

[13] Можно использовать **move_iterator** для превращения копирующих операций в перемещающие; §33.2.3.

[14] Удостоверьтесь, что ваши контейнеры допускают траверс посредством оператора range-**for**; §33.3.

[15] Используйте **bind()** для создания разных вариантов функций и функциональных объектов; §33.5.1.

[16] Заметьте, что **bind()** рано разыменовывает ссылки; используйте **ref()** для того, чтобы отложить разыменование; §33.5.1.

[17] Можно использовать лямбду или **mem_fn()** для конвертирования стиля вызова **p->f(a)** в **f(p, a)**; §33.5.2.

[18] Используйте **function**, когда вам нужна переменная, которая может содержать разные вызываемые объекты; §33.5.3.

34

Память и ресурсы

Идея может возникнуть у каждого;
важно, что вы будете с ней делать.
— Терри Пратчет

- Введение
- «Почти контейнеры»
  array; **bitset**; **vector<bool>**; **tuple**
- Указатели для управления ресурсами
  unique_ptr; **shared_ptr**; **weak_ptr**
- Аллокаторы
  Умолчательный аллокатор; Характеристики аллокаторов; Характеристики указателей; Вложенные аллокаторы
- Интерфейс сборщика мусора
- Неинициализированная память
  Временные буферы; **raw_storage_iterator**
- Советы

34.1. Введение

STL (Глава 31, Глава 32, Глава 33) является самой общей и четко структурированной частью стандартной библиотеки по управлению и манипуляции данными. В настоящей главе мы рассмотрим более специализированные средства и средства для работы с неинициализированной памятью.

34.2. «Почти контейнеры»

В стандартной библиотеке имеется некоторое число контейнеров, которые не отвечают всем требованиям STL (§31.4, §32.2, §33.1). Это, к примеру, встроенные массивы, **array** и **string**. Я иногда ссылаюсь на них как на «почти контейнеры» (§31.4), но это не совсем верно: они содержат элементы, но каждый из них имеет ограничения или нестандартные добавочные средства, что делает их чужеродными (неуклюжими) в контексте STL. Кроме того, отдельное их рассмотрение помогает систематическому изучению STL.

«Почти контейнеры»	
T[N]	Встроенный массив: смежным образом выделенная послед-ть фикс. размера N эл-тов типа T; неявно преобразуется к T*
array<T, N>	Смежным образом выделенная послед-ть фикс. размера N эл-тов типа T; как встроенный массив, но решает многие проблемы
bitset<N>	Последовательность бит фикс. размера N
vector<bool>	Последовательность бит компактно размещенная в специализации vector
pair<T, U>	Два элемента типов T и U
tuple<T...>	Послед-ть произвольного числа эл-ов произвольных типов
basic_string<C>	Послед-ть символов типа C; предоставляет строковые операции
valarray<T>	Массив числовых значений типа T; предоставляет числовые операции

Почему стандартная библиотека предоставляет так много контейнеров? Дело в том, что они служат общим, но несколько разным нуждам. Если бы стандартная библиотека не предоставляла нам эти средства, то программистам пришлось бы изобретать их каждый раз заново и реализовывать их по-своему. Например:

- **pair** и **tuple** гетерогенны; все остальные контейнеры гомогенны (однородны) — все элементы одного типа.
- Элементы у **array**, **vector** и **tuple** аллоцированы непрерывным (смежным) образом; **forward_list** и **map** являются связанными структурами.
- **bitset** и **vector<bool>** содержат биты и выполняют доступ к ним посредством прокси-объектов; все остальные контейнеры стандартной библиотеки могут работать с разными типами и выполнять доступ к элементам напрямую (самостоятельно).
- **basic_string** требует от элементов быть каким-нибудь видом символов, чтобы выполнять манипуляции со строками, такими как конкатенация и операции, зависящие от локальных сособенностей (Глава 39), а **valarray** требует, чтобы элементы были числами и допускали численные операции.

Все эти контейнеры могут трактоваться как предоставляющие специализированные сервисы, нужные программистам во многих предметных областях. Никакой отдельный контейнер не может удовлетворить всем таким нуждам, ибо многие из них противоречат друг другу. Например, «способность расти» не ладит с требованием «быть аллоцированным по фиксированным адресам», или «элементы не передвигаются при добавлении элементов» противоречит требованию смежному размещению в памяти. Кроме того, излишне общий контейнер будет демонстрировать слишком высокие накладные расходы во вногих случаях своего применения.

34.2.1. array

Тип **array**, определенный в файле **<array>**, является последовательностью элементов фиксированного размера заданного типа, где число элементов указывается на этапе компиляции. Таким образом, **array** вместе с элементами может быть размещен (аллоцирован) в стеке, внутри объекта или же в статическом хранилище. Элементы сосредоточены в области видимости, в которой определен **array**. Этот тип можно понимать как встроенный массив с жестко хранимым размером, без потенциально опасных приведений к типам указателей, плюс несколько удобных вспомогательных методов. При этом нет никаких дополнительных накладных расходов (ни по памяти, ни по времени) по сравнению со встроенными массивами. Тип **array** не следует модели дескриптора, свойственной STL контейнерам. Вместо этого **array** напрямую содержит свои элементы:

```
template<typename T, size_t N>   // массив из N элементов типа T (§ iso.23.3.2)
struct array {
/*
     Типы и операции как у vector (§ 31.4), кроме операций изменения
     размера контейнера, конструкторов, и assign() функций
*/
     void fill(const T& v);          // устанавливаем N копий v
     void swap(array&) noexcept(noexcept(swap(declval<T&>(), declval<T&>())));

     T __elem[N];   // деталь реализации
};
```

Никакой управляющей информации (например, размер) не хранится в **array**. Это означает, что перемещение (§17.5) для этого типа не эффективнее копирования (если только элементы не являются дескрипторами ресурсов с эффективным перемещением). У типа **array** нет конструктора или аллокатора.

Число элементов и значения индексов для **array** имеют типа **unsigned** (**size_t**), как и у типа **vector**, но не как у встроенных массивов. Это означает, что **array<int,-1>** может быть принято невнимательным компилятором. Остается надеяться на предупреждение.

Тип **array** позволяет выполнить инициализацию списком инициализаторов:

```
array<int,3> a1 = {1, 2, 3};
```

Число значений в списке инициализаторов должно быть равно или меньше, чем число элементов, специфицированное для **array**. Как всегда, если значения в списке инициализации представлены не для всех элементов, то оставшиеся инициализируются соответствующими умолчательными значениями. Например:

```
void f()
{
     array<string, 4> aa = {"Churchill", "Clare"};
     //
}
```

Последние два элемента будут пустыми строками.

Число элементов не является опциональным:

```
array<int> ax = {1, 2, 3}; // ошибка, размер не установлен
```

Чтобы не иметь дело со специальным случаем, число элементов может быть нулем:

```
int<int,0> a0;
```

Число элементов должно быть константным выражением:

```
void f(int n)
{
     array<string, n> aa = {"John's", "Queens' "};  // ошибка: size не константное выражение
     //
}
```

Если нужно, чтобы число элементов задавалось переменной, то следует воспользоваться типом **vector**. С другой стороны, раз число элементов **array** известно во время компиляции, то его **size()** является **constexpr**-функцией.

Для **array** нет конструктора, который копирует значение аргумента (как у **vector**; §31.3.2). Вместо этого имеется операция **fill()**:

```
void f()
{
    array<int,8> aa;      // до сих пор не инициализировано
    aa.fill(99);          // присваиваем 99 восьми копиям
    // ...
}
```

Поскольку тип **array** не следует модели дескриптора элементов, то **swap()** вынужден на самом деле обменивать значения элементов, так что эта операция для двух **array<T, N>** применяет **swap()** к **N** парам элементов типа **T**.

В случае необходимости можно явно передать **array** функции в С-стиле, которая ожидает указатель. Например:

```
void f(int* p, int sz);          // интерфейс в С-стиле

void g()
{
    array<int,10> a;

    f(a, a.size());              // ошибка: нет конверсии
    f(&a[0], a.size());          // используем в С-стиле
    f(a.data(), a.size());       // используем в С-стиле

    auto p = find(a.begin(), a.end(),777);    // C++/STL стиль использования
    // ...
}
```

Возникает вопрос: зачем использовать **array**, когда есть **vector**, который более гибок? Потому что он проще. Иногда это дает заметный выигрыш в производительности при прямом доступе к элементам, размещенным в стеке, поскольку в противном случае приходится иметь дело с элементами, размещенными в свободной памяти с косвенным доступом к ним через **vector** (дескриптор), а затем еще и освобождать память. В обратную сторону: стек есть лимитированный ресурс (особенно на некоторых встроенных системах), а переполнение стека есть сущая заноза.

Другой вопрос: зачем использовать **array**, когда есть встроенные массивы? Тип **array** знает свой размер, так что его просто использовать с алгоритмами стандартной библиотеки, и его можно копировать (используя = или инициализацию). Но на мой вкус он особенно хорош тем, что избавляет меня от противных сюрпризов из-за приведений к типам указателей. Рассмотрим следующее:

```
void h()
{
    Circle a1[10];
    array<Circle,10> a2;
    // ...
    Shape* p1 = a1;    // OK: но катастрофа уже ждет
    Shape* p2 = a2;    // ошибка: нет конверсии array<Circle,10> в Shape*
    p1[3].draw();      // катастрофа
}
```

Комментарий о катастрофе предполагает, что **sizeof(Shape)<sizeof(Circle)**, так что при индексировании **Circle[]** по **Shape*** будет получено ошибочное смещение (§27.2.1, §17.5.1.4). У всех стандартных контейнеров имеется такое преимущество перед встроенными массивами.

Тип **array** можно трактовать как **tuple** (§34.2.4), где все элементы одинакового типа. В стандартной библиотеке такой взгляд на **array** имеет поддержку. Вспомогательные функции **tuple** вроде **tuple_size** и **tuple_element** могут применяться к **array**:

```
tuple_size<array<T, N>>::value          // N
tuple_element<S, array<T, N>>::type     // T
```

Мы также можем использовать функцию **get<i>** для доступа к **i**-ому элементу:

```
template<size_t index, typename T, size_t N>
    T& get(array<T, N>& a) noexcept;
template<size_t index, typename T, size_t N>
    T&& get(array<T, N>&& a) noexcept;
template<size_t index, typename T, size_t N>
    const T& get(const array<T, N>& a) noexcept;
```

К примеру:

```
array<int,7> a = {1,2,3,5,8,13,25};
auto x1 = get<5>(a);                                    // 13
auto x2 = a[5];                                         // 13
auto sz = tuple_size<decltype(a)>::value;               // 7
typename tuple_element<5, decltype(a)>::type x3 = 13;   // x3 есть int
```

Эти функции типа предназначены для людей, которые пишут код, ожидающий **tuple**.

Для улучшения читаемости кода применяйте **constexpr**-функции (§28.2.2) и синонимы типа (§28.2.1):

```
auto sz = Tuple_size<decltype(a)>();        // 7

Tuple_element<5, decltype(a)> x3 = 13;      // x3 есть int
```

Синтаксис **tuple** предназначен для использования в обобщенном коде.

34.2.2. bitset

Состояние системы, например состояние потока ввода (§38.4.5.1), часто представляется набором флагов, отражающих бинарное состояние вроде хорошо/плохо, истина/ложь или вкл/выкл. Язык C++ поддерживает понятие небольших наборов флагов эффективным способом через побитовые операции над целыми (§11.1.1). Класс **bitset<N>** обобщает это понятие и предлагает удобные операции над последовательностью из **N** бит **[0:N)**, где **N** известно во время компиляции. Для наборов бит, которые не точно укладываются в **long long int**, использование **bitset** намного оптимальнее, чем использование целых напрямую. Для малых наборов бит **bitset** обычно оптимизируется. Если вы хотите именовать биты, а не нумеровать их, то альтернативой будет использование **set** (§31.4.3), перечислений (§8.4) или битовых полей (§8.2.7).

Тип **bitset<N>** является массивом из **N** бит. Он представлен в файле **<bitset>**. Он отличается от **vector<bool>** (§34.2.3) тем, что имеет фиксированный размер, от **set** (§31.4.3) тем, что его биты индексируются целыми, а не ассоциативно значениями, а от обоих **vector<bool>** и **set** тем, что предоставляет операции для манипуляции битами.

Невозможно адресовать отдельный бит напрямую с помощью встроенного указателя (§7.2). Таким образом, **bitset** является типом ссылки на биты (прокси). Это в целом полезная техника для адресации объектов, для которых встроенный указатель не подходит по каким-либо причинам:

```
template<size_t N>
class bitset {
public:
    class reference {    // ссылка на одиночный бит:
        friend class bitset;
        reference() noexcept;
    public:          // поддерживается нуль-терминированное индексирование в [0:b.size())
        ~reference() noexcept;
        reference& operator=(bool x) noexcept;                  // для b[i] = x;
        reference& operator=(const reference&) noexcept;        // для b[i] = b[j];
        bool operator~() const noexcept;                        // возврат ~b[i]
        operator bool() const noexcept;                         // для x = b[i];
        reference& flip() noexcept;                             // b[i].flip();
    };
    // ...
};
```

По историческим причинам **bitset** отличается по стилю от других классов стандартной библиотеки. Например, если индекс (битовая позиция) выходит из диапазона, то генерируется исключение **out_of_range**. Нет никаких итераторов. Битовые позиции отсчитываются справа налево, так что значение для **b[i]** есть **pow(2, i)**. Таким образом, битовый набор можно представлять себе как **N**-битовое бинарное число:

position:	9	8	7	6	5	4	3	2	1	0
bitset<10>(989):	1	1	1	1	0	1	1	1	0	1

34.2.2.1. Конструкторы

Можно конструировать **bitset** указанным числом нулей, из бит **unsigned long long int** или из **string**:

Конструкторы bitset<N> (§ iso.20.5.1)	
bitset bs {};	**N** нулевых бит
bitset bs {n};	Биты из **n**; **n** есть **unsigned long long**
bitset bs {s,i,n,z,o};	**n** бит [**i**:**i+n**) из **s**; **s** есть **basic_string<C, Tr, A>**; **z** – символ типа **C** использ-ся для нуля; **o** – символ типа **C** использ-ся для нуля; explicit
bitset bs {s,i,n,z};	**bitset bs {s, i, n, z, C{'1'}};**
bitset bs {s,i,n};	**bitset bs {s, i, n, C{'0'}, C{'1'}};}**
bitset bs {s,i};	**bitset bs {s, i, npos, C{'0'}, C{'1'}};**
bitset bs {s};	**bitset bs {s,0, npos, C{'0'}, C{'1'}};**
bitset bs {p,n,z,o};	**n** бит [**p**: **p+n**); **p** есть строка в С-стиле типа **C***; **z** – символ типа **C** использ-ся для нуля; **o** – символ типа **C** использ-ся для нуля; explicit
bitset bs {p,n,z};	**bitset bs {p, n, z, C{'0'}};**
bitset bs {p,n};	**bitset bs {p, n, C{'1'}, C{'0'}};**
bitset bs {p};	**bitset bs {p, npos, C{'1'}, C{'0'}};**

Позиция **npos** есть позиция за концом **string<C>**, означающая «все символы до конца» (§36.3).

Когда предоставляется аргумент типа **unsigned long long int**, то каждый бит целого используется для инициализации соответствующего бита в **bitset**. Аргумент типа **basic_string**

(§36.3) делает тот же самое, кроме того, что символ '0' дает битовое значение 0, символ '1' дает битовое значение 1, а остальные символы приводят к исключению invalid_argument. Например:

```
void f()
{
    bitset<10> b1;                          // все 0

    bitset<16> b2 = 0xaaaa;                 // 1010101010101010
    bitset<32> b3 = 0xaaaa;                 // 00000000000000001010101010101010

    bitset<10> b4 {"1010101010"};                   // 1010101010
    bitset<10> b5 {"10110111011110",4};             // 0111011110

    bitset<10> b6 {string{"1010101010"}};           // 1010101010
    bitset<10> b7 {string{"10110111011110"},4};     // 0111011110
    bitset<10> b8 {string{"10110111011110"},2,8};   // 0011011101

    bitset<10> b9 {string{"n0g00d"}};  // исключение invalid_argument сгенерировано
    bitset<10> b10 = string{"101001"}; // ошибка: нет умолчательной конверсии string в bitset
}
```

Ключевой идеей для bitset было стремление к оптимизированной реализации, размещаемой в единственном машинном слове. Интерфейс этого типа отражает это стремление.

34.2.2.2. Операции над bitset

Для bitset предусмотрены операции по доступу к индивидуальным битам и для манипуляции со всеми битами в наборе:

Операции над bitset<N> (§ iso.20.5)	
bs[i]	i-ый бит bs
bs.test(i)	i-ый бит bs; генерирует out_of_range, если i не в диапазоне [0:bs.size())
bs&=bs2	Побитовое И
bs|=bs2	Побитовое ИЛИ
bs^=bs2	Побитовое исключающее ИЛИ
bs<<=n	Логический сдвиг влево (заполнение нулями)
bs>>=n	Логический сдвиг вправо (заполнение нулями)
bs.set()	Установить каждый бит bs в 1
bs.set(i, v)	bs[i]=v
bs.reset()	Установить каждый бит bs в 0
bs.reset(i)	b[i]=0;
bs.flip()	bs[i]=~bs[i] для каждого бита в bs
bs.flip(i)	bs[i]=~bs[i]
bs2=~bs	Сделать копию с дополнением: bs2=bs, bs2.flip()
bs2=bs<<n	Сделать копию со сдвигом влево: bs2=bs, bs2<<=n
bs2=bs>>n	Сделать копию со сдвигом вправо: bs2=bs, bs2>>=n
bs3=bs&bs2	Побитовое И: bs3[i]=bs[i]&bs2[i] для каждого бита в bs
bs3=bs|bs2	Побитовое ИЛИ: bs3[i]=bs[i]|bs2[i] для каждого бита в bs
bs3=bs^bs2	Побитовое исключающее ИЛИ: bs3[i]=bs[i]^bs2[i] для каждого бита в bs
is>>bs	Прочитать из is в bs; is есть istream
os<<bs	Вывести bs в os; os есть ostream

Операции >> и << являются операциями ввода/вывода, когда их первый операнд есть iostream; в противном случае, они являются операциями сдвига, а их второй операнд при этом должен быть целым. Например:

```
bitset<9> bs ("110001111"};
cout << bs << '\n';          // пишем "110001111" в cout
auto bs2 = bs<<3;            // bs2 == "001111000";
cout << bs2 << '\n';         // пишем "001111000" в cout
cin >> bs;                   // читаем из cin
bs2 = bs>>3;                 // bs2 == "000110001" если ввод был "110001111"
cout << bs2 << '\n';         // пишем "000110001" в cout
```

При сдвиге битов реализуется логический (а не циклический) сдвиг. Это означает, что некоторые биты «вываливаются в конец», а некоторые позиции получают умолчательное значение **0**. Заметим, что поскольку **size_t** является беззнаковым типом, невозможно выполнить сдвиг по отрицательному числу. Тем не менее, **b<<-1** выполняет сдвиг на очень большое положительное число, оставляя все биты **b** (типа **bitset**) со значением **0**. Компилятор должен при этом выдать предупреждение.

Тип **bitset** поддерживает обычные операции вроде **size()**, == и т.д.

Операции над bitset<N> (§ iso.20.5) **C**, **Tr**, и **A** имеют умолч. значаения для **basic_string<C, Tr, A>**	
n=bs.to_ulong()	**n** есть **unsigned long**, соответствующий **bs**
n=bs.to_ullong()	**n** есть **unsigned long long**, соответствующий **bs**
s=bs.to_string<C, Tr, A>(c0, c1)	**s[i]=(b[i])?c1: c0**; **s** есть **basic_string<C, Tr, A>**
s=bs.to_string<C, Tr, A>(c0)	**s=bs.template to_string<C, Tr, A>(c0, C{'1'})**
s=bs.to_string<C, Tr, A>()	**s=bs.template to_string<C, Tr, A>(C{'0'}, C{'1'})**
n=bs.count()	**n** – число бит со знач-ем **1** в **bs**
n=bs.size()	**n** – число бит в **bs**
bs==bs2	Имеют ли **bs** и **b2** одно и то же значение?
bs!=bs2	**!(bs==bs2)**
bs.all()	Имеют ли все биты в **bs** значение **1**?
bs.any()	Имеет ли какой-либо бит в **bs** значение **1**?
bs.none()	**!bs.any()**
hash<bitset<N>>	Специализация **hash** для **bitset<N>**

Операции **to_ullong()** и **to_string()** обеспечивают обратные операции по отношению к конструкторам. Чтобы избежать неочевидных преобразований, предпочтительны именованные операции, а не операции приведения типов. Если в **bitset** так много значимых битов, что они не могут уместиться в **unsigned long**, то **to_ulong()** генерирует исключение **overflow_error**; так же поступает **to_ullong()** в подобном случае.

К счастью, шаблонные аргументы для **basic_string**, возвращаемые из **to_string**, получают умолчательные значения. Например, мы можем вывести бинарное представление для **int**:

```
void binary(int i)
{
  bitset<8*sizeof(int)> b = i;          // полагаем 8-bit byte (см. также § 40.2)

  cout << b.to_string<char, char_traits<char>, allocator<char>>() << '\n'; // общее реш-е
  cout << b.to_string<char>() << '\n';  // используем умолчательные traits и allocator
  cout << b.to_string<>() << '\n';      // используем одни умолчания
  cout << b.to_string() << '\n';        // используем одни умолчания
}
```

Этот код печатает биты слева направо, с наиболее значимым битом слева, так что для аргумента **123** будет следующий вывод:

```
00000000000000000000000001111011
00000000000000000000000001111011
00000000000000000000000001111011
00000000000000000000000001111011
```

В показанном примере проще напрямую использовать операцию вывода для **bitset**:

```
void binary2(int i)
{
    bitset<8*sizeof(int)> b = i;    // полагаем 8-bit байт (см. также § 40.2)
    cout << b << '\n';
}
```

34.2.3. vector<bool>

Тип **vector<bool>** из файла **<vector>** является специализацией **vector** (§31.4), предоставляющей компактное хранилище битов:

```
template<typename A>
class vector<bool, A> {        // специализация vector<T, A> (§ 31.4)
public:
    using const_reference = bool;
    using value_type = bool;
    // как vector<T, A>

    class reference {          // поддержка нуль-терминированной индексации в [0:v.size())
        friend class vector;
        reference() noexcept;
    public:
        ~reference();
        operator bool() const noexcept;
        reference& operator=(const bool x) noexcept;           // v[i] = x
        reference& operator=(const reference& x) noexcept;     // v[i] = v[j]
        void flip() noexcept;                                  // "переворачиваем" бит: v[i]=~v[i]
    };

    void flip() noexcept;                                      // переворачиваем все биты v

    // ...
};
```

Сходство с **bitset** очевидно, но в отличие от **bitset** (но так же, как в **vector<T>**) **vector<bool>** имеет аллокатор и может изменять размер.

Как и в **vector<T>**, элементы у **vector<bool>** с большими индексами имеют и большие адреса памяти:

position:	9	8	7	6	5	4	3	2	1	0
vector<bool>:	1	1	1	1	0	1	1	1	0	1

Это противоположная раскладка по сравнению с **bitset**. Кроме того, отсутствует прямое приведение целых и строк в и из **vector<bool>**.

Используйте **vector<bool>** так же, как и другие **vector<T>**, но ожидайте, что операции на единичном бите будут менее эффективными по сравнению с эквивалентной операцией на **vector<char>**. И еще, в C++ нет честной возможности эмитировать поведение встроен-

ной ссылки через прокси, так что не пытайтесь выявлять тонкие различия между rvalue/lvalue, используя **vector<bool>**.

34.2.4. Группировка значений

Стандартная библиотека предоставляет два способа группировки значений произвольных типов в единый объект:

- **pair** (§34.2.4.1) содержит два значения.
- **tuple** (§34.2.4) содержит нуль или более значений.

Тип pair используется тогда, когда мы знаем, что имеем ровно два значения. С типом **tuple** мы можем иметь дело с любым возможным числом значений.

34.2.4.1. pair

В файле **<utility>** стандартная библиотека представляет класс **pair** для манипуляции парой значений:

```
template<typename T, typename U>
struct pair {
    using first_type = T;          // тип первого элемента
    using second_type = U;         // тип второго элемента

    T first;                       // первый элемент
    U second;                      // второй элемент

    // ...
};
```

pair<T, U> (§ iso.20.3.2)	
pair p {}	Конструктор по умолчанию: **pair p {T{}, U{}};** constexpr
pair p {x, y}	**p.first** инициализируется знач-ем **x**, а **p.second** инициализируется знач-ем **y**
pair p {p2}	Создать из **pair p2: pair p {p2.first, p2.second};**
pair p {piecewise_construct,t,t2}	**p.first** создается из эл-ов **tuple t**, а **p.second** создается из эл-ов **tuple t2**
p.~pair()	Деструктор: уничтожить **t.first** и **t.second**
p2=p	Копирующее присваивание: **p2.first=p.first** и **p2.second=p.second**
p2=move{p}	Перемещающее присваивание: **p2.first=move(p.first)** и **p2.second=move(p.second)**
p.swap(p2)	Обменять значения **p** и **p2**

Операции над **pair** не генерируют исключений, если соответствующие операции над элементами этого не делают. Аналогично, копирующие или перемещающие операции существуют для **pair**, если таковые есть для элементов.

Элементы **first** и **second** — это члены, которые мы можем читать и писать напрямую. Например:

```
void f()
{
    pair<string, int> p {"Cambridge",1209};
    cout << p.first;               // печатаем "Cambridge"
    p.second += 800;               // обновим год
    // ...
}
```

Здесь **piecewise_construct** это имя объекта типа **piecewize_construct_t**, использованное для того, чтобы различать ситуации, когда конструируется **pair** с членами типа **tuple**, и когда конструируется **pair**, используя **tuple** в списке аргументов для своих членов **first** и **second**. Например:

```
struct Univ {
    Univ(const string& n, int r) : name{n}, rank{r} {}
    string name;
    int rank;
    string city = "unknown";
};

using Tup = tuple<string, int>;
Tup t1 {"Columbia",11};                          // U.S. News 2012
Tup t2 {"Cambridge",2};

pair<Tub, Tub> p1 {t1, t2};                      // пара объектов типа tuple
pair<Univ, Univ> p2 {piecewise_construct, t1, t2};   // пара объектов типа Univ
```

То есть, **p1.second** есть **t2**, то есть, **{"Cambridge",2}**. При этом **p2.second** есть **Univ{t2}**, то есть, **{"Cambridge",2,"unknown"}**.

Вспомогательные операции над pair<T, U> (§ iso.20.3.3, § iso.20.3.4)	
p==p2	p.first==p2.first && p.second==p2.second
p<p2	p.first<p2.first || (!(p2.first<p.first) && p.second<p2.second)
p!=p2	!(p==p2)
p>p2	p2<p
p<=p2	!(p2<p)
p>=p2	!(p<p2)
swap(p, p2)	p.swap(p2)
p=make_pair(x,y)	p есть pair<decltype(x), decltype(y)>, содержащий знач-е x, y; если возможно, перемещает, а не копирует x и y
tuple_size<T>::value	Размер pair типа T
tuple_element<N,T>::type	Тип first (если N==0) или second (если N==1)
get<N>(p)	Ссылка на N-ый эл-т pair p; N должен быть 0 или 1

Функция **make_pair** избегает явного упоминания типов элементов у **pair**. К примеру:

```
auto p = make_pair("Harvard",1736);
```

34.2.4.2. tuple

В файле **<tuple>** стандартная библиотека определяет класс **tuple** и множество поддерживающих функций. Класс **tuple** представляет последовательность **N** элементов произвольных типов:

```
template<typename... Types>
class tuple {
public:
    // ...
};
```

Число элементов равно нулю или положительному значению.

За деталями дизайна **tuple** и его реализации обратитесь к §28.5 и §28.6.4.

Члены tuple<Types...> (§ iso.20.4.2)	
tuple t {};	Умолч. конструктор: пустой tuple; constexpr
tuple t {args};	t имеет один эл-т для каждого эл-а args; explicit
tuple t {t2};	Создать из tuple t2
tuple t {p};	Создать из pair p
tuple t {allocator_arg_t, a, args};	Создать из args, используя аллокатор a
tuple t {allocator_arg_t, a, t2};	Создать из tuple t2, используя аллокатор a
tuple t {allocator_arg_t, a, p};	Создать из pair p, используя аллокатор a
t.~tuple()	Деструктор: уничтожить каждый эл-т
t=t2	Копирующее присваивание tuple
t=move(t2)	Перемещающее присваивание tuple
t=p	Копирующее присваивание pair p
t=move(p)	Перемещающее присваивание pair p
t.swap(t2)	Обменять значения tuple t и t2; noexcept

Операции над **tuple** корректны, если (и только если) прилагаемые операции корректны для элементов. К примеру, мы можем присвоить один **tuple** другому, если каждый элемент одного **tuple** может быть присвоен соответствующему элементу другого **tuple**. Например:

```
tuple<string, vector<double>, int> t2 = make_tuple("Hello, tuples!", vector<int>{1,2,3},'x');
```

Если для операции все элементы не генерируют исключений, то тогда то же самое считается и для операции над **tuple**. Аналогично, операция над **tuple** является **constexpr**, если таково свойство операции над всеми элементами.

Число элементов в **tuple** у пары операндов (или аргументов) должно быть одинаковым.

Отметим, что общий конструктор для **tuple** имеет атрибут **explicit**. Например, вот это не будет работать:

```
tuple<int, int, int> rotate(tuple<int, int, int> t)
{
    return {t.get<2>(), t.get<0>(), t.get<1>()};      // ошибка: явный конструктор tuple
}

auto t2 = rotate({3,7,9});                            // ошибка: явный конструктор tuple
```

Если все, что вам нужно, это пара элементов, используйте **pair**:

```
pair<int, int> rotate(pair<int, int> p)
{
    return {p.second, p.first};
}

auto p2 = rotate({3,7});
```

Дополнительные примеры представлены в §28.6.4.

Вспомогательные функции tuple<Types...> (§ iso.20.4.2.4, § iso.20.4.2.9)	
t=make_tuple(args)	Создать tuple из args
t=forward_as_tuple(args)	t есть tuple rvalue-ссылок на эл-ты из args таким образом, можно перенаправить эл-ы arg через t
t=tie(args)	t есть tuple lvalue-ссылок на эл-ты из args, таким образом, можно присваивать эл-ы args через t
t=tuple_cat(args)	Конкатенировать tuple: args есть один или более tuple; t содержит члены tuple из args в порядке арг-тов

Продолжение

tuple<Types...> Helpers (§ iso.20.4.2.4, § iso.20.4.2.9)	
tuple_size<T>::value	Число эл-ов в tuple T
tuple_elements<N, T>::type	Тип N-го эл-а tuple T
get<N>(t)	Ссылка на N-ый эл-т tuple t
t==t2	Равны ли все эл-ы t и t2? t и t2 должны иметь равное число эл-ов
t!=t2	!(t==t2)
t<t2	t лексикографически меньше, чем t2?
t>t2	t2<t
t<=t2	!(t2>t)
t>=t2	!(t2<t)
uses_allocator<T, A>::value	Может ли tuple<T> быть выделено аллокатором типа A?
swap(t, t2)	t.swap(t2)

К примеру, tie() можно использовать для извлечения элементов из tuple:

```
auto t = make_tuple(2.71828,299792458,"Hannibal");
double c;
string name;
tie(c, ignore, name) = t;       // c=299792458; name="Hannibal"
```

Имя ignore ссылается на объект типа, который игнорирует присваивания. Таким образом, ignore в tie() предполагает, что попытки присвоения его tuple позиции игнорируются. Вот альтернатива:

```
double c = get<0>(t);           // c=299792458
string name = get<2>(t);        // name="Hannibal"
```

Очевидно, что это было бы более интересным, если бы tuple пришел неизвестно откуда и мы бы не знали значений элементов. Например:

```
tuple<int, double, string> compute();
// ...
double c;
string name;
tie(c, ignore, name) = t;       // результаты в c and name
```

34.3. Указатели для управления ресурсами

Указатель указывает на объект (или не указывает). Однако, указатель не сообщает, кто владеет объектами. Таким образом, глядя на указатель, мы не имеем представления о том, кто должен удалять указуемый объект, как это делать и есть ли кто-либо для этой цели. В файле <memory> мы находим умные указатели, отражающие владение:

- unique_ptr (§34.3.1) для выражения эксклюзивного владения.
- shared_ptr (§34.3.2) для отражения совместного (разделенного) владения.
- weak_ptr (§34.3.3) для разрыва колец в зацикленных структурах совместного владения.

Эти дескрипторы ресурсов ранее были представлены в §5.2.1.

34.3.1. unique_ptr

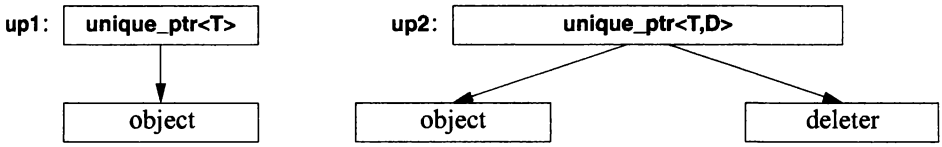
Тип unique_ptr (определяется в <memory>) предоставляет семантику строгого владения:

- **unique_ptr** владеет объектом, на который указывает. То есть, это забота **unique_ptr** уничтожать объект, на который он указывает посредством своего внутреннего указателя.
- **unique_ptr** не может быть скопирован (не имеет копирующего конструктора и копирующего присваивания). Его, однако, можно перемещать.
- **unique_ptr** хранит внутренний указатель и уничтожает указуемый объект, используя ассоциированный объект-уничтожатель, если он сам уничтожается (например, когда поток управления покидает область видимости **unique_ptr**; §17.2.2).

Вот примеры использования **unique_ptr**:

- Предоставление безопасности исключений для динамически выделяемых (аллоцируемых) объектов (§5.2.1, §13.3).
- Передача владения динамически выделенной памяти некоторой функции.
- Возврат динамически выделенной памяти из некоторой функции.
- Хранение указателей в контейнерах.

Представляйте себе **unique_ptr** будучи реализованным простым указателем («внутренний указатель») или (если есть встроенный «уничтожатель») парой указателей:



Когда **unique_ptr** уничтожается, то вызывается его «уничтожатель» для уничтожения владеемого объекта. «Уничтожатель» владеет информацией о способах уничтожения объекта. Например, он знает, что:

- При уничтожении локального объекта не надо ничего делать.
- При уничтожении пула памяти нужно возвратить объект памяти, уничтожать который или нет, зависит от того, как пул памяти был определен.
- По умолчанию (отсутствует ассоциированный «уничтожатель») **unique_ptr** использует операцию **delete**. Может использоваться специализация или оптимизация «пустого базового класса» (§28.5).

Таким образом, **unique_ptr** поддерживает общую схему управления ресурсами (§5.2).

```
template<typename T, typename D = default_delete<T>>
class unique_ptr {
public:
    using pointer = ptr;       // тип содержащегося указателя;
                               // ptr есть D::pointer, если это определено, иначе T*
    using element_type = T;
    using deleter_type = D;

    // ...
};
```

Внутренний указатель не доступен пользователям напрямую.

unique_ptr<T, D> (§ iso.20.7.1.2) **cp** – содержащийся указатель	
unique_ptr up {}	Конструктор по умолчанию: **cp=nullptr**; constexpr; noexcept
unique_ptr up {p}	**cp=p**; использ. «уничтожатель» по умолчанию; explicit; noexcept
unique_ptr up {p, del}	**cp=p**; **del** есть «уничтожатель»; noexcept

Продолжение

unique_ptr<T, D> (§ iso.20.7.1.2) cp – содержащийся указатель	
unique_ptr up {up2}	Перемещающий конструктор: cp.p=up2.p; up2.p=nullptr; noexcept
up.~unique_ptr()	Деструктор: если cp!=nullptr вызвать «уничтожатель» cp
up=up2	Перемещающее присваивание: up.reset(up2.cp); up2.cp=nullptr; up получает «уничтожатель» up2; старый объект up (если есть) уничтожается; noexcept
up=nullptr	up.reset(nullptr); т.е., уничтожить старый объект up, если такой есть
bool b {up};	Преобразование в bool: up.cp!=nullptr; explicit
x=*up	x=up.cp; только не для содержащихся массивов
x=up->m	x=up.cp->m; только не для содержащихся массивов
x=up[n]	x=up.cp[n]; только для содержащихся массивов
x=up.get()	x=up.cp
del=up.get_deleter()	del есть «уничтожатель» up
p=up.release()	p=up.cp; up.cp=nullptr
up.reset(p)	Если up.cp!=nullptr, вызвать «уничтожатель» для up.cp; up.cp=p
up.reset()	up.cp=pionter{} (вероятно nullptr); вызвать «уничтожатель» для старого знач-я up.cp
up.swap(up2)	Обменять значения up и up2; noexcept
up==up2	up.cp==up2.cp
up<up2	up.cp<up2.cp
up!=up2	!(up==up2)
up>up2	up2<up
up<=up2	!(up2>up)
up>=up2	!(up2<up)
swap(up, up2)	up.swap(up2)

Заметьте: **unique_ptr** не предоставляет копирующего конструктора и копирующего присваивания. Если бы он делал это, то потерялся бы смысл владения. Если вы чувствуете, что вам нужно копирование, то рассмотрите возможность применения **shared_ptr** (§34.3.2).

Можно иметь **unique_ptr** для встроенного массива. Пример:

```
unique_ptr<int[]> make_sequence(int n)
{
    unique_ptr p {new int[n]};
    for (int i=0; i<n; ++i)
        p[i]=i;
    return p;
}
```

Это реализуется специализацией:

```
template<typename T, typename D>
class unique_ptr<T[], D> { // специализация для массивов (§ iso.20.7.1.3)
                           // умолчательный D=default_delete<T> идет от общего unique_ptr
public:
//    ... как unique_ptr для индивидуальных объектов, но с [] вместо * и -> ...
};
```

Во избежание срезки (§17.5.1.4) **Derived[]** не принимается в качестве аргумента **unique_ptr<Base[]>**, даже если **Base** является открытым базовым классом для **Derived**. Например:

```
class Shape {
    // ...
```

```
};
class Circle : public Base {
    // ...
};

unique_ptr<Shape> ps {new Circle{p,20}};                    // OK
unique_ptr<Shape[]> pa {new Circle[] {Circle{p,20}, Circle{p2,40}};    // ошибка
```

Как лучше понимать **unique_ptr**? Как лучше всего использовать **unique_ptr**? Ясно, что это не простой указатель. Рассмотрим технический пример:

```
unique_ptr<int> f(unique_ptr<int> p)
{
    ++*p;
    return p;
}

void f2(const unique_ptr<int>& p)
{
    ++*p;
}

void use()
{
    unique_ptr<int> p {new int{7}};
    p=f(p);                 // ошибка: нет копирующего конструктора
    p=f(move(p));           // передача владения туда и обратно
    f2(p);                  // передача ссылки
}
```

Тело функции **f2()** немного меньше, чем у **f()**, и **f2()** проще вызвать, но я нахожу **f()** проще для понимания. Стиль, который иллюстрируется с помощью **f()**, открытый по отношению к владению (использование **unique_ptr** преимущественно мотивируется этим вопросом). Можно также еще раз посмотреть на дискуссию про использование не-**const** ссылок в §7.7.1. Но обозначение **f(x)**, что может модифицировать **x**, более подвержено ошибкам, чем обозначение **y=f(x)**.

По честной оценке вызов **f2()** на одну или две машинные инструкции быстрее, чем вызов **f()** (из-за необходимости размещения **nullptr** в исходном **unique_ptr**), но маловероятно, чтобы это имело большое значение. С другой стороны, доступ к содержащемуся указателю включает лишнее косвенное обращение в **f2()** по сравнению с **f()**. Это, тоже, не слишком важно в большинстве программ, так что выбор стиля программирования с **f()** или **f2()** должен базироваться на рассуждениях о качестве кода.

Вот простой пример уничтожателя, используемого для гарантированного освобождения данных, полученных из фрагмента программы на языке C, использующего **malloc()** (§43.5):

```
extern "C" char* get_data(const char* data);   // получаем data из фрагмента C-программы

using PtoCF = void(*)(void*);

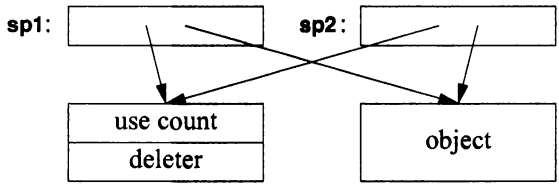
void test()
{
    unique_ptr<char, PtoCF> p {get_data("my_data"), free};
    // ... используем *p ...
}   // неявный free(p)
```

В настоящее время в стандартной библиотеке нет функции **make_unique()**, аналогичной **make_pair()** (§34.2.4.1) и **make_shared()** (§34.3.2). Однако, это можно легко определить самим:

```
template<typename T, typename ... Args>
unique_ptr<T> make_unique(Args&&... args)      // умолчательный уничтожитель
{
    return unique_ptr<T>{new T{args...}};
}
```

34.3.2. shared_ptr

Тип **shared_ptr** представляет коцепцию совместного (разделенного) владения. Он используется тогда, когда две части кода нуждаются в совместном доступе к данным, но без эксклюзивности (то есть нет никого, кто был бы четко ответственен за уничтожение объекта). Указатель **shared_ptr** относится к указателям с подсчетом ссылок, когда указуемый объект уничтожается только в случае, если число ссылок на него становится равным нулю. Можно думать о **shared_ptr** как о структуре с двумя указателями: один на объект, а второй для подсчета ссылок:



Здесь объект *deleter* («уничтожатель») — это то, что используется для уничтожения разделяемого объекта, когда число активных ссылок становится равным нулю. Умолчательный «уничтожатель» есть просто операция **delete** (активирует деструктор в случае его наличия и освобождает ранее выделенную свободную память).

Для примера рассмотрим **Node** в общем графе, который используется алгоритмом, добавляющим и устраняющим как узлы, так и связи между ними (ребра графа). Очевидно, что во избежание утечек ресурсов **Node** должен удаляться тогда и только тогда, когда никакой другой узел не ссылается на него. Можно попробовать следующее:

```
struct Node {
    vector<Node*> edges;
    // ...
};
```

При этом ответить на вопрос «сколько узлов указывают на данный узел?» не так-то просто (требует массу вспомогательного кода). Можно встроить сборщик мусора (§34.5), но это может повлечь за собой ухудшение производительности, если граф является лишь частью большого приложения. Что еще хуже, контейнер может содержать и ресурсы, отличные от памяти, такие как дескрипторы потоков, файловые дескрипторы, блокировки и т.д., и в таком случае даже сборщик мусора может допустить утечки ресурсов.

Мы можем применить **shared_ptr**:

```
struct Node {
    vector<shared_ptr<Node>> edges;
    thread worker;
    // ...
};
```

Здесь деструктор **Node** (вполне подойдет и неявно генерируемый деструктор) уничтожает свои **edges**. То есть, деструктор для каждого **edges[i]** активируется и указуемый **Node** уничтожается, если **edges[i]** был последним указателем на узел.

Не применяйте **shared_ptr** просто для того, чтобы передать указатель от одного владельца другому, для чего предназначен **unique_ptr**, который выполняет эту работу лучше и с меньшими издержками. Если вы используете подсчитываемые указатели как возвращаемые значения из фабричной функции (§21.2.4) или из чего-либо в этом роде, рассмотрите вариант модификации кода до **unique_ptr**, но не **shared_ptr**.

Не следует бездумно заменять указатели на **shared_ptr** в попытке избежать утечек памяти, так как **shared_ptr** не панацея и они не бесплатны:

- Зацикленные структуры с **shared_ptr** могут вызвать утечку ресурсов. Вам нужно логически разорвать циклическую зависимость, используя например **weak_ptr** (§34.3.3).
- Объекты с совместным владением имеют тенденцию оставаться «живыми» дольше объектов в областях видимости, вызывая тем самым больший коэффициент использования ресурсов в среднем.
- Разделяемые указатели в многопоточном окружении весьма дороги по причине необходимости отслеживания возможности ситуации гонки за данными на счетчике ссылок.
- Деструктор разделяемого объекта не выполняется в предсказуемое время, так что алгоритмы для обновления разделяемого объекта имеют больший шанс «попасть впросак», чем для неразделяемого объекта (чьи блокировки устанавливаются во время выполнения деструктора? Чьи файлы открываются?). В самом общем случае, какие объекты живы и находятся в соответствующем состоянии в непредсказуемой точке исполнения кода?
- Если узел сохраняет в активном состоянии большой объем данных, то целый каскад деструкторов вызывается при его последующем уничтожении, что для сборщика мусора может оказаться весьма длительной операцией. Это недопустимо в системах реального времени.

Ясно, что **shared_ptr** отражает концепцию совместного владения, что во многих случаях точно отражает ситуацию, но для меня лично это не идеал, так как всегда требуется очень точно отразить смысл совместного владения. Лучше, когда объект имеет вполне определенного владельца и четко определенное время существования. Когда есть выбор:

- Предпочитайте **unique_ptr**, а не **shared_ptr**.
- Предпочитайте объекты в области видимости, а не объекты в свободной памяти (в куче), указуемые через **unique_ptr**.

Тип **shared_ptr** предоставляет достаточно традиционный набор операций:

Операции над shared_ptr<T> (§ iso.20.7.2.2) **cp** – содержащийся указатель; **uc** – счетчик ссылок	
shared_ptr sp {}	Конструктор по умолчанию: **cp=nullptr**; **uc=0**; noexcept
shared_ptr sp {p}	Конструктор: **cp=p**; **uc=1**
shared_ptr sp {p,del}	Конструктор: **cp=p**; **uc=1**; использовать «уничтожатель» **del**
shared_ptr sp {p,del,a}	Конструктор: **cp=p**; **uc=1**; использовать «уничтожатель» **del** и аллокатор **a**
shared_ptr sp {sp2}	Копирующий и перемещающий конструкторы: перемещающий констр-р перемещает и устанавливает **sp2.cp=nullptr**; копирующий констр-р копирует и устанавливает **++uc** для теперь уже разделяемого **uc**
sp.~shared_ptr()	Деструктор: **--uc**; уничтожить объект, на к-рый указывает **cp**, если **uc** стал равен **0**, используя «уничтожатель» (по умолчанию – **delete**)
sp=sp2	Копирующее присваивание: **++uc** для теперь уже разделяемого **uc**; noexcept

Продолжение

Операции над shared_ptr<T> (§ iso.20.7.2.2) cp – содержащийся указатель; uc – счетчик ссылок	
sp=move(sp2)	Перемещающее присваивание: sp2.cp=nullptr для теперь уже разделяемого uc; noexcept
bool b {sp};	Преобразование в bool: sp.uc==nullptr; explicit
sp.reset()	shared_ptr{}.swap(sp); то есть, sp содержит pointer{}, а уничтожение временного shared_ptr{} уменьшает счетчик ссылок для старого объекта; noexcept
sp.reset(p)	shared_ptr{p}.swap(sp); т.е., sp.cp=p; uc==1; уничтожение временного shared_ptr уменьшает счетчик ссылок для старого объекта
sp.reset(p, d)	Как sp.reset(p), но с «уничтожателем» d
sp.reset(p, d, a)	Как sp.reset(p), но с «уничтожателем» d и аллокатором a
p=sp.get()	p=sp.cp; noexcept
x=*sp	x=*sp.cp; noexcept
x=sp->m	x=sp.cp->m; noexcept
n=sp.use_count()	n есть знач-е счетчика ссылок (0, если sp.cp==nullptr)
sp.unique()	sp.uc==1? (не проверяет sp.cp==nullptr)
x=sp.owner_before(pp)	x есть ф-ия упорядочения (строгое слабое упорядочение; §31.2.2.1) pp есть shared_ptr или weak_ptr
sp.swap(sp2)	Обменять значения sp и sp2; noexcept

Кроме того, стандартная библиотека предоставляет ряд полезных вспомогательных функций:

Вспомогательные функции shared_ptr<T> (§ iso.20.7.2.2.6, § iso.20.7.2.2.7)	
sp=make_shared(args)	sp есть shared_ptr<T> для объекта типа T, сконструированного из аргументов args; выделенного при помощи new
sp=allocate_shared(a, args)	sp есть shared_ptr<T> для объекта типа T, сконструированного из аргументов args; выделенного при помощи аллокатора a
sp==sp2	sp.cp==sp2.cp; sp или sp2 может быть nullptr
sp<sp2	less<T*>(sp.cp, sp2.cp); sp или sp2 может быть nullptr
sp!=sp2	!(sp=sp2)
sp>sp2	sp2<sp
sp<=sp2	!(sp>sp2)
sp>=sp2	!(sp<sp2)
swap(sp, sp2)	sp.swap(sp2)
sp2=static_pointer_cast(sp)	static_cast для sp2=shared_ptr<T>(static_cast<T*>(sp.cp)); noexcept
sp2=dynamic_pointer_cast(sp)	dynamic_cast для shared_ptr: sp2=shared_ptr<T>(dynamic_cast<T*>(sp.cp)); noexcept
sp2=const_pointer_cast(sp)	const_cast для shared_ptr: sp2=shared_ptr<T>(const_cast<T*>(sp.cp)); noexcept
dp=get_deleter<D>(sp)	Если sp имеет «уничтожатель» типа D, *dp есть «уничтожатель» sp; иначе, dp==nullptr; noexcept
os<<sp	Вывести sp в ostream os

Например:

```
struct S {
    int i;
    string s;
    double d;
    // ...
};

auto p = make_shared<S>(1,"Ankh Morpork",4.65);
```

Здесь p есть **shared_ptr<S>**, указывающий на объект типа **S**, аллоцированный в куче и содержащий **{1, string{"Ankh Morpork"},4.65}**.

Отметим, что в отличие от **unique_ptr::get_deleter()**, «уничтожатель» у **shared_ptr** не является функцией-членом.

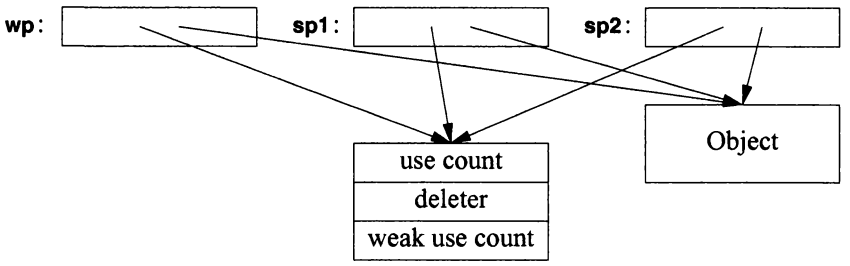
34.3.3. weak_ptr

Тип **weak_ptr** служит для указания на объект, управляемый с помощью **shared_ptr**. Для доступа к объекту **weak_ptr** можно конвертировать к типу **shared_ptr**, используя функцию-член **lock()**. **weak_ptr** позволяет получить доступ к объекту, которым владеет кто-то другой, если

- Вам нужно иметь такой доступ в случае его наличия
- Может быть уничтожен в любое время
- Должен иметь деструктор, вызываемый после последнего использования (обычно для освобождения ресурсов, не связанных с памятью)

Особенно слабые (weak) указатели используются для того, чтобы разорвать циклы в структурах данных, управляемых посредством **shared_ptr**.

Думайте о **weak_ptr** как о структуре с двумя указателями: один на потенциально разделяемый объект, а второй на структуру для подсчета ссылок у **shared_ptr** объекта:



Поле для подсчета ссылок нужно, чтобы хранить актуальной структуру подсчета, поскольку может существоать **weak_ptr** после того, как последний **shared_ptr** на объект уничтожен.

```
template<typename T>
class weak_ptr {
public:
    using element_type = T;
    // ...
};
```

Тип **weak_ptr** должен конвертироваться в **shared_ptr** для доступа к своему объекту, и поэтому этот тип предоставляет совсем немного операций:

weak_ptr<T> (§ iso.20.7.2.3) cp – содержащийся указатель; wuc – счетчик слабых ссылок	
weak_ptr wp {};	Конструктор по умолчанию: **cp=nullptr**; constexpr; noexcept
weak_ptr wp {pp};	Копирующий конструктор: **cp=pp.cp**; **++wuc**; **pp** есть **weak_ptr** или **shared_ptr**; noexcept
wp.~weak_ptr()	Деструктор: не влияет на ***cp**; **–wuc**
wp=pp	Копирование: уменьшить **wuc** и присвоить **wp** знач-е **pp**: **weak_ptr(pp).swap(wp)**; **pp** есть **weak_ptr** или **shared_ptr**; noexcept
wp.swap(wp2)	Обменять значения **wp** и **wp2**; noexcept
wp.reset()	Уменьшить **wuc** и присвоить **wp** знач-е **nullptr**: **weak_ptr{}.swap(wp)**; noexcept
n=wp.use_count()	**n** есть число **shared_ptr**, указывающих на ***cp**; **noexcept**
wp.expired()	Указывают ли какие-либо **shared_ptr** на ***cp**? noexcept
sp=wp.lock()	Создать новый **shared_ptr** на ***cp**; noexcept
x=wp.owner_before(pp)	**x** есть ф-ия упорядочения (строгое слабое упорядочение; §31.2.2.1); **pp** есть **shared_ptr** или **weak_ptr**
swap(wp, wp2)	**wp.swap(wp2)**; **noexcept**

Рассмотрим реализацию старой темы про астероиды. Все астероиды принадлежат игре. Каждый астероид отслеживает соседние астероиды и обрабатывает ситуацию столкновения. При столкновении в типичном случае один или несколько астероидов разрушаются. Каждый астероид хранит список астероидов в его непосредственной близости. Астероид не может быть уничтожен, пока на него «глядит» другой астероид (то есть выясняет возможность столкновения). Деструктор астероида должен вызываться для освобождения ресурсов. Создадим список астероидов с отслеживанием соседей:

```
void owner()
{
    // ...
    vector<shared_ptr<Asteroid>> va(100);
    for (int i=0; i<va.size(); ++i) {
        // ... вычисляем соседей нового астероида ...
        va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor])));
        launch(i);
    }
    // ...
}
```

Очевидно, что я радикально упростил владельца (owner) и дал каждому новому **Asteroid** точно одного соседа. Здесь очень важно, что мы снабжаем **Asteroid** указателем **weak_ptr** на соседа. Владелец хранит **shared_ptr** для отражения факта совместного владения. Расчет для столкновений астероидов выглядит следующим образом:

```
void collision(weak_ptr<Asteroid> p)
{
    if (auto q = p.lock()) {   // p.lock возвращает shared_ptr на объект, указуемый p
                               // ... тот Asteroid все еще существует: вычисляем ...
    }
    else {     // упс: тот Asteroid был уничтожен
        p.reset();
    }
}
```

Отметим, что даже если пользователь решит прикрыть эту игру и удалить все астероиды (путем уничтожения всех **shared_ptr**, отражающих факты владения), то каждый **Asteroid**, находящийся в середине вычислений по поводу столкновений, завершается корректно: после **p.lock()** он содержит **shared_ptr**, который не будет при этом недействительным.

34.4. Аллокаторы

STL контейнеры (§31.4) и тип **string** (Глава 36) являются дескрипторами ресурсов, которые захватывают и освобождают память, в которой хранят свои элементы. Для этого они используют *аллокаторы*. Основная задача аллокаторов состоит в предоставлении источника памяти для данного типа и местом для возврата этой памяти, когда в ней пропадает нужда. В итоге, вот базовые функции аллокатора:

```
p=a.allocate(n);      // захватываем память для n объектов типа T
a.deallocate(p, n);   // освобождаем память под n объектами типа T, указуемую посредством p
```

Пример:

```
template<typename T>
struct Simple_alloc {     // используем new[] и delete[] для аллокации/деаллокации байт
    using value_type = T;

    Simple_alloc() {}

    T* allocate(size_t n)
        {return reinterpret_cast<T*>(new char[n*sizeof(T)]);}
    void deallocate(T* p, size_t n)
        {delete[] reinterpret_cast<char*>(p);}

    // ...
};
```

Simple_alloc является простейшим аллокатором, удовлетворяющим стандартным требованиям. Обратите внимание на конвертацию в и из типа **char***: **allocate()** не активизирует конструкторы, а **deallocate()** не активизирует деструктор; они имеют дело с памятью, а не с типизированными объектами.

Можно построить собственный аллокатор для выделения памяти из произвольной области:

```
class Arena {
    void* p;
    int s;
public:
    Arena(void* pp, int ss);  // аллоцируем из p[0..ss-1]
};

template<typename T>
struct My_alloc {    // используем Arena для аллокации/деаллокации байтов
    Arena& a;
    My_alloc(Arena& aa) : a(aa) {}
    My_alloc() {}
    // обычные средства аллокатора
};
```

Когда арены созданы, объекты можно конструировать в аллоцированной памяти:

```
constexpr int sz {100000};
Arena my_arena1{new char[sz], sz};
Arena my_arena2{new char[10*sz],10*sz};

vector<int> v0;       // аллоцируем при помощи умолчательного аллокатора

vector<int, My_alloc<int>> v1 {My_alloc<int>{my_arena1}};  // конструируем в my_arena1

vector<int, My_alloc<int>> v2 {My_alloc<int>{my_arena2}};  // конструируем в my_arena2

vector<int, Simple_alloc<int>> v3;                  // конструируем в свободной памяти
```

В типичном случае сложность и громоздкость нивелируются синонимами. Например:

```
template<typename T>
    using Arena_vec = std::vector<T, My_alloc<T>>;
template<typename T>
    using Simple_vec = std::vector<T, Simple_alloc<T>>;

My_alloc<int> Alloc2 {my_arena2};       // именованный объект-аллокатор

Arena_vec<complex<double>> vcd {{{1,2}, {3,4}}, Alloc2};          // явный аллокатор
Simple_vec<string> vs {"Sam Vimes", "Fred Colon", "Nobby Nobbs"}; // умолчат-й аллокатор
```

Аллокаторы накладывают дополнительные накладные расходы на контейнер в плане объема памяти только, если его объекты на самом деле имеют состояние (как **My_alloc**).

34.4.1. Умолчательный аллокатор

Умолчательный аллокатор, применяющий операции **new** и **delete**, используется всеми контейнерами стандартной библиотеки:

```
template<typename T>
class allocator {
public:
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using pointer = T*;
    using const_pointer = const T*;
    using reference = T&;
    using const_reference = const T&;
    using value_type = T;

    template<typename U>
        struct rebind {using other = allocator<U>;};

    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template<typename U>
        allocator(const allocator<U>&) noexcept;
    ~allocator();

    pointer address(reference x) const noexcept;
    const_pointer address(const_reference x) const noexcept;

    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0); // аллоцируем n байт
    void deallocate(pointer p, size_type n);                  // деаллоцируем n байт

    size_type max_size() const noexcept;
```

```
        template<typename U, typename ... Args>
            void construct(U* p, Args&&... args);   // new(p) U{args}
        template<typename U>
            void destroy(U*p); //p->~U()
};
```

Шаблон **rebind** является архаичным синонимом. Лучше предпочесть следующее:

```
template<typename U>
using other = allocator<U>;
```

Однако ж, **allocator** был определен до того, как такие синонимы стали поддерживаться языком C++. Он был создан для того, чтобы позволить аллокаторам аллоцировать объекты произвольного типа. Рассмотрим следующее:

```
using Link_alloc = typename A::template rebind<Link>::other;
```

Если **A** есть **allocator**, тогда **rebind<Link>::other** есть синоним для **allocator<Link>**. Например:

```
template<typename T, typename A = allocator<T>>
class list {
private:
    class Link {/* ... */};

    using Link_alloc = typename A::template rebind<Link>::other;    // allocator<Link>

    Link_alloc a;   // link allocator
    A alloc;        // list allocator
    // ...
};
```

Предоставляется и более ограниченная специализация **allocator<T>**:

```
template<>
class allocator<void> {
public:
    typedef void* pointer;
    typedef const void* const_pointer;
    typedef void value_type;
    template<typename U> struct rebind {typedef allocator<U> other;};
};
```

Это помогает нам избегать специальных случаев: мы можем ссылаться на **allocator<void>** до тех пор, пока мы не разыменовываем его указатели.

34.4.2. Характеристики аллокаторов

Аллокаторы совместно характеризуются посредством **allocator_traits**. Свойство аллокатора, скажем тип его **pointer**, находится следующим образом: **allocator_traits<X>::pointer**. Можно построить аллокатор для типа, у которого нет типов членов, отвечающих требованиям аллокатора, например **int**, или для типа, спроектированного без каких-либо мыслей про аллокаторы.

По-существу, **allocator_traits** предоставляет умолчания для обычного набора синонимов типа, а также функции аллокаторов. По сравнению с умолчательным **allocator** (§34.4.1) **address()** отсутствует и добавляется **select_on_container_copy_construction()**:

```
template<typename A>   // § iso.20.6.8
struct allocator_traits {
    using allocator_type = A;
```

```
    using value_type = A::value_type;
    using pointer = value_type;                                          // трюк
    using const_pointer = Pointer_traits<pointer>::rebind<const value_type>; // трюк
    using void_pointer = Pointer_traits<pointer>::rebind<void>;          // трюк
    using const_void_pointer = Pointer_traits<pointer>::rebind<const void>; // трюк
    using difference_type = Pointer_traits<pointer>::difference_type;    // трюк
    using size_type = Make_unsigned<difference_type>;                    // трюк
    using propagate_on_container_copy_assignment = false_type;           // трюк
    using propagate_on_container_move_assignment = false_type;           // трюк
    using propagate_on_container_swap = false_type;                      // трюк

    template<typename T> using rebind_alloc = A<T, Args>;                // трюк
    template<typename T> using rebind_traits = Allocator_traits<rebind_alloc<T>>;

    static pointer allocate(A& a, size_type n) {return a.allocate(n);}   // трюк
    static pointer allocate(A& a, size_type n, const_void_pointer hint)  // трюк
        {return a.allocate(n, hint);}
    static void deallocate(A& a, pointer p, size_type n) {a.deallocate(p, n);} // трюк

    template<typename T, typename ... Args>
        static void construct(A& a, T* p, Args&&... args)                // трюк
        {::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);}
    template<typename T>
        static void destroy(A& a, T* p) {p->T();}                        // трюк

    static size_type max_size(const A& a)                                // трюк
        {return numeric_limits<size_type>::max()}
    static A select_on_container_copy_construction(const A& rhs) {return a;} // трюк
};
```

Суть трюка в том, чтобы использовать эквивалентный член у аллокатора **A**, если он есть; в противном случае используется показанные здесь умолчания. Для **allocate(n, hint)**, **A::allocate(n)** будет вызываться, если у **A** нет члена **allocate()**, принимающего hint (hint — намек, подсказка). Здесь **Args** — аргументы любого типа, требуемые **A**.

Я не сторонник никаких фокусов и трюков в определении стандартной библиотеки, но вольное использование **enable_if()** (§28.4) позволяет реализовать это в языке C++.

Чтобы улучшить читаемость объявлений, я допускаю несколько синонимов типа.

34.4.3. Характеристики указателей

Аллокаторы используют **pointer_traits** для определения свойств указателей и прокси-типов для указателей:

```
template<typename P>   // § iso.20.6.3
struct pointer_traits {
    using pointer = P;
    using element_type = T;                 // трюк
    using difference_type = ptrdiff_t;      // трюк

    template<typename U>
        using rebind = T*;                  // трюк

    static pointer pointer_to(a);           // трюк
};

template<typename T>
struct pointer_traits<T*> {
    using pointer = T*;
    using element_type = T;
```

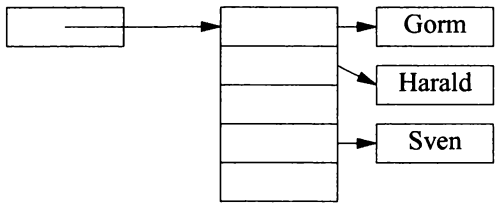
```
        using difference_type = ptrdiff_t;
        template<typename U>
            using rebind = U*;

    static pointer pointer_to(x) noexcept {return addressof(x);}
    };
```

Здесь тот же трюк, что и в случае с **allocator_traits** (§34.4.2): использовать эквивалентный член указателя **P**, если он существует, и умолчательный вариант в противном случае. Чтобы использовать **T**, шаблонный аргумент **P** должен быть первым аргументом у шаблона **Ptr<T, args>**.

34.4.4. Вложенные аллокаторы

Тонкая проблема может возникнуть при использовании контейнеров и пользовательских аллокаторов: должен ли элемент находиться в той же самой аллоцируемой области, что и сам содержащий его контейнер? К примеру, если вы используете **Your_allocator** для **Your_string** для аллоцирования элементов, а я использую **My_allocator** для аллоцирования элементов **My_vector**, то какой аллокатор следует использовать для строковых элементов в **My_vector<Your_allocator>>**?



Решением будет возможность сказать контейнеру, какой аллокатор передавать элементам. Ключевым компонентом такого решения является класс **scoped_allocator**, который предоставляет механизм отслеживания внешнего аллокатора (который будет использоваться для элементов) и внутреннего аллокатора (который будет передаваться элементам для их внутреннего применения):

```
template<typename OuterA, typename... InnerA> // § iso.20.12.1
class scoped_allocator_adaptor : public OuterA {
private:
    using Tr = allocator_traits<OuterA>;
public:
    using outer_allocator_type = OuterA;
    using inner_allocator_type = see below;
    using value_type = typename Tr::value_type;
    using size_type = typename Tr::size_type;
    using difference_type = typename Tr::difference_type;
    using pointer = typename Tr::pointer;
    using const_pointer = typename Tr::const_pointer;
    using void_pointer = typename Tr::void_pointer;
    using const_void_pointer = typename Tr::const_void_pointer;
    using propagate_on_container_copy_assignment = /* см. § iso.20.12.2 */;
    using propagate_on_container_move_assignment = /* см. § iso.20.12.2 */;
    using propagate_on_container_swap = /* см. § iso.20.12.2 */;

    // ...
};
```

Есть четыре альтернативы для аллокации векторов строк:

```
// vector и string используют собственные умолчательные аллокатора:
using svec0 = vector<string>;
svec0 v0;

// vector (и только) использует My_alloc, а string использует умолчательный аллокатор:
using Svec1 = vector<string, My_alloc<string>>;
Svec1 v1 {My_alloc<string>{my_arena1}};

// vector и string используют My_alloc:
using Xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
using Svec2 = vector<Xstring, scoped_allocator_adaptor<My_alloc<Xstring>>>;
Svec2 v2 {scoped_allocator_adaptor<My_alloc<Xstring>>{my_arena1}};

// vector использует собственный умолчательный аллокатор, а string использует My_alloc:
using Xstring2 = basic_string<char, char_traits<char>, My_alloc<char>>;
using Svec3 = vector<xstring2, scoped_allocator_adaptor<My_alloc<xstring>, My_
alloc<char>>>;
Svec3 v3 {scoped_allocator_adaptor<My_alloc<xstring2>, My_alloc<char>>{my_arena1}};
```

Очевидно, что первый вариант, **Svec0**, будет в настоящий момент самым общим, но для систем с серьезными ограничениями по производительности в связи с памятью, остальные варианты (особенно **Svec2**) будут важны и полезны. Несколько синонимов сделают этот код более читаемым, но в любом случае это не код, который вам придется писать каждый день.

В своей основе **scoped_allocator_adaptor** это аллокатор, вполне похожий на умолчательный **allocator** (§34.4.1), но который также отслеживает свой внутренний аллокатор, передаваемый для использования внутренними (вложенными, содержащимися) контейнерами, такими как **string**:

scoped_allocator_adaptor<OuterA, InnerA> (Сокращено, § iso.20.12.1)	
rebind<T>::other	Псевдоним для версии этого аллокатора, выделяющего объекты типа **T**
x=a.inner_allocator()	**x** есть внутренний аллокатор; **noexcept**
x=a.outer_allocator()	**x** есть внешний аллокатор; **noexcept**
p=a.allocate(n)	Выделить память под **n** объектов типа **value_type**
p=a.allocate(n, hint)	Выделить память под **n** объектов типа **value_type**; **hint** есть зависящая от реализации подсказка для аллокатора; часто, **hint** – указатель на объект, близко к к-рому желательно разместить ***p**
a.deallocate(p, n)	Освободить память под **n** объектов типа **value_type**, на к-рые указывает **p**
n=a.max_size()	**n** есть наибольшее возможное число выделенных объектов
t=a.construct(args)	Создать **value_type** из **args**: **t=new(p) value_type{args}**
a.destroy(p)	Уничтожить ***p**: **p->~value_type()**

34.5. Интерфейс сборщика мусора

Сборка мусора (автоматическая утилизация областей памяти без активных ссылок на них) иногда преподносится как панацея. Это не так. В частности, ресурсы, не относящиеся к памяти, вполне могут утекать и при сборщике мусора: это и файловые дескрипторы, дескрипторы потоков, блокировки и т.д. Лично я трактую сборку мусора как самое последнее средство после того, как традиционные методики предотвращения утечек были исчерпаны:

1. Используйте дескрипторы ресурсов с надлежащей семантикой приложения всюду, где это только возможно. В стандартной библиотеке это типы **string**, **vector**, **unordered_map**, **thread**, **lock_guard** и многие другие. Перемещающая семантика обеспечивает возможность эффективного возврата из функций для таких типов.
2. Применяйте **unique_ptr** для владения объектами, которые не управляют неявно собственными ресурсами (вроде указателей) и не могут вовремя удалять объекты (поскольку не имеют надлежащих деструкторов), или когда речь идет об объектах, аллоцируемых способами, требующими специального внимания («уничтожатели» — deleters).
3. Применяйте **shared_ptr** для работы с объектами, требующими совместного владения.

При правильном и согласованном применении перечисленные техники гарантируют отсутствие утечек (и никакой сборки мусора не требуется, так как нет никакого мусора). Но во множестве реальных приложений нет никакого согласованного использования этих техник, а иногда их применение и невозможно из-за нагромождения кода, структурированного разными способами. Примерами служат сложные индивидуальные приемы использования указателей, «голые» **new** и **delete**, явные приведения типов, скрывающие владение ресурсами, а также низкоуровневый код, чреватый самыми разными ошибками. В перечисленных вариантах кода сборка мусора становится последним прибежищем. Не забывайте о том, что общие финализаторы в системах сборки мусора могут работать с ресурсами и вне ресурсов памяти. Иногда сборщики мусора если и не устраняют полностью утечки, но все же могут существенно продлить время жизни работы приложения. Например, если при этом происходит удлинение времени работы с часов до нескольких дней, то это полностью спасает приложения, которые подвергаются перезапуску и ручной очистке ежедневно (или еженощно). Кроме того, сборщики мусора могут выступить в качестве инструмента для обнаружения источников утечек.

Неплохо также понимать, что системы сборки мусора могут обладать собственными источниками утечек. Например, если мы поместим указатель на объект в хэш-таблицу и «забудем» ключ к нему, то объект будет де-факто потерян. Аналогично, ресурс, на который ссылается бесконечный поток, будет жить вечно, даже если поток и не предполагал такого своего времени жизни (например он ожидает ввода, который так и не происходит). Иногда такой ресурс, который живет избыточное время, ничуть не лучше классической утечки.

С рассмотренной точки зрения становится понятно, почему сборка мусора в языке C++ факультативна (опциональна). Сборщик мусора не вызывается, если только он не будет явно проинсталлирован и активирован. Сборщик мусора даже не является частью стандартной поставки языка C++, но благодаря коммерческим поставщикам он встречается на практике. В языке C++ имеется определение того, что может делать сборщик мусора, если таковой используется, а также соответствующий ABI (Application Binary Interface — Бинарный Интерфейс Приложений).

Правила для указателей и времени жизни выражаются в терминах *безопасных указателей* (§ iso.3.7.4.3). Безопасный указатель есть грубо говоря «указатель на нечто, аллоцированное при помощи **new** или указатель на его подобъект». Здесь мы даем примеры небезопасных (замаскированных) указателей. Заставим указатели указывать на что-либо на некоторое время:

```
int* p = new int[100];
p+=10;
// ... сборщик может располагаться здесь ...
p -= 10;
*p = 10;  // можно быть уверенным, что int еще здесь?
```

Скрываем указатель на int:

```
int* p = new int;
int x = reinterpret_cast<int>(p);     // не переносимо
p = nullptr;
// ... сборщик может работать здесь ...
p = reinterpret_cast<int*>(x);
*p = 10;  // можно быть уверенным, что int еще здесь?
```

Запишем указатель в файл и прочтем его позже:

```
int* p = new int;
cout << p;
p = nullptr;
// ... сборщик может работать здесь ...
cin >> p;
*p = 10;  // можно быть уверенным, что int еще здесь?
```

Используем «трюк с хог» для компрессии двусвязанных списков:

```
using Link = pair<Value, long>;

long xor(Link* pre, Link* suc)
{
    static_assert(sizeof(Link*)<=sizeof(long),"a long is smaller than a pointer");
    return long{pre}^long{suc};
}

void insert_between(Value val, Link* pre, Link* suc)
{
    Link* p = new Link{val, xor(pre, suc)};
    pre->second = xor(xor(pre->second, suc), p);
    suc->second = xor(p, xor(suc->second, pre));
}
```

С этим трюком никакие незамаскированные указатели на связи не сохраняются.

Не применяйте такие трюки в программах, которые могут считаться работающими надежно и доступными для понимания простыми смертными — даже если вы не планируете использовать сборщик мусора. Есть и более отвратительные трюки, вроде разбрасывания битов указателя на несколько машинных слов.

Замаскированные указатели все же могут обнаруживаться внимательными сборщиками мусора, если их битовые карты хранятся в памяти вместе с неправильным типом (long и char[4]) и имеют корректное выравнивание. Про такие указатели говорят, что их можно отслеживать.

Стандартная библиотека позволяет программисту специфицировать места, где нет указателей, подлежащих поиску, и какая память не подлежит отзыву, даже если сборщик не может найти указатель на нее (§ iso.20.6.4):

```
void declare_reachable(void* p);   // объект, указуемый через p не должен утилизироваться
template<typename T>
    T* undeclare_reachable(T* p);                  // отменить declare_reachable()

void declare_no_pointers(char* p, size_t n);       // p[0:n) не содержит указателей
void undeclare_no_pointers(char* p, size_t n);     // отменить declare_no_pointers()
```

Сборщики мусора в C++ традиционно являются *консервативными сборщиками*; то есть, они не перемещают объекты в памяти, и должны предполагать, что любое слово памяти может содержать указатель. Консервативные сборщики более эффективны, чем

считают многие, особенно если программа не генерирует уж слишком много мусора, и **declare_no_pointers()** может сделать их очень эффективными путем устранения больших областей памяти из рассмотрения. Например, мы могли бы использовать **declare_no_pointers()** для того, чтобы сказать компилятору, где в нашей программе располагаются такие данные, как изображения (фотографии), освобождая сборщик от просмотра гигабайт памяти, свободной от указателей на эту память.

Программист может запросить правила безопасности указателей, которые отслеживаются системой:

```
enum class pointer_safety {relaxed, preferred, strict};

pointer_safety get_pointer_safety();
```

Стандарт гласит (§ iso.3.7.4.3): «значение указателя, не являющегося безопасным, является некорректным значением, если только полный указуемый объект не имеет времени жизни динамического хранения и не был заранее объявлен достижимым ... эффект использования недействительного указателя не определен».

Значения соответствующих энумераторов таковы:

- **relaxed**: Безопасные и небезопасные указатели трактуются одинаково (как в C и C++98). Подлежат сборке все объекты, которые не имеют безопасных или отслеживаемых указателей на них.
- **preferred**: Все, как в случае **relaxed**, но сборщик мусора может работать как детектор утечек и/или как детектор разыменования «плохих указателей».
- **strict**: Безопасные и небезопасные указатели трактуются по-разному; то есть, сборщик мусора функционирует и игнорирует небезопасные указатели.

Не существует общепринятого мнения, какие следует предпочесть альтернативы. Тут нужно обратиться к вопросам трактовки качества программной среды и ее реализации.

34.6. Неинициализированная память

В большинстве случаев следует избегать неинициализированной памяти. Это упрощает программирование и помогает избежать множества ошибок. Тем не менее, в достаточно редких случаях, например при написании алллокаторов, реализации контейнеров и когда приходится работать с аппаратурой напрямую, прямое обращение к неинициализированной памяти, известной как «сырая память», необходимо.

В дополнение к стандартному **allocator**, в заголовочном файле **<memory>** предоставляется семейство **fill*** функций, имеющих дело с неинициализированной памятью (§32.5.6). Они имеют дело с опасным использованием типа **T** для отсылки к области памяти, достаточной для хранения объекта типа **T**, вместо того, чтобы работать с корректно инициализированным объектом типа **T**. Эти функции в первую очередь предназначены для реализаторов контейнеров и алгоритмов. Например, **reserve()** и **resize()** реализуются, используя эти функции (§13.6).

34.6.1. Временные буферы

Алгоритмы часто обращаются ко временным хранилищах, которые выделяются в рамках операции, но не инициализируются до тех пор, пока не возникает нужда в конкретном местоположении. Как следствие, стандартная библиотека предоставляет пару функций для аллоцирования и освобождения (деаллоцирования) неинициализированной памяти:

```
template<typename T>
    pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t); // аллоцируем без инициализации
template<typename T>
    void return_temporary_buffer(T*);                    // деаллоцируем, не уничтожаем
```

Операция **get_temporary_buffer<X>(n)** пытается выделить память под **n** или более объектов типа **X**. В случае успеха возвращается указатель на начало выделенной неинициализированной памяти и количество объектов типа **X**, которое может в ней поместиться; в противном случае второе значение в **pair** равно нулю. Идея заключается в возможности системы в быстром выделении памяти, и что при запросе на **n** объектов фиксированного размера может быть выделено больше памяти, чем на **n** объектов. Но может быть выделено памяти и меньше, чем на n объектов, так что оптимальным сценарием будет запрос с некоторым избытком с последующим использованием того, что реально было выделено.

Буфер, полученный с помощью **get_temporary_buffer()**, должен освобождаться для иного использования вызовом **return_temporary_buffer()**. Первая из этих функций аллоцирует без инициализации, а вторая не применяет деструкторов. Функцию **get_temporary_buffer()** не следует использовать вместо **new** или **allocator::allocate()** в случае необходимости в долгосрочном использовании памяти.

34.6.2. raw_storage_iterator

Стандартные алгоритмы, пишущие в последовательность, полагают, что элементы последовательности были ранее инициализированы. То есть, алгоритмы используют присваивания, а не копирующее конструирование для записи в элементы. Следовательно, мы не можем использовать неинициализированную память в качестве непосредственной цели алгоритма. Это вызывает сожаление, так как присваивание дороже, чем инициализация. Решением является использование **raw_storage_iterator** из файла **<memory>**, в результате чего реализуется инициализация вместо присваивания:

```
template<typename Out, typename T>
class raw_storage_iterator : public iterator<output_iterator_tag, void, void, void, void> {
    Out p;
public:
    explicit raw_storage_iterator(Out pp) : p{pp} {}
    raw_storage_iterator& operator*() {return *this;}

    raw_storage_iterator& operator=(const T& val)
    {
        new(&*p) T{val}; // помещаем val в *p (§ 11.2.4)
        return *this;
    }

    raw_storage_iterator& operator++() {++p; return *this;}      // пре-инкремент
    raw_storage_iterator operator++(int)                         // post-инкремент
    {
        auto t = *this;
        ++p;
        return t;
    }
};
```

Никогда не следует использовать **raw_storage_iterator** для записи в инициализированные данные. Поэтому применение этого средства ограничивается глубинами реализаций контейнеров и алгоритмов. Рассмотрим генерирование перестановок (§32.5.5) **string** для использования в целях тестирования:

```
void test1()
{
    auto pp = get_temporary_buffer<string>(1000); // получить неинициализированную память
    if (pp.second<1000) {
        // ... обработка проблемы с аллокацией ...
    }
    auto p = raw_storage_iterator<string*, string>(pp.first); // итератор
    generate_n(p, a.size(),
        [&]{next_permutation(seed, seed+sizeof(seed)-1); return seed;});
    // ...
    return_temporary_buffer(p);
}
```

Этот пример несколько искуственный, так как я не считаю зазорным аллоцировать память и инициализировать ее умолчательным образом для строк, а затем присваивать строки для тестирования.

Заметим, что нет операций сравнения == и != для **raw_storage_iterator**, так что не пытайтесь их использовать для записи в диапазон [**b:e**). Например, **iota(b,e,0)** (§40.6) не будет работать, если **b** и **e** имеют тип **raw_storage_iterator**.

34.7. Советы

[1] Используйте **array** там, где нужна последовательность **constexpr**-размера; §34.2.1.

[2] Предпочитайте **array**, а не встроенные массивы; §34.2.1.

[3] Используйте **bitset**, если вам нужны **N** бит, а **N** не совпадает с числом бит во встроенном целом типе; §34.2.2.

[4] Избегайте типа **vector<bool>**; §34.2.3.

[5] Используя **pair**, рассмотрите возможность применения **make_pair()** для вывода типа; §34.2.4.1.

[6] Используя **tuple**, рассмотрите возможность применения **make_tuple()** для вывода типа; §34.2.4.2.

[7] Используйте **unique_ptr** для представления эксклюзивного владения; §34.3.1.

[8] Используйте **shared_ptr** для представления совместного владения; §34.3.2.

[9] Минимизируйте применение **weak_ptr**; §34.3.3.

[10] Используйте аллокаторы (лишь) тогда, когда обычная семантика **new**/**delete** недостаточна по логическим причинам, или по причинам эффективности; §34.4.

[11] Предпочитайте дескрипторы ресурсов с конкретной семантикой, а не умные указатели; §34.5.

[12] Предпочитайте **unique_ptr**, а не **shared_ptr**; §34.5.

[13] Вместо сборки мусора используйте дескрипторы ресурсов; §34.5.

[14] Применяйте полную и согласованную стратегию управления ресурсами; §34.5.

[15] Сборщики мусора оказываются на самом деле полезными в программах, в которых имеется хаотическое использование указателей; §34.5.

[16] Сборка мусора опциональна; §34.5.

[17] Не маскируйте указатели (даже если вы не используете сборку мусора); §34.5.

[18] Применяя сборку мусора, используйте **declare_no_pointers** для помощи сборщику мусора в игнорировании кода, не содержащего указателей; §34.5.

[19] Не допускайте хаотического использования неинициализированной памяти; §34.6.

35

Прочие компоненты библиотеки

Наслаждаясь ничегонеделанием,
вы ничего не теряете.
— Бертран Рассел

- Введение
- Время
  duration; **time_point**; Часы; Характеристики времени
- Рациональная арифметика времени компиляции
- Функции типа
  Характеристики типов; Генераторы типа
- Небольшие утилиты
  move() и **forward()**; **swap()**; Операции сравнения; Сравнение и хэширование **type_info**
- Советы

35.1. Введение

В стандартной библиотеке помимо основных имеется еще и довольно большое множество разрозненных, но практически полезных компонентов. Они не могут классифицироваться единым образом в отличие от базовых компонентов.

35.2. Время

В файле **<chrono>** стандартная библиотека предоставляет средства для работы с интервалами времени и временными отсчетами. Все эти средства определяются в (под) пространстве имен **std::chrono**, так что мы должны либо их явно квалифицировать через **chrono::**, либо использовать **using**-директиву:

```
using namespace std::chrono;
```

Нам часто нужно замерять время разных процессов или иметь дело с вещами, зависящими от времени. К примеру, мьютексы и блокировки стандартной библиотеки имеют опции, позволяющие потоку ждать некоторое время (**duration**) или же ждать до определенного момента времени (**time_point**).

Если нужно узнать текущее значение **time_point**, то можно вызвать функцию **now()** для тех или иных часов: **system_clock**, **steady_clock**, **high_resolution_clock**. Например:

```
steady_clock::time_point t = steady_clock::now();
// ... что-нибудь сделать здесь ...
steady_clock::duration d = steady_clock::now()-t; // взять d единиц времени
```

Часы вовзращают **time_point**, а **duration** есть разность между двумя **time_point** с одних и тех же часов. Как обычно, если вам не нужны детали, то можно воспользоваться **auto**:

```
auto t = steady_clock::now();
// ... что-нибудь тут сделать ...
auto d = steady_clock::now()-t;     // взять d единиц времени
cout << "something took " << duration_cast<milliseconds>(d).count() << "ms"; // вывести милисек.
```

Средства работы со временем предназначены для эффективной поддержки глубинных средств системы, а не для удобства реализации собственного календаря.

Так уж оказывается, что понятие «время» вовсе не такое простое, как может показаться на первый взгляд. Например, часы могут быть не совсем точными и требуется их корректировка, часы могут отличаться разной степенью погрешности и т. д. Более того, средства языка для работы с короткими интервалами (например, с наносекундами) сами должны реализовываться максимально быстро (с наименьшими накладными расходами). Отсюда становится ясно, что средства из файла **<chrono>** сами по себе достаточно сложные. Но многие случаи их использования могут быть простыми.

Средства работы со временем в стиле языка C можно найти в §43.6.

35.2.1. duration

В файле **<chrono>** стандартная библиотека определяет тип **duration** для представления интервала времени между двумя временными точками (**time_point**; §35.2.2):

```
template<typename Rep, typename Period = ratio<1>>
class duration {
public:
    using rep = Rep;
    using period = Period;
    // ...
};
```

duration<Rep,Period> (§ iso.20.11.5)	
duration d {};	Конструктор по умолчанию: **d** становится {**Rep**{},**Period**{}}; constexpr
duration d {r};	Конструктор из **r**; **r** должно быть преобразуемо в **Rep** без сужения constexpr; explicit
duration d {d2};	Копирующий конструктор: **d** получает то же знач-е, что и **d2**; **d2** должно быть преобразуемо в **Rep** без сужения; constexpr
d=d2	**d** получает то же знач-е, что и **d2**; **d2** должно быть представимо, как **Rep**
r=d.count()	**r** есть число тиков в **d**; constexpr

Можно определить **duration** с конкретным значением **period**. Например:

```
duration<long long,milli> d1 {7};   // 7 милисекунд
duration<double,pico> d2 {3.33};    // 3.33 пикосекунд
duration<int,ratio<1,1>> d3 {};     // 0 секунд
```

period содержит число отсчетов (тиков) часов.

```
cout << d1.count() << '\n';         // 7
cout << d2.count() << '\n';         // 3.33
cout << d3.count() << '\n';         // 0
```

Естественно, что значение **count()** зависит от **period**:

```
d2=d1;
cout << d1.count() << '\n';          // 7
cout << d2.count() << '\n';          // 7e+009
if (d1!=d2) cerr<<"insane!";
```

Здесь **d1** и **d2** эквивалентны, но выдают разные значения **count()**.

Нужна осторожность, чтобы избежать усечения или потери точности при инициализации (даже при использовании {}-нотации). Например:

```
duration<int, milli> d {3};          // OK
duration<int, milli> d {3.5};        // ошибка: 3.5 в int это сужение

duration<int, micro> us {ms};        // OK
duration<int, milli> ms2 {us};       // ошибка: можно потерять много микросекунд
```

В стандартной библиотеке имеются осмысленные арифметические операции для типа **duration**:

duration<Rep,Period> (продолжение) **(§ iso.20.11.5)** **r** есть **Rep**; арифм. операции работают над **common_type** представления	
++d	**++d.r**
d++	**duration{d.r++}**
--d	**--d.r**
d--	**duration{d.r--}**
+d	**d**
-d	**duration{-d.r}**
d+=d2	**d.r+=d2.r**
d-=d2	**d.r-=d2.r**
d%=d2	**d.r%=d2.r.count()**
d%=r	**d.r+=r**
d*=r	**d.r+=r**
d/=r	**d.r+=r**

period имеет отношение к системе единиц, так что нет операций = или +=, принимающих «значения вообще». Если бы такие операции были, то тогда разрешалось бы сложение величины **5** неизвестной системы единиц с длиной в метрах. Рассмотрим следующее:

```
duration<long long,milli> d1 {7};    // 7 милисекунд
d1 += 5; // ошибка

duration<int,ratio<1,1>> d2 {7};     // 7 секунд
d2 = 5;  // ошибка
d2 += 5; // ошибка
```

Что здесь может означать **5**? **5** секунд? **5** милисекунд? Что-то еще? Если вы знаете, что все это означает, будьте точными в формулировках. Пример:

```
d1 += duration<long long,milli>{5};  // OK: милисекунды
d3 += decltype(d2){5};               // OK: секунды
```

Арифметика с **duration** для разных представлений допустима до тех пор, пока комбинация имеет смысл (§35.2.4):

duration<Rep,Period> (продолжение) (§ iso.20.11.5)	
r есть Rep; арифм. операции работают над common_type представления	
d3=d+d2	constexpr
d3=d-d2	constexpr
d3=d%d2	constexpr
d2=d%r	d2=d%r.count(); constexpr
d2=d*x	x есть duration или Rep; constexpr
d2=r*d	constexpr
d2=d/x	x есть duration или Rep; constexpr

Поддерживаются сравнения и явные приведения между duration с разными представлениями:

duration<Rep,Period> (продолжение) (§ iso.20.11.5)	
d=zero()	0 для Rep: d=duration{duration_values<rep>::zero()}; constexpr
d=min()	Наименьшее значение Rep (меньше либо равно zero()): d=duration{duration_values<rep>::min()}; constexpr
d=max()	Наибольшее значение Rep (больше либо равно zero()): d=duration{duration_values<rep>::max()}; constexpr
d==d2	Сравнение ведется в common_type для d и d2; constexpr
d!=d2	!(d==d2)
d<d2	Сравнение ведется в common_type для d и d2; constexpr
d<=d2	!(d>d2)
d>d2	Сравнение ведется в common_type для d и d2; constexpr
d>=d2	!(d<d2)
d2=duration_cast<D>(d)	Преобразовать duration d в duration типа D; неявные преобразования не использ-ся ни для представления, ни для периода constexpr

Стандартная библиотека предоставляет несколько удобных синонимов, используя системы единиц из <ratio> (§35.3):

```
using nanoseconds = duration<si64,nano>;
using microseconds = duration<si55,micro>;
using milliseconds = duration<si45,milli>;
using seconds = duration<si35>;
using minutes = duration<si29,ratio<60>>;
using hours = duration<si23,ratio<3600>>;
```

Здесь siN означает «зависящий от реализации знаковый целый тип из как минимум N битов».

duration_cast используется для получения duration с известными единицами измерения. Например:

```
auto t1 = system_clock::now();
f(x); // сделать что-то
auto t2 = system_clock::now();

auto dms = duration_cast<milliseconds>(t2-t1);
cout << "f(x) took " << dms.count() << " milliseconds\n";

auto ds = duration_cast<seconds>(t2-t1);
cout << "f(x) took " << ds.count() << " seconds\n";
```

Причина, по которой нужно приведение в этом примере, состоит в том, что мы отбрасываем информацию: на моей системе **system_clock** отсчитывает в **nanoseconds**.

В ином случае мы можем просто попытаться сконструировать подходящий **duration**:

```
auto t1 = system_clock::now();
f(x); // что-нибудь сделать
auto t2 = system_clock::now();

cout << "f(x) took " << milliseconds(t2-t1).count() << " milliseconds\n"; // ошибка: урезание
cout << "f(x) took " << microseconds(t2-t1).count() << " microseconds\n";
```

Точность часов зависит от реализации.

35.2.2. time_point

В файле **<chrono>** стандартная библиотека определяет тип **time_point** для представления точки времени в заданной эпохе, измеряемой данным **clock**:

```
template<typename Clock, typename Duration = typename Clock::duration>
class time_point {
public:
    using clock = Clock;
    using duration = Duration;
    using rep = typename duration::rep;
    using period = typename duration::period;
    // ...
};
```

Здесь *epoch* это диапазон времени, определяемый посредством **clock**, измеренный в терминах **duration**, начиная с **duration::zero()**:

time_point<Clock,Duration> (§ iso.20.11.6)	
time_point tp {};	Конструктор по умолчанию: начало эпохи: **duration::zero()**
time_point tp {d};	Конструктор: точка времени **d** эпохи: **time_point{}+d**; explicit
time_point tp {tp2};	Конструктор: **tp** указ-т на ту же точку времени, что и **tp2**; тип **duration** у **tp2** должен иметь неявное преобр-ние к таковому у **tp**
d=tp.time_since_epoch()	**d** есть хранимый **duration** у **tp**
tp=tp2	**tp** ссылается на ту же точку времени, что и **tp2**; тип **duration** у **tp2** должен иметь неявное преобр-ние к таковому у **tp**

Например:

```
void test()
{
    time_point<steady_clock,milliseconds> tp1(milliseconds(100));
    time_point<steady_clock,microseconds> tp2(microseconds(100*1000));

    tp1=tp2;  // ошибка: будет урезание
    tp2=tp1;  // OK

    if (tp1!=tp2) cerr << "Insane!";
}
```

Как и для **duration**, для **time_point** предоставляется осмысленная арифметика и сравнения:

time_point<Clock,Duration> (продолжение) (§ iso.20.11.6)	
tp+=d	Передвинуть tp вперед: tp.d+=d
tp-=d	Передвинуть tp назад: tp.d-=d
tp2=tp+d	tp2=time_point<Clock>{tp.time_since_epoc()+d}
tp2=d+tp	tp2=time_point<Clock>{d+tp.time_since_epoc()}
tp2=tp-d	tp2=time_point<Clock>{tp.time_since_epoc()-d}
d=tp-tp2	d=duration{tp.time_since_epoc()-tp2.time_since_epoc()}
tp2=tp.min()	tp2=time_point(duration::min()); static; constexpr
tp2=tp.max()	tp2=time_point(duration::max()); static; constexpr
tp==tp2	tp.time_since_epoch()==tp2.time_since_epoch()
tp!=tp2	!(tp==tp2)
tp<tp2	tp.time_since_epoch()<tp2.time_since_epoch()
tp<=tp2	!(tp2<tp)
tp>tp2	tp2<tp
tp>=tp2	!(tp<tp2)
tp2=time_point_cast<D>(tp)	Преобразовать time_point tp в time_point<C,D>: time_point<C,D>(duration_cast<D>(t.time_since_epoch()))

Например:

```
void test2()
{
  auto tp = steady_clock::now();
  auto d1 = time_point_cast<hours>(tp).time_since_epoch().count()/24; // дни с начала эпохи

  using days = duration<long,ratio<24*60*60,1>>;        // длительность дня
  auto d2 = time_point_cast<days>(tp).time_since_epoch().count(); // дни с начала эпохи

  if (d1!=d2) cout << "Impossible!\n";
}
```

Операции с **time_point**, не выполняющие доступ к часам, могут быть **constexpr**, но в настоящее время это не гарантируется.

35.2.3. Часы

В конечном счете **time_point** и **duration** порождаются аппаратными часами. В файле **<chrono>** стандартная библиотека предоставляет интерфейсы к часам. Класс **system_clock** реализует «время с настенных часов», получаемое из системных часов реального времени:

```
class system_clock {
public:
    using rep = /* зависящий от реализации знаковый тип */;
    using period = /* зависящее от реализации частное<> */;
    using duration = chrono::duration<rep,period>;
    using time_point = chrono::time_point<system_clock>;
    // ...
};
```

Все функции-члены и члены-данные являются статическими. Мы напрямую не имеем дело с объектами-часами. Вместо этого мы используем типы часов:

Функции часов (§ iso.20.11.7)	
is_steady	Является ли этот тип часов равномерным? То есть, **c.now()<=c.now()** для всех последовательных вызовов **now()** и время между тиками постоянно? static
tp=now()	**tp** есть **time_point** у **system_clock** на момент вызова; noexcept
t=to_time_t(tp)	**t** есть **time_t** (§43.6) для the **time_point tp**; noexcept
tp=from_time_t(t)	**tp** есть **time_point** для **time_t t**; noexcept

К примеру:

```
void test3()
{
    auto t1 = system_clock::now();
    f(x); // что-нибудь сделать
    auto t2 = system_clock::now();
    cout << "f(x) took " << duration_cast<milliseconds>(t2-t1).count() << " ms";
}
```

В системе имеются три типа именованных часов:

Типы часов (§ iso.20.11.7)	
system_clock	Часы реального времени системы; системные часы могут быть переведены (перескочить вперед или назад) для соответствия внешним часам
steady_clock	Часы, на которых время идет равномерно вперед; то есть, время не передвигается назад и время между тиками постоянно
high_resolution_clock	Часы с наименьшим шагом времени в системе

Эти три типа часов не обязательно должны быть различными; имена часов в стандартной библиотеке могут быть синонимами.

Можно выявить базовые свойства часов следующим образом:

```
cout << "min " << system_clock::duration::min().count()
   << ", max " << system_clock::duration::max().count()
   << ", " << (treat_as_floating_point<system_clock::duration>::value ? "FP" : "integral") << '\n';

cout << (system_clock::is_steady?"steady\n": "not steady\n");
```

Когда я запустил этот пример на моей системе, то получил следующее:

```
min -9223372036854775808, max 9223372036854775807, integral
not steady
```

Другие системы и другие часы могут дать иные результаты.

35.2.4. Характеристики времени

Реализация средств из **<chrono>** зависит от нескольких стандартных средств, известных как *характеристики времени*.

Правила приведения для **duration** и **time_point** зависят от того, являются ли их представления целыми или значениями с плавающей запятой:

```
template<typename Rep>
struct treat_as_floating_point : is_floating<Rep> { };
```

Предоставляется несколько стандартных значений:

duration_values<Rep> (§ iso.20.11.4.2)	
r=zero()	r=Rep(0); static; constexpr
r=min()	r=numeric_limits<Rep>::lowest(); static; constexpr
r=max()	r=numeric_limits<Rep>::max(); static; constexpr

Общий тип для **duration** выявляется посредством вычисления их наибольшего общего знаменателя (GCD):

```
template<typename Rep1, typename P1, typename Rep2, typename P2>
struct common_type<duration<Rep1,P1>, duration<Rep2, P2>> {
    using type = duration<typename common_type<Rep1,Rep2>::type, GCD<P1,P2>> ;
};
```

Это делает **type** синонимом для **duration** с наибольшим возможным периодом тиков (отсчетов часов), так что оба аргумента у **duration** будут конвертироваться в него без необходимости в операции деления. Это предполагает, что **common_type<R1,P1,R2,P2>::type** может содержать любое значение из **duration<R1,R2>** и **duration<R2,P2>** без ошибок усечения. Однако, **duration** со значениями чисел с плавающей запятой могут иметь ошибки округления.

```
template<typename Clock, typename Duration1, typename Duration2>
struct common_type<time_point<Clock, Duration1>, time_point<Clock, Duration2>> {
    using type = time_point<Clock, typename common_type<Duration1, Duration2>::type>;
};
```

Чтобы иметь **common_type**, два **time_points** должны иметь общий тип часов. Их **common_type** есть **time_point** с **common_type** их **duration**.

35.3. Рациональная арифметика времени компиляции

В файле **<ratio>** находится класс **ratio**, который предоставляет рациональную арифметику времени компиляции. Стандартная библиотека использует **ratio** для представления на этапе компиляции временных длительностей и временных точек (§35.2):

```
template<intmax_t N, intmax_t D = 1>
struct ratio {
    static constexpr intmax_t num;
    static constexpr intmax_t den;

    using type = ratio<num,den>;
};
```

Идея состоит в том, чтобы представить числитель и знаменатель рационального числа как шаблонные аргументы-значения. Знаменатель не должен быть нулем.

Арифметические операции ratio (§ iso.20.10.4)	
z=ratio_add<x,y>	z.num=x::num*y::den+y::num*x::den; z.den=x::den*y::den
z=ratio_subtract<x,y>	z.num=x::num*y::den-y::num*x::den; z.den=x::den*y::den
z=ratio_multiply<x,y>	z.num=x::num*y::num; z.den=x::den*y::den
z=ratio_divide<x,y>	z.num=x::num*y::den; z.den=x::den*y::num
ratio_equal<x,y>	x::num==y::num && x::den==y::den

Продолжение

Арифметические операции ratio (§ iso.20.10.4)	
ratio_not_equal<x,y>	!ratio_equal<x,y>::value
ratio_less<x,y>	x::num*y::den < y::num*x::den
ratio_less_equal<x,y>	!ratio_less_equal<y,x>::value
ratio_not_equal<x,y>	!ratio_less<y,x>::value
ratio_greater<x,y>	ratio_less<y,x>::value
ratio_greater_equal<x,y>	!ratio_less<x,y>::value

Например:

```
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::num == 1, "problem: 1/3+1/6 != 1/2");
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::den == 2, "problem: 1/3+1/6 != 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::num == 1, "problem: 1/3*3/2 != 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::den == 2, "problem: 1/3*3/2 != 1/2");
```

Очевидно, что это неудобно для выражения чисел и арифметики. В файле **<chrono>** мы находим традиционную нотацию (например, + и *) для рациональной арифметики времени (§35.2). Аналогично, для помощи в выражении значений единиц стандартная библиотека предоставляет общие имена для порядков величин:

```
using yocto = ratio<1,1000000000000000000000000>;      // условно поддерживается
using zepto = ratio<1,1000000000000000000000>;         // условно поддерживается
using atto  = ratio<1,1000000000000000000>;
using femto = ratio<1,1000000000000000>;
using pico  = ratio<1,1000000000000>;
using nano  = ratio<1,1000000000>;
using micro = ratio<1,1000000>;
using milli = ratio<1,1000>;
using centi = ratio<1,100>;
using deci  = ratio<1,10>;
using deca  = ratio<10,1>;
using hecto = ratio<100,1>;
using kilo  = ratio<1000,1>;
using mega  = ratio<1000000,1>;
using giga  = ratio<1000000000,1>;
using tera  = ratio<1000000000000,1>;
using peta  = ratio<1000000000000000,1>;
using exa   = ratio<1000000000000000000,1>;
using zetta = ratio<1000000000000000000000,1>;         // условно поддерживается
using yotta = ratio<1000000000000000000000000,1>;      // условно поддерживается
```

Пример использования показан в §35.2.1.

35.4. Функции типа

В файле **<type_traits>** стандартная библиотека определяет функции типа (§28.2) для определения свойств типов (§35.4.1) и для генерации новых типов из существующих типов (генераторы типов; §35.4.2). Эти функции типа преимущественно используются во время компиляции для поддержки простого и непростого метапрограммирования.

35.4.1. Характеристики типов

В файле **<type_traits>** стандартная библиотека предоставляет большой набор функций типа, которые помогают программисту определить свойства типа или пары типов. Их имена по большей части объясняют сами себя. Первичные предикаты типа проверяют фундаментальные свойства типа:

Первичные предикаты типа (§ iso.20.9.4.1)	
is_void<X>	Является ли **X** **void**?
is_integral<X>	Является ли **X** целым типом?
is_floating_point<X>	**X** – вещественный тип?
is_array<X>	**X** – встроенный массив?
is_pointer<X>	**X** – указатель (не считая указателей на члены)?
is_lvalue_reference<X>	**X** есть **lvalue**-ссылка?
is_rvalue_reference<X>	**X** есть **rvalue**-ссылка?
is_member_object_pointer<X>	**X** – указатель на не-**static** поле данных?
is_member_function_pointer<X>	**X** – указатель на не-**static** функцию-член?
is_enum<X>	**X** есть **enum** (обычный или **class enum**)?
is_union<X>	**X** есть **union**?
is_class<X>	**X** есть **class** (включая **struct**'ы, но не **enum**'ы)?
is_function<X>	**X** является функцией?

Элементы этой таблицы возвращают значение, используемое как логическое значение. Для доступа к этим значениям используйте суффикс **::value**. Пример:

```
template<typename T>
void f(T& a)
{
    static_assert(std::is_floating_point<T>::value ,"FP type expected");
    // ...
}
```

Если вам надоест нотация **::value**, то можете определить **constexpr**-функцию (§28.2.2):

```
template<typename T>
constexpr bool Is_floating_point<T>()
{
    return std::is_floating_point<T>::value;
}

template<typename T>
void f(T& a)
{
    static_assert(Is_floating_point<T>(),"FP type expected");
    // ...
}
```

Некоторые функции типа запрашивают информацию о комбинациях фундаментальных свойств:

Композитные предикаты типа (§ iso.20.9.4.2)	
is_reference<X>	**x** – ссылка (lvalue- или rvalue-ссылка)?
is_arithmetic<X>	**x** – арифметический тип (целый или вещественный; §6.2.1)?
is_fundamental<X>	**x** – фундаментальный тип (§6.2.1)?
is_object<X>	**x** – объектный тип (не функция)?
is_scalar<X>	**x** – скалярный тип (не класс или функция)?
is_compound<X>	**x** – пользовательский тип (**!is_fundamental<X>**)?
is_member_pointer<X>	**x** – указатель на не-**static** член данных или функцию-член?

Композитные (комбинированные) предикаты типа удобны с точки зрения нотации. Например, **is_reference<X>** возвращает **true**, если **X** либо lvalue-ссылка, либо rvalue-ссылка.

Как и первичные предикаты типа, предикаты свойств типа выполняют тесты фундаментальных аспектов типа:

Предикаты свойств типа (§ iso.20.9.4.3)	
is_const<X>	**X** есть **const**?
is_volatile<X>	**X** есть **volatile** (§41.4)?
is_trivial<X>	**X** есть тривиальный тип (§8.2.6)?
is_trivially_copyable<X>	Может ли **X** копироваться, перемещаться и уничтожаться как простой набор бит (§8.2.6)?
is_standard_layout<X>	**X** – тип со стандартной раскладкой в памяти (§8.2.6)?
is_pod<X>	**X** есть POD (§8.2.6)?
is_literal_type<X>	Имеет ли **X** **constexpr**-конструктор (§10.4.3)?
is_empty<X>	Имеет ли **X** члены, которые занимают место в объекте?
is_polymorphic<X>	Имеет ли **X** виртуальные функции?
is_abstract<X>	Имеет ли **X** чистые виртуальные функции?
is_signed<X>	**X** – арифметический знаковый тип?
is_unsigned<X>	**X** – арифметический беззнаковый тип?
is_constructible<X,args>	Может ли **X** быть соконструирован из **args**?
is_default_constructible<X>	Может ли **X** быть соконструирован из {}?
is_copy_constructible<X>	Может ли **X** быть соконструирован из **X&**?
is_move_constructible<X>	Может ли **X** быть соконструирован из **X&&**?
is_assignable<X,Y>	Может ли **X** быть присвоено знач-е **Y**?
is_copy_assignable<X>	Может ли **X** быть присвоено знач-е **X&**?
is_move_assignable<X>	Может ли **X** быть присвоено знач-е **X&&**?
is_destructible<X>	Может ли **X** быть уничтоженным (т.е., **~X()** не был подавлен)?

Например:

```
template<typename T>
class Cont {
    T* elem;        // хранение элементов в массиве, указуемым elem
    int sz;         // sz элементов
    // ...
    Cont(const Cont& a)     // копирующий конструктор
        :sz(a.sz), elem(new T[a.elem])
    {
        static_assert(Is_copy_constructable<T>(),"Cont::Cont(): no copy");
        if (Is_trivially_copyable<T>())
            memcpy(elem,a.elem,sz*sizeof(T));       // memcopy оптимизация
        else
            uninitialized_copy(a.begin(),a.end(),elem); // используем копир. конструкторы
    }
    // ...
}
```

Такая оптимизация не так уж и необходима, потому что **uninitialized_copy()** скорее всего уже была оптимизированы в таком ключе.

Чтобы класс был пустым, у него не должно быть виртуальных функций, не должно быть никаких виртуальных базовых классов, и не должно быть никаких базовых классов, для которых **!is_empty<Base>::value**.

Предикаты свойств типа не выполняют на самом деле проверки доступа в зависимости от того, где они используются. Вместо этого, они дают ожидаемый результат для использования вне членов и друзей. Например:

```
class X {
public:
    void inside();
private:
    X& operator=(const X&);
    ~X();
};

void X::inside()
{
    cout << "inside =: " << is_copy_assignable<X>::value << '\n';
    cout << "inside ~: " << is_destructible<X>::value << '\n';
}

void outside()
{
    cout << "outside =: " << is_copy_assignable<X>::value << '\n';
    cout << "outside ~: " << is_destructible<X>::value << '\n';
}
```

Как **inside()**, так и **outside()**, выдадут **00**, что свидетельствует о том, что **X** не является уничтожаемым и не имеет копирующего присваивания. Также, если вам нужно устранить операцию, то используйте **=delete** (§17.6.4), а не полагайтесь на **private**.

Предикаты свойств типа (продолжение) **(§ iso.20.9.4.3)**	
is_trivially_constructible<X,args>	Может ли **X** быть сконструирован из **args**, используя только тривиальные операции?
is_trivially_default_constructible<X>	
is_trivially_copy_constructible<X>	§8.2.6
is_trivially_move_constructible<X>	
is_trivially_assignable<X,Y>	
is_trivially_copy_assignable<X>	
is_trivially_move_assignable<X>	
is_trivially_destructible<X>	

К примеру, рассмотрим вопрос о том, как бы мы могли оптимизировать деструктор для контейнерного типа:

```
template<class T>
Cont::~Cont()   // деструктор для контейнера Cont
{
    if (!is_trivially_destructible<T>())
        for (T* p = elem; p!=p+sz; ++p)
            p->~T();
}
```

Предикаты свойств типа (продолжение) **(§ iso.20.9.4.3)**	
is_nothrow_constructible<X,args>	Может ли **X** быть сконструирован из **args**, используя только **noexcept**-операции?
is_nothrow_default_constructible<X>	
is_nothrow_copy_constructible<X>	
is_nothrow_move_constructible<X>	
is_nothrow_assignable<X,Y>	
is_nothrow_copy_assignable<X>	
is_nothrow_move_assignable<X>	
is_nothrow_destructible<X>	
has_virtual_destructor<X>	Имеет ли **X** виртуальный деструктор?

Как и **sizeof(T)**, запрос свойства вовзращает числовое значение, имеющее отношение к аргументу типа:

Запросы свойств типов (§ iso.20.9.5)	
n=alignment_of<X>	**n=alignof(X)**
n=rank<X>	Если **X** – массив, **n** есть число измерений; иначе, **n==0**
n=extent<X,N>	Если **X** – массив, **n** есть число эл-ов в **N**-ом измерении; иначе, **n==0**
n=extent<X>	**n=extent<X,0>**

Например:

```
template<typename T>
void f(T a)
{
    static_assert(Is_array<T>(), "f(): not an array");
    constexpr int dn {Extent<a,2>()}; // число элементов по второму измерению
    // ..
}
```

Здесь я снова использовал **constexpr**-версии функций типа, возвращающих числовые значения (§28.2.2).

Отношения типов являются предикатами над двумя типами:

Отношения типов (§ iso.20.9.6)	
is_same<X,Y>	**X** того же типа, что и **Y**?
is_base_of<X,Y>	Является ли **X** базовым по отношению к **Y**?
is_convertible<X,Y>	Может ли **X** быть неявно преобразованным в **Y**?

Пример:

```
template<typename T>
void draw(T t)
{
    static_assert(Is_same<Shape*,T>() || Is_base_of<Shape,Remove_pointer<T>>(), "");
    t->draw();
}
```

35.4.2. Генераторы типа

В файле **<type_traits>** стандартная библиотека предоставляет функции типа для порождения типа, отталкиваясь от других типов как аргументов.

Модификация const и volatile (§ iso.20.9.7.1)	
remove_const<X>	Подобно **X**, но любой **const** верхнего уровня убран
remove_volatile<X>	Подобно **X**, но любой **volatile** верхнего уровня убран
remove_cv<X>	Подобно **X**, но любой **const** или **volatile** верхнего уровня убран
add_const<X>	Если **X** – ссылка, функция или **const**, тогда **X**; иначе **const X**
add_volatile<X>	Если **X** – ссылка, функция или **volatile**, тогда **X**; иначе **volatile X struct**
struct add_cv<X>	Добавить **const** и **volatile**: **add_const<typename add_volatile<T>::type>::type**

Преобразователи типа вовзращают тип. Для доступа к этому типу используйте суффикс **::type**. Пример:

```
template<typename K, typename V>
class My_map {
{
    pair<typename add_const<K>::type,V> default_node;
    // ...
};
```

Если вам надоест использовать **::type**, то определите синоним типа (§28.2.1):

```
template<typename T>
using Add_const = typename add_const<T>::type;

template<typename K, typename V>
class My_map {
{
    pair<Add_const<K>,V> default_node;
    // ...
};
```

Можно применить вспомогательную библиотеку, предоставляющую такие синонимы на систематической основе для преобразователей типов стандартной библиотеки.

Модификация ссылок (§ iso.20.9.7.2, § iso.20.9.7.6)	
remove_reference<X>	Если **X** – ссылочный тип, тип, на который ссылаются; иначе, **X**
add_lvalue_reference<X>	Если **X** есть rvalue-ссылка **Y&&**, **Y&**; иначе, **X&**
add_rvalue_reference<X>	Если **X** есть ссылка, **X**; иначе, **X&&** (§7.7.3)
decay<X>	Тип, передаваемый по значению для функуионального аргумента типа **X**

Функции типа для добавления и устранения ссылок важны при написании шаблонов, которые должны работать с аргументом, который может быть ссылкой, а может и не быть ею. Например:

```
template<typename T>
void f(T v)
{
    Remove_reference<T> x = v;      // копия v
    T y = v;                        // может копия v, а может и ссылка на x
    ++x;                            // инкрементируем локальную переменную
    ++y;
    // ...
}
```

Здесь **x** является копией **v**, но если **T** является ссылочным типом, то **y** будет ссылкой на **v**:

```
void user()
{
    int val = 7;
    f(val);     // вызов f<int&>(): ++y в f() инкрементирует val
    f(7);       // вызов f<int>(): ++y в f инкрементирует локальную копию
}
```

В обоих вызовах **++x** будет инкрементировать локальную копию.

Модификация знака (§ iso.20.9.7.3)	
make_signed<X>	Убрать любой (явный или неявный) модификатор **unsigned** и добавить **signed**; **X** должен быть интегральным типом (кроме **bool** или перечисления)
make_unsigned<X>	Убрать любой (явный или неявный) модификатор **signed** и добавить **unsigned**; **X** должен быть интегральным типом (кроме **bool** или перечисления)

Для встроенных массивов мы иногда хотим получить тип элемента или устранить измерение:

Модификация массива (§ iso.20.9.7.4)	
remove_extent<X>	Если **X** – массив, тип элемента; иначе **X**
remove_all_extents<X>	Если **X** – массив, базовый тип (после удаления всех модификаторов массива); иначе **X**

Например:

```
int a[10][20];
Remove_extent<decltype(a)> a10;        // an array[10]
Remove_all_extents<decltype(a)> i;     // an int
```

Можно создать указательный тип, указывающий на произвольный тип, или найти тип, на который есть указывание:

Модификация указателей (§ iso.20.9.7.5)	
remove_pointer<X>	Если **X** – указатель, указываемый тип; иначе **X**
add_pointer<X>	**remove_reference<X>::type***

Например:

```
template<typename T>
void f(T x)
{
    Add_pointer<T> p = new Remove_reference<T>{};
    T* p = new T{};          // не будет работать, если T ссылка
    // ...
}
```

Имея дело с памятью на самом нижнем уровне системы, приходится иногда учитывать выравнивание (§6.2.9):

Выравнивание (§ iso.20.9.7.6)	
aligned_storage<n,a>	Тип POD размером как минимум **n** и на чье выравнивание делится **a**
aligned_storage<n>	**aligned_storage<n,def>** где **def** есть наибольшее выравнивание, требующееся для какого-либо объектного типа **T** с **sizeof(T)<=n**
aligned_union<n,X...>	Тип POD размером как минимум **n**, который может содержать **union** с членами типа **X**

Вот возможная реализация **aligned_storage**:

```
template<std::size_t N, std::size_t A>
struct aligned_storage {
  using type = struct { alignas(A) unsigned char data[N]; }; // N chars выровненные под A (§6.2.9)
};
```

Вот еще функции типа для выбора типа, для вычисления общих типов и т. д. — здесь отобраны предположительно наиболее полезные:

Прочие преобразования (§ iso.20.9.7.6)	
enable_if<b,X>	**X**, если **b==true**; иначе, нет члена **::type**, и в большинстве случаев возникнет ошибка подстановки (§23.5.3.2)
enable_if<b>	**enable_if<b,void>**
conditional<b,T,F>	**T**, если **b==true**; иначе **F**
common_type<X>	Общий тип для набора параметров **X**; два типа являются общими, если могут использ-ся в качестве типов ветвей **?:**-выражения
underlying_type<X>	Нижележащий тип **X** (§8.4); **X** должен быть перечислением
result_of<FX>	Тип результата **F(X)**; **FX** должен быть типом **F(X)**, где **F** вызывается со списком аргументов **X**

Примеры с **enable_if** и **conditional** приведены в §28.3.1.1 и §28.4.

Часто полезно найти тип, который может использоваться для операций над более чем одним типом, вроде результата сложения двух связанных друг с другом значений, но имеющих разный тип. Функция типа **common_type** находит такие общие типы. Очевидно, что тип служит общим типом для самого себя:

```
template<typename ...T>
struct common_type;

template<typename T>
struct common_type<T> {
    using type = T;
};
```

Общий тип двух типов дают нам правила для операции **?:** (§11.1.3):

```
template<typename T, typename U>
struct common_type<T, U> {
    using type = decltype(true ? declval<T>() : declval<U>());
};
```

Функция типа **declval<T>()** возвращает тип невычисляемой переменной типа **T**.

Общий тип для N типов находится рекурсивным применением правил для N==1 и N==2:

```
template<typename T, typename U, typename ... V>
struct common_type<T, U, V...> {
    using type = typename common_type<typename common_type<T, U>::type, V...>::type;
};
```

Например:

```
template<typename T, typename U>
using Common_type = typename common_type<T,U>::type;

Common_type<int,double> x1;                 // x1 есть double
Common_type<int,string> x2;                 // ошибка: нет общего типа
Common_type<int,short,long,long long> x3;   // x3 есть long
Common_type<Shape*,Circle*>x4;              // x4 есть Shape*
Common_type<void*,double*,Shape*> x5;       // x5 есть void*
```

Result_of используется для извлечения типа результата для вызываемого типа:

```
int ff(int) { return 2; }                    // функция

typedef bool (*PF)(int);                     // указатель на функцию

struct Fct {                                 // функциональный объект
     double operator()(string);
     string operator()(int,int);
};

auto fx = [](char ch) { return tolower(ch); };      // лямбда

Result_of<decltype(&ff)()> r1 = 7;           // r1 есть int
Result_of<PF(int)> r2 = true;                // r2 есть bool
Result_of<Fct(string)> r3 = 9.9;             // r3 есть double
Result_of<Fct(int,int)> r4 = "Hero";         // r4 есть string
Result_of<decltype(fx)(char)> r5 = 'a';      // r5 есть char
```

Отметим, что **Result_of** может различить две версии **Fct::operator()()**.

Достаточно удивительно, но то же самое не приложимо к глобальным функциям (не членам). Пример:

```
int f();                                     // функция
string f(int);
Result_of<decltype(&f)()> r1 = 7;            // ошибка: не разрешен указатель на функцию
```

К сожалению, мы не выполняем разрешение указателей на функции, но почему я использовал **Result_of** таким кружным способом, вместо того чтобы:

```
Result_of<ff> r1 = 7;                        // ошибка: нет спецификации аргументов,
                                             // и ff скорее функция чем тип
Result_of<ff()> r1 = 7;                      // ошибка: аргумент для Result_of должен быть типом
Result_of<decltype(f)()> r2 = 7;             // ошибка: decltype(f) тип функции,
                                             // а не указатель на тип функции
Result_of<decltype(f)*()> r3 = 7;            // OK: r3 есть int
```

Естественно, что **Result_of** чаще всего встречается в шаблонах, где нам нелегко найти ответ в тексте программы. Пример:

```
template<typename F, typename A>
auto temp(F f, A a) -> Result_of<F(A)>
{
     // ...
}

void f4()
{
     temp(ff,1);
     temp(fx,'a');
     temp(Fct(),"Ulysses");
}
```

Заметим, что функция **ff** конвертируется в указательную функцию в вызове, так что опора на указатели на функции в **Result_of** не так уж странна, как может показаться на первый взгляд.

declval() (§ iso.20.2.4)	
declval<T>()	Возвращает rvalue для T: **typename add_rvalue_reference<T>::type**; никогда не используйте возвращаемое знач-е **declval**

Функция типа **declval()** необычна для стандартной библиотеки, потому что она действительно функция (без необходимости пользователям делать обертку вокруг нее). Она возвращает значение, которое не должно использоваться. Намерением является использование **decltype<X>** в качестве типа там, где требуется тип для переменной типа **X**. Например:

```
template<typename T, size_t N>
void array<T,N> swap(array& y) noexcept(noexcept(swap(declval<T&>(), declval<T&>())))
{
    for (int i=0; i<a.size(); ++i)
        swap((*this)[i],a[i]);
}
```

Смотри также определение **common_type**.

35.5. Небольшие утилиты

Эти утилиты небольшие по размеру, но их применение важно. Их трудно сгруппировать с чем-либо еще.

35.5.1. move() и forward()

В файле **<utility>** находим некоторые из наиболее важных небольших функций:

Прочие преобразования (§ iso.20.9.7.6)	
x2=forward(x)	**x2** есть rvalue; **x** не может быть; noexcept
x2=move(x)	**x2** есть rvalue; noexcept
x2=move_if_noexcept(x)	Если **x** может быть перемещено, **x2=move(x)**; иначе **x2=x**; noexcept

Функция **move()** по существу есть просто приведение к rvalue:

```
template<typename T>
Remove_reference<T>&& move(T&& t) noexcept
{
    return static_cast<Remove_reference<T>&&>(t);
}
```

На мой вкус, **move()** стоило назвать **rvalue()**, поскольку она на самом деле ничего не перемещает. Вместо этого она производит rvalue для своего аргумента, так что объект, на который ссылаются, может быть перемещен.

Функция **move()** применяется для того, чтобы сообщить компилятору, что объект не будет больше использоваться в данном контексте, так что его значение может быть перемещено, а объект останется пустым. Этот простейший пример является реализацией функции **swap()** (§35.5.2).

Функция **forward()** производит rvalue только из rvalue:

```
template<typename T>
T&& forward(Remove_reference<T>& t) noexcept
{
    return static_cast<T&&>(t);
}

template<typename T>
T&& forward(Remove_reference<T>&& t) noexcept;
{
    static_assert(!Is_lvalue_reference<T>,"forward of lvalue");
    return static_cast<T&&>(t);
}
```

Пара **forward()** функций всегда доступна, и выбор между ними должен выполняться на основе правил разрешения перегрузки. В таком случае, любое lvalue относится к первому варианту, а любое rvalue — ко второму. Например:

```
int i = 7;
forward(i);      // вызов первой версии
forward(7);      // вызов второй версии
```

static_assert тут поставлен для тех «слишком умных» программистов, которые вызовут вторую версию с явным аргументом шаблона и lvalue.

Типичным примером использования **forward()** является корректное перенаправление аргумента из одной функции в другую (§23.5.2.1, §28.6.3). Хорошим примером является **make_shared<T>(x)** (§34.3.2) из стандартной библиотеки.

Используйте **move()**, если есть намерение «украсть представление» объекта с перемещающей операцией, и используйте **forward()** для перенаправления. Таким образом, **forward()** безопасна, в то время как **move()** отмечает **x** для уничтожения (деструкции), так что использовать **move(x)** нужно с осторожностью.

35.5.2. swap()

В файле **<utility>** стандартная библиотека предоставляет общий вариант **swap()** и специализацию для встроенных массивов:

Прочие преобразования (§ iso.20.2.2)	
swap(x,y)	Обменять значения **x** и **y**; **x** и **y** передаются, как не-**const** ссылки; noexcept, если операции копирования **x** и **y** noexcept
swap(a1n,a2n)	**a1n** and **a2n** передаются, как ссылки на массивы: **T(&)[N]**; noexcept, если операции копирования ***a1n** и ***a2n** noexcept

Достаточно очевидная реализация **swap()** может быть таковой:

```
template<typename T>
void swap(T& a, T& b) noexcept(Is_nothrow_move_constructible<T>()
                && Is_nothrow_move_assignable<T>())
{
    T tmp {move(a)};
    a = move(b);
    b = move(tmp);
}
```

Это означает, что **swap()** не может использоваться для обмена rvalue:

```
vector<int> v1 {1,2,3,4};
swap(v,vecor<int>{});          // ошибка: второй аргумент есть rvalue
v.clear();                     // очистка
```

35.5.3. Операции сравнения

В файле **<utility>** находятся операции сравнения для произвольных типов в рамках подпространства имен **rel_ops**:

Операции сравнения в std::rel_ops (§ iso.20.2.1)	
x!=y	**!(x==y)**
x>y	**y<x**
x<=y	**!(y<x)**
x>=y	**!(x<y)**

Это требует от программиста быть уверенным, что **x==y** и **x<y** работают. Например:

```
struct Val {
    double d;
    bool operator==(Val v) const { return v.d==d; }
};

void my_algo(vector<Val>& vv)
{
    using namespace std::rel_ops;

    for (int i=0; i<ww.size(); ++i)
        if (0>ww[i]) ww[i]=abs(ww[i]); // OK: > из rel_ops
}
```

Трудно использовать **rel_ops** без загрязнения пространства имен. В частности:

```
namespace Mine {
    struct Val {
        double d;
        bool operator==(Val v) const { return v.d==d; }
    };

    using namespace std::rel_ops;
}
```

Это может открыть нахождение общих шаблонов из **rel_ops** посредством зависящего от аргументов поиска (§14.2.4) и привести к их применению для типов, для которых это не является подходящим. Более безопасно применять **using**-директивы в локальных областях видимости.

35.5.4. Сравнение и хэширование type_info

В файле **<typeindex>** стандартная библиотека предоставляет поддержку сравнения и хэширования для типа **type_index**. **type_index** создается из **type_info** (§22.5) специально для того, чтобы допустить такое сравнение и хэширование.

Операции над type_index (§ iso.20.13)	
tip есть указатель на **type_info**, представленное **type_index**	
type_index	**ti {tinf}**; **ti** представляет **type_info tinf**; noexcept
ti==ti2	**ti** и **ti2** представляют один и тот же **type_info**: ***ti.tip==*ti2.tip)**; noexcept
ti!=ti2	**!(ti==ti2)**; noexcept
ti<ti2	**ti.tip->before(ti2.tip)**; noexcept
ti<=ti2	**!ti2.tip->before(ti.tip)**; noexcept
ti>ti2	**ti2.tip->before(ti.tip)**; noexcept
ti>=ti2	**!ti.tip->before(ti2.tip)**; noexcept
n=ti.hash_code()	**n=ti.tip->hash_code()**
p=name()	**p= ti.tip->name()**
hash<type_index>	Специализация **hash** (§31.4.3.4)

Например:

```
unordered_map<type_index,type_info*> types;
// ...
types[type_index{something}] = &typeid(something);
```

35.6. Советы

[1] Используйте средства из **<chrono>**, такие как **steady_clock**, **duration** и **time_point** для работы со временем; §35.2.

[2] Предпочитайте средства из **<clock>** средствам из **<ctime>**; §35.2.

[3] Для определения длительностей промежутков времени в известных единицах используйте **duration_cast**; §35.2.1.

[4] Для получения текущего времени используйте **system_clock::now()**; §35.2.3.

[5] Во время компиляции можно запрашить свойства типов; §35.4.1.

[6] Используйте **move(obj)** только тогда, когда значение **obj** не будет больше использоваться; §35.5.1.

[7] Используйте **forward()** для перенаправления; §35.5.1.

36

Строки

Предпочитайте стандарт излишней оригинальности.
— Странк & Уайт

- Введение
- Классификация символов
  Функции классификации; Характеристики символов
- Строки
  string против строк в С-стиле; Конструкторы; Важнейшие операции; Ввод/вывод; Числовые преобразования; Операции в стиле STL; Семейство функций **find**; Подстроки
- Советы

36.1. Введение

В стандартной библиотеке в файле **<cctype>** имеются средства классификации символов (§36.2), строки с ассоциированными операциями — в файле **<string>** (§36.3), регулярные выражения для поиска совпадений находятся в файле **<regex>** (Глава 37), и поддержка строк в С-стиле в файле **<cstring>** (§43.4). Работа с разными наборами символов, кодировками и соглашениями (местные особенности) обсуждаются в Главе 39.

Упрощенная реализация класса **string** представлена ранее в §19.3.

36.2. Классификация символов

В стандартной библиотеке имеются функции классификации для помощи пользователям в их манипуляции со строками (и другими последовательностями символов), а также предоставляются характеристики типов символов (в помощь реализаторам операций над строками).

36.2.1. Функции классификации

В файле **<cctype>** стандартная библиотека предоставляет функции для классификации символов из базового набора символов:

Классификация символов	
isspace(c)	Является ли **c** символом-разделителем (пробел ' ', гориз. табуляция **'\t'**, перевод строки **'\n'**, верт. табуляция **'\v'**, перевод страницы **'\f'**, возврат каретки **'\r'**)?
isalpha(c)	Является ли **c** буквой (**'a'..'z'**, **'A'..'Z'**)? внимание: не знаком подчеркивания '_'

Продолжение

Классификация символов	
isdigit(c)	Является ли **c** десятичной цифрой (**'0'..'9'**)?
isxdigit(c)	Является ли **c** шестнадцатерич. цифрой (десятичн. цифра или **'a'..'f'** или **'A'..'F'**)?
isupper(c)	Является ли **c** буквой в верх. регистре?
islower(c)	Является ли **c** буквой в ниж. регистре?
isalnum(c)	**isalpha(c)** или **isdigit(c)**
iscntrl(c)	Является ли **c** контрольным символом (ASCII 0..31 и 127)?
ispunct(c)	Не является ли **c** ничем из вышеперечисленного?
isprint(c)	Является ли с доступным для печати (ASCII ' '..'~')?
isgraph(c)	**isalpha(c)** и **isdigit(c)** и **ispunct(c)**? внимание: не пробелом

Кроме того, стандартная библиоенка предоставляет две полезные функции для устранения различения регистра символов:

Верхний и нижний регистры	
toupper(c)	**c** или аналог **c** в верх. регистре
tolower(c)	**c** или аналог **c** в ниж. регистре

Эквивалентные функции для «широких» символов располагаются в файле **<cwtype>**.

Функции классификации символов чувствительны к тегу локализации **"C"** (§39.5.1, §39.5.2), а функции для других типов локализации расположены в файле **<locale>** (§39.5.1).

Одна из причин, по которой эти функции классификации полезны, более замысловата и неочевидна, чем это может показаться на первый взгляд. Например, новичок может написать следующее:

```
if ('a'<ch && ch<'z')      // символ
```

Это избыточно и наверняка не столь эффективно, как следующее:

```
if (islower(ch))      // символ нижнего регистра
```

К тому же, нет никакой гарантии, что символы идут смежным образом в пространстве кодировки. Более того, использование стандартной классификации символов намного проще применять для конвертирования в иной тип локализации:

```
if (islower,danish)      // символ нижнего регистра Датского языка
                         // ("danish" имя для Датской схемы локализации)
```

Заметьте, что для датского языка имеется на три символа нижнего регистра больше, чем в английском языке, так что первоначальный тест с использованием **'a'** и **'z'** будет неправильным.

36.2.2. Характеристики символов

Как показано в §23.2, строковый шаблон может в принципе использовать любой тип с надлежащими копирующими операциями в качестве своего типа символов. Однако, эффективность можно улучшить и реализацию упростить для типов, у которых нет пользовательских копирующих операций. Следовательно, стандартный тип **string** требует, чтобы его символьный тип был типом POD (§8.2.6). Это также помогает вводу/выводу строк быть простым и эффективным.

Свойства символьных типов определены в их **char_traits** — это специализация шаблона:

```
template<typename C> struct char_traits { };
```

Все характеристики символов определены в пространстве имен **std**, а стандартные представлены в файле **<string>**. Самые общие **char_traits** сами по себе не имеют характеристик; только специализации **char_traits** для конкретных типов символов имеют таковые. Рассмотрим **char_traits<char>**:

```
template<>
struct char_traits<char> {                  // операции char_traits не генерируют исключений
    using char_type = char;
    using int_type = int;                   // тип целого значения для символа
    using off_type = streamoff;             // смещение в потоке
    using pos_type = streampos;             // позиция в потоке
    using state_type = mbstate_t;           // состояние мультибайтового потока (§39.4.6)
    // ...
};
```

Стандарт определяет четыре специализации для **char_traits** (§ iso.21.2.3):

```
template<> struct char_traits<char>;
template<> struct char_traits<char16_t>;
template<> struct char_traits<char32_t>;
template<> struct char_traits<wchar_t>;
```

Члены стандартных **char_traits** являются статическими функциями:

Статические члены char_traits<C> (§ iso.21.2)	
c=to_char_type(i)	Преобразование **int_type** в **char_type**
i=to_int_type(c)	Преобразование **char_type** в **int_type**
eq_int_type(c,c2)	**to_int_type(c)==to_int_type(c2)**
eq(c,c2)	Трактуется ли **c**, как равное **c2**?
lt(c,c2)	Трактуется ли **c**, как меньшее, чем **c2**?
i=compare(p,p2,n)	Лексикографическое сравнение [**p**:**p+n**) и [**p2**:**p2+n**)
assign(c,c2)	**c=c2** для **char_type**
p2=assign(p,n,c)	Присвоить **n** копий **c** в [**p**:**p+n**); **p2=p**
p3=move(p,p2,n)	Копировать [**p**:**p+n**) в [**p2**:**p2+n**); [**p**:**p+n**) и [**p2**:**p2+n**) могут перекрываться; **p3=p**
p3=copy(p,p2,n)	Копировать [**p**:**p+n**) в [**p2**:**p2+n**); [**p**:**p+n**) и [**p2**:**p2+n**) не могут перекрываться; **p3=p**
n=length(p)	**n** есть число символов в [**p**:**q**), где ***q** – первый эл-т такой, что **eq(q,charT{})**
p2=find(p,n,c)	**p** указывает на первый символ **c** в [**p**:**p+n**) или **nullptr**
i=eof()	**i** есть значение **int_type**, обозначающее конец файла
i=not_eof(i)	**i**, если **!eq_int_type(i,eof())**; иначе, **i** может иметь любое значение, кроме **eof()**

Сравнение функцией **eq()** часто не сводится просто к операции **==**. Например, чувствительные к регистру **char_traits** определяют свой вариант **eq()** так, что **eq('b','B')** будет возвращать **true**.

Поскольку **copy()** не гарантирует защиты в случае перекрывающихся диапазонов, она может быть быстрее, чем **move()**.

Функция сравнения **compare()** использует **lt()** и **eq()** для сравнения символов. Она возвращает **int**, где **0** означает точное совпадение, отрицательное число означает, что первый аргумент лексикографически идет впереди второго, а позитивное число означает что первый аргумент лексикографически идет после второго аргумента.

Функции ввода/вывода используются при реализации низкоуровневого ввода/вывода (§38.6).

36.3. Строки

В файле **<string>** стандартная библиотека определяет общий строковый шаблон **basic_string**:

```
template<typename C,
    typename Tr = char_traits<C>,
    typename A = allocator<C>>
class basic_string {
public:
    using traits_type = Tr;
    using value_type = typename Tr::char_type;
    using allocator_type = A;
    using size_type = typename allocator_traits<A>::size_type;
    using difference_type = typename allocator_traits<A>::difference_type;
    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = typename allocator_traits<A>::pointer;
    using const_pointer = typename allocator_traits<A>::const_pointer;
    using iterator = /* зависит от реализации */;
    using const_iterator = /* зависит от реализации */;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    static const size_type npos = -1;  // целое обозначает конец-строки

    // ...
};
```

Элементы (символы) хранятся смежным образом, так что низкоуровневые операции ввода могут безопасно использовать последовательность символов **basic_string** в качестве источника или цели.

basic_string предоставляет сильную гарантию (§13.2): если операция из **basic_string** генерирует исключение, то строка остается нетронутой.

Имеются специализации для нескольких стандартных символьных типов:

```
using string = basic_string<char>;
using u16string = basic_string<char16_t>;
using u32string = basic_string<char32_t>;
using wstring = basic_string<wchar_t>;
```

Аналогично случаю с контейнерами (Глава 31), **basic_string** не предполагается использовать в качестве базового класса, и он предоставляет перемещающую семантику, так что может быть эффективно реализован возврат по значению.

36.3.1. string против строк в С-стиле

В книге я привожу примеры тех моментов, которые отличают использование **string** от использования строк в С-стиле (§43.4), популярного у программистов на языке С и раннем С++.

Рассмотрим пример составления адреса электронной почты путем конкатенации идентификатора пользователя и доменного имени:

```
string address(const string& identifier, const string& domain)
{
    return identifier + '@' + domain;
}
```

```
void test()
{
    string t = address("bs","somewhere");
    cout << t << '\n';
}
```

Это тривиально. А теперь рассмотрим некий вариант в С-стиле. Там строка — это указатель на массив символов с терминирующим нулем. Пользователь управляет выделением памяти и ответственен за ее возвращение (освобождение):

```
char* address(const char* identifier, const char* domain)
{
    int iden_len = strlen(identifier);
    int dom_len = strlen(domain);
    char* addr = (char*)malloc(iden_len+dom_len+2);    // не забудьте про место для 0 и '@'
    strcpy(identifier,addr);
    addr[iden_len] = '@';
    strcpy(domain,addr+iden_len+1);
    return addr;
}

void test2()
{
    char* t = address("bs","somewhere");
    printf("%s\n",t);
    free(t);
}
```

Я все тут сделал правильно? Надеюсь, что да. По крайней мере, я получил ожидаемый вывод. Как и большинство опытных программистов на С, я добился корректной версии, которую часто предъявляют новичкам, не знающим всех подводных камней. Реализация функции **address()** в С-стиле содержит некоторое количество трюков с манипуляцией указателями, а ее использование требует от того, кто вызывает эту функцию, не забывать освобождать возвращаемую память. Так какой вариант предпочтете вы?

Иногда приходится слышать, что С-строки эффективнее. Однако ж, в большинстве случаев **string** выполняет меньшее количество выделений динамической памяти и парных к ним освобождений ее, чем это делает С-эквивалент (из-за оптимизации малых блоков памяти и перемещающей семантики; §19.3.3, §19.3.1). Кроме того, **strlen()** имеет оценку log(N), в то время как **string::size()** — это простое чтение. С-код читает каждую входную строку дважды, а **string**-версия делает это (траверс по входной строке) единожды. Так что вопрос о превосходстве С-версии в эффективности не так уж очевиден.

Принципиальная разница между С-версией и версией с классом **string** состоит в том, что второй вариант опирается на «твердый» тип со стандартной семантикой, в то время как С-версия — это набор некоторых соглашений, поддержанный набором полезных функций. Рассмотрим вопрос о присваиваниях и сравнении:

```
void test3()
{
    string s1 = "Ring";
    if (s1!="Ring") insanity();
    if (s1<"Opera")cout << "check";
    string s2 = address(s1,"Valkyrie");

    char s3[] = "Ring";
    if (strcmp(s3,"Ring")!=0) insanity();
    if (strcmp(s3,"Opera")<0) cout << "check";
    char* s4 = address(s3,"Valkyrie");
    free(s4);
}
```

Наконец, рассмотрим сортировку:

```
void test4()
{
vector<string> vs = {"Grieg", "Williams", "Bach", "Handel" };
sort(vs.begin(),vs.end());        // полагаем, что я не определил sort(vs)

const char* as[] = {"Grieg", "Williams", "Bach", "Handel" };
qsort(as,sizeof(*as),sizeof(as)/sizeof(*as),(int(*)(const void*,const void*))strcmp);
}
```

Функция сортировки **qsort()** в С-стиле представлена в §43.7. Функция **sort()** столь же быстра, как и функция **qsort()**, так что нет причины из-за мнимой потери эффективности выбирать низкоуровневый и чреватый ошибками стиль программирования.

36.3.2. Конструкторы

Тип **basic_string** предлагает большое число конструкторов:

Конструкторы basic_string<C,Tr,A> (§iso.21.4.2) **x** может быть **basic_string**, строкой в С-стиле или **initializer_list<char_type>**	
basic_string	**s** {**a**}; **s** – пустая строка с аллокатором **a**; explicit
basic_string s {};	Конструктор по умолчанию: **basic_string s** {**A**{}};
basic_string s {**x**,**a**};	**s** получает символы из **x**; использует аллокатор **a**
basic_string s {**x**};	Перемещ-й и копирующий конструкторы: **basic_string s** {**x**,**A**{}};
basic_string s {**s2**,**pos**,**n**,**a**};	**s** получает символы **s2**[**pos**:**pos+n**); использует аллокатор **a**
basic_string s {**s2**,**pos**,**n**};	**basic_string s** {**s2**,**pos**,**n**,**A**{};}
basic_string s {**s2**,**pos**};	**basic_string s** {**s2**,**pos**,**string::npos**,**A**{}};
basic_string s {**p**,**n**,**a**};	Инициализировать **s** из [**p**:**p+n**); **p** есть строка в С-стиле; использ. аллокатор **a**
basic_string s {**p**,**n**};	**basic_string s** {**p**,**n**,**A**{}};
basic_string s {**n**,**c**,**a**};	**s** содержит **n** копий символа **c**; использует аллокатор **a**
basic_string s {**n**,**c**};	**basic_string s** {**n**,**c**,**A**{};}
basic_string s {**b**,**e**,**a**};	**s** получает символы из [**b**:**e**); использует аллокатор **a**
basic_string s {**b**,**e**};	**basic_string s** {**b**,**e**,**A**{}};
s.~basic_string()	Деструктор: освобождает все ресурсы
s=x	Копирование: **s** получает символы из **x**
s2=move(s)	Перемещение: **s2** получает символы из **s**; noexcept

Самые часто применяемые варианты — они же самые простые:

```
string s0;                          // пустая строка
string s1 {"As simple as that!"};   // конструируем из строки в С-стиле
string s2 {s1};                     // копирующий конструктор
```

На практике почти всегда деструктор активируется неявным образом.

Для **string** нет конструктора, который принимает лишь число элементов:

```
string s3 {7};      // ошибка: нет string(int)
string s4 {'a'};    // ошибка: нет string(char)
string s5 {7,'a'};  // OK: 7 'a's
string s6 {0};      // опасно: передаем nullptr
```

Объявление **s6** показывает ошибку, иногда совершаемую программистами, использующими С-строки:

```
const char* p = 0; // устанавливаем p в "no string"
```

К сожалению, компилятор не может отловить определение **s6** или даже (что еще хуже) случай с **const char***, содержащие **nullptr**:

```
string s6 {0};        // опасно: passing nullptr
string s7 {p};        // OK или нет в зависимоти от значения p
string s8 {"OK"};     // OK: передача указателя на C-строку
```

Не пытайтесь инициализировать **string** чререз **nullptr**. В лучшем случае, вы получите ужасную ошибку во время выполнения программы. А в худшем — таинственное неопределенное поведение.

Если вы попытаетесь конструировать **string** большим числом символов, чем это позволяет ваша реализация, то конструктуор сгенерирует исключение **std::length_error**. Пример:

```
string s9 {string::npos,'x'};     // генерируем length_error
```

Значение **string::npos** представляет собой позицию вне области памяти в соответствии с длиной строки и обычно используется в смысле «конец **string**». Пример:

```
string ss {"Fleetwood Mac"};
string ss2 {ss,0,9};                  // "Fleetwood"
string ss3 {ss,10,string::npos};      // "Mac"
```

Заметьте, что обозначение для подстроки есть (position,length), а не [start,end).

Для типа **string** литералов не существует. С такой целью может применяться пользовательский литерал (§19.2.6), например, **"The Beatles's** и **"Elgar's**. Не забывайте про суффикс **s**.

36.3.3. Важнейшие операции

Тип **basic_string** предлагает сравнения, управление размером и емкостью, а также операции доступа:

Сравнение basic_string<C,Tr,A> (§iso.21.4.8) **s** или **s2** но не оба могут быть строкой в С-стиле	
s==s2	**s** равна **s2**? сравнивает знач-я, используя **traits_type**
s!=s2	**!(s==s2)**
s<s2	**s** лексикографически раньше **s2**?
s<=s2	**s** лексикографически раньше или равно **s2**?
s>s2	**s** лексикографически позже **s2**?
s>=s2	**s** лексикографически позже или равно **s2**?

По поводу иных операций сравнения см. §36.3.8.

Механизм размера и емкости для **basic_string** тот же самый, что и для **vector** (§31.3.3):

Размер и емкость basic_string<C,Tr,A> (§iso.21.4.4)	
n=s.size()	**n** есть число символов в **s**
n=s.length()	**n=s.size()**
n=s.max_size()	**n** – наибольшее возможное значение **s.size()**
s.resize(n,c)	Сделать **s.size()==n**; добавленные эл-ы получают знач-е **c**
s.resize(n)	**s.resize(n,C{})**
s.reserve(n)	Гарантировать, что **s** может содержать **n** симв. без дальнейшей аллокации
s.reserve()	Никакого эффекта: **s.reserve(0)**
n=s.capacity()	**s** может содержать **n** симв. без дальнейшей аллокации
s.shrink_to_fit()	Сделать **s.capacity==s.size()**
s.clear()	Сделать **s** пустой
s.empty()	Пуста ли **s**?
a=s.get_allocator()	**a** есть аллокатор **s**

Функции **resize()** и **reserve()**, которые могут привести к выходу за значение **max_size()**, будут генерировать исключение **std::length_error**.

Пример:

```
void fill(istream& in, string& s, int max)
    // используем s для низкоуровнего ввода
{
    s.reserve(max);                 // достаточно ли выделено памяти?
    in.read(&s[0],max);
    const int n = in.gcount();      // число прочитанных символов
    s.resize(n);
    s.shrink_to_fit();              // устраните избыточную емкость
}
```

Здесь я «забыл» использовать число прочитанных символов. Это неряшливо.

Доступ к basic_string<C,Tr,A> (§iso.21.4.5)	
s[i]	Индексирование: **s[i]** – ссылка на **i**-й эл-т **s**; нет проверки диапазона
s.at(i)	Индексирование: **s.at(i)** – ссылка на **i**-й эл-т **s**; генерирует **range_error**, если **s.size()<=i**
s.front()	**s[0]**
s.back()	**s[s.size()-1]**
s.push_back(c)	Добавить символ **c**
s.pop_back()	Удалить последний символ из **s**: **s.erase(s.size()-1)**
s+=x	Добавить **x** в конец **s**; **x** может быть символом, **string**, строкой в С-стиле или **initializer_list<char_type>**
s=s1+s2	Конкатенация: оптимизированная версия **s=s1**; **s+=s2**;
n2=s.copy(s2,n,pos)	**s** получает символы из **s2[pos:n2)**, где **n2** есть **min(n,s.size()-pos)**; генерирует **out_of_range**, если **s.size()<pos**
n2=s.copy(s2,n)	**s** получает все символы из **s2**; **n=s.copy(s2,n,0)**
p=s.c_str()	**p** версия в виде С-строки (с завершающим нулем) символов в **s**; типа const **C***
p=s.data()	**p=s.c_str()**
s.swap(s2)	Обменять значения **s** и **s2**; noexcept
swap(s,s2)	**s.swap(s2)**

При использовании **at()** выход за корректный диапазон приводит к генерации исключения **std::out_of_range**. А в таком же случае для операций **+=()**, **push_back()** и **+** будет генерироваться **std::length_error**.

Не существует неявного приведения от **string** к **char***. Вместо этого стандартная библиотека предоставляет возможность явного приведения с помощью функции **c_str()** к типу **const char***.

Тип **string** допускает содержание нулевого символа (то есть, **'\0'**). При использовании функций, таких как **strcmp()**, что предполагает конвертирование в С-строку результата работы функции **s.c_str()** или **s.data()**, наличие нулевого символа может преподнести сюрприз.

36.3.4. Ввод/вывод

Для **basic_string** вывод (запись) может выполняться операцией **<<** (§38.4.2), а ввод (чтение) — операцией **>>** (§38.4.1):

Ввод/вывод basic_string<C,Tr,A> (§iso.21.4.8.9)	
in>>s	Прочитать отделенное пробельными символами слово в s из in
out<<s	Вывести s в out
getline(in,s,d)	Читать символы из in в s, пока не встретится символ d; d удаляется из in, но не добавляется к s
getline(in,s)	getline(in,s,'\n'), где '\n' расширяется, чтобы соответствовать типу символа string

Операция ввода, при которой будет превзойден **max_size()**, сгенерирует исключение **std::length_error**.

Функция **getline()** устраняет терминирующий символ (по умолчанию это '\n') из потока ввода, но не внедряет его в строку. Это упрощает работу со строками. Пример:

```
vector<string> lines;
for (string s; getline(cin,s);)
    lines.push_back(s);
```

Все операции ввода/вывода возвращают ссылку на поток, так что операции могут объединяться в цепочку. Например:

```
string first_name;
string second_name;
cin >> first_name >> second_name;
```

Перед чтением целевая строка опустошается и расширяется так, чтобы можно было вместить читаемые символы. Операция ввода может завершаться достижением символа «конец файла» (§38.3).

36.3.5. Числовые преобразования

В файле **<string>** стандартная библиотека предоставляет набор функций для извлечения числовых значений из их символьных представлений внутри **string** или **wstring** (заметьте, не из **basic_string<C,Tr,A>**). Желательные числовые типы кодируются в именах функций:

Числовые преобразования (§iso.21.5) s может быть string или wstring	
x=stoi(s,p,b)	Строка в int; x есть целое; начинает с s[0] если p!=nullptr, в *p записывается число символов, использованное под x; b есть основание числа (между 2 и 36 включительно)
x=stoi(s,p)	x=stoi(s,p,10); десятичные числа
x=stoi(s)	x=stoi(s,nullptr,10); десятичные числа; не сообщает число символов
x=stol(s,p,b)	Строка в long
x=stoul(s,p,b)	Строка в unsigned long
x=stoll(s,p,b)	Строка в long long
x=stoull(s,p,b)	Строка в unsigned long long
x=stof(s,p)	Строка в float
x=stod(s,p)	Строка в double
x=stold(s,p)	Строка в long double
s=to_string(x)	s есть представление x в виде string; x должен быть целым или вещественным значением
ws=to_wstring(x)	s есть представление x в виде wstring; x должен быть целым или вещественным значением

Каждая из этих **sto*** функций имеет три варианта, вроде **stoi**. Пример:

```
string s = "123.45";
auto x1 = stoi(s);          // x1 = 123
auto x2 = stod(s);          // x2 = 123.45
```

Второй аргумент у **sto*** функций является указателем, который используется для индикации позиции, до которой продвинулся поиск в строке числового значения. Например:

```
string ss = "123.4567801234";
size_t dist = 0;                // сюда помещаем число прочитанных символов
auto x = stoi(ss,&dist);        // x = 123 (an int)
++dist;                         // игнорируем точку
auto y = stoll(&ss[dist]);      // x = 4567801234 (a long long)
```

Это не мой любимый способ извлечения нескольких чисел из строки. Я предпочитаю использовать **string_stream** (§38.2.2).

Начальный пробельный символ пропускается. Пример:

```
string s = " 123.45";
auto x1 = stoi(s);          // x1 = 123
```

Базовый аргумент может быть значением из диапазона **[2:36]** с «цифрами» из набора **0123456789abcdefghhijklmnopqrstuvwxyz**, значения которых определяются по их позиции в последовательности. Например:

```
string s4 = "149F";
auto x5 = stoi(s4);             // x5 = 149
auto x6 = stoi(s4,nullptr,10);  // x6 = 149
auto x7 = stoi(s4,nullptr,8);   // x7 = 014
auto x8 = stoi(s4,nullptr,16);  // x8 = 0x149F

string s5 = "1100101010100101"; // бинарное
auto x9 = stoi(s5,nullptr,2);   // x9 = 0xcaa5
```

Если конвертирующая функция не обнаруживает в строке чисел, то она генерирует исключение **invalid_argument**. Если она находит число, которое не может отобразить в своем целевом типе, она генерирует исключение **out_of_range**; дополнительно при конверсии в типы с плавающей запятой она устанавливает **errno** в значение **ERANGE** (§40.3). Например:

```
stoi("Hello, World!");          // генерирует std::invalid_argument
stoi("12345678901234567890");   // генерирует std::out_of_range; errno=ERANGE
stof("123456789e1000");         // генерирует std::out_of_range; errno=ERANGE
```

Функции **sto*** кодируют целевый тип в своем имени. Это делает их неудобными для обобщенного кода, где цель может быть параметром шаблона. В таких случаях рассмотрите **to<X>** (§25.2.5.1).

36.3.6. Операции в стиле STL

Тип **basic_string** предлагает обычный набор итераторов:

Строковые итераторы basic_string<C,Tr,A> (§iso.21.4.3) Все операции noexcept	
p=s.begin()	**p** есть **iterator** на первый символ **s**
p=s.end()	**p** есть **iterator** на следующий за последним символ **s**
p=s.cbegin()	**p** есть **const_iterator** на первый символ

Продолжение

Строковые итераторы basic_string<C,Tr,A> (§iso.21.4.3) Все операции noexcept	
p=s.cend()	p есть const_iterator на следующий за последним символ s
p=s.rbegin()	p есть начало обратной последовательности s
p=s.rend()	p есть конец обратной последовательности s
p=s.crbegin()	p есть const_iterator на начало обратной последовательности s
p=s.crend()	p есть const_iterator на конец обратной последовательности s

Поскольку **string** имеет необходимые члены-типы и функции для получения итераторов, то его можно использовать вместе со стандартными алгоритмами (Глава 32). Например:

```
void f(string& s)
{
    auto p = find_if(s.begin(),s.end(),islower);
    // ...
}
```

Самые общие операции над строками предоставляются непосредственно типом **string**. Предположительно эти операции могут оптимизироваться сильнее, чем обобщенные алгоритмы.

Стандартные алгоритмы (Глава 32) действительно не столь полезны, как кто-то мог бы подумать. Дело в том, что обобщенные алгоритмы имеют тенденцию считать, что отдельные элементы имеют самостоятельное значение в изоляции друг от друга. Для строк это не так.

Тип **basic_string** предлагает сложные присваиваания:

Присваивание basic_string<C,Tr,A> (§iso.21.4.6.3) Все операции возвращают строку, к которой применяются	
s.assign(x)	s=x; x может быть string, строкой в С-стиле или initializer_list<char_type>
s.assign(move(s2))	Перемещение: s2 есть string; noexcept
s.assign(s2,pos,n)	s получает символы s2[pos:pos+n)
s.assign(p,n)	s получает символы [p:p+n); p есть строка в С-стиле
s.assign(n,c)	s получает n копий символа c
s.assign(b,e)	s получает символы из [b:e)

Для типа **basic_string** имеются функции **insert()**, **append()** и **erase()**:

Вставка и удаление символов basic_string<C,Tr,A> (§iso.21.4.6.2, §iso.21.4.6.4, §iso.21.4.6.5) Все операции возвращают строку, к которой применяются	
s.append(x)	Добавить x в конец s; x может быть символом, string, строкой в С-стиле или initializer_list<char_type>
s.append(b,e)	Добавить [b:e) в конец s
s.append(s2,pos,n)	Добавить s2[pos:pos+n) в конец s
s.append(p,n)	Добавить символы [p:p+n) в конец s; p есть строка в С-стиле
s.append(n,c)	Добавить n копий символа c в конец s
s.insert(pos,x)	Вставить x перед s[pos]; x может быть символом, string, строкой в С-стиле или initializer_list<char_type>

Продолжение

Вставка и удаление символов basic_string<C,Tr,A> (§iso.21.4.6.2, §iso.21.4.6.4, §iso.21.4.6.5) Все операции возвращают строку, к которой применяются	
s.insert(p,c)	Вставить **c** перед итератором **p**
s.insert(p,n,c)	Вставить **n** копий **c** перед итератором **p**
insert(p,b,e)	Вставить [**b**:**e**) перед итератором **p**
s.erase(pos)	Удалить символы в конце **s**, начиная с **s[pos]**; **s.size()** становится **pos**
s.erase(pos,n)	Удалить **n** символов из **s**, начиная с **s[pos]**; **s.size()** становится **max(pos,s.size()-n)**

Например:

```
void add_middle(string& s, const string& middle)      // добавляем среднее имя
{
    auto p = s.find(' ');
    s.insert(p,' '+middle);
}

void test()
{
    string dmr = "Dennis Ritchie";
    add_middle(dmr,"MacAlistair");
    cout << dmr << '\n';
}
```

Как и в случае типа **vector**, функция **append()** более эффективна, чем **insert()**.

В последующем я использую обозначение **s[b:e)** для обозначения последовательности элементов [**b**:**e**) из **s**:

Замена символов basic_string<C,Tr,A> (§iso.21.4.6.6) Все операции возвращают строку, к которой применяются	
s.replace(pos,n,s2,pos2,n2)	Заменить **s[pos:pos+n)** на **s2[pos2:pos2+n2)**
s.replace(pos,n,p,n2)	Заменить **s[pos:pos+n)** на [**p**:**p+n2**); **s2** есть **string** или строка в С-стиле
s=s.replace(pos,n,s2)	Заменить **s[pos:pos+n)** на **s2**; **s2** есть **string** или строка в С-стиле
s.replace(pos,n,n2,c)	Заменить **s[pos:pos+n)** на **n2** копий символа **c**
s.replace(b,e,x)	Заменить [**b**:**e**) на **x**; **x** есть **string**, или строка в С-стиле или **initializer_list<char_type>**
s.replace(b,e,p,n)	Заменить [**b**:**e**) на [**p**:**p+n**)
s.replace(b,e,n,c)	Заменить [**b**:**e**) на **n** копий символа **c**
s.replace(b,e,b2,e2)	Заменить [**b**:**e**) на [**b2**:**e2**)

Функция **replace()** заменяет одну подстроку на другую и корректирует размер строки соответственно. Например:

```
void f()
{
    string s = "but I have heard it works even if you don't believe in it";
    s.replace(0,4,"");                  // удаляем начальную "but "
    s.replace(s.find("even"),4,"only");
    s.replace(s.find(" don't"),6,"");       // удаляем заменой на ""
    assert(s=="I have heard it works only if you believe in it");
}
```

Код, полагающийся на «магические» константы, вроде числа символов, чреват ошибками.

Функция **replace()** возвращает ссылку на объект, для которого она была вызвана. Это можно использовать для создания цепочки вызовов:

```
void f2()
{
    string s = "but I have heard it works even if you don't believe in it";
    s.replace(0,4,"").replace(s.find("even"),4,"only").replace(s.find(" don't"),6,"");
    assert(s=="I have heard it works only if you believe in it");
}
```

36.3.7. Семейство функций find

Имеется чрезвычайно большое число функций для поиска подстрок. Как обычно, **find()** ищет, начиная с **s.begin()**, в то время как **rfind()** выполняет поиск в обратном порядке, начиная с **s.end()**. Функции семейства **find** используют **string::npos** для индикации «не найдено».

Поиск элемента basic_string<C,Tr,A> (§iso.21.4.7.2) **x** может быть символом, **string** или строкой в С-стиле. Все операции noexcept	
pos=s.find(x)	Найти **x** в **s**; **pos** есть индекс первого найденного символа или **string::npos**
pos=s.find(x,pos2)	**pos=find(basic_string(s,pos2)**
pos=s.find(p,pos2,n)	**pos=s.find(basic_string{p,n},pos2)**
pos=s.rfind(x,pos2)	Найти **x** в **s[0:pos2)**; **pos** есть индекс первого найденного символа **x**, ближайшего к концу **s**, или **string::npos**
pos=s.rfind(x)	**pos=s.rfind(p,string::npos)**
pos=s.rfind(p,pos2,n)	**pos=s.rfind(basic_string{p,n},pos2)**

Например:

```
void f()
{
    string s {"accdcde"};

    auto i1 = s.find("cd");    // i1==2 s[2]=='c' && s[3]=='d'
    auto i2 = s.rfind("cd");   // i2==4 s[4]=='c' && s[5]=='d'
}
```

Функции **find_*_of()** функции отличаются от **find()** и **rfind()** поиском одиночных символов, а не целой последовательности символов:

Поиск элементов из набора в basic_string<C,Tr,A> (§iso.21.4.7.4) **x** может быть символом, **string**, или строкой в С-стиле; **p** есть строка в С-стиле. Все операции noexcept	
pos2=s.find_first_of(x,pos)	Найти символ из **x** в **s[pos:s.size())**; **pos2** есть позиция первого символа из **x** в **s[pos:s.size())** или **string::npos**
pos=s.find_first_of(x)	**pos=s.find_first_of(s2,0)**
pos2=s.find_first_of(p,pos,n)	**pos2=s.find_first_of(pos,basic_string{p,n})**
pos2=s.find_last_of(x,pos)	Найти символ из **x** в **s[0:pos)**; **pos2** есть позиция символа из **x**, ближайшего к концу **s**, или **string::npos**
pos=s.find_last_of(x)	**pos=s.find_first_of(s2,0)**
pos2=s.find_last_of(p,pos,n)	**pos2=s.find_last_of(pos2,basic_string{p,n})**

Продолжение

Поиск элементов из набора в basic_string<C,Tr,A> (§iso.21.4.7.4) x может быть символом, string, или строкой в С-стиле; p есть строка в С-стиле. Все операции noexcept	
pos2=s.find_first_not_of(x,pos)	Найти символ не из x в s[pos:s.size()); pos2 – позиция первого символа из x, не найденного в s[pos:s.size()) или string::npos
pos=s.find_first_not_of(x)	pos=s.find_first_not_of(s2,0)
pos2=s.find_first_not_of(p,pos,n)	pos2=s.find_first_not_of(pos,basic_string{p,n})
pos2=s.find_last_not_of(x,pos)	Найти символ не из x в s[0:pos); pos – позиция символа из x, ближайшего к концу s или string::npos
pos=s.find_last_not_of(x)	pos=s.find_first_not_of(s2,0)
pos2=s.find_last_not_of(p,pos,n)	pos=s.find_last_not_of(pos,basic_string{p,n})

Пример:

```
string s {"accdcde"};

auto i1 = s.find("cd");                 // i1==2 s[2=='c' && s[3]=='d'
auto i2 = s.rfind("cd");                // i2==4 s[4]=='c' && s[5]=='d'

auto i3 = s.find_first_of("cd");        // i3==1 s[1]=='c'
auto i4 = s.find_last_of("cd");         // i4==5 s[5]=='d'
auto i5 = s.find_first_not_of("cd");    // i5==0 s[0]!='c' && s[0]!='d'
auto i6 = s.find_last_not_of("cd");     // i6==6 s[6]!='c' && s[6]!='d'
```

36.3.8. Подстроки

Тип basic_string предлагает низкоуровневое понятие подстроки:

Подстроки basic_string<C,Tr,A> (§iso.21.4.7.8)	
s2=s.substr(pos,n)	s2=basic_string(&s[pos],m), где m=min(s.size()-n,n)
s2=s.substr(pos)	s2=s.substr(pos,string::npos)
s2=s.substr()	s2=s.substr(0,string::npos)

Заметьте, что substr() создает новую строку:

```
void user()
{
    string s = "Mary had a little lamb";
    string s2 = s.substr(0,4);      // s2 == "Mary"
    s2 = "Rose";                    // не меняет s
}
```

Можно сравнивать подстроки:

Сравнение basic_string<C,Tr,A> (§iso.21.4.7.9)	
n=s.compare(s2)	Лексикографическое сравнение s и s2; используя char_traits<C>::compare() для сравнения; n=0, если s==s2; n<0, если s<s2; n>0, если s2>s; noexcept;
n2=s.compare(pos,n,s2)	n2=basic_string{s,pos,n}.compare(s2)
n2=s.compare(pos,n,s2,pos2,n2)	n2=basic_string{s,pos,n}.compare(basic_string{s2,pos2,n2})
n=s.compare(p)	n=compare(basic_string{p}); p есть строка в С-стиле

Продолжение

Сравнение basic_string<C,Tr,A> (§iso.21.4.7.9)	
n2=s.compare(pos,n,p)	n2=basic_string{s,pos,n}.compare(basic_string{p}); p есть строка в С-стиле
n2=s.compare(pos,n,p,n2)	n2=basic_string{s,pos,n}.compare(basic_string{p,n2}); p есть строка в С-стиле

Например:

```
void f()
{
    string s = "Mary had a little lamb";
    string s2 = s.substr(0,4);      // s2 == "Mary"
    auto i1 = s.compare(s2);        // i1 положительно
    auto i2 = s.compare(0,4,s2);    // i2==0
}
```

Помните, что такое явное использование констант для маркировки позиций и длин чревато ошибками.

36.4. Советы

[1] Используйте классификацию символов, а не ручную проверку символов по диапазону; §36.2.1.

[2] При проектировании строковых абстракций используйте **character_traits** для реализации операций над символами; §36.2.2.

[3] Можно применять **basic_string** для создания строк с символами любого типа; §36.3.

[4] Используйте **string** в качестве типа переменных и членов, а не его базовые классы; §36.3.

[5] Отдавайте предпочтение **string**-операциям перед строковыми функциями в С-стиле; §36.3.1.

[6] Возвращайте **string** по значению (полагаясь на перемещающую семантику); §36.3.2.

[7] Используйте **string::npos** для индикации «остальная часть **string**»; §36.3.2.

[8] Не передавайте **nullptr string**-функциям, ожидающим строку в С-стиле; §36.3.2.

[9] Тип **string** выполняет рост и сжатие по необходимости; §36.3.3.

[10] Используйте **at()**, а не итераторы или **[]**, когда нужна проверка диапазона; §36.3.3, §36.3.6.

[11] Используйте итераторы или **[]** вместо **at()**, когда требуется оптимизация скорости; §36.3.3, §36.3.6.

[12] При использовании типа **string** перехватывайте где-либо исключения **length_error** и **out_of_range**; §36.3.3.

[13] Используйте функцию **c_str()** для создания С-строки из типа **string** лишь по необходимости; §36.3.3.

[14] Ввод для **string** чувствителен к регистру и не вызывает переполнения; §36.3.4.

[15] Предпочитайте **string_stream** или обобщенные функции извлечения значения (вроде **to<X>**) перед прямым использованием функций **str*** для числовой конверсии; §36.3.5.

[16] Используйте операции **find()** для локализации значений в **string** (не используйте явные циклы); §36.3.7.

[17] Прямо или косвенно, используйте **substr()** для чтения подстрок и **replace()** для записи подстрок; §36.3.8.

37

Регулярные выражения

Если код и комментарии не соответствуют друг другу,
то они скорее всего неверны.
— Норм Шрайер

- Регулярные выражения
  Нотация регулярных выражений
- **regex**
  Результаты поиска соответствия; Форматирование
- Функции регулярных выражений
  regex_match(); **regex_search()**; **regex_replace()**
- Итераторы регулярных выражений
  regex_iterator; **regex_token_iterator**
- **regex_traits**
- Советы

37.1. Регулярные выражения

В файле **<regex>** стандартная библиотека предоставляет регулярные выражения:

- **regex_match()**: Для сопоставления регулярного выражения строке (известного размера).
- **regex_search()**: Для поиска строки, соответствующей регулярному выражения в потоке (произвольной длины) данных.
- **regex_replace()**: Для поиска строк, соответствующих регулярному выражения в потоке данных произвольной длины и их замены.
- **regex_iterator**: итерирование по полным и частичным соответствиям.
- **regex_token_iterator**: итерирование по несоответствиям.

Результатом работы **regex_search()** служит коллекция соответствий, в типичном случае представленная как **smatch**:

```
void use()
{
    ifstream in("file.txt");      // входной файл
    if (!in) cerr << "no file\n";

    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"};    // U.S. шаблон почтового кода
```

```
    int lineno = 0;
    for (string line; getline(in,line);) {
        ++lineno;
        smatch matches;    // соответствующие друг другу строки идут сюда
        if (regex_search(line, matches, pat)) {
            cout << lineno << ": " << matches[0] << '\n';      // полное соответствие
            if (1<matches.size() && matches[1].matched)
                cout << "\t: " << matches[1] << '\n'; // частичное соответствие
        }
    }
}
```

Эта функция читает файл, выполняя поиск почтовых кодов США, таких как **TX77845** и **DC 20500-0001**. Тип **smatch** — это контейнер результатов поиска по регулярным выражениям. Здесь **matches[0]** соответствует полному совпадению, **matches[1]** — это опциональное совпадение по четырем цифрам. Я использовал «сырую строку» (§7.3.2.1), которая особенно удобна для регулярных выражений, поскольку последние имеют тенденцию содержать большое количество обратных наклонных черт. В случае традиционных строк определение образца для поиска выглядело бы так:

```
regex pat {"\\w{2}\\s*\\d{5}(-\\d{4})?"};    // шаблон U.S. почтового кода
```

Синтаксис и семантика регулярных выражений разрабатывались с тем, чтобы их можно было компилировать в конечные автоматы для эффективной реализации [Cox,2007]. Тип **regex** выполняет эту компиляцию во время работы программы.

37.1.1. Нотация регулярных выражений

Библиотека регулярных выражений может распознавать несколько вариантов нотации регулярных выражений (§37.2). Здесь я представляю умолчательную нотацию, вариант стандарта ECMA, который применяется в языке JavaScript.

Синтаксис регулярных выражений базируется на специальных символах (символах со специальным смыслом):

Специальные символы регулярных выражений			
.	Любой одиночный символ ("wildcard")	\	След. символ имеет специальное значение
[	Начало класса символов	*	Ноль или более
]	Конец класса символов	+	Один или более
{	Начало квантификатора	?	Опционально (ноль или один)
}	Конец квантификатора	|	Перечисление (или)
(	Начало группы	^	Начало строки; отрицание
)	Конец группы	$	Конец строки

Например, можно специфицировать линейку символов, начинающуюся с нуля или более букв **A**, за которыми следуют одна или более букв **B**, за которыми может следовать (опционально) буква **C**:

```
^A*B+C?$
```

Вот примеры последовательностей символов, которые отвечают запросу:

```
AAAAAAAAAAAABBBBBBBBBC
BC
B
```

А вот примеры несоответствий:

```
AAAAA   // нет B
    AAAABC // начальные пробелы
AABBCC // слишком много C
```

Часть образца рассматривается как подобразец (отдельно извлекаемый из **smatch**), если он заключен в круглые скобки.

Образец для поиска может маркироваться как опциональный или повторяющийся (по умолчанию — единичное вхождение) добавлением суффикса:

Повторение	
{ n }	Точно **n** раз
{ n, }	**n** или более раз
{n,m}	Как минимум **n** и как максимум **m** раз
*	Ноль или более, то есть, **{0,}**
+	Один или более, то есть, **{1,}**
?	Опционально (ноль или один), то есть, **{0,1}**

Пример:

```
A{3}B{2,4}C*
```

Вот примеры соответствий:

```
AAABBC
AAABBB
```

А это примеры несоответствий:

```
AABBC          // слишком мало A
AAABC          // слишком мало B
AAABBBBBCCC    // слишком много B
```

Знак **?** после нотации повторения означает «ленивый» или «нежадный». При поиске образца в исходной строке будет искаться наиболее короткое соответствие, а не наиболее длинное. По умолчанию ищется самый длинный вариант. Рассмотрим следующее:

```
ababab
```

Образец **(ab)*** соответствует всей **ababab**. Однако, **(ab)*?** соответствует только первым **ab**. Наиболее общие классификационные характеристики символов имеют имена:

Символьные классы	
alnum	Любой буквенно-цифровой символ
alpha	Любой буквенный символ
blank	Любой символ-разделитель, кроме разделителей строк
cntrl	Любой управляющий символ
d	Любая десятичная цифра
digit	Любая десятичная цифра
graph	Любой графический символ
lower	Любой символ нижнего регистра
print	Любой печатный символ
punct	Любой знак пунктуации
s	Любой пробельный символ

Продолжение

Символьные классы	
space	Любой пробельный символ
upper	Любой символ верхнего регистра
w	Любой буквенно-цифровой символ или знак подчеркивания
xdigit	Любая шестнадцатеричная цифра

Некоторые классы символов имеют еще и сокращенную нотацию:

Сокращения символьных классов		
\d	Десятичная цифра	**[[:digit:]]**
\s	Пробельный символ (пробел, табуляция и т.д.)	**[[:space:]]**
\w	Буква (**a-z**) или цифра (**0-9**) или подчеркивание (_)	**[_[:alnum:]]**
\D	Не **\d**	**[^[:digit:]]**
\S	Не **\s**	**[^[:space:]]**
\W	Не **\w**	**[^_[:alnum:]]**

Кроме того, языки, поддерживающие регулярные выражения, часто предоставляют нестандартную нотацию:

Нестандартные (но распостраненные) сокращения символьных классов		
\l	Символ нижнего регистра	**[[:lower:]]**
\u	Символ верхнего регистра	**[[:upper:]]**
\L	Не **\l**	**[^[:lower:]]**
\U	Не **\u**	**[^[:upper:]]**

Ради переносимости используйте полные имена классов, а не сокращения.

Для примера рассмотрим образец поиска, который описывает идентификаторы C++: знак подчеркивания или букву, за которыми следует потенциально пустая последовательность букв, цифр или знаков подчеркивания. Для иллюстрации тонкостей, я также включил несколько ложных попыток:

```
[:alpha:][:alnum:]*                 // неверно: символы из множества ":alph" с последующими ...
[[:alpha:]][[:alnum:]]*             // неверно: не принимает знак подчеркивания ('_' не alpha)
([[:alpha:]]|_)[[:alnum:]]*         // неверно: знак подчеркивания не входит в alnum
([[:alpha:]]|_)([[:alnum:]]|_)*     // OK, но грубо
[[:alpha:]_][[:alnum:]_]*           // OK: включаем знак подчеркивания
[_[:alpha:]][_[:alnum:]]*           // также OK
[_[:alpha:]]\w*                     // \w эквивалентно [_[:alnum:]]
```

Наконец, вот функция, которая использует простейшую версию **regex_match()** (§37.3.1) для проверки, является ли строка идентификатором:

```
bool is_identifier(const string& s)
{
    regex pat {"[_[:alpha:]]\\w*"};
    return regex_match(s,pat);
}
```

Обратите внимание на дублирование обратных наклонных черт для включения обратных наклонных черт в обычный строковый литерал. Как всегда, обратные наклонные черты применяются для спецификации специальных символов:

Специальные символы (§iso.2.14.3, §6.2.3.2)	
\n	Перевод строки
\t	Табуляция
\\	Одна обратная наклонная черта
\xhh	Символы Unicode, представленные двумя шестнадцатеричными цифрами
\uhhhh	Символы Unicode, представленные четырьмя шестнадцатеричными цифрами

Чтобы внести некоторую сумятицу, я покажу еще два варианта использования обратных наклонных черт:

Специальные символы (§iso.28.5.2, §37.2.2)	
\b	Первый или последний символ слова ("пограничный символ")
\B	Не \b
\i	i-й **sub_match** в текущем шаблоне

Применение «сырых строк» устраняет многие проблемы с обозначениями. Пример:

```
bool is_identifier(const string& s)
{
    regex pat {R"([_[:alpha:]]\w*)"};
    return regex_match(s,pat);
}
```

Вот примеры образцов для поиска:

```
Ax*             // A, Ax, Axxxx
Ax+             // Ax, Axxx      не A
\d-?\d          //1-2, 12       не 1--2
\w{2}-\d{4,5}   // Ab-1234, XX-54321, 22-5432      Цифры в \w
(\d*:)?(\d+)    // 12:3, 1:23, 123, :123   не 123:
(bs|BS)         // bs, BS        не bS
[aeiouy]        // a, o, u       английские гласные, не x
[^aeiouy]       // x, k          не английские гласные, не e
[a^eiouy]       // a, ^, o, u    английские гласные или ^
```

Группа (подобразец), которая потенциально представима через **sub_match**, отделяется круглыми скобками. Если вам нужны круглые скобки сами по себе, используйте **(?** вместо простой **(**. Например:

```
(\s|:|,)*(\d*)      // пробелы, двоеточие, и/или запятые с последующим числом
```

Предполагая, что нам не интересны символы перед числом (возможно разделители), составляем такую схему:

```
(?\s|:|,)*(\d*)     // пробелы, двоеточие, и/или запятые с последующим числом
```

Это избавит механизм регулярных выражений от необходимости хранить первые символы: вариант с **(?** имеет только один подобразец:

Примеры групп в регулярных выражениях	
\d*\s\w+	Нет групп (подобразцов)
(\d*)\s(\w+)	Две группы
(\d*)(\s(\w+))+	Две группы (группы не вкладываются друг в друга)
(\s*\w*)+	Одна группа, но один или более подобразцов; только последний подобразец сохраняется в **sub_match**
<(.*?)>(.*?)</\1>	Три группы; \1 означает «то же, что и группа 1»

Последний образец полезен для того, чтобы парсить XML. С его помощью находят маркеры тэг/конец-тэга. Обратите внимание, что я указал нежадное соответствие, **.*?**, для подобразца между тэгом и концом тэга. Если бы я использовал просто **.***, то следующие входные данные вызвали бы проблемы:

```
Always look for the <b>bright</b> side of <b>life</b>.
```

Жадный поиск для первого подобразца найдет соответствие между первым символом **<** и последним **>**. Жадный поиск на втором подобразце найдет соответствие между первым **** и последним ****. Оба будут корректными результатами, но не теми, которые вероятнее всего хотел программист.

Возможно варьирование деталей нотации регулярных выражений, если использовать опции (§37.2). Например, если используется **regex_constants::grep**, то **a?x:y** — это последовательность пяти обычных символов, потому что **?** в **grep** не означает «опционально».

Относительно исчерпывающего изложения регулярных выражений смотри [Friedl, 1997].

37.2. regex

Регулярные выражения реализуются движком соответствия (обычно конечным автоматом), конструируемым из последовательности символов, такой как **string**:

```
template<class C, class traits = regex_traits<C>>
class basic_regex {
public:
    using value_type = C;
    using traits_type = traits;
    using string_type = typename traits::string_type;
    using flag_type = regex_constants::syntax_option_type;
    using locale_type = typename traits::locale_type;

    ~basic_regex(); // не виртуальный; basic_regex не будет базовым классом
    // ...
};
```

regex_traits будут представлены в §37.5.

Как и **string**, **regex** является синонимом для версии, использующей **char**:

```
using regex = basic_regex<char>;
```

Смысл образцов регулярных выражений управляется посредством констант **syntax_option_type**, определяемых идентично друг другу в **regex_constants** и **regex**:

Константы basic_regex<C,Tr> (syntax_option_type, §iso.28.5.1)	
icase	При сравнении не учитывается регистр
nosubs	Совпадения подобразцов не сохраняются в результатах сравнения
optimize	Предпочитать быстрое сравнение быстрому созданию объектов
collate	Диапазоны символов вида **[a-b]** чувствительны к локализации
ECMAScript	Синтаксис регулярных выражений, используемый ECMAScript в ECMA-262 (с незначительными изменениями; §iso.28.13)
basic	Синтаксис регулярных выражений, используемый базовыми регулярными выражениями POSIX
extended	Синтаксис регулярных выражений, используемый расширенными регулярными выражениями POSIX

Продолжение

Константы basic_regex<C,Tr> (syntax_option_type, §iso.28.5.1)	
awk	Синтаксис регулярных выражений, используемый POSIX awk
grep	Синтаксис регулярных выражений, используемый POSIX grep
egrep	Синтаксис регулярных выражений, используемый POSIX grep -E

Лучше использовать умолчания, если только нет каких-либо особых причин поступать по-другому.

Объект **regex** можно конструировать из **string** или аналогичных последовательностей символов:

Конструкторы basic_regex<C,Tr> (§iso.28.8.2)	
basic_regex r {};	Конструктор по умолчанию: пустой шаблон; флаги сответствуют **regex_constants::ECMAScript**
basic_regex r {x,flags};	**x** может быть **basic_regex**, **string**, строкой в С-стиле или **initializer_list<value_type>** с синтаксисом, определяемым **flags**; **explicit**
basic_regex r {x};	**basic_regex{x,regex_constants::ECMAScript}**; explicit
basic_regex r {p,n,flags};	Конструировать **r** из символов [**p:p+n**) с синтаксисом, определяемым **flags**
basic_regex r {p,n};	**basic_regex{p,n,regex_constants::ECMAScript}**
basic_regex r {b,e ,flags}	Конструировать **r** из символов [**b:e**) с синтаксисом, определяемым **flags**
basic_regex r {b,e};	**basic_regex{b,e,regex_constants::ECMAScript}**

Основное применение **regex** относится к функциям поиска и замены (§37.3), но есть несколько операций с самим **regex**:

Операции над basic_regex<C,Tr> (§iso.28.8)	
r=x	Копирующее присваивание: **x** может быть **basic_regex**, строкой в С-стиле, **basic_string** или **initializer_list<value_type>**
r=move(r2)	Перемещающее присваивание
r=r.assign(r2)	Копирование или перемещение
r=r.assign(x,flags)	Копирование или перемещение; флаги **r** получают знач-е от **flags** **x** может быть **basic_string**, строкой в С-стиле или **initializer_list<value_type>**
r=r.assign(x)	**r=r.assign(x,regex_constants::ECMAScript)**
r=r.assign(p,n,flags)	Шаблон **r** получает знач-е из [**p:p+n**), а флаги **r** – из **flags**
r=r.assign(b,e,flags)	Шаблон **r** получает знач-е из [**b:e**) а флаги **r** – из **flags**
r=r.assign(b,e)	**r=r.assign(b,e,regex_constants::ECMAScript)**
n=r.mark_count()	**n** – число выделенных подобразцов **r**
x=r.flags()	**x** есть **flags** из **r**
loc2=r.imbue(loc)	**r** получает **loc**; **loc2** есть предыдущая **locale** из **r**
loc=r.getloc()	**loc** есть **locale** из **r**
r.swap(r2)	Обменять значения **r** и **r2**

Можете определить **locale** или **regex** вызовом **getloc()** и узнать от **flags()**, какие флаги используются, но к сожалению нет стандартного способа прочесть исходный образец. Если вам это нужно, то лучше сохранить копию строки, использованной для инициализации. Например:

```
regex pat1 {R"(\w+\d*)"};       // нельзя вывести шаблон в pat1

string s {R"(\w+\d*)"};
regex pat2 {s};
cout << s << '\n';              // шаблон (образец) в pat2
```

37.2.1. Результаты поиска соответствия

Результаты поиска соответствия по регулярному выражению собираются в объекте типа **match_results**, который содержит один или более объектов типа **sub_match**:

```
template<class Bi>
class sub_match : public pair<Bi,Bi> {
public:
    using value_type = typename iterator_traits<Bi>::value_type;
    using difference_type = typename iterator_traits<Bi>::difference_type;
    using iterator = Bi;
    using string_type = basic_string<value_type>;

    bool matched; // true если *this содержит соответствие
    // ...
};
```

Bi должен быть двунаправленным итератором (§33.1.2). **sub_match** можно рассматривать как пару итераторов на строке, в которой идет поиск соответствий.

Операции над sub_match<Bi>	
sub_match sm {};	Конструктор по умолчанию: пустая последовательность; constexpr
n=sm.length()	n – число символов в соответствии
s=sm	Неявное преобразование **sub_match** в **basic_string**; **s** есть **basic_string**, содержащая символы, имеющие соответствие
s=sm.str()	**s** есть **basic_string**, содержащая символы, имеющие соответствие
x=sm.compare(x)	Лексикографическое сравнение: **sm.str().compare(x)**; **x** может быть **sub_match**, **basic_string** или строкой в С-стиле
x==y	**x** равен ли **y**? **x** и **y** могут быть **sub_match** или **basic_string**
x!=y	!(x==y)
x<y	**x** лексикографически раньше **y**
x>y	y<x
x<=y	!(x>y)
x>=y	!(x<y)
sm.matched	**true**, если **sm** содержит соответствие; **false** в ином случае

Например:

```
regex pat ("<(.*?)>(.*?)</(.*?)>");

string s = "Always look for the <b> bright </b> side of <b> death </b>";

if (regex_search(s1,m,p2))
    if (m[1]==m[3]) cout << "match\n";
```

Здесь **match** является выводом.

match_results является контейнером для нескольких **sub_match**:

```
template<class Bi, class A = allocator<sub_match<Bi>>
class match_results {
public:
    using value_type = sub_match<Bi>;
    using const_reference = const value_type&;
    using reference = const_reference;
    using const_iterator = /* зависит от реализации */;
    using iterator = const_iterator;
    using difference_type = typename iterator_traits<Bi>::difference_type;
    using size_type = typename allocator_traits<A>::size_type;
    using allocator_type = A;
    using char_type = typename iterator_traits<Bi>::value_type;
    using string_type = basic_string<char_type>;

    ~match_results();    // не виртуальный

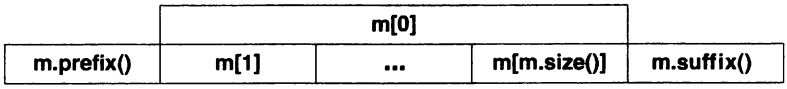
    // ...
};
```

Bi должен быть двунаправленным итератором (§33.1.2).

Как для **basic_string** и **basic_ostream**, несколько стандартных синонимов даются для наиболее общих **match_results**:

```
using cmatch = match_results<const char*>;              // C-строка
using wcmatch = match_results<const wchar_t*>;         // широкая C-строка
using smatch = match_results<string::const_iterator>;  // строка
using wsmatch = match_results<wstring::const_iterator>; // wстрока
```

match_results дают доступ к строке-образцу поиска, к **sub_matches**, а также к символам перед и после соответствия:



Для **match_results** имеется стандартный набор операций:

Операции с соответствиями regex<C,Tr> (§iso.28.9, §iso.28.10)	
match_results m {};	Конструктор по умолчанию: использует **allocator_type{}**
match_results m {a};	Использовать аллокатор **a**; explicit
match_results m {m2};	Копирующий и перемещающий конструкторы
m2=m	Копирующее присваивание
m2=move(m)	Перемещающее присваивание
m.~match_results()	Деструктор: освободить все ресурсы
m.ready()	Содержит ли **m** полное соответствие?
n=m.size()	**n-1** есть число подобразцов в **m**; **n==0**, если не было соответствия
n=m.max_size()	**n** – наибольшее возможное число **sub_match**-ей для **m**
m.empty()	**m.size()==0?**
r=m[i]	**r** есть **const** ссылка на **i**-й **sub_match** в **m**; **m[0]** представляет полное соответствие; если **i>= size()**, **m[i]** ссылается на **sub_match**, представляющий несопоставленное подвыражение
n=m.length(i)	**n=m[i].length()**; число символов в **m[i]**
n=m.length()	**n=m.length(0)**
pos=m.position(i)	**pos=m[i].first**; первый символ в **m[i]**

Продолжение

Операции с соответствиями regex<C,Tr> (§iso.28.9, §iso.28.10)	
s=m.str(i)	s=m[i].str(); строковое представление m[i]
s=m.str()	s=m.str(0)
sm=m.prefix()	sm есть sub_match, представляющий несопоставленные символы в m во входной строке перед соответствием
sm=m.suffix()	sm есть sub_match, представляющий несопоставленные символы в m во входной строке после соответствия
p=m.begin()	p указывает на первый sub_match в m
p=m.end()	p указывает на sub_match, следующий за последним в m
p=m.cbegin()	p указывает на первый sub_match в m (const iterator)
p=m.cend()	p указывает на sub_match, следующий за последним в m (const iterator)
a=m.get_allocator()	a есть аллокатор m
m.swap(m2)	Обменять состояния m и m2
m==m2	Значения sub_match-ей m и m2 равны?
m!=m2	!(m==m2)

Мы можем индексировать regex_match для доступа к sub_match, например m[i]. Если индекс, i, отсылает к несуществующему sub_match, то результат для m[i] представляет собой несопоставленный sub_match. Например:

```
void test()
{
    regex pat ("(AAAA)(BBB)?");
    string s = "AAAA";
    smatch m;
    regex_search(s,m,pat);

    cout << boolalpha;
    cout << m[0].matched << '\n';    // правда: нашли соответствие
    cout << m[1].matched << '\n';    // правда: первый sub_match
    cout << m[2].matched << '\n';    // ложь: нет второго sub_match
    cout << m[3].matched << '\n';    // ложь: не может быть третьего sub_match для pat
}
```

37.2.2. Форматирование

В regex_replace() форматирование выполняется с помощью функции format():

Форматирование regex<C,Tr> (§iso.28.10.5) Форматирование управляется опциями match_flag_type	
out=m.format(out,b,e,flags)	Копировать [b:e) в out; подменяя форматирующие символы на соответствия из m
out=m.format(out,b,e)	out=m.format(out,b,e,regex_constants::format_default)
out=m.format(out,fmt,flags)	out=m.format(out,begin(fmt),end(fmt),flags); fmt может быть basic_string или строкой в С-стиле
out=m.format(out,fmt)	out=m.format(out,fmt,regex_constants::format_default)
s=m.format(fmt,flags)	Создать копию fmt в s; подменяя форматирующие символы на соответствия из m; fmt может быть basic_string или строкой в С-стиле
s=m.format(fmt)	s=m.format(fmt,regex_constants::format_default)

Форматы могут содержать форматирующие символы:

Форматирующие символы для подмены	
$&	Полное совпадение
$'	Символы перед соответствием
$'	Символы после соответствия
$i	i-е частичное совпадение, напр., **$1**
$ii	ii-е частичное совпадение, напр., **$12**
$$	Не совпадение, символ **$**

Примеры смотри в §37.3.3.

Детали форматирования функцией **format()** контролируются набором опций (флагов):

Опции форматирования **regex<C,Tr>** (**regex_constants::match_flag_type**; §iso.28.5.2)	
format_default	Использовать правила ECMAScript (ECMA-262) (§iso.28.13)
format_sed	Использовать синтаксис POSIX sed
format_no_copy	Копировать только совпадения
format_first_only	Заменять только первое совпадение

37.3. Функции регулярных выражений

Функции, применяющие схемы регулярных выражений к данным, включают в себя **regex_search()** для поиска в последовательности символов, **regex_match()** для сопоставления с последовательностью символов фиксированной длины, и **regex_replace()** для замены найденных образцов.

Детали сопоставления контролируются набором опций (флагов):

Опции сопоставления **regex<C,Tr>** (**regex_constants::match_flag_type**; §iso.28.5.2)	
match_not_bol	Символ **^** не рассматривается, как «начало строки»
match_not_eol	Символ **$** не рассматривается, как «конец строки»
match_not_bow	**\b** не соответствует последовательности [**first,first**)
match_not_eow	**\b** не соответствует последовательности [**last,last**)
match_any	Если возможно более одного совпадения, принимается любое совпадение
match_not_null	Не сопоставлять с пустой последовательностью
match_continuous	Сопоставлять только часть последовательности, начинающуюся с **first**
match_prev_avail	**--first** есть допустимая позиция итератора

37.3.1. regex_match()

Для выявления совпадения по образцу для всей последовательности известной длины используется **regex_match()**:

Сопоставление регулярных выражений (§iso.28.11.2) Сопоставление управляется опциями **match_flag_type** (§37.3)	
regex_match(b,e,m,pat,flags)	Совпадает ли [**b:e**) с **regex**-шаблоном **pat**? помещает рез-т в **match_results m**; использует опции **flags**
regex_match(b,e,m,pat)	**reg ex_match(b,e,m,pat,regex_constants::match_default)**
regex_match(b,e,pat,flags)	Совпадает ли [**b:e**) с **regex**-шаблоном **pat**? использует опции **flags**
regex_match(b,e,pat)	**reg ex_match(b,e,pat,regex_constants::match_default)**

Продолжение

Сопоставление регулярных выражений (§iso.28.11.2) Сопоставление управляется опциями **match_flag_type** (§37.3)	
regex_match(x,m,pat,flags)	Совпадает ли **x** с **regex**-шаблоном **pat**? **x** может быть **basic_string** или строкой в С-стиле; помещает рез-т в **match_results m**; использует опции **flags**
regex_match(x,m,pat)	**regex_match(x,m,pat,regex_constants::match_default)**
regex_match(x,pat,flags)	Совпадает ли **x** с **regex**-шаблоном **pat**? **x** может быть **basic_string** или строкой в С-стиле; использует опции **flags**
regex_match(x,pat)	**regex_match(x,pat,regex_constants::match_default)**

Для примера рассмотрим простенькую программу проверки формата таблицы. Если у таблицы формат ожидаемый, то программа выводит в **cout** сообщение «all is well», а в противном случае она пишет сообщение об ошибке в **cerr**. Таблица есть последовательность рядов, каждый из которых состоит из четырех полей, разделенных знаком табуляции, кроме заглавного ряда, который может иметь три поля. Пример:

Class	Boys	Girls	Total
1a	12	15	27
1b	16	14	30
Total	28	29	57

Программа читает заглавный (титульный) ряд, а затем выполняет суммирования по каждой строке до тех пор, пока не достигнет последней, озаглавленной «Total»:

```
int main()
{
    ifstream in("table .txt"); // входной файл
    if (!in) cerr << "no file\n";

    string line;        // входной буфер
    int lineno = 0;

    regex header {R"(^[\w ]+(\t[\w ]+)*$)"};      // слова, разделямые tab-ом
    regex row {R"(^([\w ]+)(\t\d+)(\t\d+)(\t\d+)$)"}; // метка за кот-й идут три числа, разд-ные tab-ом

    if (getline(in,line)) { // проверить и удалить заголовочную строку
        smatch matches;
        if (!regex_match(line,matches,header))
            cerr << "no header\n";
    }

    int boys = 0;                               // текущие итоги (суммы)
    int girls = 0;

    while (getline(in,line)) {
        ++lineno;
        smatch matches;                         // частичные совпадения идут сюда

        if (!regex_match(line,matches,row))
            cerr << "bad line: " << lineno << '\n';

        int curr_boy = stoi(matches[2]);    // stoi() см. в §36.3.5
        int curr_girl = stoi(matches[3]);
        int curr_total = stoi(matches[4]);
        if (curr_boy+curr_girl != curr_total) cerr << "bad row sum \n";

        if (matches[1]=="Total") { // последняя строка
            if (curr_boy != boys) cerr << "boys do not add up\n";
```

```
                if (curr_girl != girls) cerr << "girls do not add up\n";
                cout << "all is well\n";
                return 0;
            }
            boys += curr_boy;
            girls += curr_girl;
        }

        cerr << "didn't find total line\n")
        return 1;
    }
```

37.3.2. regex_search()

Для поиска образца в части последовательности, вроде файла, используется **regex_search()**:

Поиск по регулярным выражениям (§iso.28.11.3) Сопоставление управляется опциями **match_flag_type** (§37.3)	
regex_search(b,e,m,pat,flags)	Содержит ли [**b**:**e**) совпадение с **regex**-шаблоном **pat**? помещает рез-т в **match_results m**; использует опции **flags**
regex_search(b,e,m,pat)	**reg ex_search(b,e,m,regex_constants::match_default)**
regex_search(b,e,pat,flags)	Содержит ли [**b**:**e**) совпадение с **regex**-шаблоном **pat**? использует опции **flags**
regex_search(b,e,pat)	**reg ex_search(b,e,pat,regex_constants::match_default)**
regex_search(x,m,pat,flags)	Содержит ли **x** совпадение с **regex**-шаблоном **pat**? **x** может быть **basic_string** или строкой в С-стиле; помещает рез-т в **match_results m**; использует опции **flags**
regex_search(x,m,pat)	**regex_search(x,m,pat,regex_constants::match_default)**
regex_search(x,pat,flags)	Содержит ли **x** совпадение с **regex**-шаблоном **pat**? **x** может быть **basic_string** или строкой в С-стиле; использует опции **flags**
regex_search(x,pat)	**regex_search(x,pat,regex_constants::match_default)**

К примеру, я мог бы поискать наиболее полные ошибки в правописании моего имени:

```
regex pat {"[Ss]tro?u?v?p?stra?o?u?p?b?"};

smatch m;
for (string s; cin>>s; )
    if (regex_search(s,m,pat))
        if (m[0]!="stroustrup" && m[0]!="Stroustrup" )
            cout << "Found: " << m[0] << '\n';
```

При соответствующем вводе будет получен следующий список ошибок в написании **Stroustrup**:

```
Found: strupstrup
Found: Strovstrup
Found: stroustrub
Found: Stroustrop
```

Заметьте, что **regex_search()** найдет образец, даже если он «спрятан» среди других символов. Например, будет найден **strustrub** в **abstrustrubal**. При необходимости соответствия образцу каждого символа во входной строке, используйте **regex_match()** (§37.3.1).

37.3.3. regex_replace()

Для выполнения простых подстановок по образцу в части последовательности, вроде файла, используйте **regex_replace()**:

Подстановка по регулярным выражениям (§iso.28.11.4) Сопоставление управляется опциями **match_flag_type** (§37.3)	
out=regex_replace(out,b,e,pat,fmt,flags)	Копировать [**b**:**e**) в **out**, сравнивая с **regex**-шаблоном **pat**; когда найдено совпадение с **pat**, скопировать его в **out**, используя формат **fmt** под управлением **flags**; **fmt** может быть **basic_string** или строкой в С-стиле
out=regex_replace(out,b,e,pat,fmt)	**out=regex_replace(out,b,e,pat,fmt,** **regex_constants::match_defaults)**
s=regex_replace(x,pat,fmt,flags)	Копировать **x** в **s**, сравнивая с **regex**-шаблоном **pat**; когда найдено совпадение с **pat**, скопировать его в **s**, используя формат **fmt** под управлением **flags**; **x** может быть **basic_string** или строкой в С-стиле; **fmt** может быть **basic_string** или строкой в С-стиле
s=regex_replace(x,pat,fmt)	**s=regex_replace(x,pat,fmt,** **regex_constants::match_defaults)**

Копирование формата выполняется функцией **regex** по имени **format()** (§37.2.2) с помощью префиксного использования знака **$**, например **$&** для полного совпадения, и **$2** для следующего частичного совпадения. Вот пример небольшой тестовой программы, которая принимает строку пар из слов и чисел, и выводит их как {слово, число}:

```
void test1()
{
    string input {"x 1 y2 22 zaq 34567"};
    regex pat {"(\w+)\s(\d+)"};      // слово пробел число
    string format {"{$1,$2}\n"};

    cout << regex_replace(input,pat,format);
}
```

Вывод программы таков:

```
{x,1}
 {y2,22}
 {zaq,34567}
```

По умолчанию, **regex_match()** копирует несопоставленные символы в свой вывод, так что два лидирующих несопоставленных с **pat** пробела присутствуют в выводе.

Чтобы устранить эти пробелы, мы можем использовать опцию **format_no_copy** (§37.2.2):

```
cout << regex_replace(input,pat,format,regex_constants::format_no_copy);
```

Теперь вывод таков:

```
{x,1}
{y2,22}
{zaq,34567}
```

Неполные соответствия не обязаны быть отсортированными в рамках вывода:

```
void test2()
{
    string input {"x 1 y 2 z 3"};
    regex pat {"(\w)\s(\d+)"};        // слово пробел число
    string format {"$2: $1\n"};

    cout << regex_replace(input,pat,format,regex_constants::format_no_copy);
}
```

Результат работы таков:

```
1: x
22: y2
34567: zeq
```

37.4. Итераторы регулярных выражений

Функция **regex_search()** помогает нам найти единственное совпадение образца в потоке данных. А что, если нам нужно что-то сделать со всеми совпадениями? Если данные организованы как последовательность легко распознаваемых физических строк или записей, то мы можем итерировать по ним и использовать **regex_match()** для каждого из них. Если все, что нам нужно сделать с каждым актом совпадения, это простая подстановка, то мы можем использовать **regex_replace()**. Если же мы хотим итерировать по всей последовательности символов, делая что-нибудь с каждым совпадением, то используем **regex_iterator**.

37.4.1. regex_iterator

Итератор **regex_iterator** является двунаправленным итератором, который при инкрементировании продвигается по последовательности к следующему совпадению с образцом:

```
template<class Bi,
         class C = typename iterator_traits<Bi>::value_type,
         class Tr = typename regex_traits<C>::type>
class regex_iterator {
public:
    using regex_type = basic_regex<C,Tr>;
    using value_type = match_results<Bi>;
    using difference_type = ptrdiff_t;
    using pointer = const value_type*;
    using reference = const value_type&;
    using iterator_category = forward_iterator_tag;
    // ...
}
```

regex_traits будут рассматриваться в §37.5.
Предоставляется обычный набор синонимов:

```
using cregex_iterator = regex_iterator<const char*>;
using wcregex_iterator = regex_iterator<const wchar_t*>;
using sregex_iterator = regex_iterator<string::const_iterator>;
using wsregex_iterator = regex_iterator<wstring::const_iterator>;
```

Для **regex_iterator** предоставляется минимальный набор операций:

regex_iterator<Bi,C,Tr> (§iso.28.12.1)	
regex_iterator p {};	p есть конец последовательности
regex_iterator p {b,e,pat,flags};	Итерировать по [b:e), в поиске совпадений с pat, используя опции flags
regex_iterator p {b,e,pat};	p инициализируется знач-ем {b,e,pat,regex_constants::match_default}
regex_iterator p {q};	Копирующий конструктор (нет перемещ. констр-ра)
p=q	Копирующее присваивание (нет пермещ. присв-я)
p==q	Указывает ли p на тот же sub_match, что и q?
p!=q	!(p==q)
c=*p	c есть текущий sub_match
x=p->m	x=(*p).m
++p	Пусть p указывает на следующее совпадение с образцом p
q=p++	q=p, тогда ++p

Так как regex_iterator является двунаправленным итератором, то мы не можем напрямую итерировать в istream.

В качестве примера мы можем вывести все разделяемые пробельными символами слова в string:

```
void test()
{
    string input = "aa as; asd ++e^asdf asdfg";
    regex pat {R"(\s+(\w+))"};
    for (sregex_iterator p(input.begin(),input.end(),pat); p!=sregex_iterator{}; ++p)
        cout << (*p)[1] << '\n';
}
```

Вот вывод этой программы:

```
as
asd
asdfg
```

Заметьте, что мы пропускаем первое слово, aa, поскольку оно не имеет предшествующего пробела. Если мы упростим образец для поиска до R"((\ew+))", то получим

```
aa
as
asd
e
asdf
asdfg
```

Невозможно писать посредством regex_iterator, и regex_iterator{} - это конец последовательности.

37.4.2. regex_token_iterator

regex_token_iterator является адаптором для regex_iterator, который итерирует по sub_match найденных match_results:

```
template<class Bi,
    class C = typename iterator_traits<Bi>::value_type,
    class Tr = typename regex_traits<C>::type>
class regex_token_iterator {
```

```
public:
    using regex_type = basic_regex<C,Tr>;
    using value_type = sub_match<Bi>;
    using difference_type = ptrdiff_t;
    using pointer = const value_type*;
    using reference = const value_type&;
    using iterator_category = forward_iterator_tag;
// ...
```

regex_traits рассматриваются в §37.5.

Предоставляется обычный набор синонимов:

```
using cregex_token_iterator = regex_token_iterator<const char*>;
using wcregex_token_iterator = regex_token_iterator<const wchar_t*>;
using sregex_token_iterator = regex_token_iterator<string::const_iterator>;
using wsregex_token_iterator = regex_token_iterator<wstring::const_iterator>;
```

Для **regex_token_iterator** предусмотрен минимальный набор операций:

regex_token_iterator (§iso.28.12.2)	
regex_token_iterator p {};	**p** есть конец последовательности
regex_token_iterator p {b,e,pat,x,flags};	**x** перечисляет индексы **sub_match**-ей, выбранных для итерации или **0**, что означает «все совпадение» или **-1**, что означает «представить каждую несопоставленную послед-ть символов, как **sub_match**»; **x** может быть **int, initializer_list<int>, const vector<int>&** или **const int (&sub_match)[N]**
regex_token_iterator	**p {b,e,pat,x};** **p** инициализируется знач-ем **{b,e,pat,x,regex_constants::match_default}**
regex_token_iterator p {b,e,pat};	**p** инициализируется знач-ем **{b,e,pat,0,regex_constants::match_default}**
regex_token_iterator p {q};	Копирующий конструктор (нет перемещ. констр-ра)
p.~regex_token_iterator()	Деструктор: освободить все ресурсы
p=q	Копирующее присваивание (нет пермещ. присв-я)
p==q	Указывает ли **p** на тот же **sub_match**, что и **q**?
p!=q	**!(p==q)**
c=*p	**c** есть текущий **sub_match**
x=p->m	**x=(*p).m**
++p	Пусть **p** указывает на следующее совпадение с образцом **p**
q=p++	**q=p, тогда ++p**

Аргумент **x** содержит список всех **sub_match**, вовлекаемых в итерирование. Например, вот итерирование по совпадениям **1** и **3**:

```
void test1()
{
    string input {"aa::bb cc::dd ee::ff"};
    regex pat {R"((\w+)([[:punct:]]+)(\w+)\s*)"};
    sregex_token_iterator end {};
    for (sregex_token_iterator p{input.begin(),input.end(),pat,{1,3}}; p!=end; ++p)
        cout << *p << '\n';
}
```

Вывод программы таков:

```
aa
bb
cc
dd
ee
ff
```

Опция **-1** инвертирует стратегию сообщений о совпадениях, представляя каждую последовательность символов, которая не отвечает **sub_match**. Часто об этом говорят как о расщеплении по лексемам (то есть о расщеплении потока символов на лексемы), потому что когда ваш образец соответствует разделителям лексем, опция **-1** оставляет вас с лексемами. Например:

```
void test2()
{
    string s {"1,2 , 3 ,4,5, 6 7"}; // ввод
    regex pat {R"(\s*,\s*)"};       // запятая - разделитель лексем
    copy(sregex_token_iterator{s.begin(),s.end(),pat,-1)},
        sregex_token_iterator{},
        ostream_iterator<string>{cout,"\n"});
}
```

Вывод программы таков:

```
1
2
3
4
5
6 7
```

Это можно эквивалентно выполнить с помощью явного цикла:

```
void test3()
{
    sregex_token_iterator end{};
    for (sregex_token_iterator p {s.begin(),s.end(),pat,-1}; p!=end; ++p)
        cout << *p << '\n';
}
```

37.5. regex_traits

regex_traits<T> представляет соответствие между типом символов, типом строки и схемой локализации, что требуется реализатору **regex**:

```
template<class C>
struct regex_traits {
public:
    using char_type = C;
    using string_type = basic_string<char_type>;
    using locale_type = locale;
    using char_class_type = /* зависящий о реализации тип битовой маски */;
    // ...
};
```

Стандартная библиотека предоставляет специализации **regex_traits<char>** и **regex_traits<wchar_t>**.

Операции над regex_traits<C> (§iso.28.7)	
regex_traits tr {};	Создать умолчательный **regex_trait<C>**
n=length(p)	**n** есть число символов в С-строке **p**; **n=char_traits<C>::length()**; static
c2=tr.translate(c)	**c2=c**, то есть, нет эффекта
c2=tr.translate_nocase(c)	**use_facet<ctype<C>>(getloc()).tolower(c)**; §39.4.5
	s=tr.transform(b,e) **s** есть строка, с помощью к-рой можно сравнить [**b:e**) с другими строками; §39.4.1
s=tr.transform_primary(b,e)	**s** есть строка, с помощью к-рой можно сравнить [**b:e**) с другими строками; без учета регистра; §39.4.1
s=tr.lookup_collatename(b,e)	**s** есть **string** с именем элемента сравнения, названного [**b:e**) или пустая строка
m=tr.lookup_classname(b,e,ign)	**m** есть строка с именем маски классификации для классификационного имени [**b:e**); без учета регистра, если **ign==true**
m=tr.lookup_classname(b,e)	**m=tr.lookup_classname(b,e,false)**
tr.isctype(c,m)	**c** классифицируется, как **m**? **m** есть **class_type**
i=tr.value(c,b)	**i** есть целочисленное значение в **c** по основанию **b**; **b** должно быть **8**, **10**, или **16**
loc2=tr.imbue(loc)	Сделать **loc** локализацией **tr**; **loc2** есть предыдущая лок-ция **tr**
loc=tr.getloc()	**loc** есть локализация **tr**

Классификационное имя — это одна из символьных классификаций, перечисленных в §37.1.1, вроде **alpha**, **s** и **xdigit**.

37.6. Советы

[1] Для наиболее общих случаев использования регулярных выражений применяйте **regex**; §37.1.

[2] Различные варианты нотации регулярных выражений могут удовлетворить разным стандартам; §37.1.1, §37.2.

[3] Умолчательный вариант нотации регулярных выражений соответствует стандарту ECMAScript (JavaScript); §37.1.1.

[4] Ради переносимости используйте нотацию классов символов вместо нестандартных сокращений; §37.1.1.

[5] Не перебарщивайте с регулярными выражениями, а то получите язык «только для записи»; §37.1.1.

[6] Отдавайте предпочтение «сырым» строковым литералам во всех случаях, кроме простейших; §37.1.1.

[7] Запись **\i** позволяет задать подобразец в терминах предыдущего подобразца; §37.1.1.

[8] Используйте **?** для придания образцам «ленивости»; §37.1.1, §37.2.1.

[9] **regex** может использовать нотации ECMAScript, POSIX, awk, grep и egrep; §37.2.

[10] Храните копию строки образца, если вам она нужна для вывода; §37.2.

[11] Используйте **regex_search()** для поиска в потоках символов и **regex_match()** для поиска в фиксированных раскладках; §37.3.2, §37.3.1.

38

Потоки ввода/вывода

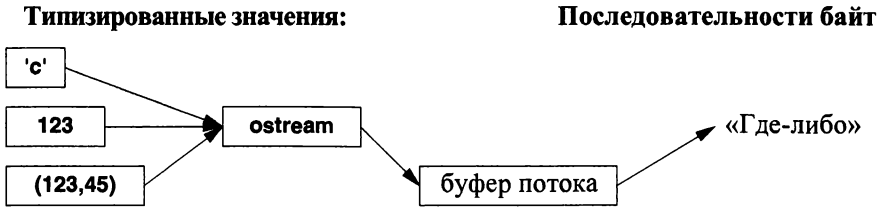
Что вы видите, то и получаете.
— Брайан Керниган

- Введение
- Иерархия потоков ввода/вывода
  Файловые потоки; Строковые потоки
- Обработка ошибок
- Операции ввода/вывода
  Операции ввода; Операции вывода; Манипуляторы; Состояние потока; Форматирование
- Буферизация
  Потоки вывода и буферы; Потоки ввода и буферы; Итераторы буферов
- Советы

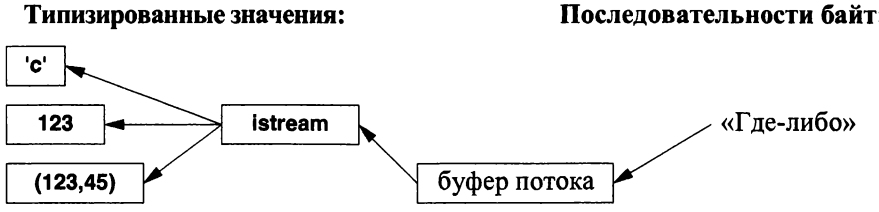
38.1. Введение

Библиотека потоков ввода/вывода обеспечивает форматированный и бесформатный буферированный ввод/вывод текста и числовых значений. Определения средств потокового ввода/вывода расположены в файлах **<istream>**, **<ostream>** и т. д.; см. §30.2.

ostream конвертирует типизированные объекты в поток символов (байт):



istream конвертирует поток символов (байт) в типизированные объекты:

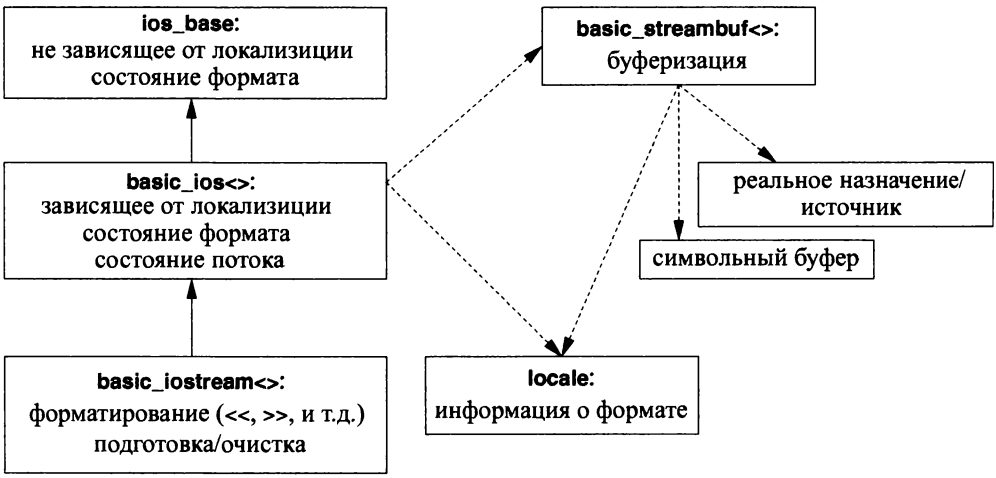


iostream является потоком, который может действовать и как **istream**, и как **ostream**. Буферы на диаграммах являются буферами потоков (**streambufs**; §38.6). Они нужны для

установления соответствия между **iostream** и какого-либо устройства, файла или памяти. Операции с **istream** и **ostream** рассматриваются в §38.4.1 и §38.4.2.

Для использования библиотеки ввода/вывода не нужно знать детали ее реализации. Поэтому я представляю лишь общие идеи, необходимые для понимания и использования потоков. Если вы хотите реализовать собственные потоки, совместимые со стандартом, то вам потребуется еще много информации: точная и полная копия стандарта, руководство по вашей операционной системе, примеры работающего кода, и все это в дополнение к материалу, рассмотренному в данной главе.

Можно отобразить ключевые компоненты системы ввода/вывода графически:



Непрерывные линии со стрелками означают «наследует от». Точечные линии со стрелками означают «указывает на». Классы, маркированные знаками <>, являются шаблонами, параметризуемыми типом символов, и содержащими схему локализации.

Операции ввода/вывода:

- Безопасны по типам и чувствительны к типам
- Расширяемы (когда разрабатывается новый тип, соответствующие операции ввода/вывода могут добавляться без модификации существующего кода)
- Чувствительны к схемам локализации (Глава 39)
- Эффективны (хотя их потенциал и не всегда реализован полностью)
- В состоянии взаимодействовать и сосуществовать со стандартным вводом/выводом языка C (§43.3)
- Включают форматированные, неформатированные операции и операции на уровне символов

Класс **basic_iosream** имеет базовыми классами **basic_istream** (§38.6.2) и **basic_ostream** (§38.6.1):

```
template<typename C, typename Tr = char_traits<C>>
class basic_iostream :
    public basic_istream<C,Tr>, public basic_ostream<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;
    using off_type = typename Tr::off_type;
    using traits_type = Tr;
```

```
        explicit basic_iostream(basic_streambuf<C,Tr>* sb);
        virtual ~basic_iostream();
    protected:
        basic_iostream(const basic_iostream& rhs) = delete;
        basic_iostream(basic_iostream&& rhs);

        basic_iostream& operator=(const basic_iostream& rhs) = delete;
        basic_iostream& operator=(basic_iostream&& rhs);
        void swap(basic_iostream& rhs);
    };
```

Шаблонные параметры специфицируют тип символов и их характеристики, нужные для манипуляции символами (§36.2.2).

Заметьте, что нет никаких копирующих операций: совместное владение или клонирование довольно сложного состояния потока реализовать непросто, да и стоить это будет дорого. Предполагается, что перемещающие операции используются производными классами, и поэтому они **protected**. Перемещение **iostream** без перемещения его состояния (определяющего производный класс, например **fstream**) спровоцирует ошибку.

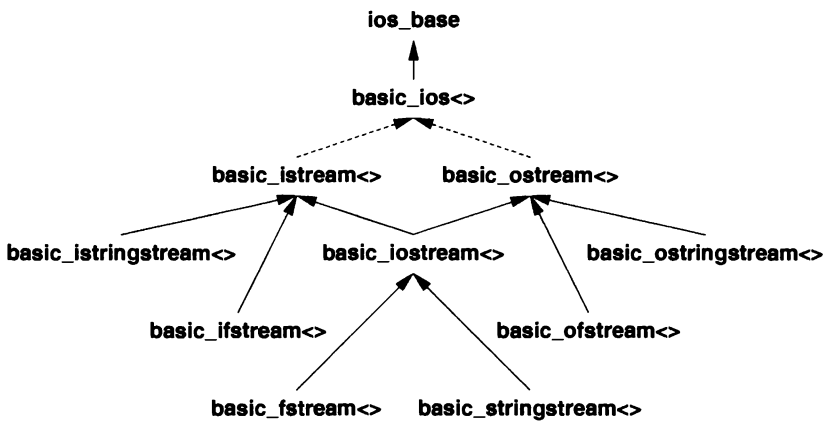
Существуют три стандартных потока:

Стандартный поток ввода/вывода	
cout	Стандартный символьный поток вывода (обычно на экран)
cin	Стандартный символьный поток ввода (обычно с клавиатуры)
cerr	Стандартный небуферированный поток вывода ошибок
clog	Стандартный поток вывода ошибок
wcin	Расширенный поток – аналог **cin**
wcout	Расширенный поток – аналог **cout**
wcerr	Расширенный поток – аналог **cerr**
wclog	Расширенный поток – аналог **clog**

Объявления для типов потоков и потоковых объектов предоставляются в файле **<iosfwd>**.

38.2. Иерархия потоков ввода/вывода

Поток **istream** может быть присоединен к устройству ввода (например, к клавиатуре), к файлу или к строке (**string**). Средства потокового ввода/вывода организованы в классовую иерархию:



Если класс имеет суффикс <>, то это шаблон, параметризуемый типом символов. Точечная линия указывает на виртуальный базовый класс (§21.3.5).

В ключевом классе **basic_os** имеется большая часть реализации и многие операции. Но большинство пользователей не видит этого класса: он (по большей части) является деталью реализации потоков. Его описание дается в §38.4.4. Его средства рассматриваются в контексте их функциональности (например, форматирование; §38.4.5).

38.2.1. Файловые потоки

В файле **<fstream>** стандартная библиотека определяет потоки в файл и из файла:

- **ifstream** для чтения из файла
- **ofstream** для записи в файл
- **fstream** для чтения из файла и записи в файл

Файловые потоки следуют одной и той же схеме. Поэтому я описываю только лишь **fstream**:

```
template<typename C, typename Tr=char_traits<C>>
class basic_fstream
: public basic_iostream<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;     // для позиций в файле
    using off_type = typename Tr::off_type;     // для смещений в файле
    using traits_type = Tr;
    // ...
};
```

Набор операций для **fstream** довольно простой:

basic_fstream<C,Tr> (§iso.27.9)	
fstream fs {};	**fs** – файловый поток, не присоединенный к файлу
fstream fs {s,m};	**fs** – файловый поток, с файлом **s**, открытым в режиме **m**; **s** может быть **string** или строкой в С-стиле
fstream fs {fs2};	Перемещающий конструктор: **fs2** перемещается в **fs**; **fs2** становится неприсоединенным
fs=move(fs2)	Перемещающее присваивание: **fs2** перемещается в **fs**; **fs2** становится неприсоединенным
fs.swap(fs2)	Обменять состояния **fs** и **fs2**
p=fs.rdbuf()	**p** есть указатель на файловый буфер потока **fs (basic_filebuf<C,Tr>)**
fs.is_open()	**fs** открыт?
fs.open(s,m)	Открыть файл, названный **s**, в режиме **m** и связать с ним **fs**; устанавливает **failbit** у **fs**, если не может открыть файл; **s** может быть **string** или строкой в С-стиле
fs.close()	Закрыть файл, связанный с **fs** (если есть)

Кроме того, строковые потоки замещают защищенные виртуальные функции **basic_ios**: **underflow()**, **pbackfail()**, **overflow()**, **setbuf()**, **seekoff()** и **seekpos()** (§38.6).

У файлового потока нет копирующих операций. Если вам нужно, чтобы два имени ссылались на один и тот же файловый поток, то следует воспользоваться ссылкой или указателем, или же осторожно манипулировать файловыми **streambuf** (§38.6).

Если не получается открыть **fstream**, то поток оказывается в состоянии **bad()** (§38.3).

Есть шесть синонимов файловых потоков, определенных в **<fstream>**:

```
using ifstream = basic_ifstream<char>;
using wifstream = basic_ifstream<wchar_t>;
using ofstream = basic_ofstream<char>;
using wofstream = basic_ofstream<wchar_t>;
using fstream = basic_fstream<char>;
using wfstream = basic_fstream<wchar_t>;
```

Можно открыть файл в одном из нескольких режимов, как это специфицировано в **ios_base** (§38.4.4):

Режимы потоков (§iso.27.5.3.1.4)	
ios_base::app	Добавление (т. е., дописывать к концу файла)
ios_base::ate	«В конец» (открыть и перейти в конец файла)
ios_base::binary	Бинарный режим; может быть системно-зависимое поведение
ios_base::in	Для чтения
ios_base::out	Для записи
ios_base::trunc	Урезать файл до 0 в длину

В любом случае эффект открытия файла может зависеть от операционой системы, и если операционная система не открывает файл в заявленном режиме, то получается поток в состоянии **bad()** (§38.3). Например:

```
ofstream ofs("target");    // "o" от "output" подразумевается ios::out
if (!ofs)
    error("couldn't open 'target' for writing");
fstream ifs;               // "i" от "input" подразумевается ios::in
ifs.open("source",ios_base::in);
if (!ifs)
    error("couldn't open 'source' for reading");
```

Позиционирование в файле рассматривается в §38.6.1.

38.2.2. Строковые потоки

В файле **<sstream>** стандартная библиотека определяет потоки в строку и из строк (объектов типа **string**):

- **istringstream** для чтения из **string**
- **ostringstream** для записи в **string**
- **stringstream** для чтения из **string** и записи в **string**

Строковые потоки следуют одной и той же схеме, так что я рассмотрю лишь **stringstream**:

```
template<typename C, typename Tr = char_traits<C>, typename A = allocator<C>>
class basic_stringstream
    : public basic_iostream<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type; // для позиций в строке
    using off_type = typename Tr::off_type; // для смещений в строке
    using traits_type = Tr;
    using allocator_type = A;

    // ...
```

Операции для **stringstream** таковы:

basic_stringstream<C,Tr,A> (§iso.27.8)	
stringstream ss {m};	**ss** есть пустой строковый поток в режиме **m**
stringstream ss {};	Констр-р по умолчанию: **stringstream ss {ios_base::outlios_base::in};**
stringstream ss {s,m};	**ss** есть строковый поток с буфером, инициализированным из **string s**, в режиме **m**
stringstream ss {s};	**stringstream ss {s,ios_base::outlios_base::in};**
stringstream ss {ss2};	Перемещающий конструктор: **ss2** перемещается в **ss**; **ss2** становится пустым
ss=move(ss2)	Перемещающее присваивание: **ss2** перемещается в **ss**; **ss2** становится пустым
p=ss.rdbuf()	**p** указывает на строковый буфер потока **ss (basic_stringbuf<C,Tr,A>)**
s=ss.str()	**s** есть **string**, содержащий копию символов из **ss: s=ss.rdbuf()->str()**
ss.str(s)	Буфер **ss** инициализируется из **string s: ss.rdbuf()->str(s)**; если **ss** имеет режим **ios::ate** («в конец»), значения, записанные в **ss**, добавляются после символов из **s**; иначе, значения перетирают символы из **s**
ss.swap(ss2)	Обменять состояния **ss** и **ss2**

Режимы открытия рассматриваются в §38.4.4. Для **istringstream** умолчательный режим есть **ios_base::in**. Для **ostringstream** умолчательный режим есть **ios_base::out**.

Кроме того, строковые потоки замещают защищенные виртуальные функции класса **basic_ios**: **underflow()**, **pbackfail()**, **overflow()**, **setbuf()**, **seekoff()** и **seekpos()** (§38.6).

У строковых потоков нет копирующих операций. Если вам нужны два имени для отсылки к одному и тому же строковому потоку, то следует использовать ссылку или указатель.

Имеется шесть синонимов для строковых потоков, определенных в файле **<sstream>**:

```
using istringstream = basic_istringstream<char>;
using wistringstream = basic_istringstream<wchar_t>;
using ostringstream = basic_ostringstream<char>;
using wostringstream = basic_ostringstream<wchar_t>;
using stringstream = basic_stringstream<char>;
using wstringstream = basic_stringstream<wchar_t>;
```

Например:

```
void test()
{
    ostringstream oss {"Label: ",ios::ate};  // пишем в конец
    cout << oss.str() << '\n';       // пишет "Label: "
    oss<<"val";
    cout << oss.str() << '\n';       // пишет "Label: val" ("val" добавляется после "Label: ")

    ostringstream oss2 {"Label: "};          // пишем в начало
    cout << oss2.str() << '\n';      // пишет "Label: "
    oss2<<"val";
    cout << oss2.str() << '\n';      // пишет "valel: " (val перетирает "Label: ")
}
```

Я предпочитаю использовать **str()** только для вычитывания результата из **istringstream**. Невозможно напрямую вывести строковый поток; нужно применить **str()**:

```
void test2()
{
    istringstream iss;
    iss.str("Foobar");    // наполним iss
```

```
    cout << iss << '\n';        // пишет 1
    cout << iss.str() << '\n'; // OK: пишем "Foobar"
}
```

Причина в использовании потенциально удивляющей 1 состоит в том, что **iostream** конвертирует это для проверки состояния:

```
if (iss) {  // последнее действие над iss успешно; iss в состоянии good() или eof()
    // ...
}
else {
    // обработка ошибки
}
```

38.3. Обработка ошибок

Поток **iostream** может находиться в одном из четырех состояний, определеных в **basic_ios** в файле **<ios>** (§38.4.4):

Состояния потока (§iso.27.5.5.4)	
good()	Предыдущие операции с **iostream** завершились успешно
eof()	Достигнут конец файла («end-of-file»)
fail()	Произошло нечто неожиданное (напр., искали цифру, а нашли '**x**')
bad()	Произошло нечто неожиданное и серьезное (напр., ошибка чтения с диска)

Любая операция над потоком, находящимся не в состоянии **good()**, не дает никакого эффекта. Можно использовать **iostream** в качестве условия. В таком случае условие есть true, если состояние у **iostream** есть **good()**. Это составляет идиому чтения потока значений:

```
for (X x; cin>>x;) { // читаем в буфер типа X
    // ... делаем что-нибудь с x ...
}
// мы попадаем сюда, когда >> не может прочитать еще X из cin
```

После неудачи чтения, мы могли бы очистить поток и продолжать:

```
int i;
if (cin>>i) {
    // ... используем i ...
} else if (cin.fail()){ // возможно, ошибка форматирования
    cin.clear();
    string s;
    if (cin>>s) {    // попробуем обработать с помощью строки
        // ... используем s ...
    }
}
```

Альтернативно можно обрабатывать ошибки с помощью исключений:

Исключения basic_ios<C,Tr> (§38.4.4, §iso.27.5.5)	
st=ios.exceptions()	**st** есть **iostate** из **ios**
ios.exceptions(st)	Установить **iostate ios**'а равным **st**

Например, можно заставить **cin** генерировать исключение **basic_ios::failure**, когда его состояние устанавливается в **bad()** (например, посредством **cin.setstate(ios_base::badbit)**):

```
cin.exceptions(cin.exceptions()|ios_base::badbit);
```

Например:

```
struct Io_guard {    // дескрипторный класс для исключений iostream
    iostream& s;
    auto old_e = s.exceptions();
    Io_guard(iostream& ss, ios_base::iostate e) :s{ss} { s.exceptions(s.exceptions()|e); }
    ~Io_guard() { s.exceptions(old_e); }
};

void use(istream& is)
{
    Io_guard guard(is.ios_base::badbit);
    // ... используем is ...
}
catch (ios_base::badbit) {
    // ... бросаем это дело! ...
}
```

Я предпочитаю обрабатывать ошибки в **iostream** посредством исключений тогда, когда я думаю, что нет возможности их исправить. Обычно это все **bad()** исключения.

38.4. Операции ввода/вывода

Сложность операций ввода/вывода вытекает из традиций, из необходимости в высокой производительности и попытке отвечать ожиданиям пользователей. Здесь мы ведем изложение для традиционного набора строчных символов английского языка (ASCII). Способы работы с иными наборами символов и другими естественными языками рассматриваюися в Главе 39.

38.4.1. Операции ввода

Операции ввода предоставляется классом **istream** (§38.6.2), находящимся в файле **<istream>**, за исключением операций чтения в **string**; последние находятся в файле **<string>**. Класс **basic_istream** преимущественно предназначен на роль базового класса для более специфических классов ввода, таких как **istream** и **istringstream**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_istream : virtual public basic_ios<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;
    using off_type = typename Tr::off_type;
    using traits_type = Tr;

    explicit basic_istream(basic_streambuf<C,Tr>* sb);
    virtual ~basic_istream(); // освобождает все ресурсы

    class sentry;
    // ...
protected:
    // перемещение есть, копирования нет:
    basic_istream(const basic_istream& rhs) = delete;
    basic_istream(basic_istream&& rhs);
    basic_istream& operator=(const basic_istream& rhs) = delete;
    basic_istream& operator=(basic_istream&& rhs);
    // ...
};
```

Для пользователей **istream** класс **sentry** является деталью реализации. Он предоставляет общий код для стандартной библиотеки и пользовательских операций ввода. Код, который нужно выполнять с самого начала — вроде очистки и привязки потока — предоставляется конструктором класса **sentry**. Пример:

```
template<typename C, typename Tr = char_traits<C>>
basic_ostream<C,Tr>& basic_ostream<C,Tr>::operator<<(int i)
{
    sentry s {*this};
    if (!s) {    // проверяем, все ли в порядке для начала вывода
        setstate(failbit);
        return *this;
    }

    // ... выводим int ...
    return *this;
}
```

Класс **sentry** используется скорее реализаторами операций ввода, а не конечными пользователями.

38.4.1.1. Форматированный ввод

Форматированный ввод (преимущественно) предоставляется операцией **>>**:

Форматированный ввод (§iso.27.7.2.2, §iso.21.4.8.9)	
in>>x	Прочитать из **in** в **x** в соответствии с типом **x**; **x** может быть арифметическим типом, указателем, **basic_string**, **valarray**, **basic_streambuf** или любым типом, для к-рого пользователь предоставил подходящий **operator>>()**
getline(in,s)	Прочитать строку из **in** в **string s**

Встроенные типы известны классу **istream** (и **ostream**), так что если **x** имеет встроенный тип, то **cin>>x** означает **operator>>(cin,x)** (§18.2.5). Таким образом, ввод у **iostream** является чувствительным к типам, изначально безопасным по типам и расширяемым. Разработчик нового типа может предоставить операции ввода/вывода без необходимости в прямом доступе к реализации **iostream**.

Если целью операции **>>** служит указатель на функцию, то эта функция будет активироваться с **istream** в качестве аргумента. К примеру, **cin>>pf** равносильно **pf(cin)**. Это служит базисом для манипуляторов ввода, таких как **skipws** (§38.4.5.2). Манипуляторы потоков вывода используются чаще, чем манипуляторы потоков ввода. Поэтому эта техника будет пояснена позже в §38.4.3.

Если не утверждается обратное, операция **istream** возвращает ссылку на свой **istream**, так что можно объединять операции в цепочку:

```
template<typename T1, typename T2>
void read_pair(T1& x, T2& y)
{
    cin >> c1 >> x >> c2 >> y >> c3;
    if (c1!='{' || c2!=',' || c3!='}') {          // ошибка формата входных данных
        cin.setstate(ios_base::badbit);           // устанавливаем badbit
        throw runtime_error("bad read of pair");
    }
}
```

По умолчания операция **>>** пропускает пробельные символы. Пример:

```
for (int i; cin>>i && 0<i;)
cout << i << '\n';
```

Здесь получают последовательность разделенных пробельными символами положительных чисел и выводят по одному на строку.

Пропуск пробельных символов можно подавить с помощью **noskipws** (§38.4.5.2).

Операции ввода не являются виртуальными. Это означает, что пользователь не может писать **in>>base**, где **base** из классовой иерархии, и автоматически разрешить операцию **>>** к соответствующему производному классу. Однако, такое поведение можно достичь простыми приемами; см. §38.4.2.1. Более того, эту технику можно расширить на случаи чтения объектов по существу произвольных типов из входного потока; см. §22.2.4.

38.4.1.2. Неформатированный ввод

Неформатированный ввод можно использовать ради «тонкого» чтения и ради потенциального выигрыша в производительности. Одним из вариантов использования неформатированного ввода является реализация форматированного ввода:

Неформатированный ввод (§iso.27.7.2.3, §iso.27.7.2.3)	
x=in.get()	Прочитать один символ из **in** и вернуть его целочисленное значение; возвращает **EOF** в случае конца файла
in.get(c)	Прочитать символ из **in** в **c**
in.get(p,n,t)	Прочитать максимум **n** символов из **in** в [**p**:...); использ. **t** в качестве терминирующего символа
in.get(p,n)	**in.g et(p,n,'\n')**
in.getline(p,n,t)	Прочитать максимум **n** символов из **in** в [**p**:...); использ. **t** в качестве терминирующего символа; удалить терминирующий символ из **in**
in.getline(p,n)	**in.g etline(p,n,'\n')**
in.read(p,n)	Прочитать максимум **n** символов из **in** в [**p**:...)
x=in.gcount()	**x** есть число прочитанных символов самой последней операцией неформатированного ввода из **in**
in.putback(c)	Поместить **c** обратно в потоковый буфер **in**
in.unget()	Откатить потоковый буфер **in** на один символ, так, чтобы следующий прочитанный символ соответствовал предыдущему
in.ignore(n,d)	Извлекать символы из **in** и отбрасывать их, покуда либо **n** символов не будет отброшено или **d** не будет найден (и отброшен)
in.ignore(n)	**in.ignore(n,traits::eof())**
in.ignore()	**in.ignore(1,traits::eof())**
in.swap(in2)	Обменять значения **in** и **in2**

Если у вас есть выбор, используйте форматированный ввод (§38.4.1.1), а не эти низкоуровневые функции ввода.

Простая функция **get(c)** полезна, когда нужно составить значение из символов. Другая функция **get()** и функция **getline()** читает последовательность символов в область фиксированного размера [**p**:...). Чтение происходит до тех пор, пока не будет прочтено максимальное число символов или не встретится терминирующий символ (по умолчанию это **'\n'**). Они помещают **0** в конец символов; **getline()** устраняет терминирующий символ из входа, а **get()** этого не делает. Например:

```
void f() // старомодное низкоуровневое чтение строки
{
    char word[MAX_WORD][MAX_LINE]; // MAX_WORD массивов по MAX_LINE симв. каждый
    int i = 0;
    while(cin.getline(word[i++],MAX_LINE,'\n') && i<MAX_WORD)
        /* ничего не делать */ ;
    // ...
}
```

Для этих функций не сразу очевидно, что служит окончанием чтения:

- Найден терминирующий символ.
- Прочтено максимальное число символов.
- Встречен символ «конец файла».
- Произошла ошибка форматирования.

Последние два случая обрабатываются просмотром состояния файла (§38.3). Необходимые действия во всех этих случаях различны.

Функция **read(p,n)** не пишет **0** в массив после того, как прочтены символы. Ясно, что операции форматированного ввода проще использовать и они менее подвержены ошибкам, чем неформатированные операции.

Следующие функции зависят от деталей взаимодействия между буфером потока (§38.6) и реальным источником данных, и их нужно использовать лишь по необходимости и с осторожностью:

Неформатированный ввод (§iso.27.7.2.3)	
x=in.peek()	**x** есть текущий считываемый символ; **x** не извлекается из потокового буфера **in** и будет следующим считываемым символом
n=in.readsome(p,n)	Если **rdbuf()->in_avail()==-1**, вызвать **setstate(eofbit)**; иначе, прочитать **min(n,most rdbuf()->in_avail())** символов в **[p:...)**; **n** есть число прочитанных символов
x=in.sync()	Синхронизовать буферы: **in.rdbuf()->pubsync()**
pos=in.tellg()	**pos** есть позиция указателя чтения **in**
in.seekg(pos)	Переместить указатель чтения **in** на позицию **pos**
in.seekg(off,dir)	Переместить указатель чтения **in** на **off** символов в направлении **dir**

38.4.2. Операции вывода

Операции вывода относятся к **ostream** (§38.6.1), и находятся в файле **<ostream>**, за исключением тех, что пишут в **string**; последние находятся в **<string>**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_ostream : virtual public basic_ios<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;
    using off_type = typename Tr::off_type;
    using traits_type = Tr;

    explicit basic_ostream(basic_streambuf<char_type ,Tr>* sb);
    virtual ~basic_ostream(); // освобождает все ресурсы

    class sentry; // см. §38.4.1
    // ...
protected:
    // перемещение есть, копирования нет:
    basic_ostream(const basic_ostream& rhs) = delete;
    basic_ostream(basic_ostream&& rhs);
    basic_ostream& operator=(basic_ostream& rhs) = delete;
    basic_ostream& operator=(const basic_ostream&& rhs);
    // ...
};
```

ostream предоставляет форматированный вывод, неформатированный вывод (вывод символов) и простые операции над своим **streambuf** (§38.6):

Операции вывода (§iso.27.7.3.6, §iso.27.7.3.7, §iso.21.4.8.9)	
out<<x	Вывести **x** в **out** в соответствии с типом **x**; **x** может быть арифметическим типом, указателем, **basic_string**, **bitset**, **complex**, **valarray** или любым типом, для к-рого пользователь предоставил подходящий **operator<<()**
out.put(c)	Вывести символ **c** в **out**
out.write(p,n)	Вывести символы [**p**:**p**+**n**) в **out**
out.flush()	Очистить буфер, выведя по назначению
pos=out.tellp()	**pos** есть позиция указателя записи **out**
out.seekp(pos)	Переместить указатель записи **out** на позицию **pos**
out.seekp(off,dir)	Переместить указатель **out** на **off** символов в направлении **dir**

Если не оговорено обратное, операции **ostream** возвращают ссылку на свой **ostream**,так что можно собирать операции в цепочку. Например:

```
cout << "The value of x is " << x << '\n';
```

Заметьте, что величины типа **char** выводятся как символы, а не как короткие целые. Например:

```
void print_val(char ch)
{
    cout << "the value of '" << ch << "' is " << int{ch} << '\n';
}

void test()
{
    print_val('a');
    print_val('A');
}
```

Вывод этой программы таков:

```
the value of 'a' is 97
the value of 'A' is 65
```

Обычно довольно легко писать версии операции >> для пользовательских типов:

```
template<typename T>
struct Named_val {
    string name;
    T value;
};

ostream& operator<<(ostream& os, const Named_val& nv)
{
    return os << '{' << nv.name << ':' << nv.value << '}';
}
```

Это будет работать для любого **Named_val<X>**, где **X** располагает операцией <<. Ради общности, операция << должна быть определена для **basic_string<C,Tr>**.

38.4.2.1. Виртуальные функции вывода

Члены класса **ostream** не виртуальные. Операции, которые пользователь может добавить, не являются членами, так что они тоже не могут быть виртуальными. Причина этого

состоит в намерении добиться близкой к оптимальной производительности для простых операций вроде помещения символа в буфер. Здесь производительность критически важна, так что операции должны быть встраиваемыми. А виртуальные функции используются ради гибкости в операциях, имеющих дело с переполнением буферов (§38.6).

Однако, программисту иногда требуется вывести объект, для которого известен лишь базовый класс. Так как точный тип объекта неизвестен, то корректный вывод невозможно достичь лишь добавлением операции << для нового типа. Вместо этого, следует предоставить виртуальную функцию вывода в абстрактном базовом классе:

```
class My_base {
public:
    // ...
    virtual ostream& put(ostream& s) const = 0; // пишет *this в s
};

ostream& operator<<(ostream& s, const My_base& r)
{
    return r.put(s); // используем правильный put()
}
```

То есть, **put()** является виртуальной функцией, гарантирующей, что правильная операция вывода используется в <<.

Теперь мы можем написать:

```
class Sometype : public My_base {
public:
    // ...
    ostream& put(ostream& s) const override;    // настоящая функция вывода
};

void f(const My_base& r, Sometype& s) // используем <<, к-рый вызывает правильный put()
{
    cout << r << s;
}
```

Таким образом, виртуальная функция **put()** интегрируется в единое целое с **ostream** и <<. Эта техника используется для определения операций, действующих как виртуальные функции, но у которых выбор на этапе выполнения реализуется при помощи второго аргумента. Это аналогично технике, которая под именем *двойной диспетчеризации* (*double dispatch*) используется для выбора операции на основе двух динамических типов (§22.3.1). Аналогично можно действовать для того, чтобы сделать виртуальными операции ввода (§22.2.4).

38.4.3. Манипуляторы

Если указатель на функцию дается в качестве второго аргумента для <<, то вызывается указуемая функция. Например, **cout<<pf** означает **pf(cout)**. Такая функция называется *манипулятором*. Манипуляторы, принимающие аргументы, полезны на практике. Например:

```
cout << setprecision(4) << angle;
```

При этом выводится значение переменной **angle** типа с плавающей запятой с четырьмя цифрами. Для этого **setprecision** возвращает объект, инициализируемый значением **4**, и вызывающий **cout.precision(4)** при активировании. Такой манипулятор является объектом-функцией, который активизируется посредством <<, а не посредством **()**. Точный тип такого объекта-функции зависит от реализации, но он может быть определен следующим образом:

```
struct smanip {
    ios_base& (*f)(ios_base&,int);        // вызываемая функция
    int i;                                // используемое значение
    smanip(ios_base&(*ff)(ios_base&,int), int ii) :f{ff}, i{ii} { }
};

template<typename C, typename Tr>
basic_ostream<C,Tr>& operator<<(basic_ostream<C,Tr>& os, const smanip& m)
{
    m.f(os,m.i);    // вызываем метод f из m со значением, хранящимся в m
    return os;
}
```

Теперь мы можем определить **setprecision()** следующим образом:

```
inline smanip setprecision(int n)
{
    auto h = [](ios_base& s, int x) -> ios_base& { s.precision(x); return s; };
    return smanip(h,n);        // создает объект-функцию
}
```

Явная спецификация возвращаемого типа лямбды требуется при возврате ссылки. Пользователь не может копировать **ios_base**.

Мы теперь можем написать:

```
cout << setprecision(4) << angle;
```

Программист может определять новые манипуляторы в стиле **smanip**. При этом не требуется модификация определений шаблонов и классов стандартной библиотеки.

Манипуляторы стандартной библиотеки описываются в §38.4.5.2.

38.4.4. Состояние потока

В файле **<ios>** стандартная библиотека предоставляет базовый класс **ios_base**, определяющий большую часть интерфейса к потоковым классам:

```
template<typename C, typename Tr = char_traits<C>>
class basic_ios : public ios_base {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;
    using off_type = Tr::off_type;
    using traits_type = Tr;
    // ...
};
```

Класс **basic_ios** контролирует состояние потока, реализуя следующее:

- Соответствие между потоком и его буферами (§38.6)
- Опции форматирования (§38.4.5.1)
- Использование **locale** (Глава 39)
- Обработка ошибок (§38.3)
- Связь с другими потоками и стандартными средствами ввода/вывода языка C (§38.4.4)

Это, возможно, самый сложный класс в стандартной библиотеке.

Класс **ios_base** содержит информацию, не зависящую от шаблонных аргументов:

```
class ios_base {
public:
    using fmtflags = /* тип, зависящий от реализации */;
    using iostate = /* тип, зависящий от реализации */;
    using openmode = /* тип, зависящий от реализации */;
    using seekdir = /* тип, зависящий от реализации */;

    class failure;  // класс для обработки исключений
    class Init;     // инициализирует стандартные iostream'ы
};
```

Типы, зависящие от реализации, являются типами битовых масок; они поддерживают побитовые логические операции вроде **&** и **|**. Примерами служат типы **int** (§11.1.2) и **bitset** (§34.2.2).

Класс **ios_base** отвечает за связь **iostream** с **stdio** (стандартный ввод/вывод языка С) (§43.3):

Основные функции ios_base (§iso.27.5.3.4)	
ios_base b {};	Конструктор по умолчанию; protected
ios.~ios_base()	Деструктор; virtual
b2=sync_with_stdio(b)	Если **b==true** синхронизирует **ios** с stdio; в противном случае, разделяемые буферы могут быть испорчены; **b** содержит предыдущее состояние синхронизации; static
b=sync_with_stdio()	**b=sync_with_stdio(true)**

Вызов **sync_with_stdio(true)** перед первой операцией с **iostream** гарантирует, что операции ввода/вывода **iostream** и **stdio** (§43.3) будут разделять одни и те же буферы. Вызов **sync_with_stdio(false)** перед первой операцией с **iostream** предотвращает разделение буферов, а также может существенно повысить производительность на некоторых платформах.

Отметим, что **ios_base** не имеет копирующих или перемещающих операций.

Константы iostate состояния потока ios_base (§iso.27.5.3.1.3)	
badbit	Произошло нечто неожиданное и серьезное (напр., ошибка чтения с диска)
failbit	Произошло нечто неожиданное (напр., искали цифру, а нашли '**x**')
eofbit	Достигнут конец ввода (напр., конец файла)
goodbit	Все хорошо

Функции для чтения битов (**good()**, **fail()** и т. д.) в потоке предоставляются классом **basic_ios**.

Константы openmode режима открытия ios_base (§iso.27.5.3.1.4)	
app	Добавление (добавить вывод в конец потока)
ate	В конец (переместить в конец потока)
binary	Не применять форматирование к символам
in	Поток ввода
out	Поток вывода
trunc	Урезать поток перед использованием (установить размер потока в ноль)

Точный смысл **ios_base::binary** зависит от реализации. Но чаще всего это означает, что символы маппируются (отображаются) в байт. Например:

```
template<typename T>
char* as_bytes(T& i)
{
    return static_cast<char*>(&i); // рассматриваем эти данные, как байты
}

void test()
{
    ifstream ifs("source",ios_base::binary); // поток в бинарном режиме
    ofstream ofs("target",ios_base::binary); // поток в бинарном режиме

    vector<int> v;

    for (int i; ifs.read(as_bytes(i),sizeof(i));) // читаем байты из бинарного файла
        v.push_back(i);

    // ... делаем что-нибудь с v ...

    for (auto i : v)                              // пишем байты в бинарный файл:
        ofs.write(as_bytes(i),sizeof(i));
}
```

Используйте бинарный ввод/вывод в случае, когда объекты представляются просто «мешком с битами» и не имеют очевидного и разумного представления в виде строки символов. Потоки звука и видео служат примерами.

Для **seekg()** (§38.6.2) и **seekp()** (§38.6.2) требуют указать направление:

Константы seekdir позиционирования ios_base (§iso.27.5.3.1.5)	
beg	От начала текущего файла
cur	От текущей позиции
end	С конца файла в обратном направлении

Классы, производные от **basic_ios**, форматируют вывод и извлекают объекты, базируясь на информации, хранимой в их **basic_io**.

Можно суммировать операции **ios_base**:

basic_ios<C,Tr> (§iso.27.5.5)	
basic_ios ios {p};	Создать **ios** с буфером потока **p**
ios.~basic_ios()	Уничтожить **ios**: освобождает все ресурсы **ios**
bool b {ios};	Преобразование в **bool**: **b** устанавливается равным **!ios.fail()**; explicit
b=!ios	**b=ios.fail()**
st=ios.rdstate()	**st** принимает знач-е **iostate** из **ios**
ios.clear(st)	Установить **iostate** из **ios** равным **st**
ios.clear()	Привести **iostate** из **ios** к состоянию good
ios.setstate(st)	Добавить **st** к **ios** из **iostate**
ios.good()	**ios** в порядке (установлен **goodbit**)?
ios.eof()	**ios** в состоянии end-of-file?
ios.fail()	**ios** в состоянии fail?
ios.bad()	**ios** в состоянии bad?
st=ios.exceptions()	**st** – биты исключений **iostate** из **ios**
ios.exceptions(st)	Установить биты исключений **ios** из **iostate** равными **st**
p=ios.tie()	**p** указывает на ранее привязанный поток или **nullptr**
p=ios.tie(os)	Связать выходной поток **os** с **ios**; **p** указывает на ранее привязанный поток или **nullptr**

Продолжение

basic_ios<C,Tr> (§iso.27.5.5)	
p=ios.rdbuf()	pis указывает на буфер потока ios
p=ios.rdbuf(p2)	Назначает ios тот буфер потока, на к-рый указывет p2; p указывает на предыдущий буфер потока
ios3=ios. copyfmt(ios2)	Копирует состояние форматирования из io2 в ios; вызывает любую callback-функцию из ios2 типа copyfmt_event; копирует значения, на к-рые указывают ios2.pword и ios2.iword; ios3 есть предыдущее состояние форматирования
c=ios.fill()	c есть заполняющий символ ios
c2=ios.fill(c)	Установить c в кач-ве заполняющего символа ios; c2 есть предыдущий заполняющий символ
loc2=ios.imbue(loc)	Установить ios в кач-ве локализации loc; loc2 есть предыдущая локализация
c2=narrow(c,d)	c2 есть значение типа char получаемое из c типа char_type, d есть значение по умолчанию: use_facet<ctype<char_type>>(g etloc()).narrow(c,d))
c2=widen(c)	c2 есть значение типа char_type получаемое из c типа char type: use_facet<ctype<char_type>>(g etloc()).widen(c))
ios.init(p)	Вернуть ios в состояние по умолчанию и использовать буфер потока, указываемый p; protected
ios.set_rdbuf(p)	Устанавливает для ios буфер потока, указываемый p; protected
ios.move(ios2)	Операция копирования и замещения; protected
ios.swap(ios2)	Обменивает состояния ios и ios2; protected; noexcept

Конвертирование ios (включая istream и ostream) в bool существенно для обычной идиомы чтения многих значений:

```
for (X x; cin>>x;) {
    // ...
}
```

Здесь возвратом у cin>>x служит ссылка на ios (член cin). А ios неявно конвертируется в bool, представляя состояние cin. Таким образом, мы можем эквивалентно написать:

```
for (X x; !(cin>>x).fail();) {
    // ...
}
```

Здесь tie() используется для того, чтобы гарантировать, что вывод от связанного потока появляется до ввода от потока, к которому он привязан. Например, cout связан с cin:

```
cout << "Please enter a number: ";
int num;
cin >> num;
```

Этот код явным образом не вызывает cout.flush(), так что если бы cout не был связан с cin, то пользователь увидел бы приглашение ко вводу.

Операции ios_base (§iso.27.5.3.5, §iso.27.5.3.6)	
i=xalloc()	i есть индекс новой пары (iword,pword); static
r=iob.iword(i)	r есть ссылка на i-ый long
r=iob.pword(i)	r есть ссылка на i-ый void*
iob.register_callback(fn,i)	Регистрировать ф-ию обратного вызова fn к iword(i)

Иногда нужно кое-что добавить к состоянию потока. Например, может потребоваться оповестить поток о том, как выводить тип **complex** — в полярных или декартовых координатах? В классе **ios_base** есть функция **xalloc()** для выделения памяти под такую несложную информацию в потоке. Возвращаемое при этом значение идентифицирует пару областей, к которым можно выполнить доступ по **iword()** и **pword()**.

Иногда нужно, чтобы изменения в состоянии потока получал пользователь или реализатор. Функция **register_callback()** регистрирует функцию, которая будет вызывана по соответствующему событию. Таким образом, обращение к **imbue()**, **copyfmt()** или **~ios_base()** вызовут функции, зарегистрированные для **imbue_event**, **copyfmt_event** или **erase_event**, соответственно. При изменении состояния зарегистрированные функции вызываются с аргументом **i**, предоставляемым их **register_callback()**.

Типы **event** и **event_callback** определяются в **ios_base**:

```
enum event {
      erase_event,
      imbue_event,
      copyfmt_event
};
using event_callback = void (*)(event, ios_base&, int index);
```

38.4.5. Форматирование

Формат потокового ввода/вывода управляется (контролируется) комбинацией типа объекта, состояния потока (§38.4.4), состояния форматирования (§38.4.5.1), информации локализации (Глава 39) и явными операциями (например, манипуляторами; §38.4.5.2)

38.4.5.1. Состояние форматирования

В файле **<ios>** стандартная библиотека определяет набор констант форматирования, имеющих тип битовых масок (зависят от реализации), и являющихся членами класса **ios_base**:

Константы **fmtflags** форматирования **ios_base** (§iso.27.5.3.1.2)	
boolalpha	Использовать символьное представление **true** и **false**
dec	Целые по основанию 10
hex	Целые по основанию 16
oct	Целые по основанию 8
fixed	Формат чисел с плавающей запятой dddd.dd
scientific	Формат чисел с плавающей запятой d.ddddEdd
internal	Заполнитель между префиксом (таким как **+**) и числом
left	Заполнитель после значения
right	Заполнитель перед значением
showbase	Перед восьмеричными числами выводить **0**, а перед шестнадцатеричными выводить **0x**
showpoint	Всегда показывать десятичную запятую (напр., **123.**)
showpos	Показывать **+** для положительных чисел (напр., **+123**)
skipws	Пропускать пробельные символы при вводе
unitbuf	Сбрасывать буфер после каждой операции вывода
uppercase	Использовать верхний регистр при выводе чисел, напр., **1.2E10** и **0X1A2**
adjustfield	Установить расположение значения: **left**, **right** или **internal**
basefield	Установить основание целых чисел: dec, oct или hex
floatfield	Установить формат чисел с плавающей запятой: scientific или fixed

Как ни странно, но нет флагов **defaultfloat** и **hexfloat**. Для получения эквивалентов используйте манипуляторы **defaultfloat** и **hexfloat** (§38.4.5.2) или напрямую манипулируйте **ios_base**:

```
ios.unsetf(ios_base::floatfield);      // использовать формат по умолчанию
ios.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield);
                                       // использовать шестнадцатеричное представление
```

Состояние форматирования **iostream** можно читать и писать операциями, предоставляемыми в его **ios_base**:

Операции над флагами fmtflags форматирования ios_base (§iso.27.5.3.2)	
f=ios.flags()	**f** есть флаги форматирования **ios**
f2=ios.flags(f)	Присвоить флагам **ios** значение **f**; **f2** – старое значение флагов
f2=ios.setf(f)	Присвоить флагам **ios** значение **f**; **f2** – старое значение флагов
f2=ios.setf(f,m)	**f2=ios.setf(f&m)**
ios.unsetf(f)	Очистить флаги **f** в **ios**
n=ios.precision()	**n** есть точность (число цифр после запятой) **ios**
n2=ios.precision(n)	Установить точность **ios** равной **n**; **n2** есть старая точность
n=ios.width()	**n** есть ширина поля **ios**
n2=ios.width(n)	Установить ширину поля **ios** равной **n**; **n2** есть старая ширина поля

Точность это целое значение, определяющее число цифр, используемых для отображения числа с плавающей запятой;

- *Общий формат* (**defaultfloat**) позволяет реализации выбрать формат представления значения в стиле, который лучше всего сохраняет значение в имеющемся для этого объеме памяти. Точность влияет на максимальное число цифр числа.
- *Научный формат* (**scientific**) представляет значение с одной цифрой перед запятой и с экспонентой (показателем степени). Точность задает максимальное число цифр после десятичной запятой (в смысле, точки).
- *Фиксированный формат* (**fixed**) представляет значение в виде целой части, за которой идет десятичная точка и дробная часть. Точность специфицирует максимальное число цифр после десятичной точки. Пример см. в §38.4.5.2.

Значения с плавающей запятой округляются, а не усекаются, и **precision()** не влияет на вывод целых. Пример:

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

Получается такой вывод:

```
1234.5679 1234.5679 123456
1235 1235 123456
```

Функция **width()** специфицирует минимальное количество символов для последующей операции **<<** стандартной библиотеки с числовыми значениями, **bool**, строками в С-стиле, символами, указателями, **string** и **bitset** (§34.2.2). Например:

```
cout.width(4);
cout << 12;    // выводит 12, предваряемые двумя пробелами
```

Заполняющий символ может уточняться с помощью функции **fill()**. Пример:

```
cout.width(4);
cout.fill('#');
cout << "ab"; // выводит ##ab
```

Умолчательным заполняющим символом служит пробел, а умолчательным значением размера поля служит **0**, означающий «столько символом, сколько надо». Размер поля можно вернуть к умолчательному значению следующим образом:

```
cout.width(0); // "столько символов, сколько надо"
```

Вызов **width(n)** устанавливает минимальное число символов, равным **n**. Если предоставляется большее число символов, то все они выводятся:

```
cout.width(4);
cout << "abcdef";   // выводит abcdef
```

При этом вывод не усекается до **abcd**. Лучше получить правильный вывод, пусть и некрасивый, чем красивый, но неправильный.

Вызов **width(n)** влияет только на один последующий вывод операцией **<<**. Пример:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;        // выводит ##12:13
```

Здесь на выводе получается **##12:13**, а не **##12###:##13**.

Если становится утомительным явный контроль посредством форматирующих опций для многих отдельных операций, то можно скомбинировать их в пользовательском манипуляторе (§38.4.5.3).

Класс **ios_base** разрешает пользователю установить **locale** для **iostream** (Глава 39):

Операции с locale в ios_base (§iso.27.5.3.3)	
loc2=ios.imbue(loc)	Установить **loc**, как локализацию **ios**; **loc2** есть старая локализация
loc=ios.getloc()	**loc** есть локализация **ios**

38.4.5.2. Стандартные манипуляторы

Стандартная библиотека предоставляет манипуляторы, соответствующие различным состояниям форматирования и изменениям состояний. Стандартные манипуляторы определяются в файлах **<ios>**, **<istream>**, **<ostream>** и **<iomanip>** (для манипуляторов с аргументами):

Манипуляторы ввода/вывода из <ios> (§iso.27.5.6, §iso.27.7.4)	
s<<boolalpha	Использовать символьное представление **true** и **false** (ввод и вывод)
s<<noboolalpha	**s.unsetf(ios_base::boolalpha)**
s<<showbase	Перед восьмеричными числами выводить **0**, а перед шестнадцатеричными выводить **0x**
s<<noshowbase	**s.unsetf(ios_base::showbase)**
s<<showpoint	Всегда показывать десятичную запятую
s<<noshowpoint	**s.unsetf(ios_base::showpoint)**
s<<showpos	Показывать **+** для положительных чисел
s<<noshowpos	**s.unsetf(ios_base::showpos)**
s<<uppercase	Использовать верхний регистр при выводе чисел, напр., **1.2E10** и **0X1A2**
s<<nouppercase	Использовать нижний регистр при выводе чисел, напр., **1.2e10** и **0x1a2**

Продолжение

Манипуляторы ввода/вывода из <ios> (§iso.27.5.6, §iso.27.7.4)	
s<<unitbuf	Сбрасывать буфер после каждой операции вывода
s<<nounitbuf	Не сбрасывать буфер после каждой операции вывода
s<<internal	Заполнитель между знаком и числовым значением
s<<left	Заполнитель после значения
s<<right	Заполнитель перед значением
s<<dec	Целые по основанию 10
s<<hex	Целые по основанию 16
s<<oct	Целые по основанию 8
s<<fixed	Формат чисел с плавающей запятой dddd.dd
s<<scientific	Формат чисел с плавающей запятой d.ddddEdd
s<<hexfloat	Использовать основание 16 для мантиссы и экспоненты, символ **p** означает начало экспоненты, напр., **A.1BEp-C** и **a.bcdef**
s<<defaultfloat	Формат чисел с плавающей запятой по умолчанию
s>>skipws	Пропускать пробельные символы
s>>noskipws	**s.unsetf(ios_base::skipws)**

Эти операции возвращают ссылку на их первый аргумент (поток), **s**. Например:

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n'; // выводит 1234,4d2,2322
```

Можно явно установить формат вывода для чисел с плавающей запятой:

```
constexpr double d = 123.456;

cout << d << "; "
     << scientific << d << "; "
     << hexfloat << d << "; "
     << fixed << d << "; "
     << defaultfloat << d << '\n';
```

Отсюда получается такой вывод:

```
123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456
```

Формат для чисел с плавающей запятой распространяется на последующие операции с плавающей запятой:

Манипуляторы ввода/вывода из <ostream> (§iso.27.5.6, §iso.27.7.4)	
os<<endl	Вывести **'\n'** и очистить
os<<ends	Вывести **'\0'**
os<<flush	Очистить поток

Перед уничтожением **ostream** он очищается, когда **istream** у **tie()** требует ввода (§38.4.4), и когда реализация считает его желательным. Редко когда требуется явная очистка потока. Аналогично, **<<endl** можно рассматривать эквивалентным **<<'\n'**, но последний вариант (скорее всего) более эффективен. Я обнаружил, что

```
cout << "Hello, World!\n";
```

и пишется, и читается проще, чем

```
cout << "Hello, World!" << endl;
```

Если вам на самом деле нужна частая очистка, то рассмотрите применение **cerr** и **unitbuf**.

Манипуляторы ввода/вывода из <iomanip> (§iso.27.5.6, §iso.27.7.4)	
s<<resetiosflags(f)	Очистить флаги **f**
s<<setiosflags(f)	Установить флаги **f**
s<<setbase(b)	Выводить целые по основанию **b**
s<<setfill(int c)	Сделать **c** заполняющим символом
s<<setprecision(n)	Точность **n** символов
s<<setw(n)	Размер следующего поля **n** символов
is>>get_money(m,intl)	Читать из **is**, используя фасет **money_g** взятый из **is**; **m** имеет тип **long double** или **basic_string**; если **intl==true**, использовать стандартные трехсимвольные названия валют
is>>get_money(m)	**s>>get_money(m,false)**
os<<put_money(m,intl)	Вывести **m** в **os**, используя фасет **money_put**, взятый из **os**; этот **money_put** определяет приемлемые для **m** типы; если **intl==true**, использовать стандартные трехсимвольные названия валют
os<<put_money(m)	**s<<put_money(m,false)**
is>>get_time(tmp,fmt)	Читать в ***tm** в соответствии с форматом **fmt**, используя фасет **time_put**, взятый из **is**
os<<put_time(tmp,fmt)	Вывести ***tm** в **os** в соответствии с форматом **fmt**, используя фасет **time_put**, взятый из **os**

Фасеты времени рассматриваются в §39.4.4, а форматы времени — в §43.6. Пример:

```
cout << '(' << setw(4) << setfill('#') << 12 << ") (" << 12 << ")\n"; // выводит (##12) (12)
```

Манипуляторы istream (§iso.27.5.6, §iso.27.7.4)	
s>>skipws	Пропускать пробельные символы (в **<ios>**)
s>>noskipws	**s.unsetf(ios_base::skipws)** (в **<ios>**)
is>>ws	Съедать пробельные символы (в **<istream>**)

По умолчанию **>>** пропускает пробелы (§38.4.1). Это умолчание можно модифицировать посредством **>>skipws** и **>>noskipws**. Например:

```
string input {"0 1 2 3 4"};
istringstream iss {input};
string s;
for (char ch; iss>>ch;)
    s += ch;
cout << s;      // выводит "01234"

istringstream iss2 {input};
iss>>noskipws;
for (char ch; iss2>>ch;)
    s += ch;
cout << s;      // выводит "0 1 2 3 4"
}
```

Если вы хотите явным образом разбираться с пробелами и по-прежнему использовать **>>**, то помогут **noskipws** и **>>ws**.

38.4.5.3. Пользовательские манипуляторы

Пользователь может определять манипуляторы в стиле стандартных манипуляторов. Здесь я рассмотрю еще один стиль, который мне кажется полезным для форматирования чисел с плавающей запятой.

Форматирование контролируется целой кучей функций (§38.4.5.1). К примеру, влияние **precision()** сохраняется для всех операций вывода, а **width()** относится только к одной последующей операции вывода. Что мне нужно, так это то, что может упростить вывод чисел с плавающей запятой предустановленного формата без влияния на будущие операции с потоком. Основная идея состоит в том, что следует определить класс, представляющий форматы, а затем операцию <<, которая выводит значение в **ostream** согласно формату. Например:

```
Form gen4 {4};      // общий формат, точность 4

void f(double d)
{
    Form sci8;
    sci8.scientific().precision(8);          // научный формат, точность 8
    cout << d << ' ' << gen4(d) << ' ' << sci8(d) << ' ' << d << '\n';

    Form sci {10,ios_base::scientific}; // научный формат, точность 10
    cout << d << ' ' << gen4(d) << ' ' << sci(d) << ' ' << d << '\n';
}
```

Вызов **f(1234.56789)** осуществит следующий вывод:

```
1234.57 1235 1.23456789e+003 1234.57
1234.57 1235 1.2345678900e+003 1234.57
```

Заметьте, что использование **Form** не влияет на состояние потока, так что последняя операция над **d** имеет тот же формат, что и первая операция.

А вот упрощенная реализация:

```
class Form;    // наш тип форматирования

struct Bound_form {      // Form плюс значение
    const Form& f;
    double val;
};

class Form {
    friend ostream& operator<<(ostream&, const Bound_form&);

    int prc;    // точность
    int wdt;    // ширина 0 значит "столько символов, сколько надо"
    int fmt;    // общий, научный или фиксированный (§38.4.5.1)
    // ...

public:
    explicit Form(int p =6, ios_base::fmtflags f =0, int w =0) : prc{p}, fmt{f}, wdt{w} {}

    Bound_form Form::operator()(double d) const       // создаем Bound_for m для *this и d
    {
        return Bound_form{*this,d};
    }

    Form& scientific() { fmt = ios_base::scientific; return *this; }
    Form& fixed() { fmt = ios_base::fixed; return *this; }
    Form& general() { fmt = 0; return *this; }

    Form& uppercase();
    Form& lowercase();
    Form& precision(int p) { prc = p; return *this; }
```

```
    Form& width(int w) { wdt = w; return *this; }     // применяется ко всем типам
    Form& fill(char);

    Form& plus(bool b = true);              // явный плюс
    Form& trailing_zeros(bool b = true);    // выводить хвостовые нули
    // ...
};
```

Идея в том, что **Form** содержит всю информацию, нужную для форматирования одного элемента данных. Умолчание выбрано разумным для многих случаев, а разные члены-функции устанавливают отдельные отличительные аспекты форматирования. Операция () используется для привязки значения к формату, который будет использоваться для его вывода. Тогда **Bound_form** (то есть, **Form** плюс значение) может затем выводиться в заданный поток подходящей операцией <<:

```
ostream& operator<<(ostream& os, const Bound_form& bf)
{
    ostringstream s;          // §38.2.2
    s.precision(bf.f.prc);
    s.setf(bf.f.fmt,ios_base::floatfield);
    s << bf.val;              // составляем строку в s
    return os << s.str();     // выводим в os
}
```

В качестве упраженения можете написать не такую уж упрощенную версию.

Заметьте, что эти объявления превращают комбинацию << и () в тернарную операцию; **cout<<sci4{d}** собирает в единую функцию **ostream**, формат и значение перед выполнением реального вычисления.

38.5. Итераторы потоков

В файле **<iterator>** стандартная библиотека предоставляет итераторы, позволяющие трактовать потоки ввода и вывода как последовательности [input:begin:end-of-input) и [output-begin:end-of-output):

```
template<typename T,
    typename C = char,
    typename Tr = char_traits<C>,
    typename Distance = ptrdiff_t>
class istream_iterator
    :public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
    using char_type = C;
    using traits_type = Tr;
    using istream_type = basic_istream<C,Tr>;
    // ...
};

template<typename T, typename C = char, typename Tr = char_traits<C>>
class ostream_iterator: public iterator<output_iterator_tag, void, void, void, void> {
    using char_type = C;
    using traits_type = Tr;
    using ostream_type = basic_ostream<C,Tr>;
    // ...
};
```

Например:

```
copy(istream_iterator<double>{cin}, istream_iterator<double,char>{},
    ostream_iterator<double>{cout,";\n"});
```

Когда **ostream_iterator** конструируется со вторым аргументом-строкой (типа **string**), то эта строка выводится как терминирующая после каждого значения элемента. Так что, если вы предоставляете **1 2 3** для вызова **call()**, то вывод будет следующим:

```
1;
2;
3;
```

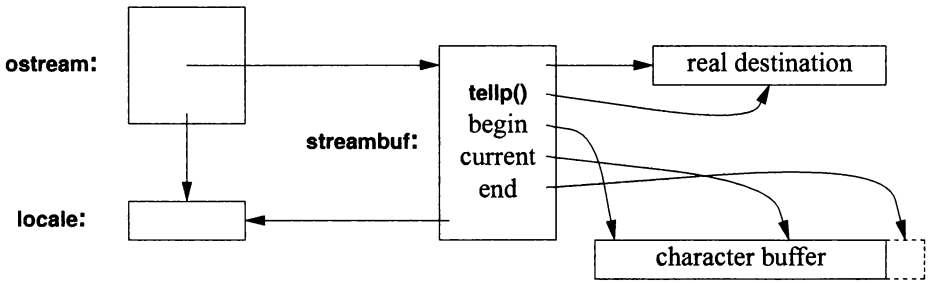
Операции для **iterator_stream** такие же, как и для других итераторных адапторов (§33.2.2):

Функции итераторов потоков (§iso.24.6)	
istream_iterator p {st};	Итератор для потока ввода **st**
istream_iterator p {p2};	Копирующий конструктор: **p** есть копия **istream_iterator p2**
ostream_iterator p {st};	Итератор для потока вывода **st**
ostream_iterator p {p2};	Копирующий конструктор: **p** есть копия **ostream_iterator p2**
ostream_iterator p {st,s};	Итератор для потока ввода **st**; использует С-строку **s** как разделитель между эл-тами вывода
p=p2	**p** есть копия **p2**
p2=++p	**p** и **p2** указывают на следующий эл-т
p2=p++	**p2=p,++p**
***p=x**	Вставить **x** перед **p**
***p++=x**	Вставить **x** перед **p**, затем инкрементировать

За исключением конструкторов, эти операции используются в основном обобщенными алгоритмами, такими как **copy()**, а не напрямую.

38.6. Буферизация

Концептуально, поток вывода кладет символы в буфер. Когда-нибудь позже, символы будут переписаны туда, куда надо. Такой буфер называется **streambuf**. Его определение находится в файле **<streambuf>**. Разные варианты **streambuf** реализуют разные стратегии буферизации. В типичном случае **streambuf** хранит символы в массиве до тех пор, пока не замаячит переполнение, после чего символы будут переписаны в другое место по назначению. Таким образом, **ostream** можно графически отобразить следующим образом:



Набор шаблонных аргументов для **ostream** и его **streambuf** должен быть одинаковым, и они определяют тип символов, используемых в символьном буфере.

Случай **istream** такой же, за исключением того, что символы следуют в обратном направлении.

Небуферированный ввод/вывод есть буферированный ввод/вывод, в котором **streambuf** немедленно передает каждый символ далее по назначению, а не хранит символы вплоть до набора достаточного их числа в плане эффективной передачи по назначению.

Ключевым классом в механизме буферирования является класс **basic_streambuf**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_streambuf {
public:
    using char_type = C;                          // тип символа
    using int_type = typename Tr::int_type;       // целый тип в который
                                                  // может быть преобразован символ
    using pos_type = typename Tr::pos_type;       // для позиций в буфере
    using off_type = typename Tr::off_type;       // для смещений в буфере
    using traits_type = Tr;
    // ...
    virtual ~basic_streambuf();
};
```

Как обычно, пара синонимов предоставляется для наиболее общих случаев:

```
using streambuf = basic_streambuf<char>;
using wstreambuf = basic_streambuf<wchar_t>;
```

Многие из открытых (**public**) операций класса **basic_streambuf** просто обращаются к защищенным (**protected**) виртуальным функциям, что гарантирует правильную реализацию функций в производных классах для конкретного вида буфера:

Открытые (public) операции basic_streambuf<C,Tr> (§iso.27.6.3)	
sb.~basic_streambuf()	Деструктор: освободить все ресурсы; virtual
loc=sb.getloc()	**loc** есть локализация **sb**
loc2=sb.pubimbue(loc)	**sb.imbue(loc)**; **loc2** – указатель на предыд. локализацию
psb=sb.pubsetbuf(s,n)	**psb=sb.setbuf(s,n)**
pos=sb.pubseekoff(n,w,m)	**pos=sb.seekoff(n,w,m)**
pos=sb.pubseekoff(n,w)	**pos=sb.seekoff(n,w)**
pos=sb.pubseekpos(n,m)	**pos=sb.seekpos(n,m)**
pos=sb.pubseekpos(n)	**pos=sb.seekpos(n,los_base::inlios_base::out)**
sb.pubsync()	**sb.sync()**

Все конструкторы защищенные, потому что **basic_streambuf** проектировался как базовый класс.

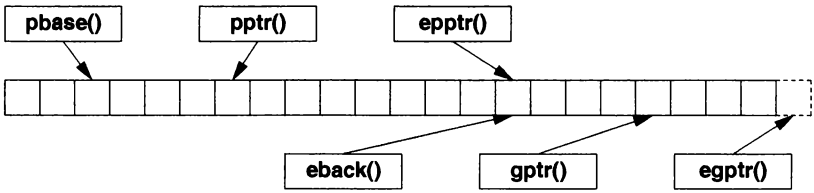
Защищенные (protected) операции protected basic_streambuf<C,Tr> (§iso.27.6.3)	
basic_streambuf sb {};	Создать **sb** без символьного буфера и с глобальной локализацией
basic_streambuf sb {sb2};	**sb** есть копия **sb2** (у них общий символьный буфер)
sb=sb2	**sb** есть копия **sb2** (у них общий символьный буфер); старые ресурсы **sb** освобождаются
sb.swap(sb2)	Обменять состояния **sb** и **sb2**
sb.imbue(loc)	**loc** становится локализацией **sb**; virtual
psb=sb.setbuf(s,n)	Установить буфер **sb**; **psb=&sb**; **s** есть **const char***, **n** есть **streamsize**; virtual
pos=sb.seekoff(n,w,m)	Позиционировать с отступом **n**, направлением **w** и режимом **m**; **pos** есть получившаяся позиция или **pos_type(off_type(-1))**, что обозначает ошибку; virtual

Продолжение

Защищенные (protected) операции protected basic_streambuf<C,Tr> (§iso.27.6.3)	
pos=sb.seekoff(n,w)	pos=sb.seekoff(n,way,ios_base::inlios_base::out)
pos=sb.seekpos(n,m)	Переместить на позицию n в режиме m; pos есть получившаяся позиция или pos_type(off_type(-1)), что обозначает ошибку; virtual
n=sb.sync()	Синхронизировать символьные буферы с реальным назначением или источником; virtual

Конкретный смысл виртуальных функций формируется производными классами.

Класс **streambuf** имеет выходную область памяти (*put area*), в которую операция **<<** и другие функции вывода пишут данные (§38.4.2), а также входную область памяти (*get area*), из которой операция **>>** и другие функции ввода читают данные (§38.4.1). Каждая область описывается начальным указателем, текущим указателем и указателем на байт после конца области:



Переполнение обрабатывается виртуальными функциями **overflow()**, **underflow()** и **uflow()**.

Насчет позиционирования смотри §38.6.1.

Интерфейсы put и get разделены по **public** и **protected** секциям:

Открытые функции интерфейсов Put и Get basic_streambuf<C,Tr> (§iso.27.6.3)	
n=sb.in_avail()	Если ввод готов, n=sb.egptr()-sb.gptr(); иначе возвращает sb.showmanyc()
c=sb.snextc()	Увеличить указатель чтения sb, затем c=*sb.gptr()
n=sb.sbumpc()	Увеличить указатель чтения sb
c=sb.sgetc()	Если символов для чтения не осталось, c=sb.underflow(); иначе c=*sb.gptr()
n=sb.sgetn(p,n)	n=sb.xsg etn(p,n); p есть char*
n=sb.sputbackc(c)	Поместить c обратно в буфер и уменьшить gptr; n=Tr::to_int_type(*sb.gptr()), если символ помещен успешно; иначе n=sb.pbackfail(Tr::to_int_type(c))
n=sb.sungetc()	Уменьшить указатель чтения; n=Tr::to_int_type(*sb.gptr()) если чтение отменено успешно; иначе n=sb.pbackfail(Tr::to_int_type())
n=sb.sputc(c)	Если места в буфере нет, n=sb.overflow(Tr::to_int_type(c)); иначе *sb.sptr()=c; n=Tr::to_int_type(c)
n=sb.sputn(s,n)	n=sb.xsputn(s,n); s есть const char*

Защищенный интерфейс предоставляет простые, эффективные и типично встраиваемые функции, манипулирующие put и get указателями. Кроме того, там есть виртуальные функции, подлежащие замещению в производных классах.

Защищенные функции интерфейсов Put и Get basic_streambuf<C,Tr> (§iso.27.6.3)	
sb.setg(b,n,end)	Область чтения [b,e); текущий указатель чтения n
pc=sb.eback()	[pc:sb.egptr()) есть область чтения
pc=sb.gptr()	pc есть указатель чтения
pc=sb.egptr()	[sb.eback():pc) есть область чтения
sb.gbump(n)	Увеличить указатель чтения sb
n=sb.showmanyc()	«Показать, сколько осталось символов»; n есть оценка кол-ва символов, к-рые можно прочитать без вызова sb.underflow() или n=-1 что означает, что символов для чтения нет; virtual
n=sb.underflow()	В области чтения нет больше символов; наполнить область чтения; n=Tr::to_int_type(c), где c есть новый текущий символ; virtual
n=sb.uflow()	Как sb.underflow(), но передвигает указатель после чтения нового текущего символа; virtual
n=sb.pbackfail(c)	Операция возвращения символа провалилась; n=Tr::eof(), если замещающий pbackfail() не смог вернуть символ; virtual
n=sb.pbackfail()	n=sb.pbackfail(Tr::eof())
sb.setp(b,e)	Область записи [b,e) текущий указатель записи b
pc=sb.pbase()	[pc:sb.epptr()) есть область записи
pc=sb.pptr()	pc есть указатель записи
pc=sb.epptr()	[sb.pbase(),pc) есть область записи
sb.pbump(n)	Добавить 1 к указателю записи
n2=sb.xsgetn(s,n)	s есть const char*; выполнять sb.sgetc(*p) ля каждого p в [s:s+n); n2 есть кол-во прочитанных символов; virtual
n2=sb.xsputn(s,n)	s есть [const char*; sb.sputc(*p) для каждого p в [s:s+n); n2 есть кол-во записанных символов; virtual
n=sb.overflow(c)	Сбросить область записи, затем n=sb.sputc(c); virtual
n=sb.overflow()	n=sb.overflow(Tr::eof())

Функция **showmanyc()** — это странная функция, предназначенная для того, чтобы позволить пользователю узнать что-либо про состояние машинной системы ввода. Она возвращает оценку количества символов, которые «вскоре» могут быть прочитаны, скажем из-за очистки системных буферов, а не из-за ожидания чтения с диска. Функция **showmanyc()** возвращает **-1**, если она не гарантирует чтения символа без обнаружения конца файла. Это все весьма низкоуровневые вопросы и они сильно зависят от реализации. Не используйте **showmanyc()** без тщательного изучения документации по вашей системе, а также без проведения серии экспериментов.

38.6.1. Потоки вывода и буферы

ostream предоставляет операции по конвертированию значений разных типов в последовательности символов согласно соглашениям (§38.4.2) и явным директивам форматирования (§38.4.5). Кроме того, **ostream** предоставляет операции, напрямую имеющие дело с его **streambuf**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_ostream : virtual public basic_ios<C,Tr> {
public:
    // ...
    explicit basic_ostream(basic_streambuf<C,Tr>* b);
    pos_type tellp();                          // получить текущую позицию
```

```
    basic_ostream& seekp(pos_type);                    // установить текущую позицию
    basic_ostream& seekp(off_type, ios_base::seekdir); // установить текущую позицию

    basic_ostream& flush(); // очистить буфер (отправить по реальному назначению)

    basic_ostream& operator<<(basic_streambuf<C,Tr>* b);    // писать из b
};
```

Функции **basic_ostream** замещают их эквиваленты из **basic_ios** (базового класса для **basic_ostream**).

ostream конструируется со **streambuf**-аргументом, который определяет, как прочитанные символы обрабатываются и куда они на самом деле перемещаются. Например, **ostringstream** (§38.2.2) или **ofstream** (§38.2.1) создаются при инициализации **ostream** подходящим **streambuf** (§38.6).

Функции **seekp()** используются для позиционирования **ostream** для записи. Суффикс **p** указывает на то, что эта позиция для помещения (putting) символов в поток. Эти функции не имеют никакого эффекта, если только поток не прикреплен к чему-либо, для чего позиционирование имеет смысл, например, когда речь идет о файле. При этом **pos_type** представляет позицию символа в файле, а **off_type** представляет собой смещение от точки, указуемый **ios_base::seekdir**.

Позиции в потоке стартуют с **0**, так что мы можем думать о файле как о массиве из **n** символов. Например:

```
int f(ofstream& fout)     // fout – некий файл
{
    fout << "0123456789";
    fout.seekp(8);                  // 8 от начала
    fout << '#';                    // добавить '#' и передвинуть позицию (+1)
    fout.seekp(-4,ios_base::cur);   // 4 назад
    fout << '*';                    // добавить '*' и передвинуть позицию (+1)
}
```

Если файл исходно пуст, мы получаем:

```
01234*67#9
```

Нет аналогичного способа выполнить произвольный доступ к элементам простых **istream** или **ostream**. Попытка спозиционировать перед началом или за концом файла в типичном случае переводит поток в состояние **bad()** (§38.4.4). В разных операционных системах это поведение может отличаться (например, может быть выполнено изменение размера файла).

Операция **flush()** позволяет пользователю очистить буфер, не ожидая переполнения.

Можно использовать **<<** для записи **streambuf** напрямую в **ostream**. Это полезно для реализаторов механизмов ввода/вывода.

38.6.2. Потоки ввода и буферы

istream предоставляет операции для чтения символов и конвертирования их в значения разных типов (§38.4.1). Дополнительно, **istream** предоставляет операции, напрямую имеющие дело с его **streambuf**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_istream : virtual public basic_ios<C,Tr> {
public:
    // ...
    explicit basic_istream(basic_streambuf<C,Tr>* b);
    pos_type tellg();                          // получить текущую позицию
```

```
    basic_istream& seekg(pos_type);                    // установить текущую позицию
    basic_istream& seekg(off_type, ios_base::seekdir); // установить текущую позицию

    basic_istream& putback(C c);    // поместить с обратно в буфер
    basic_istream& unget();         // возвратить последний считанный символ
    int_type peek();                // взглянуть на символ, подлежащий чтению

    int sync();                     // очистить буфер (flush)

    basic_istream& operator>>(basic_streambuf<C,Tr>* b); // читать в b
    basic_istream& get(basic_streambuf<C,Tr>& b, C t = Tr::newline());

    streamsize readsome(C* p, streamsize n);   // читать не более n симв.
};
```

Функции **basic_istream** замещают их эквиваленты в **basic_ios** (базовый класс для **basic_istream**).

Эти функции позиционирования работают так же, как и в **ostream** (§38.6.1). Суффикс **g** указывает на то, что это позиция, используемая для извлечения (getting) символов из потока. Суффиксы **p** и **g** нужны потому, что мы можем наследовать **iostream** от **istream** и **ostream** одновременно, и такой поток должен отслеживать как get-позицию, так и put-позицию.

Функция **putback()** разрешает программе поместить символ назад в **istream** так, что он будет символом для следующего прочтения. Функция **unget()** помещает самый последний прочитанный символ назад. К сожалению, не все случаи помещения символа назад в поток возможны. Все что гарантируется, это возможность поместить назад символ сразу после его успешного чтения из потока. Функция **peek()** читает следующий символ и оставляет его в потоке, так что его можно будет прочесть снова. Таким образом, **c=peek()** логически эквивалентно **(c=get(),unget(),c)**. Установка **failbit** может сгенерировать исключение (§38.3).

Очистка **istream** выполняется функцией **sync()**. Для некоторых потоков это не выполняется корректно, так как может потребоваться повторное чтение символов из реального источника, что не всегда возможно или желательно (например, для потока, подсоединенного к сети). Как следствие, **sync()** возвращает **0** в случае успеха, а в противном случае, эта функция устанавливает **ios_base::badbit** (§38.4.4) и возвращает **-1**. Установка **badbit** может вызвать генерацию исключения (§38.3). Вызов **sync()** для буфера, прикрепленном к **ostream**, очищает буфер и выводит данные.

Функция **get()** и операция **<<**, которые напрямую читают из **streambuf**, предназначены в первую очередь для реализаторов средств ввода/вывода.

Функция **readsome()** представляет собой низкоуровневую операцию, позволяющую пользователю заглянуть в поток, чтобы узнать, есть ли там еще символы, доступные для чтения. См. также **in_avail()** (§38.6).

38.6.3. Итераторы буферов

В файле **<iterator>** стандартная библиотека предоставляет **istreambuf_iterator** и **ostreambuf_iterator**, чтобы разрешить пользователю (преимущественно реализатору нового вида **iostream**) итерировать по содержимому потокового буфера. В частности, эти итераторы широко используются в **locale_facet** (Глава 39).

38.6.3.1. istreambuf_iterator

istreambuf_iterator читает поток символов из **istream_buffer**:

```
template<typename C, typename Tr = char_traits<C>> // §iso.24.6.3
class istreambuf_iterator
    :public iterator<input_iterator_tag, C, typename Tr::off_type, /*неуточненный*/, C> {
public:
    using char_type = C;
    using traits_type = Tr;
    using int_type = typename Tr::int_type;
    using streambuf_type = basic_streambuf<C,Tr>;
    using istream_type = basic_istream<C,Tr>;
    // ...
};
```

Член **reference** из базового класса **iterator** не используется и поэтому не специфицируется.

Если вы используете **istreambuf_iterator** в качестве входного итератора, то эффект от этого такой же, как и у других входных итераторов: поток символов может читаться с помощью выражения **c=*p++**:

istreambuf_iterator<C,Tr> (§iso.24.6.3)	
istreambuf_iterator p {};	p есть итератор конца потока; noexcept; constexpr
istreambuf_iterator p {p2};	Копирующий конструктор; noexcept
istreambuf_iterator p {is};	p есть итератор для is.rdbuf(); noexcept
istreambuf_iterator p {psb};	p есть итератор для istreambuf *psb; noexcept
istreambuf_iterator p {nullptr};	p есть итератор конца потока
istreambuf_iterator p {prox};	p указывает на istreambuf обозначаемый с помощью prox; noexcept
p.~istreambuf_iterator()	Деструктор
c=*p	c есть символ, возвращаемый sgetc() streambuf'a
p->m	m – член *p, если это объект класса
p=++p	sbumpc() streambuf'a
prox=p++	Пусть prox обозначает ту же позицию, что и p; затем ++p
p.equal(p2)	Находятся ли p и p2, оба или ни один из них в конце потока?
p==p2	p.equal(p2)
p!=p2	!p.equal(p2)

Вы не можете полагаться на два итератора, ссылающихся на один и тот же символ, пока ввод продолжается.

38.6.3.2. ostreambuf_iterator

ostreambuf_iterator пишет поток символов в **ostream_buffer**:

```
template<typename C, typename Tr = char_traits<C>> // §iso.24.6.4
class ostreambuf_iterator
    :public iterator<output_iterator_tag, void, void, void, void> {
public:
    using char_type = C;
    using traits_type = Tr;
    using streambuf_type = basic_streambuf<C,Tr>;
    using ostream_type = basic_ostream<C,Tr>;
    // ...
};
```

При его использовании эффект такой же, как и других итераторов вывода: поток символов может писаться с помощью выражения ***p++=c**:

ostreambuf_iterator<C,Tr> (продолжение) (§iso.24.6.4)	
ostreambuf_iterator p {os};	**p** есть итератор для **os.rdbuf()**; noexcept
ostreambuf_iterator p {psb};	**p** есть итератор для **istreambuf *psb**; noexcept
p=c	Если **!p.failed()**, вызвать **sputc(c)** **streambuf**'а
***p**	Ничего не делается
++p	Ничего не делается
p++	Ничего не делается
p.failed()	Достиг ли **eof** вызов **sputc()** у **streambuf**'а **p**? noexcept

38.7. Советы

[1] Определяйте **<<** и **>>** для пользовательских типов со значениями, имеющими осмысленные текстовые представления; §38.1, §38.4.1, §38.4.2.

[2] Используйте **cout** для нормального вывода и **cerr** для вывода ошибок; §38.1.

[3] Есть **iostream** для обычных символов и для широких символов, и вы можете определить **iostream** для любого вида символов; §38.1.

[4] Имеются стандартные **iostream** для стандартных потоков ввода/вывода, для файлов и строк типа **string**; §38.2.

[5] Не пытайтесь копировать файловый поток; §38.2.1.

[6] Бинарный ввод/вывод системно зависим; §38.2.1.

[7] Проверяйте, подключен ли файловый поток к файлу перед тем, как использовать его; §38.2.1.

[8] Предпочитайте **ifstream** и **ofstream** простому **fstream**; §38.2.1.

[9] Используйте **stringstream** для форматирования в памяти; §38.2.2.

[10] Используйте исключения для перехвата редких случаев **bad()**-ошибок ввода/вывода; §38.3.

[11] Используйте состояние потока **fail** для обработки потенциально исправляемых ошибок ввода/вывода; §38.3.

[12] Нет необходимости в модификации **istream** или **ostream** при добавлении новых операций **<<** и **>>**; §38.4.1.

[13] При реализации примитивных операций **iostream** применяйте **sentry**; §38.4.1.

[14] Предпочитайте форматированный ввод неформатированному (низкоуровневому); §38.4.1.

[15] Ввод в **string** не сопряжен с переполнением; §38.4.1.

[16] Будьте внимательны с терминирующим критерием, используя **get()**, **getline()** и **read()**; §38.4.1.

[17] По умолчанию операция **>>** пропускает пробелы; §38.4.1.

[18] Можно определить **<<** (или **>>**) так, что они будут вести себя как виртуальные функции, базируясь на втором операнде; §38.4.2.1.

[19] Отдавайте предпочтение манипуляторам, а не флагам состояния, при контроле ввода/вывода; §38.4.3.

[20] Используйте **sync_with_stdio(true)**, если вы хотите смешивать C-стиль с вводом/выводом в стиле **iostream**; §38.4.4.

[21] Используйте **sync_with_stdio(false)** для оптимизации **iostream**; §38.4.4.

[22] Связывайте потоки, используемые для интерактивного ввода/вывода; §38.4.4.

[23] Используйте **imbue()**, чтобы **iostream** отражал «культурные различия» **locale**; §38.4.4.

[24] Спецификации **width()** относятся только к одной последующей операции ввода/вывода; §38.4.5.1.

[25] Спецификации **precision()** относятся ко всем последующим операциям вывода для значений с плавающей запятой; §38.4.5.1.

[26] Спецификации формата для числового типа с плавающей запятой (например, **scientific**) относятся ко всем последующим операциям вывода для значений с плавающей запятой; §38.4.5.1.

[27] Используя стандартные манипуляторы с аргументами, включайте в код директиву **#include <iomanip>**; §38.4.5.2.

[28] Вам редко когда придется использовать **flush()**; §38.4.5.2.

[29] Не используйте **endl**, кроме как по эстетическим причинам; §38.4.5.2.

[30] Если надоест применять **iostream**-форматирование, напишите свой собственный манипулятор; §38.4.5.3.

[31] Можно достичь эффекта (и эффективности) тернарной операции, определив простой функциональный объект; §38.4.5.3.

39

Локализация

В Риме будь римлянином.
— Поговорка

- Культурные особенности
- Класс **locale**

  Именованные **locale**; Сравнение строк
- Класс **facet**

  Доступ к **facet** из **locale**; Простой пользовательский **facet**; Применение **locale** и **facet**
- Стандартные фасеты

  Операции сравнения для **string**; Форматирование чисел; Форматы денег; Форматирование дат и времени; Классификация символов; Конверсия кодов символов; Сообщения
- Полезные интерфейсы

  Классификация символов; Конверсия символов; Конвертирование строк; Конвертирование буферов
- Советы

39.1. Культурные особенности

locale является объектом, отражающим набор культурных (местных) предпочтений, например: как нужно сравнивать строки, каким образом числа представляются в текстах для людей и как они представляются для внешнего машинного хранения. Понятие культурных особенностей поддается расширению путем определения пользователем новых классов **facet** и представлению их классам **locale** с тем, чтобы расширить поддержку, имеющуюся в стандартной библиотеке, на почтовые коды и телефонные номера. Начальная цель подобного рода средств в стандартной библиотеке заключалась в контроле за внешним видом информации, которую пишут в **ostream**, и за форматами данных, читаемых из **istream**.

В данной главе рассматривается использование класса **locale**: то, как он конструируется из разных **facet**, и как все это влияет на потоковый ввод/вывод.

Не только язык C++, но и многие операционные системы и прикладные среды озабочены вопросом о разных культурных особенностях. Они распространяются на всю систему, и в принципе не зависят от языка программирования. Поэтому этот вопрос для языка C++ стоит так, чтобы обеспечить доступ к информации с самыми разными представлениями на разных системах. В частности, **locale** служит интерфейсом к системной информации, которая мало совместима между собой на разных платформах.

Вот простая задача — написание программы, которую будут использовать в разных странах мира. Написание программы в стиле, отвечающем этому требованию, часто на-

зывают *интернационализацией* программы или ее *локализацией* (адаптация программы к разным условиям в разных странах). Многие сущности в программах могут по-разному отображаться в разных странах. Поэтому мы должны писать наши процедуры ввода/вывода, учитывая эти различия. Например:

```
void print_date(const Date& d)          // вывести в соответствующем формате
{
    switch(where_am_I) {                // пользовательский индикатор стиля
    case DK:            // e.g., 7. marts 1999
        cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
        break;
    case ISO:           // e.g., 1999-3-7
        cout << d.year() << " - " << d.month() << " / " << d.day();
        break;
    case US:            // e.g., 3/7/1999
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
    // ...
    }
}
```

Данный стиль написания кода справляется с задачей, но он неуклюж и сложен в дальнейшей поддержке. В частности, этот стиль должен применяться согласованным образом, чтобы быть уверенным, что все идет по одному образцу. Если вдруг потребуется писать дату другим образом, нужно будет модифицировать исходный код приложения. Что еще сильнее усложняет задачу, так это то, что даты — не единственное, что отличается в разных культурных средах.

Из-за этого стандартная библиотека предоставляет возможность расширений для учета культурных различий. Библиотека **iostream** полагается на эти возможности при работе со встроенными типами и с типами, определяемыми пользователем (§38.1). К примеру, рассмотрим простой цикл, копирующий пары (Date,double), которые могут представлять собой набор измерений или набор транзакций:

```
void cpy(istream& is, ostream& os)      // copy (Date,double) stream
{
    Date d;
    double volume;

    while (is >> d >> volume)
        os << d << ' '<< volume << '\n';
}
```

Естественно, что настоящая программа будет что-то делать с этими парами, а также не забудет про обработку ошибок.

Как мы заставим эту программу читать файл, согласующийся с французскими стандартами (где в роли «десятичной точки» используется запятая; например **12,5** означает двенадцать с половиной), и писать далее это по американским правилам? Мы можем определить **locale** и операции ввода/вывода так, что **cpy()** будет выполнять конверсию между этими соглашениями:

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2)
{
  fin.imbue(locale{"en_US.UTF-8"}); // американский английский
  fout.imbue(locale{"fr_FR.UTF-8"}); // французский
  cpy(fin,fout);                    // читаем американский английский, пишем французский
  // ...
```

```
    fin2.imbue(locale{"fr_FR.UTF-8"});  // французский
    fout2.imbue(locale{"en_US.UTF-8"}); // американский английский
    cpy(fin2,fout2);                    // читаем французский, пишем американский английский
    // ...
}
```

Для такого потока данных

```
Apr 12, 1999 1000.3
Apr 13, 1999 345.45
Apr 14, 1999 9688.321
...
3 juillet 1950 10,3
3 juillet 1951 134,45
3 juillet 1952 67,9
...
```

программа выдаст следующее:

```
12 avril 1999 1000,3
13 avril 1999 345,45
14 avril 1999 9688,321
...
July 3, 1950 10.3
July 3, 1951 134.45
July 3, 1952 67.9
...
```

Большая часть оставшегося материала данной главы посвящена механизму, обеспечивающему такую возможность, а также объяснениям, как использовать эти механизмы. Большинству программистов нет нужды работать с внутренними деталями **locale**, и они вряд ли будут явно манипулировать типом **locale**. Скорее всего, они ограничатся стандартным **locale** и свяжут его с потоком (§38.4.5.1).

Концепция локализации (интернационализации) проста. Но практические моменты делают эту реализацию интригующей:

1. Класс **locale** инкапсулирует культурные соглашения, такие как внешний вид написания дат. Эти соглашения варьируются многими тонкими способами и не очень систематическим образом. Эти соглашения не имеют ничего общего с языками программирования, так что языки программирования не могут ничего стандартизировать в этом отношении.
2. Концепция **locale** должна быть расширяемой, так как невозможно даже просто перечислить все возможные культурные особенности, которые могут потребоваться пользователям.
3. Класс **locale** используется в операциях (ввод/вывод, сортировка), от которых пользователи ждут высокой производительности.
4. **locale** должен быть невидимым большинству программистов, которые ждут корректной работы, не заботясь о том, как все это устроено.
5. **locale** должен быть доступен разработчикам средств, имеющих дело с культурной информацией, выходящей за рамки, которые предоставляются стандартной библиотекой.

Можно сказать, что механизмы для создания **locale**, обеспечивающие простоту их использования, образуют (в некотором смысле) свой собственный язык программирования.

locale составляется из нескольких **facet**, контролирующих отдельные аспекты, такие как символы пунктуации в выводе чисел с плавающей запятой (**decimal_point()**; §39.4.2)

и формат для чтения денежных сумм (**moneypunct**; §39.4.3). Объекты **facet** относятся к классу, производному от **locale::facet** (§39.3). Можно трактовать **locale** как контейнер для **facet** (§39.2, §39.3.1).

39.2. Класс locale

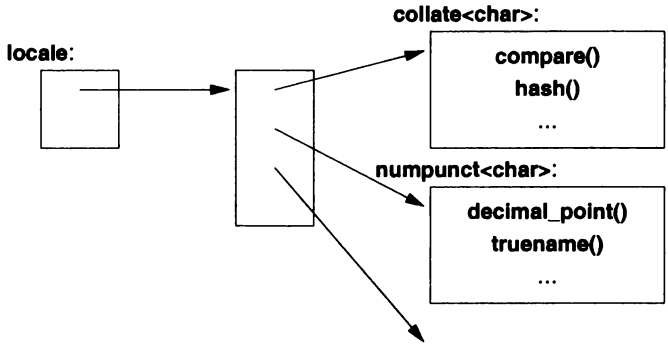
Класс **locale** и связанные с ним средства представлены в файле **<locale>**.

Функции locale (§iso.22.3.1)	
`locale loc {};`	loc есть копия текущей глобальной локализации; noexcept
`locale loc {loc2};`	Копирующий конструктор: loc содержит копию loc2; loc.name()==loc2.name(); noexcept
`locale loc {s};`	Инициализировать loc в locale с именем s; s может быть string или С-строкой; loc.name()==s; explicit
`locale loc {loc2,s,cat};`	loc есть копия loc2 за исключением фасета категории cat, к-рый копируется из locale{s}; s может быть string или С-строкой; если loc2 имеет имя, loc тоже имеет имя
`locale loc {loc2,pf};`	loc есть копия loc2 за исключением фасета *pf в случае pf!=nullptr; loc не имеет имени
`locale loc {loc2,loc3,cat};`	loc есть копия loc2 за исключением фасета категории cat, к-рый копируется из loc3; loc имеет имя, если loc2 и loc3 оба имеют имен
`loc.~locale()`	Деструктор; не virtual; noexcept
`loc2=loc`	Присваивание: loc2 есть копия loc; noexcept
`loc3=loc.combine<F>(loc2)`	loc3 есть копия loc за исключением фасета F, к-рый копируется из loc2; loc3 не имеет имени
`s=loc.name()`	s is the name of есть имя locale loc'а или "*"
`loc==loc2`	Если loc имеет тот же locale что и loc2?
`loc!=loc2`	!(loc==loc2)
`loc()(s,s2)`	Сравнить basic_string<C>s s и s2, используя фасет collate<C> из loc
`loc2=global(loc)`	Установить глобальную locale в loc; loc2 есть предыд. глобальная locale
`loc=classic()`	loc классическая локализация «С»

Если **locale** для данного имени не существует или не существует **facet** для него, то операция именования **locale** генерирует исключение **runtime_error**.

Именование **locale** довольно любопытно. Когда вы создаете новый **locale** из существующего плюс некоторый **facet**, то результирующий **locale** имеет имя, зависящее от реализации. Для **locale** без имени функция **name()** возвращает "*".

Можно считать **locale** интерфейсом к **map<id,facet*>**, то есть к тому, что позволяет нам использовать **locale::id** для нахождения соответствующего объекта класса, производного от **locale::facet**. В реальности реализация **locale** следует эффективному варианту такой идеи. Вот возможная раскладка в памяти:



Здесь **collate<char>** и **numpunct<char>** являются стандартными фасетами (§39.4). Все фасеты являются производными от **locale::facet**.

locale допускает копирование, дешевое и эффективное. Следовательно, он почти наверняка реализован как дескриптор к специализированному **map<id,facet*>**, который образует основную часть реализации. Доступ к **facet** из **locale** должен быть эффективным и быстрым. Так что специализированный **map<id,facet*>** должен оптимизироваться так, чтобы обеспечивать быстрый доступ (как в массивах). Доступ к **facet** из **locale** выполняется с помощью нотации **use_facet<Facet>(loc)**; см. §39.3.1.

Богатый набор **facet** предоставляется стандартной библиотекой. Стандартные фасеты сгруппированы в категории, такие как **numeric** или **collate** (§39.4):

Категории facet (§iso.22.3.1)	
collate	Напр., **collate**; §39.4.1
ctype	Напр., **ctype**; §39.4.5
numeric	Напр., **num_put**, **num_get**, **numpunct**; §39.4.2
monetary	**money_put**, **money_get**, **moneypunct**; §39.4.3
time	Напр., **time_put**, **time_get**; §39.4.4
messages	**messages**; §39.4.7
all	**collate** | **ctype** | **monetary** | **numeric** | **time** | **messages**
none	

У программиста нет средств именования (предоставления строки именования) для вновь создаваемых **locale**. Строки именования или создаются средой исполнения, либо создаются из комбинации имен конструкторами **locale**.

Программист может заменять **facet** в существующих категориях (§39.4, §39.4.2.1). Но он не может определять новые категории. Понятие категории относится только к стандартным фасетам, и это понятие не расширяемо. Многие пользовательские фасеты не принадлежат ни к какой категории.

Если **locale x** не имеет строки именования, то неизвестно, влияет ли **locale::global(x)** на глобальную локализацию языка С. Это означает, что в языке С++ нет надежного способа установить локализацию для языка С иную, чем та, что извлечена из среды исполнения. То же и в обратную сторону: нет надежного способа из С-кода установить глобальную локализацию для С++ (кроме как вызвав для этого С++-функцию). В смешанной среде попытка иметь разные локализации для С и С++ — это прямой путь к вероятным ошибкам.

До настоящего времени локализация преимущественно используется неявным образом в потоках ввода/вывода. Каждый **istream** и **ostream** имеет собственный **locale**. По умолчанию этот **locale** есть глобальный **locale** (§39.2.1) во время создания потока. Лока-

лизацию для потока можно установить с помощью функции **imbue()** и можно извлечь ее копию функцией **getloc()** (§38.4.5.1).

Установка глобальной локализации не влияет на существующие потоки; они по-прежнему будут использовать схемы локализации, приписанные им до указанного выше изменения.

39.2.1. Именованные locale

locale конструируется из другого **locale** и набора **facet**. Простейшим способом создания **locale** является копирование существующего. Например:

```
locale loc1;                          // копия текущей глобальной локализации
locale loc2 {""};                     // копия "локализации, предпочитаемой пользователем"

locale loc3 {"C"};                    // копия "C"-локализации
locale loc4 {locale::classic()};      // копия "C"-локализации

locale loc5 {"POSIX"};                // копия лок-ции, именуемой "POSIX"
locale loc6 {"Danish_Denmark.1252"};  // копия лок-ции, именуемой "Danish_Denmark.1252"
locale loc7 {"en_US.UTF-8"};          // копия лок-ции, именуемой "en_US.UTF-8"
```

Смысл **locale{"C"}** определяется стандартом как «классическая локализация языка C»; мы ее используем на протяжении всей книги. Остальные имена **locale** зависят от реализации.

Можно трактовать **locale{""}** как «локализацию, предпочтительную для пользователя». Она устанавливается на основе лингвистических правил среды исполнения программы. Чтобы увидеть ее, напишите следующее:

```
locale loc("");
cout << loc.name() << '\n';
```

На моем ноутбуке с Windows, я получил следующее:

```
English_United States.1252
```

А на Linux-компьютере — вот что:

```
en_US.UTF-8
```

Имена локализаций не стандартизируются языком C++. Множество организаций, таких как POSIX и Microsoft, поддерживают свои собственные стандарты поверх разных языков программирования. Например:

Примеры имен локализаций GNU (Основано на POSIX)	
ja_JP	Японский для Японии
da_DK	Датский для Дании
en_DK	Английский для Дании
de_CH	Немецкий для Швейцарии
de_DE	Немецкий для Германии
en_GB	Английский для Великобритании
en_US	Английский для США
fr_CA	Французский для Канады
de_DE	Немецкий для Германии
de_DE@euro	Немецкий для Германии с символом евро €
de_DE.utf8	Немецкий для Германии с использованием UTF-8
de_DE.utf8@euro	Немецкий для Германии с использованием UTF-8 и символом евро €

POSIX рекомендует имя языка (маленькими буквами), за которым через подчеркивание идет имя страны заглавными буквами, опционально специфицированное кодировкой, например sv_FI@euro (Шведский для Финляндии, включая символ Евро).

Примеры имен локализаций Microsoft
Arabic_Qatar.1256
Basque.Spain.1252
Chinese_Singapore.936
English_United Kingdom.1252
English_United States.1252
French_Canada.1252
Greek_Greece.1253
Hebrew_Israel.1255
Hindi_India.1252
Russian_Russia.1251

У Microsoft за именем языка следует имя страны, опционально дополненное номером кодовой страницы.

У большинства операционных систем есть возможности установки умолчательной локализации для программы. В типичном случае это делается с помощью переменных среды, таких как **LC_ALL**, **LC_COLLATE** и **LANG**. Часто подходящий для конечного пользователя вариант локализации определяется при его первоначальном выборе рабочего языка, например для Аргентинского Испанского умолчательная (предпочтительная для пользователя) локализация **locale{""}** означает **locale{"es_AR"}**. Но эти имена не являются единым стандартом для разных платформ. Таким образом, чтобы использовать именованные локализации на конкретном компьютере, нужно изучить системную документацию и провести эксперименты.

Нежелательно погружать строки именования для **locale** непосредственно в текст программы. Упоминание имен файлов или переменных среды сужает возможности программы по ее переносимости; на других платформах приходится выискивать все такие явные указания и переделывать их вручную. Лучше извлекать **locale** из программной среды исполнения (например, используя **locale("")**). Альтернативно, можно запросить у пользователя ввести соответствующую строку. Пример:

```
void user_set_locale(const string& question)
{
    cout << question;  // "Нужна другая локализация? – введите ее имя"
    string s;
    cin >> s;
    locale::global(locale{s}); // установить указ-ую польз-лем лок-цию в кач-ве глобальной
}
```

Можно предоставить пользователю (не эксперту) возможность выбрать из списка альтернатив (функция должна этот список вывести, для чего она должна знать, где система хранит **locale**; для Linux это каталог **/usr/share/locale**).

Если строка именования не соответствует существующей схеме локализации, то конструктор генерирует исключение **runtime_error** (§30.4.1.1). Пример:

```
void set_loc(locale& loc, const char* name)
try
{
    loc = locale{name};
}
```

```
catch (runtime_error&) {
        cerr << "locale
        // ...
}
```

Если у **locale** есть имя, то **name()** вернет его. А если имени нет, то **name()** вернет **string("*")**. В основном, строка именования должна ссылаться на **locale**, хранящийся в среде исполнения. Еще строка именования может использоваться в отладочных целях. Например:

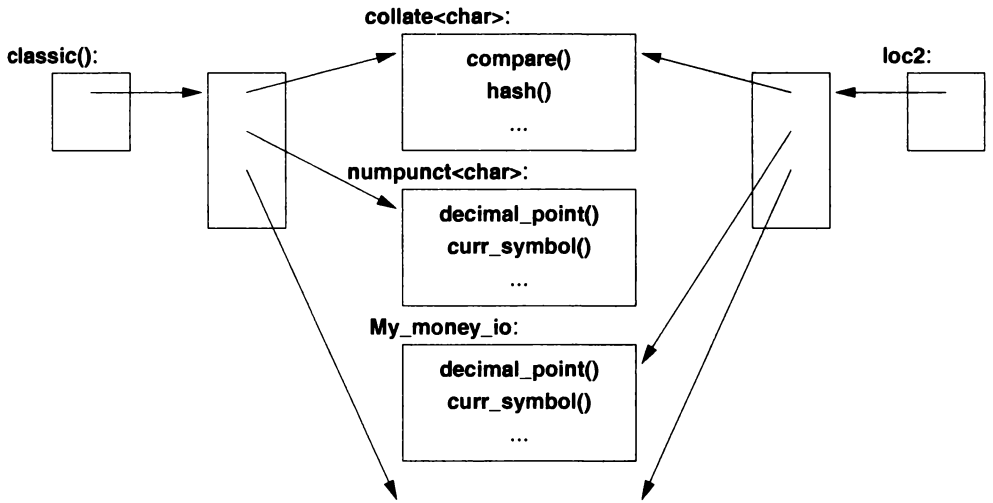
```
void print_locale_names(const locale& my_loc)
{
        cout << "name of current global locale: " << locale().name() << "\n";
        cout << "name of classic C locale: " << locale::classic().name() << "\n";
        cout << "name of "user's preferred locale": " << locale("").name() << "\n";
        cout << "name of my locale: " << my_loc.name() << "\n";
}
```

39.2.1.1. Конструирование новых locale

Новый **locale** конструируется из существующего **locale** добавлением (или замещением) **facet**. В типичном случае новый **locale** есть минимальная вариация существующего. Пример:

```
void f(const locale& loc, const My_money_io* mio)    // My_money_io определен в §39.4.3.1
{
locale loc1(locale{"POSIX"},loc,locale::monetary);    // monetary фасеты из loc
locale loc2 = locale(locale::classic(), mio);         // classic плюс mio
// ...
}
```

Здесь **loc1** является копией для **POSIX locale**, модифицированной для использования денежных **facet** (§39.4.3). Аналогично, **loc2** является копией **C locale**, модифицированной для использования **My_money_io** (§39.4.3.1). Результирующие локализации можно отобразить следующим образом:



Если **Facet*** аргумент (здесь, **My_money_io**) есть **nullptr**, то результирующий **locale** есть просто копия **locale**-аргумента.

При конструировании locale(loc,f) аргумент f должен идентифицировать конкретный тип facet. Тут простого facet* не достаточно. Например:

```
void g(const locale::facet* mio1, const money_put<char>* mio2)
{
    locale loc3 = locale(locale::classic(), mio1);  // ошибка: не известен тип фасета
    locale loc4 = locale(locale::classic(), mio2);  // OK: тип фасета известен (moneyput<char>)
    // ...
}
```

locale использует тип аргумента Facet* для определения типа facet на этапе компиляции. Конкретно, реализация facet использует facet::id (§39.3) для нахождения этого facet в locale (§39.3.1). Конструктор

```
template<class Facet> locale(const locale& x, Facet* f);
```

является единственным механизмом, предложенным программисту в C++ для предоставления facet, который будет использоваться в locale. Остальные locale предоставляются реализаторами как именованные схемы локализации (§39.2.1). Именованные локализации извлекаются из среды исполнения программы.

Набор конструкторов locale разработан так, чтобы тип facet определялся бы путем вывода типа (из шаблонного параметра Facet), или этот тип происходил бы от другого locale. Специфицирование категории (category) позволяет уточнить тип facet неявно, потому что locale знает facet в категориях. Из этого следует, что класс locale может (и должен) отслеживать типы facet, чтобы он мог манипулировать ими с минимальными накладными расходами.

Член locale::id используется locale для идентификации типов facet (§39.3).

Не существует возможности модифицировать locale. Вместо этого операции locale предоставляют способы создания новых locale из уже существуюших. Неизменяемость созданных locale повышает эффективность их использования в плане производительности. Можно вызывать виртуальные функции facet и кэшировать их возвраты. Например, istream может узнать символ, использующийся в качестве десятичной точки, и как представляется значение true, без вызова decimal_point() каждый раз при чтении числа и truename() — каждый раз при чтении значения типа bool (§39.4.2). Только после вызова imbue() для потока (§38.4.5.1) эти вызовы могут возвратить иные значения.

39.2.2. Сравнение строк

Сравнение двух строк в соответствии со схемой локализации является наиболее частым случаем использования locale вне проблематики ввода/вывода в потоках. Отсюда становится понятным, почему такая операция предоставляется самим locale, чтобы пользователи не нуждались в построении своих собственных процедур сравнения на основе фасета collate (§39.4.1). Функции сравнения строк определяются как operator()() у locale. Пример:

```
void user(const string s1, const string s2, const locale& my_locale)
{
    if (my_locale(s,s2)) {      // is s<s2 в соответствии с my_locale?
        // ...
    }
}
```

Имея функцию сравнения в виде операции (), становится возможным напрямую использовать ее для предиката (§4.5.4). Например:

```
void f(vector<string>& v, const locale& my_locale)
{
    sort(v.begin(),v.end());                // sort с применением < для сравнения элементов
    // ...
    sort(v.begin(),v.end(),my_locale);      // sort согласно правилам my_locale
    // ...
}
```

По умолчанию, функция **sort()** стандартной библиотеки использует **<** для числового значения символов из набора реализации в целях определения порядка следования символов (§32.6, §31.2.2.1).

39.3. Класс facet

Можно трактовать **locale** как коллекцию **facet**, которые отражают какой-либо один конкретный культурный аспект, например способ отображения числа при его выводе (**num_put**), как даты читаются из входных потоков (**time_get**), и как числа сохраняются в файлах (**codecvt**). Стандартные фасеты перечисляются в §39.4.

Пользователь может определить новый **facet**, например для того, чтобы указать форму отображения сезонов при печати (§39.3.2).

facet представляется в программе в виде объекта класса, производного от **std::locale::facet**. Как и все остальные средства **locale**, определения для фасетов располагаются в файле **<locale>**:

```
class locale::facet {
protected:
    explicit facet(size_t refs = 0);
    virtual ~facet();
    facet(const facet&) = delete;
    void operator=(const facet&) = delete;
};
```

Класс **facet** предназначен быть базовым классом и не имеет открытых (**public**) функций. Его конструктор защищенный (**protected**), чтобы предотвращать создание «простых фасетов», а его деструктор виртуальный, что гарантирует надлежащее уничтожение объектов производных классов.

Управлять фасетами можно посредством указателей, хранящихся в **locale**. Аргумент **0** у конструктора **facet** означает, что **locale** должен удалить **facet** в случае, когда на последний не остается никаких действительных ссылок. В противном случае (аргумент не нуль) **locale** никогда не удаляет **facet**. Это редкий случай, когда время жизни **facet** контролируется программистом явным образом, а не косвенно через **locale**.

Каждый вид интерфейса **facet** должен иметь отдельный **id**:

```
class locale::id {
public:
    id();
    void operator=(const id&) = delete;
    id(const id&) = delete;
};
```

Механизмы **locale** используют **id** для идентификации **facet** (§39.2, §39.3.1). В очевидной реализации **locale**, **id** используется как индекс в векторе указателей на **facet**, в результате чего получается эффективный **map<id,facet*>**.

Данные для создания нового (производного) **facet** определяются в рамках производного класса. Это означает, что пользователь, определяющий **facet**, имеет полный кон-

троль над этими данными, и эти данные могут в полной мере отразить концепцию нового **facet**.

facet разрабатывался как неизменяемый, так что все члены-функции пользовательского **facet** должны быть константными.

39.3.1. Доступ к facet из locale

Для доступа используются две шаблонные функции:

Функции доступа locale (§iso.22.3.2)	
f=use_facet<F>(loc)	**f** есть ссылка на фасет **F** в **loc**; генерирует **bad_cast**, если в **loc** нет **F**
has_facet<F>(loc)	Имеет ли **loc** фасет **F**? noexcept

Можно трактовать эти функции так, что они выполняют поиск в их **loc**-аргументе (типа **locale**) для шаблонного параметра **F**. Но можно рассматривать функцию **use_facet** и по-другому — как явное преобразование типа **locale** к специфическому **facet**. Это допустимо, так как **locale** может иметь лишь один **facet** заданного типа. Например:

```
void f(const locale& my_locale)
{
    char c = use_facet<numpunct<char>>(my_locale).decimal_point() // исп-ем станд-ный фасет
    // ...

    if (has_facet<Encrypt>(my_locale)) { // в my_locale есть фасет Encrypt?
            const Encrypt& f = use_facet<Encrypt>(my_locale); // извлекаем фасет Encrypt
            const Crypto c = f.g et_crypto();  // используем фасет Encrypt
            // ...
    }
    // ...
}
```

Гарантируется, что стандартные **facet** доступны для всех **locale** (§39.4), так что для стандартных фасетов не нужно использовать **has_facet**.

Еще один вариант трактовки **facet::id** состоит в том, чтобы считать его механизмом оптимальной реализации полиморфизма времени компиляции. Можно использовать **dynamic_cast** для получения результатов, похожих на те, что дает **use_facet**. Но специализированный **use_facet** может быть реализован эффективнее, чем общий **dynamic_cast**.

id идентифицирует скорее интерфейс и поведение, а не класс. То есть, если два класса **facet** имеют одинаковый интерфейс и реализуют одинаковую семантику, они должны идентифицироваться одним и тем же **id**. Например, **collate<char>** и **collate_byname<char>** взаимозаменяемы в рамках **locale**, так что оба идентифицируются посредством **collate<char>::id** (§39.4.1).

Если мы определим **facet** с новым интерфейсом — вроде **Encrypt** в **f()** — то мы должны определить соответствующий **id** для его идентификации (см. §39.3.2 и §39.4.1).

39.3.2. Простой пользовательский facet

Стандартная библиотека предоставляет стандартные фасеты для наиболее значимых случаев культурных различий: для национальных наборов символов, для отображения чисел в потоках ввода/вывода и др. В рамках изучения механизмов **facet** в отрыве от вопросов широко используемых типов и вопросов эффективности я представлю вариант **facet** для тривиального пользовательского типа:

```
enum Season { spring, summer, fall, winter }; // очень простой пользовательский тип
```

Стиль ввода/вывода, очерченный в данном примере, может с небольшими вариациями применяться для большинства простых пользовательских типов:

```
class Season_io : public locale::facet {
public:
    Season_io(int i = 0) : locale::facet{i} { }
    ~Season_io() { }   // обеспечивает возможность уничтожения объектов Season_io (§39.3)

    virtual const string& to_str(Season x) const = 0;   // строковое представление для x
    virtual bool from_str(const string& s, Season& x) const = 0; // кладет Season для s в x

    static locale::id id;    // объект идентификации фасета (§39.2, §39.3, §39.3.1)
};

locale::id Season_io::id; // определяем идентифицирующий объект
```

Для простоты данный **facet** лимитирован строками символов типа **char**.

Класс **Season_io** предоставляет общий и абстрактный интерфейс для всех фасетов **Season_io**. Чтобы обеспечить представление **Season** в рамках ввода/вывода для конкретного **locale**, мы наследует от **Season_io**, определяя **to_str()** и **from_str()** надлежащим образом.

Вывод для **Season** прост. Если поток располагает фасетом от **Season_io**, то его можно использовать для конвертации значения **x** в строку. В противном случае, мы можем просто вывести значение типа **int** для **Season**:

```
ostream& operator<<(ostream& os, Season x)
{
    locale loc {os.getloc()};  // извлекаем stream's locale (§38.4.4)

    if (has_facet<Season_io>(loc))
        return os << use_facet<Season_io>(loc).to_str(x); // строковое представление

    return os << static_cast<int>(x);                     // целочисленное представление
}
```

Ради эффективности и гибкости стандартные фасеты преимущественно работают прямо над буферами потоков (§39.4.2.2, §39.4.2.3). Но для простого пользовательского типа, такого как **Season**, нет нужды опускаться на столь низкий уровень абстракции (уровень **streambuf**).

Как обычно, ввод несколько сложнее, чем вывод:

```
istream& operator>>(istream& is, Season& x)
{
    const locale& loc {is.getloc()};          // извлекаем stream's locale (§38.4.4)

    if (has_facet<Season_io>(loc)) {
        const Season_Io& f {use_facet<Season_Io>(loc)}; // берем фасет Season_io

        string buf;
        if (!(is>>buf && f.from_str(buf,x)))  // читаем алфавитное представление
            is.setstate(ios_base::failbit);
        return is;
    }

    int i;
    is >> i;                                  // читаем числовое представление
    x = static_cast<Season>(i);
    return is;
}
```

Обработка ошибок проста и следует стилю обработки ошибок для встроенных типов. То есть, если входная строка не представляет **Season** в выбранном **locale**, то поток переводится в состояние **fail**. Если исключения не заблокированы, то в этом случае генерируется исключение **ios_base::failure** (§38.3).

Вот простая тестовая программа:

```
int main()
    // тривиальный тест
{
    Season x;

    // используем локал-цию по умолчанию (нет фасета Season_io) – ввод/вывод целых:
    cin >> x;
    cout << x << endl;

    locale loc(locale(),new US_season_io{});
    cout.imbue(loc);    // используем локализацию с фасетом Season_io
    cin.imbue(loc);     // используем локализацию с фасетом Season_io

    cin >> x;
    cout << x << endl;
}
```

При входных данных

```
2
summer
```

вывод программы будет таков:

```
2
summer
```

Чтобы этого добиться, мы должны наследовать класс **US_season_io** от **Season_io**, и определить соответствующее строковое представление для сезонов:

```
class US_season_io : public Season_io {
    static const string seasons[];
public:
    const string& to_str(Season) const;
    bool from_str(const string&, Season&) const;

    // обратите внимание: отсутствует US_season_io::id
};

const string US_season_io::seasons[] = {
    "spring",
    "summer",
    "fall",
    "winter"
};
```

Затем мы замещает функции **Season_io**, которые конвертируют строковые представления в перечисления:

```
const string& US_season_io::to_str(Season x) const
{
    if (x<spring || winter<x) {
        static const string ss = "no-such-season";
        return ss;
    }
    return seasons[x];
}
```

```
bool US_season_io::from_str(const string& s, Season& x) const
{
    const string* p = find(begin(seasons),end(seasons),s);
    if (p==end)
        return false;

    x = Season(p-begin(seasons));
    return true;
}
```

Заметьте, что поскольку **US_season_io** есть просто реализация интерфейса **Season_io**, то я не определял **id** для **US_season_io**. Фактически, если мы хотим использовать **US_season_io** в качестве **Season_io**, то мы не должны давать **US_season_io** собственного **id**. Операции над **locale**, такие как **has_facet** (§39.3.1), полагаются на то, что фасеты реализуют одну и ту же концепцию, если идентифицируются одним и тем же **id** (§39.3).

Единственный интересный вопрос для реализации: «что делать, если на выход подан некорректный **Season**?». Естественно, такого быть не должно. Но в общем случае некорректные значения пользовательских типов все-таки попадают на вывод программ, и поэтому нужно как-то реагировать на такие случаи. Можно, конечно, сгенерировать исключение, но в простых программах, ориентированных на пользователя-человека, лучше выдать на выход сообщение «вне диапазона значений» как текстовое представление значения, не попадающего в корректный диапазон. Заметьте, что для ввода обработка ошибок возлагается на операцию **>>**, в то время как для вывода это закреплено за функцией **to_str()** (функция **facet**). В больших промышленных программах функции **facet** работают как с ошибками ввода, так и вывода.

Дизайн программы с **Season_io** полагается на производные классы, предоставляющие строки, специфичные для **locale**. Альтернативно, **Season_io** сам мог бы извлекать эти строки из **locale**-репозитария (см. §39.4.7). В качестве упражнения оставляем реализацию единственного класса **Season_io**, которому строки с сезонами передаются как аргументы конструктора.

39.3.3. Применение locale и facet

Основное применение **locale** в рамках стандартной библиотеки сосредоточено в потоках ввода/вывода. Но механизм **locale** весьма общий и расширяемый, позволяющий учитывать культурные различия в общем случае. Фасет **messages** (§39.4.7) служит примером фасета, не имеющего ничего общего с потоками ввода/вывода. Что угодно, например ввод/вывод, не базирующийся на потоках, может использовать **locale**. Любой пользователь может воспользоваться **locale** для организации программного обеспечения, чувствительного к культурным особенностям разных стран и народов.

Из-за общности механизмов **locale**/**facet** возможности пользовательских **facet** практически неограничены. Для представления фасетов очевидными кандидатами являются даты, часовые пояса, телефонные номера, номера соцобеспечения, коды товаров, температурные показания, пары значений (единица, величина) вообще, почтовые коды, размеры одежды, книжные ISBN-коды и т. д.

Как и любые слишком мощные механизмы, фасеты стоит использовать с осторожностью. Если что-то можно реализовать в виде фасетов, то это не значит, что это нужно делать обязательно и что не подходят иные способы реализации. Все зависит от сложности написания кода, от трудности в его сопровождении, а также от надежности в работе и степени эффективности (производительности).

39.4. Стандартные фасеты

Стандартные фасеты определяются в файле **<locale>**:

Стандартные фасеты (§iso.22.3.1.1.1)			
collate	Сравнение строк	**collate<C>**	§39.4.1
numeric	Числовой ввод/вывод	**numpunct<C>** **num_get<C,In>** **num_put<C,Out>**	§39.4.2
monetary	Ввод/вывод денежных	**moneypunct<C>** **moneypunct<C,International>** **money_get<C,In>** **money_put<C,Out>**	§39.4.3
time	Ввод/вывод времени	**time_put<C,Out>** **time_put_byname<C,Out>** **time_get<C,In>**	§39.4.4
ctype	Классификация символов	**ctype<C>** **codecvt<In,Ex,SS>** **codecvt_byname<In,Ex,SS>**	§39.4.5
messages	Извлечение сообщений	**messages<C>**	§39.4.7

Детали рассматриваются в указанных параграфах.

Здесь **C** означает тип символов (§36.1). Гарантируется, что указанные фасеты определены для **char** и **wchar_t**. Кроме того, гарантируется, что **ctype<C>** поддерживает **char16_t** и **char32_t**. Для иных типов символов можно опираться на специфические черты конкретных реализаций, или самим предоставить подходящие версии фасетов для искомого типа **X**. К примеру, **codecvt<X,char,mbstate_t>** (§39.4.6) может потребоваться для управления конвертированием между **X** и **char**.

В приведенной выше таблице **International** может быть **true** или **false**; **true** означает, что используется трехсимвольное представление валюты (§39.4.3.1) вроде **USD** и **BRL**.

Параметр **SS** используется для индикации состояний клавиши Shift у представления многобайтных символов (§39.4.6). В файле **<cwchar>** определяется **mbstate_t** с целью отражения разных состояний конверсии в рамках зависящих от реализации правил поддержки многобайтных символов. Эквивалентом **mbstate_t** для произвольного символьного типа **X** является **char_traits<X>::state_type** (§36.2.2).

В таблице **In** и **Out** — это итераторы ввода и вывода, соответственно (§33.1.2, §33.1.4). Предоставляя **_put** и **_get** фасеты с этими аргументами шаблона, программист может определить **facet**, который имеет доступ к нестандартным буферам (§39.4.2.2). Буферы, ассоциированные с **iostream** — это буферы потоков, так что итераторы для них имеют тип **ostreambuf_iterator** (§38.6.3, §39.4.2.2). Как следствие, функция **failed()** доступна для обработки ошибок (§38.6.3).

Каждый стандартный фасет имеет **_byname**-версию. Фасет **F_byname** наследуется от фасета **F**. Фасет **F_byname** предоставляет интерфейс, аналогичный интерфейсу **F**, за исключением того, что добавляет конструктор, принимающий строковый аргумент, именующий **locale** (см. §39.4.1). **F_byname(name)** предоставляет надлежащую семантику для **F**, определенную в **locale(name)**. Например:

```
sort(v.begin(),v.end(),collate_byname{"da_DK"}); // sort используя сравнение строк из "da_DK"
```

Идея состоит в том, чтобы выбрать версию стандартного **facet** из именованного **locale** (§39.2.1) в среде исполнения программы. Подразумевается, что **_byname**-конструкторы работают медленно по сравнению с конструкторами, которые не консультируются с окружением. Почти всегда быстрее сконструировать **locale** и затем осуществить доступ к его **facet**, чем использовать **_byname** фасеты во многих местах программы. Поэтому однократное чтение **facet** из окружения и дальнейшее использование сохраненной в памяти копии является хорошей идеей. Например:

```
locale dk {"da_DK"};      // читаем датскую локал-ю (включая все фасеты) единожды
                          // а затем используем dk и его фасеты по необходимости
void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col {use_facet<collate<char>>(dk)};
    const ctype<char>& ctyp {use_facet<ctype<char>>(dk)};

    locale dk1 {loc,&col};    // сравнение датских строк
    locale dk2 {dk1,&ctyp};   // сравнение датских строк и классификация датских символов

    sort(v.begin(),v.end(),dk2);
    // ...
}
```

Здесь **dk2** будет использовать строки в датском стиле, но сохранит умолчательные соглашения по числам.

Понятие категорий дает более простой способ манипуляции стандартными фасетами в **locale**. Например, имея **dk**, можно сконструировать **locale**, который читает и сравнивает строки по правилам датского языка, но сохраняющий синтаксис чисел, используемый в C++:

```
locale dk_us(locale::classic(),dk,collate|ctype);    // датские буквы, обычные числа
```

Представление конкретных стандартных фасетов содержит дополнительные примеры использования **facet**. В частности, обсуждение **collate** (§39.4.1) вскрывает много общих структурных аспектов фасетов.

Часто стандартные фасеты зависят друг от друга. Например, **num_put** зависит от **numpunct**. В случае, если вы имеете детальные знания об индивидуальных стандартных фасетах, вы можете с уверенностью смешивать и сопоставлять разные **facet** или добавлять новые версии стандартных фасетов. Другими словами, за исключением простых операций (таких как **imbue()** для **iostream** и **collate** для **sort()**) механизмы **locale** не предназначены для того, чтобы новички могли работать с ними напрямую. Исчерпывающее обсуждение схем локализации имеется в [Langer,2000].

Дизайн индивидуальных фасетов часто довольно запутанный. Часть причин кроется в том, что вопросы культурных различий сами по себе довольно запутанные и находятся вне контроля разработчиков библиотеки, а также потому, что средства стандартной библиотеки C++ должны быть в значительной степени совместимы с тем, что предлагается стандартной библиотекой языка C, а также многочисленными стандартами компьютерных платформ.

С другой стороны, набор средств, предоставляемый с помощью **locale** и **facet**, весьма общий и гибкий. Фасеты могут содержать любые данные, а операции с фасетами могут манипулировать с данными желаемым образом. Кроме крайних случаев ограничений соглашениями, дизайн **facet** может быть простым и ясным (§39.3.2).

39.4.1. Операции сравнения для string

Стандартный фасет **collate** предоставляет способы сравнения для массивов символов:

```
template<class C>
class collate : public locale::facet {
public:
    using char_type = C;
    using string_type = basic_string<C>;

    explicit collate(size_t = 0);

    int compare(const C* b, const C* e, const C* b2, const C* e2) const
        { return do_compare(b,e,b2,e2); }

    long hash(const C* b, const C* e) const
        { return do_hash(b,e); }
    string_type transform(const C* b, const C* e) const
        { return do_transform(b,e); }

    static locale::id id; // объект идентификации фасета (§39.2, §39.3, §39.3.1)

protected:
    ~collate();          // внимание: protected destructor

    virtual int do_compare(const C* b, const C* e, const C* b2, const C* e2) const;
    virtual string_type do_transform(const C* b, const C* e) const;
    virtual long do_hash(const C* b, const C* e) const;
};
```

Здесь определяются два интерфейса:

- **public** интерфейс для пользователей **facet**.
- **protected** интерфейс для реализаторов классов, наследуемых от **facet**.

Аргумент конструктора специфицирует, кто ответственный за уничтожение **facet** — **locale** или пользователь. Умолчательное значение (**0**) означает «пусть сам **locale** управляет» (§39.3).

У всех стандартных фасетов одинаковая структура, так что все **facet** могут характеризоваться следующими ключевыми функциями:

Фасет collate<C> (§iso.22.4.4.1)
int compare(const C* b, const C* e, const C* b2, const C* e2) const;
long hash(const C* b, const C* e) const;
string_type transform(const C* b, const C* e) const;

Определяя **facet**, можете взять **collate** в качестве образца. При этом просто определите **do_*** версии ключевых функций, обеспечивающих функциональность **facet**. Полные объявления функций перечисляются для того, чтобы дать достаточно информации для написания замещающих **do_*** функций. За примером обратитесь к §39.4.1.1.

Функция **hash()** вычисляет хэш-значение для ее входной строки. Это может быть полезно для построения хэш-таблиц.

Функция **transform()** производит строку, которая при сравнении с другой трансформированной строкой (результатом вызова **transform()**) дает такой же результат, как и при сравнении исходных строк. То есть:

```
cf.compare(cf.transform(s),cf.transform(s2)) == cf.compare(s,s2)
```

Функция **transform()** предназначена для оптимизации кода, в котором приходится сравнивать строку со многими другими строками. Это полезно, например, при поиске в наборе строк.

Функция **compare()** выполняет сравнение строк согласно правилам, определенным для конкретного **collate**. Она возвращает:

1 если первая строка лексикографически больше второй
0 если строки идентичны
-1 если вторая строка лексикографически больше первой

Например:

```
void f(const string& s1, const string& s2, const collate<char>& cmp)
{
    const char* cs1 {s1.data()}; // поскольку compare() работает с char[]
    const char* cs2 {s2.data()};

    switch (cmp.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size()) {
    case 0:  // строки идентичны согласно cmp
        // ...
        break;
    case -1:  // s1 < s2
        // ...
        break;
    case 1: // s1 > s2
        // ...
        break;
    }
}
```

Функции-члены **collate** сравнивают диапазон **[b:e)** для **C**, а не **basic_string** или терминированных нулем строк в стиле языка C. В частности, **C** с числовым значением **0** трактуется как обычный символ, а не терминирующий символ.

Тип **string** стандартной библиотеки не зависит от **locale**. То есть, он сравнивает строки согласно правилам набора символов реализации (§6.2.3). Более того, стандартный тип **string** не предоставляет прямого способа спецификации критерия сравнения (Глава 36). Для выполнения сравнения с учетом **locale** нужно воспользоваться функцией **compare()** у **collate**. Пример:

```
void f(const string& s1, const string& s2, const string& name)
{
    bool b {s1==s2};    // сравниваем по правилам для набора символов реализации

    const char* s1b {s1.data()};                    // берем начало данных
    const char* s1e {s1.data()+s1.size()}           // берем конец данных
    const char* s2b {s2.data()};
    const char* s2e {s2.data()+s2.size()}

    using Col = collate<char>;

    const Col& global {use_facet<Col>(locale{})};       // из текущей глобаль. локализации
    int i0 {global.compare(s1b,s1e,s2b,s2e)};

    const Col& my_coll {use_facet<Col>(locale{""})}; // из предпочтительной локал-ции
    int i1 {my_coll.compare(s1b,s1e ,s2b,s2e)};

    const Col& n_coll {use_facet<Col>(locale{name})}; // из locale с именем name
    int i2 {n_coll.compare(s1b,s1e,s2b,s2e)};
}
```

С точки зрения нотации было бы удобнее использовать функцию **compare()** не напрямую, а косвенно посредством **operator()** у **locale** (§39.2.2). Например:

```
void f(const string& s1, const string& s2, const string& name)
{
    int i0 = locale{}(s1,s2);          // из текущей глобаль. локализации
    int i1 = locale{""}(s1,s2);        // из предпочтительной локал-ции
    int i2 = locale{name}(s1,s2);      // из locale с именем name
    // ...
}
```

Нетрудно предположить, что в некоторых случаях **i0**, **i1** и **i2** различаются. Рассмотрим следующую последовательность слов из словаря немецкого языка:

Dialekt, Diät, dich, dichten, Dichtung

По соглашению существительные пишутся с заглавной буквы, но порядок следования от этого не меняется.

А вот сортировка немецких слов, чувствительная к регистру букв, поместит все слова, начинающиеся на **D**, перед словами, начинающимися на **d**:

Dialekt, Diät, Dichtung, dich, dichten

Буква **ä** трактуется как «некоторый вид буквы **a**», так что она идет перед **c**. Но в наиболее распространенных наборах символов числовое значение для **ä** больше, чем числовое значение для **a**. Следовательно, **int('c')<int('ä')**, и простая сортировка на базе числовых кодов даст

Dialekt, Dichtung, Diät, dich, dichten

Будет интересным упражнением написать функцию сравнения, которая упорядочивает последовательность слов согласно словарю.

39.4.1.1. Именованный collate

collate_byname является версией **collate** для **locale**, именованного строковым аргументом конструктора:

```
template<class C>
class collate_byname : public collate<C> {  // внимание: ни нового id, ни новых функций
public:
    typedef basic_string<C> string_type;

    explicit collate_byname(const char*, size_t r = 0); // констр-ние из именованной локал-ции
    explicit collate_byname(const string&, size_t r = 0);

protected:
    ~collate_byname();          // внимание:  protected destructor

    int do_compare(const C* b, const C* e, const C* b2, const C* e2) const override;
    string_type do_transform(const C* b, const C* e) const override;
    long do_hash(const C* b, const C* e) const override;
};
```

Таким образом, **collate_byname** можно использовать для извлечения **collate** из **locale**, именованного из среды исполнения программы (§39.4). Очевидным способом хранения **facet** в среде исполнения будет их хранение в файле. Менее гибкой альтернативой будет представление **facet** в тексте программы и данных в **_byname**-фасете.

39.4.2. Форматирование чисел

Числовой вывод выполняется фасетом **num_put**, пишущим в буфер потока (§38.6). Числовой ввод выполняется фасетом **num_get**, читающим из буфера потока. Формат, используемый **num_put** и **num_get** определяется фасетом числовой пунктуации **numpunct**.

39.4.2.1. Числовая пунктуация

Фасет **numpunct** определяет формат ввода/вывода для встроенных типов, таких как **bool**, **int** и **double**:

Фасет numpunct<C> (§iso.22.4.6.3.1)	
C decimal_point() const;	Напр., '.'
C thousands_sep() const;	Напр., ','
string grouping() const;	Напр., "" означает «без группировки»
string_type truename() const;	Напр., **"true"**
string_type falsename() const;	Напр., **"false"**

Символы строки, возвращаемой **grouping()**, читаются как последовательность значений укороченных целых типов. Каждое число специфицирует число цифр в группе. Символ **0** специфицирует самую правую группу (наименее значимые цифры), символ **1** специфицирует группу левее означенной, и т. д. Таким образом, **"\004\002\003"** описывает число вроде **123-45-6789** (при условии, что знак - используется как разделительный символ). При необходимости, последнее число в схеме группирования используется для повторения, так что "\003" эквивалентно "\003\003\003". Чаще всего группирование используется для улучшения читаемости больших чисел. Функции **grouping()** и **thousands_sep()** определяют формат как для ввода, так и для вывода целых чисел и целых частей чисел с плавающей запятой.

Можно определить новый стиль пунктуации, наследуя от **numpunct**. Например, я мог бы определить фасет **My_punct** для записи целых значений, используя пробелы для группировки цифр в наборы по три, и для значений с плавающей запятой, используя европейский стиль с запятой в качестве «десятичной точки»:

```
class My_punct : public numpunct<char> {
public:
    explicit My_punct(size_t r = 0) :numpunct<char>(r) { }
protected:
    char do_decimal_point() const override { return ','; }        // запятая
    char do_thousands_sep() const override { return '_'; }        // знак подчеркивания
    string do_grouping() const override { return "\003"; }        // 3-цифровые группы
};

void f()
{
    cout << "style A: " << 12345678
         << " *** " << 1234567.8
         << " *** " << fixed << 1234567.8 << '\n';
    cout << defaultfloat;      // вернем формат float по умолчанию
    locale loc(locale(),new My_punct);
    cout.imbue(loc);
    cout << "style B: " << 12345678
         << " *** " << 1234567.8
         << " *** " << fixed << 1234567.8 << '\n';
}
```

Отсюда вот что получается:

```
style A: 12345678 *** 1.23457e+06 *** 1234567.800000
style B: 12_345_678 *** 1_234_567,800000 *** 1_234_567,800000
```

Заметьте, что **imbue()** хранит копию своего аргумента в своем потоке. Следовательно, поток может полагаться на **locale** потока после того, как оригинальная копия этого **locale** уже уничтожена (удалена). Если у **iostream** установлен его флаг **boolalpha** (§38.4.5.1), то строки, возвращаемые **truename()** и **falsename()**, используются для представления **true** и **false**, соответственно; в ином случае используются **1** и **0**.

Предоставляется **_byname**-версия (§39.4, §39.4.1) для **numpunct**:

```
template<class C>
class numpunct_byname : public numpunct<C> {
    // ...
};
```

39.4.2.2. Числовой вывод

При записи в буфер потока (§38.6) **ostream** полагается на фасет **num_put**:

Фасет num_put<C,Out=ostreambuf_iterator<C>> (§iso.22.4.2.2) Помещает значение **v** в позицию буфера **b** в потоке **s**
Out put(Out b, ios_base& s, C fill, bool v) const;
Out put(Out b, ios_base& s, C fill, long v) const;
Out put(Out b, ios_base& s, C fill, long long v) const;
Out put(Out b, ios_base& s, C fill, unsigned long v) const;
Out put(Out b, ios_base& s, C fill, unsigned long long v) const;
Out put(Out b, ios_base& s, C fill, double v) const;
Out put(Out b, ios_base& s, C fill, long double v) const;
Out put(Out b, ios_base& s, C fill, const void* v) const;

Значением возврата для **put()** служит позиция итератора на одну позицию далее последнего записанного символа.

Умолчательная специализация для **num_put** (та, для которой итератор для доступа к символам имеет тип **ostreambuf_iterator<C>**) является частью стандартных **locale** (§39.4). Для записи в иные места, используя **num_put**, мы должны определить надлежащую специализацию. Например, вот простой **num_put** для записи в **string**:

```
template<class C>
class String_numput : public num_put<C,typename basic_string<C>::iterator> {
public:
    String_numput() :num_put<C,typename basic_string<C>::iterator>{1} { }
};
```

Поскольку я не думал внедрять **String_numput** в **locale**, то я использовал аргумент конструктора для удержания обычных правил времени жизни. Вот пример использования:

```
void f(int i, string& s, int pos)     // форматирование i в s с позиции pos
{
    String_numput<char> f;
    f.put(s.begin()+pos,cout,' ',i);  // форматируем i в s;
                                      // используем правила форматирования потока cout
}
```

Аргумент **ios_base** (здесь это **cout**) предоставляет информацию о состоянии форматирования и **locale**. Например:

```
void test(iostream& io)
{
    locale loc = io.getloc();

    wchar_t wc = use_facet<ctype<char>>(loc).widen(c);                    // из char в C
    string s = use_facet<numpunct<char>>(loc).decimal_point();            // по умолчанию: "."
    string false_name = use_facet<numpunct<char>>(loc).falsename(); // по умолчанию: "false"
}
```

Стандартный **facet**, такой как **num_put<char>**, в типичном случае используется неявно посредством стандартной функции потока ввода/вывода. Следовательно, большинство программистов не имеет необходимости знать о нем. Но использование таких стандартных фасетов функциями стандартной библиотеки весьма интересно, потому что они знают, как потоки ввода/вывода работают и как используются **facet**. Как всегда, стандартная библиотека служит примером интересных приемов программирования.

Используя **num_put**, реализатор **ostream** может написать:

```
template<class C, class Tr>
basic_ostream<C,Tr>& basic_ostream<C,Tr>::operator<<(double d)
{
    sentry guard(*this);            // см. §38.4.1
    if (!guard) return *this;

    try {
        if (use_facet<num_put<C,Tr>>(g etloc()).put(*this,*this,this->fill(),d).failed())
            setstate(badbit);
    }

    catch (...) {
        handle_ioexception(*this);
    }
    return *this;
}
```

Здесь много чего происходит, например, **sentry** гарантирует, что выполняются все префиксные и суффиксные операции (§38.4.1). Мы получаем **locale** у **ostream** вызовом его функции-члена **getloc()** (§38.4.5.1). Извлекаем **num_put** из **locale**, используя **use_facet** (§39.3.1). После этого мы вызываем соответствующую функцию **put()**, чтобы она сделала всю реальную работу. Можно сконструировать **ostreambuf_iterator** из **ostream** (§38.6.3), и **ostream** может быть неявно приведен к базовому классу **ios_base** (§38.4.4), так что легко получаются два первых аргумента **put()**.

Вызов **put()** возвращает свой аргумент — итератор вывода. Этот итератор вывода получается из **basic_ostream**, так что это **ostreambuf_iterator**. Функция **failed()** (§38.6.3) пригодна для тестирования успеха, что позволяет нам установить состояние потока надлежащим образом.

Я не использовал **has_facet**, потому что стандартные **facet** (§39.4) гарантированно присутствуют во всех **locale**. Если все же эта гарантия нарушается, то генерируется исключение **bad_cast** (§39.3.1).

Функция **put()** вызывает виртуальную функцию **do_put()**. Следовательно, возможно исполнение пользовательского кода и **operator<<()** должен быть готов к обработке исключения, генерируемого замещающей **do_put()**. Также **num_put** может не существовать для некоторых типов символов, так что **use_facet()** может сгенерировать **bad_cast** (§39.3.1). По-

ведение операции << для встроенных типов, таких как **double**, определяется стандартом языка C++. Поэтому не стоит вопрос о том, что должна делать функция **handle_exception()**, а стоит вопрос о том, как она должна действовать согласно стандарту. Если установлен **badbit** в данном **ostream** (§38.3), то исключение просто повторно генерируется. Иначе исключение обрабатывается путем установки состояния потока и продолжением работы. В любом случае, **badbit** должен быть установлен для потока (§38.4.5.1):

```
template<class C, class Tr>
void handle_ioexception(basic_ostream<C,Tr>& s)      // вызывается из catch-блока
{
    if (s.exceptions()&ios_base::badbit) {
        try {
            s.setstate(ios_base::badbit);            // может генер-вать basic_ios::failure
        }
        catch(...) {
                // ... ничего не делаем ...
        }
        throw;    // повторная генерация исключения
    }
    s.setstate(ios_base::badbit);
}
```

Здесь **try**-блок нужен, поскольку **setstate()** може сгенерировать **basic_ios::failure** (§38.3, §38.4.5.1). Однако, если **badbit** установлен для состояния исключений, то **operator<<()** должен повторно сгенерировать исключение, что приведет к вызову **handle_ioexception()** (а не к генерации **basic_ios::failure**).

Операция << для встроенных типов, таких как **double**, должна реализовываться посредством прямой записи в буфер потока. Когда << определяется для пользовательских типов, то можно избежать больших сложностей, определяя вывод пользовательских типов в терминах вывода уже существующих типов (§39.3.2).

39.4.2.3 Ввод чисел

При чтении из потокового буфера (§38.6) **istream** полагается на фасет **num_get**:

Фасет num_get<In = istreambuf_iterator<C>> (§iso.22.4.2.1) Читает [b:e) в v, используя правила форматирования из s, при ошибках, устанавливает r
In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, long long& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long long& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

В своей основе **num_get** организован так же, как **num_put** (§39.4.2.2). Но поскольку он читает (а не пишет), то **get()** нуждается в паре итераторов ввода, и в аргументе-ссылке, обозначающем цель ввода.

Переменная **r** типа **iostate** устанавливается, чтобы отразить состояние потока. Если значение желаемого типа не может быть прочитано, то в **r** устанавливается **failbit**; если достигается маркер конца чтения, то в **r** устанавливается **eofbit**. Операция ввода использует **r** для определения того, как нужно установить состояние потока. Если не встречено никаких ошибок, то прочитанное значение присваивается **v**; в противном случае значение **v** никак не прописывается (остается неизменным).

Здесь **sentry** используется для гарантирования того, что суффиксные и постфиксные операции потока будут выполняться (§38.4.1). В частности, им также гарантируется, что попытка чтения выполняется только тогда, когда поток находится в корректном состянии, пригодном для начала чтения. Например, реализатор **istream** может написать следующее:

```
template<class C, class Tr>
basic_istream<C,Tr>& basic_istream<C,Tr>::operator>>(double& d)
{
    sentry guard(*this);        // см. §38.4.1
    if (!guard) return *this;

    iostate state = 0;          // все хорошо
    istreambuf_iterator<C,Tr> eos;
    try {
      double dd;
      use_facet<num_get<C,Tr>>(getloc()).get(*this,eos,*this,state,dd);
      if (state==0 || state==eofbit) d = dd; // присваиваем только, если вызов get() успешен
      setstate(state);
    }
    catch (...) {
      handle_ioexception(*this);   // см. §39.4.2.2
    }
    return *this;
}
```

Я принял меры к тому, чтобы цель чтения (цель операции >>) не модифицировалась до того, когда операция чтения не завершится успешно.

Исключения в **istream** генерируются функцией **setstate()** в случае ошибок (§38.3).

Определяя **numpunct**, такой как **My_punct** из §39.4.2.1, мы можем читать, используя нестандартную пунктуацию. Например:

```
void f()
{
    cout << "style A: "
    int i1;
    double d1;
    cin >> i1 >> d1;    // читаем по стандартному формату "12345678"

    locale loc(locale::classic(),new My_punct);
    cin.imbue(loc);
    cout << "style B: "
    int i2;
    double d2;
    cin >> i1 >> d2;    // читаем по формату "12_345_678"
}
```

Если нам нужно читать действительно нестандартные числовые форматы, то нам придется заместить **do_get()**. Например, мы могли бы определить **num_get** так, чтобы можно было читать римские цифры вроде **XXI** и **MM**.

39.4.3. Форматы денег

Форматирование денежных величин технически эквивалентно форматированию простых чисел (§39.4.2). Но визуальное представление денежных величин еще более вариативно в связи существующими культурными различиями. Например, отрицательные финансовые итоги (потери, дебет), вроде **-1.25**, в некоторых контекстах должны представляться положительными числами в круглых скобках: **(1.25)**. Кроме того, в некоторых контекстах для этой цели принято выводить цветные изображения чисел.

Не существует стандартного «денежного типа». Вместо этого, денежные фасеты должны явно использоваться для числовых значений, отражающих денежные суммы. Например:

```
struct Money {                      // простой тип для хранения денежной суммы
    using Value = long long;        // для валют, к-рые подвергались инфляции
    Value amount;
};

// ...

void f(long int i)
{
    cout << "value= " << i << " amount= " << Money{i} << '\n';
}
```

Задачей денежных фасетов является помощь в написании операций вывода **Money**, чтобы печатались денежные величины согласно местным соглашениям (см. §39.4.3.2). Вывод будет варьироваться согласно **locale** у **cout**. Вот возможный вывод:

```
value= 1234567 amount= $12345.67
value= 1234567 amount= 12345,67 DKK
value= 1234567 amount= CAD 12345,67
value= -1234567 amount= $-12345.67
value= -1234567 amount= -€12345.67
value= -1234567 amount= (CHF12345,67)
```

В денежных вопросах точность в отношении мельчайших единиц денег считается существенной. Как следствие, я принял общее соглашение, что целые значения представляют суммы в центах (пенсах и т. д.), а не число долларов (фунтов и т. д.). Это соглашение поддерживается функцией **frac_digits()** у **moneypunct** (§39.4.3.1). Визуальное представление «десятичной точки» определяется посредством **decimal_point**().

Фасеты **money_get** и **money_put** предоставляют функции, выполняющие ввод/вывод на основе формата, определяемого фасетом **money_base**.

Простой тип **Money** может использоваться для контроля форматов ввода/вывода и для хранения денежных величин. В первом случае мы приводим значения иных типов, используемых для хранения денежных сумм, к типу **Money** перед их записью, и читаем в переменные типа **Money** перед их конвертированием в иные типы. Хранение денежных величин согласно типу **Money** менее подвержено ошибкам, так как при этом мы не можем уже забыть привести значения к типу **Money** перед их записью, и мы не получим ошибок ввода, пытаясь читать денежные величины способом, нечувствительным к актуальному **locale**.

39.4.3.1. Денежная пунктуация

Фасет, контролирующий представление денежных величин, **moneypunct**, напоминает фасет, контролирующий простые числа, **numpunct** (§39.4.2.1):

```
class money_base {
public:
    enum part {         // части раскладки значения
        none, space, symbol, sign, value
    };

    struct pattern {    // спецификация раскладки
        char field[4];
    };
};

template<class C, bool International = false>
class moneypunct : public locale::facet, public money_base {
public:
    using char_type = C;
    using string_type = basic_string<C>;
    // ...
};
```

Функции-члены **moneypunct** определяют раскладку для денег при вводе и выводе:

Фасет moneypunct<C,International>> (§iso.22.4.6.3)	
C decimal_point() const;	Напр., '.'
C thousands_sep() const;	Напр., ','
string grouping() const;	Напр., "" означает «без группировки»
string_type curr_symbol() const;	Напр., "$"
string_type positive_sign() const;	Напр., ""
string_type negative_sign() const;	Напр., "-"
int frac_digits() const;	Число цифр после ".", напр., **2**
pattern pos_format() const;	**symbol, space, sign, none,** или **value**
pattern neg_format() const;	**symbol, space, sign, none,** или **value**
static const bool intl = International;	Международ. денежные форматы

Средства, предоставляемые **moneypunct**, предназначены преимущественно для реализаторов фасетов **money_put** и **money_get** (§39.4.3.2, §39.4.3.3).

Имеется версия **_byname** (§39.4, §39.4.1) для **moneypunct**:

```
template<class C, bool Intl = false>
class moneypunct_byname : public moneypunct<C, Intl> {
    // ...
};
```

Члены **decimal_point()**, **thousands_sep()** и **grouping()** работают так же, как и в случае **numpunct**.

Члены **curr_symbol()**, **positive_sign()** и **negative_sign()** возвращают строку, которую нужно использовать для представления знака валюты (например, **$**, **INR**, **DKK** и т. д.), знака плюс и знака минус, соответственно. Если шаблонный аргумент **International** есть **true**, то член **intl** тоже будет **true**, и интернациональное представление символов валюты (четырехсимвольная строка в стиле языка C) будет использоваться:

```
"USD"
"DKK"
"EUR"
```

Последний (невидимый) символ это терминирующий нуль. Трехбуквенные обозначения валют определены стандартом ISO-4217. Когда **International** есть **false**, то используются местные символы, вроде $, £.

Возвращаемый функциями **pos_format()** или **neg_format()** **pattern** состоит из четырех частей типа **part**, определяя последовательность, в которой числовое значение, в котором находятся символ валюты, положительный (или отрицательный) знак и символ пробела. Наиболее простые форматы легко представляются с помощью следующих образцов:

```
+$ 123.45      // { sign, symbol, space, value } где positive_sign() возвращает "+"
$+123.45       // { symbol, sign, value, none } где positive_sign() возвращает "+"
$123.45        // { symbol, sign, value, none } где positive_sign() возвращает ""
$123.45-       // { symbol, value, sign, none }
-123.45 DKK    // { sign, value, space, symbol }
($123.45)      // { sign, symbol, value, none } где negative_sign() возвращает "()"
(123.45DKK)    // { sign, value, symbol, none } где negative_sign() возвращает "()"
```

Представление отрицательных значений круглыми скобками достигается посредством возврата у **negative_sign()** строки, содержащей два символа **()**. Первый символ знаковой строки помещается туда, где **sign** находится в образце, а остальная часть этой строки помещается после остальных частей в образце. Чаще всего это средство используется в финансовом сообществе, но возможны и иные сферы использования. Пример:

```
-$123.45       // { sign, symbol, value, none } где negative_sign() возвращает "-"
*$123.45 silly // { sign, symbol, value, none } где negative_sign() возвращает "* silly"
```

Величины **sign**, **value** и **symbol** должны присутствовать в образце ровно по одному разу. Остальные значения могут быть пробелами (**space**) или **none** («ничто»). Там, где появляется **space**, может находиться и большее число пробелов. Там, где присутствует **none**, там (за исключением конца образца) могут находиться нуль или более пробелов.

Заметьте, что эти строгие правила запрещают некоторые вполне очевидные варианты:

```
pattern pat = { sign, value, none, none };    // ошибка: нет symbol
```

Функция **frac_digits()** указывает, где помещается **decimal_point()**. Часто денежные суммы выражаются в мельчайших единицах (§39.4.3). Часто это одна сотая основной единицы (например, ¢ это одна сотая от $), так что **frac_digits()** часто дает значение **2**.

Вот простой формат, определенный как **facet**:

```
class My_money_io : public moneypunct<char,true> {
public:
    explicit My_money_io(size_t r = 0) :moneypunct<char,true>(r) { }

    char_type do_decimal_point() const { return '.'; }
    char_type do_thousands_sep() const { return ','; }
    string do_grouping() const { return "\003\003\003"; }

    string_type do_curr_symbol() const { return "USD "; }
    string_type do_positive_sign() const { return ""; }
    string_type do_negative_sign() const { return "()"; }

    int do_frac_digits() const { return 2; } // две цифры после десятичной точки

    pattern do_pos_format() const { return pat; }
    pattern do_neg_format() const { return pat; }
private:
    static const pattern pat;
};

const pattern My_money_io::pat { sign, symbol, value, none };
```

39.4.3.2. Вывод денежных сумм

Фасет **money_put** пишет денежные величины согласно формату, специфицированному **moneypunct**. Конкретно, **money_put** предоставляет функцию **put()**, которая помещает надлежащим образом сформатированный символ в буфер потока:

Фасет money_put<C,Out = ostreambuf_iterator<C>> (§iso.22.4.6.2) Помещает значение v в позицию буфера b
Out put(Out b, bool intl, ios_base& s, C fill, long double v) const; Out put(Out b, bool intl, ios_base& s, C fill, const string_type& v) const;

Аргумент **intl** указывает, используется ли четырехсимвольный интернациональный стиль для символа валюты, или местный символ (§39.4.3.1):

Имея **money_put**, мы можем определить операцию вывода для **Money** (§39.4.3):

```
ostream& operator<<(ostream& s, Money m)
{
    ostream::sentry guard(s); // см. §38.4.1
    if (!guard) return s;

    try {
        const money_put<char>& f = use_facet<money_put<char>>(s.getloc());
        if (m==static_cast<long long>(m)) {      // m можно представить как long long
            if (f.put(s,true,s,s.fill(),m).failed())
                s.setstate(ios_base::badbit);
        }
        else {
            ostringstream v;
            v << m;                              // преобразуем в строковое представление
            if (f.put(s,true,s,s.fill(),v.str()).failed())
                s.setstate(ios_base::badbit);
        }
    }
    catch (...) {
        handle_ioexception(s); // см. §39.4.2.2
    }
    return s;
}
```

Если точности **long long** не хватает, чтобы точно представить денежное значение, то я конвертирую значение в его строковое представление и вывожу это, используя **put()**, принимающий **string**.

39.4.3.3. Ввод денежных сумм

Фасет **money_get** читает денежные суммы согласно формату, специфицированному **moneypunct**. Конкретно, **money_get** предоставляет функцию **get()**, которая извлекает надлежащим образом сформатированное строковое представление из буфера потока:

Фасет money_get<C,In = istreambuf_iterator<C>> (§iso.22.4.6.1) Читает [b:e) в v, используя правила форматирования из s, при ошибках, устанавливает r
In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const; In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

Правильно определенная пара фасетов **money_get** и **money_put** обеспечат вывод в форме, который может быть (в обратном направлении) прочитан без ошибок или потери точности. Например:

```
int main()
{
    Money m;
    while (cin>>m)
        cout << m << "\n";
}
```

Вывод этой простой программы должен быть приемлемым для ввода. Более того, вывод второго прогона этой программы, получив вывод первого прогона, должен быть идентичным вводу.

Приемлемая операция ввода для **Money** будет таковой:

```
istream& operator>>(istream& s, Money& m)
{
  istream::sentry guard(s); // см. _io.sentr y_
  if (guard) try {
      ios_base::iostate state = 0; // все хорошо
      istreambuf_iterator<char> eos;
      string str;

      use_facet<money_get<char>>(s.getloc()).get(s,eos,true,state,str);

      if (state==0 || state==ios_base::eofbit) { // присваиваем только, если вызов get() успешен
          long long i = stoll(str);    // §36.3.5
          if (errno==ERANGE) {
              state |= ios_base::failbit;
          }
          else {
              m = i; // присваиваем только, если преобразование в long long успешно
          }
          s.setstate(state);
      }
  }
  catch (...) {
      handle_ioexception(s);   // см. §39.4.2.2
  }
  return s;
}
```

Я использую **get()**, который читает в **string**, потому что чтение в **double** и последующее конвертирование в **long long** может привести к потере точности.

Наибольшее значение, представимое через **long double**, может быть меньше максимального значения, представимого через **long long**.

39.4.4. Форматирование дат и времени

Форматы дат и времени контролируется через **time_get<C,In>** и **time_put<C,Out>**. Представление дат и времени используется в **tm** (§43.6).

39.4.4.1. time_put

Фасет **time_put** берет точку времени, представленную как **tm**, и производит последовательность символов, представляющую время, используя **strftime()** (§43.6) или ее эквиваленты:

Фасет time_put<C,Out = ostreambuf_iterator<C>> (§iso.22.4.5.1)
Out put(Out s, ios_base& f, C fill, const tm* pt, const C* b, const C* e) const;
Out put(Out s, ios_base& f, C fill, const tm* pt, char format, char mod = 0) const;
Out do_put(Out s, ios_base& ib, const tm* pt, char format, char mod) const;

Вызов **s=put(s,ib,fill,pt,b,e)** копирует [**b**:**e**) в поток вывода **s**. Для каждого форматирующего символа **x** (возврат **strftime()**) с опциональным модификатором **mod** он вызывает **do_put(s,ib,pt,x,mod)**. Возможные значения модификатора есть **0** (умолчание для «none»), **E** или **O**. При замещении **p=do_put(s,ib,pt,x,mod)** форматирует части у ***pt** в **s**, и возвращает значение, указывающее на позицию в **s** после последнего записанного символа.

Имеется **_byname-** версия (§39.4, §39.4.1) для **messages**:

```
template<class C, class Out = ostreambuf_iterator<C>>
class time_put_byname : public time_put<C,Out>
{
    // ...
};
```

39.4.4.2. time_get

Основная идея заключается в том, что **get_time** может читать то, что получено посредством **put_time**, используя тот же **strftime()** формат (§43.6):

```
class time_base {
public:
    enum dateorder {
        no_order,       // значит mdy
        dmy,            // значит "%d%m%y"
        mdy,            // значит "%m%d%y"
        ymd,            // значит %y%m%d"
        ydm             // значит %y%d%m"
    };
};

template<class C, class In = istreambuf_iterator<C>>
class time_get : public locale::facet, public time_base {
public:
    using char_type = C;
    using iter_type = In;
    // ...
}
```

Кроме чтения согласно формату есть еще операции для проверки **dateorder** и для чтения отдельных частей представления дат и времени, вроде **weekday** и **monthname**:

Фасет time_get<C,In> (§iso.22.4.5.1) Читает [b:e) в *pt
dateorder do_date_order() const;
In get_time(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt) const; In get_date(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt) const; In get_weekday(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt) const; In get_monthname(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt) const; In get_year(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt) const;
In get(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt, char format, char mod) const; In get(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt, char format) const; In get(In b, In e, ios_base& ib, ios_base::iostate& err, tm* pt, C* fmtb, C* fmte) const;

Функция **get_*()** читает из [**b**:**e**) в ***pt**, получая **locale** из **b**, и устанавливая **err** в случае ошибки. Она возвращает итератор, указывающий на первый непрочитанный символ из [**b**:**e**).

Вызов **p=get(b,e,ib,err,pt,format,mod)** читает согласно форматирующему символу **format** и модифицирующему символу **mod**, как это специфицировано посредством **strftime()**. Если **mod** не задано, то используется **mod==0**.

Вызов **get(b,e,ib,err,pt,fmtb,fmte)** использует формат (возврат функции **strftime()**) в виде строки **[fmtb:fmte)**. Эта перегруженная версия вместе с еще одной (имеющей умолчательный модификатор) не имеет интерфейсов **do_get()**. Вместо этого, они реализуются посредством вызовов **do_get()** для первого **get()**.

Основное применение фасетов дат и времени заключается в том, что они обеспечивают чувствительный к **locale** ввод/вывод для класса **Date**. Рассмотрим вариант **Date** из §16.3:

```
class Date {
public:
    explicit Date(int d ={}, Month m ={}, int year ={});
    // ...
    string to_string(const locale& = locale()) const;
};

istream& operator>>(istream& is, Date& d);
ostream& operator<<(ostream& os, Date d);
```

Имея **to_string()**, операция вывода становится тривиальной:

```
string Date::to_string(const locale& loc) const
{
    ostringstream os;
    os.imbue(loc);
    return os << *this;
}

ostream& operator<<(ostream& os, Date d)
{
    return os<<to_string(d,os.getloc());
}
```

Операция ввода должна следить за состоянием:

```
istream& operator>>(istream& is, Date& d)
{
    if (istream::sentry guard{is}) {
        ios_base::iostate err = goodbit;
        struct tm t;
        use_facet<time_get<char>>(is.getloc()).get_date(is,0,is,err,&t); // читаем в t
        if (!err) {
            Month m = static_cast<Month>(t.tm_mon+1);
            d = Date(t.tm_day,m,t.tm_year+1900);
        }
        is.setstate(err);
    }
    return is;
}
```

+1900 нужны, потому что **1900** является нулевым годом для **tm** (§43.6).

Имеется **_byname**-версия (§39.4, §39.4.1) для **messages**:

```
template<class C, class In = istreambuf_iterator<C>>
class time_get_byname : public time_get<C, In> {
    // ...
};
```

39.4.5. Классификация символов

Читая символы из входных потоков, бывает необходимо классифицировать символы, чтобы учитывать смысл того, что читается. Например, чтобы прочитать число, процедура ввода должна знать, какие символы являются цифрами. В §10.2.2 рассматривался вопрос о пользе классификации символов при разборе входных данных.

Естественно, что классификация символов зависит от используемого алфавита. Как следствие, фасет **ctype** назначается для классификации символов в **locale**.

Классы символов описываются перечислением, названным **mask**:

```
class ctype_base {
public:
    enum mask {        // реальные значения зависят от реализации
        space = 1,          // пробельные символы (в "C" locale: ' ', '\n', '\t', ...)
        print = 1<<1,       // символы печати
        cntrl = 1<<2,       // управляющие символы
        upper = 1<<3,       // символы в верхнем регистре
        lower = 1<<4,       // символы в нижнем регистре
        alpha = 1<<5,       // алфавитные символы
        digit = 1<<6,       // десятичные цифры
        punct = 1<<7,       // символы пунктуации
        xdigit = 1<<8,      // шестнадцатеричные цифры
        blank = 1 << 9;     // пробел и знак табуляции
        alnum=alpha|digit,  // алфавитно-цифровые символы
        graph=alnum|punct
    };
};

template<class C>
class ctype : public locale::facet, public ctype_base {
public:
    using char_type = C;
    // ...
};
```

Этот **mask** не зависит от конкретного типа символов. Поэтому это перечисление располагается в нешаблонном базовом классе.

Ясно, что **mask** соответствует традиционной классификации в C и C++ (§36.2.1). Однако, для различных наборов символов различные числовые значения для символов попадают в разные классы. Например, для набора символов ASCII целое число **125** представляет символ }, являющийся символом пунктуации (**punct**). Однако в датском национальном наборе символов **125** представляет гласную å, которая в датском **locale** должна классифицироваться как **alpha**.

Название классификации (mask — маска) отражает тот факт, что в традиционных эффективных реализациях для небольших наборов символов классификация реализуется в виде таблицы, в которой каждый вход содержит биты (маску), представляющие классификационную единицу. Например:

```
table['P'] == upper|alpha
table['a'] == lower|alpha|xdigit
table['1'] == digit|xdigit
table[' '] == space|blank
```

Для этой классификации **table[c]&m** ненулевое, если символ **c** есть **m**, и **0** в противном случае.

Фасет **ctype** определяется следующим образом:

Фасет ctype<C> (§iso.22.4.1.1)

bool is(mask m, C c) const;
const C* is(const C* b, const C* e, mask* v) const;
const C* scan_is(mask m, const C* b, const C* e) const;
const C* scan_not(mask m, const C* b, const C* e) const;

C toupper(C c) const;
const C* toupper(C* b, const C* e) const;
C tolower(C c) const;
const C* tolower(C* b, const C* e) const;

C widen(C c) const; const char* widen(const char* b, const char* e, C* b2) const;
char narrow(C c, char def) const; const C* narrow(const C* b, const C* e, char def, char* b2) const;

Вызов **is(m,c)** проверяет, принадлежит ли символ **c** к классификации **m**. Например:

```
int count_spaces(const string& s, const locale& loc)
{
    const ctype<char>& ct = use_facet<ctype<char>>(loc);
    int i = 0;

    for(auto p = s.begin(); p!=s.end(); ++p)
        if (ct.is(ctype_base::space,*p))        // символ пробела, в соответствии с ct
            ++i;
    return i;
}
```

Отметим, что также возможно использовать **is()** для проверки того, принадлежит ли символ к одной из классификаций. Например:

```
ct.is(ctype_base::space|ctype_base::punct,c); // c в ct – пробел. сим-л или знак пунктуации?
```

Вызов **is(b,e,v)** выясняет классификацию каждого символа в [**b**:**e**) и помещает ее в соответствующую позицию в массиве **v**.

Вызов **scan_is(m,b,e)** возвращает указатель на первый символ из [**b**:**e**), что принадлежит **m**. Если никакой символ не классифицируется как **m**, то возвращается **e**. Как всегда, для стандартных фасетов открытая функция-член реализуется посредством вызова соответствующей **do_** виртуальной функции. Вот возможная простая реализация:

```
template<class C>
const C* ctype<C>::do_scan_is(mask m, const C* b, const C* e) const
{
    while (b!=e && !is(m,*b))
        ++b;
    return b;
}
```

Вызов **scan_not(m,b,e)** возвращает указатель на первый символ из [**b**:**e**), который не есть **m**. Если все символы классифицируюися как **m**, то возвращается **e**.

Вызов **toupper(c)** возвращает версию символа **c** в верхнем регистре, если такая версия существует в наборе символов, и возвращается само **c** в противном случае.

Вызов **toupper(b,e)** конвертирует каждый символ из диапазона [**b**:**e**) в верхний регистр и возвращает **e**. Вот возможная простая реализация:

```
template<class C>
const C* ctype<C>::to_upper(C* b, const C* e)
{
    for (; b!=e; ++b)
        *b = toupper(*b);
    return e;
}
```

Функция **tolower()** аналогична функции **toupper()**, за исключением того, что она конвертирует символы в нижний регистр.

Вызов **widen(c)** преобразует символ **c** в соответствующее значение **C**. Если набор символов для **C** предоставляет несколько символов, соответствующих **c**, то стандарт постулирует выбор «простейшей разумной трансформации». Например:

```
wcout << use_facet<ctype<wchar_t>>(wcout.getloc()).widen('e');
```

Будет выведен разумный эквивалент для символа **e** из **locale** для **wcout**.

Трансляция между несвязанными представлениями символов, вроде ASCII и EBCDIC, может также быть выполнена посредством **widen()**. К примеру, предположим, что существует **locale ebcdic**:

```
char EBCDIC_e = use_facet<ctype<char>>(ebcdic).widen('e');
```

Вызов **widen(b,e,v)** берет каждый символ из диапазона **[b:e)** и помещает расширенную версию в соответствующую позицию в массиве **v**.

Вызов **narrow(ch,def)** производит значение **char**, соответствующее символу **ch** из типа **C**. Опять-таки, нужно использовать «простейшее разумное преобразование». Если такого соответствующего **char** нет, то возвращается **def**.

Вызов **narrow(b,e,def,v)** берет каждый символ из диапазона **[b:e)** и кладет его узкую версию на соответствующую позицию в массиве **v**.

Итак, **narrow()** конвертирует из набора широких символов в набор узких символов, а **widen()** выполняет обратную операцию. Для символа **c** из узкого набора можно ожидать:

```
c == narrow(widen(c),0)  // не гарантируется
```

Это имеет место при условии, что символ, представляемый **c**, имеет только одно представление в узком наборе. Это, однако, не гарантируется. Если символы, представимые как **char**, не являются подмножеством символов из широкого набора (**C**), то можно ожидать аномалий и потенциальных проблем в коде, трактующем символы традиционным образом.

Аналогично, для символа **ch** из широкого набора можно ожидать:

```
widen(narrow(ch,def)) == ch || widen(narrow(ch,def)) == widen(def)    // не гарантируется
```

Однако, несмотря на то, что это часто имеет место, это не может гарантироваться для набора символов, представимых несколькими значениями в широком наборе, и единственным значением в узком наборе. Например, цифра **7** часто имеет несколько разных представлений в широком наборе. Причина состоит в том, что в типичном случае широкий набор имеет несколько традиционных наборов в качестве подмножеств, и символы из узкого набора реплицируются в широком наборе ради простоты преобразований.

Для каждого символа из базового набора символов (§6.1.2) гарантируется, что

```
widen(narrow(ch_lit,0)) == ch_lit
```

Например:

```
widen(narrow('x',0)) == 'x'
```

Функции **narrow()** и **widen()** стараются уважать классификации символов по мере возможности. Например, если справедливо **is(alpha,c)**, то справедливо и **is(alpha,narrow(c,'a'))**, и **is(alpha,widen(c))**, пока **alpha** является действительной маской для используемого **locale**.

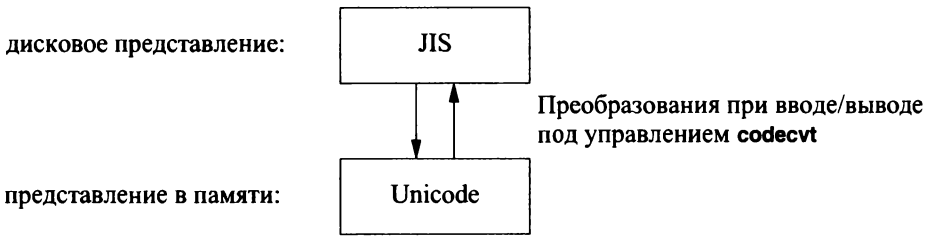
Реализация **iostream** критически зависит от рассмотренных средств. Полагаясь на **<iostream>** и **<string>**, пользователь может избежать многих случаев использования фасета **ctype**.

Предоставляется также и **_byname**-версия (§39.4, §39.4.1) для **ctype**:

```
template<class C>
class ctype_byname : public ctype<C> {
    // ...
};
```

39.4.6. Конверсия кодов символов

Иногда представление символов, хранимых в файле, отличается от желательного представления этих символов в памяти. Например, японские символы часто хранятся в файлах, где индикаторы указывают на то, каким из четырех общих наборов символов (kanjii, katakana, hiragana и romaji) соответствует данная последовательность символов. Смысл каждого символа зависит от индикатора (индикатор это shift-состояние); при этом только kanjii требует более одного байта в памяти для своего представления. В памяти этими символами легче манипулировать в многобайтовых представлениях, где каждый символ имеет один и тот же размер. Такие символы (например, Unicode символы) в типичном случае сопоставляются широкому типу (**wchar_t**; §6.2.3). Как следствие, фасет **codecvt** предоставляет механизм для конвертирования символов из одного представления в другое по мере того, как они читаются или пишутся. Например:



Этот механизм конвертации кодов достаточно общий, чтобы обеспечивать произвольные конверсии символьных представлений. Этот механизм позволяет нам писать программы с подходящим внутренним представлением символов (хранимых в **char**, **wchar_t** и т. д.), и принимать разные представления символов из потока ввода, используя соответствующий **locale** потоков **iostream**. Альтернатива может быть только в модификации самой программы или в конвертировании форматов входных и выходных файлов.

Фасет **codecvt** обеспечивает конверсию между различными наборами символов во время перемещения символов из буфера потока во внешнее хранилище:

```
class codecvt_base {
public:
    enum result { // индикаторы результата
        ok, partial, error, noconv
    };
};

template<class In, class Ex, class SS>
class codecvt : public locale::facet, public codecvt_base {
```

```
public:
    using intern_type = In;
    using extern_type = Ex;
    using state_type = SS;
    // ...
};
```

Фасет codecvt<In,Ex,SS> (§iso.22.5) using CI = const In; using CE = const Ex;
result in(SS& st, CE* b, CE* e, CE*& next, In* b2, In* e2, In*& next2) const;
result out(SS& st, CI* b, CI* e, CI*& next, Ex* b2, Ex* e2, Ex*& next2) const;
result unshift(SS& st, Ex* b, Ex* e, Ex*& next) const;
int encoding() const noexcept;
bool always_noconv() const noexcept;
int length(SS& st, CE* b, CE* e, size_t max) const;
int max_length() const noexcept;

Фасет **codecvt** используется **basic_filebuf** (§38.2.1) для чтения или записи символов. **basic_filebuf** получает этот фасет из **locale** потока (§38.1).

Шаблонный аргумент **State** это тип, используемый для хранилища state-состояния потока, который подвергается конверсии. Этот аргумент также может использоваться для идентификации различных конверсий путем своей специализации. Это полезно, поскольку символы различных кодировок (различных наборов символов) могут храниться в объектах одного и того же типа. Например:

```
class JISstate { /* .. */ };

p = new codecvt<wchar_t,char,mbstate_t>;    // стандартный char в wchar_t
q = new codecvt<wchar_t,char,JISstate>;     // JIS в wide char
```

Без разных **State**-аргументов не было бы способа, которым **facet** мог бы узнать, какую кодировку принять для потока **char**. Тип **mbstate_t** из файла **<cwchar>** или **<wchar.h>** определяет стандартную системную конверсию между **char** и **wchar_t**.

Можно создать новый **codecvt** как производный класс, идентифицируемый по имени. Например:

```
class JIScvt : public codecvt<wchar_t,char,mbstate_t> {
    // ...
};
```

Вызов **in(st,b,e,next,b2,e2,next2)** читает каждый символ из **[b:e)** и пытается его конвертировать. Если символ конвертируется, то **in()** пишет его конвертированную форму в соответствующую позицию диапазона **[b2:e2)**; в противном случае **in()** прекращает работу в этой точке. Перед возвратом **in()** сохраняет позицию за последним прочитанным символом в **next** (следующий символ для чтения). Возврат **result** функции **in()** указывает объем выполненной работы:

codecvt_base result (§iso.22.4.1.4)	
ok	Преобразованы все символы из интервала **[b:e)**
partial	Преобразованы не все символы из интервала **[b:e)**
error	Символ не может быть преобразован
noconv	Не потребовалось преобразований

Заметьте, что **partial**-конверсия не обязательно ошибочна. Возможно, следует прочесть больше символов, чтобы завершить мультибайтовый символ, пригодный для записи, или же нужно очистить выходной буфер для приема новых символов.

Аргумент **st** типа **state_type** индицирует состояние потока ввода символов в самом начале работы при вызове функции **in()**. Это важно, когда внешнее представление символов использует shift-состояния. Отметим, что **st** это аргумент в виде неконстантной ссылки: в конце вызова **st** содержит shift-состояние входной последовательности. Это помогает программисту иметь дело с **partial**-конверсиями и выполнять конверсию длинных последовательностей несколькими вызовами **in()**.

Вызов **out(st,b,e,next,b2,e2,next2)** конвертирует **[b:e)** из внутреннего во внешнее представление тем же способом, каким **in()** делает это в противоположном направлении.

Поток символов должен стартовать и финишировать в нейтральном (не-shift) состоянии. В типичном случае это **state_type{}**.

Вызов **unshift(ss,b,e,next)** смотрит на **st** и размещает символы из **[b:e)** как требуется для перевода последовательности символов назад в не-shift (**unshift**) состояние. Работа **unshift()** (результат этой работы) и использование **next** похожи на **out()**.

Вызов **length(st,b,e,max)** возвращает число символов, которые **in()** может конвертировать из **[b:e)**.

Возвращаемые **encoding()** значения означают:

- **-1** Кодировка внешних символов использует состояние (например, использует shift и unshift последовательности).
- **0** Кодировка использует варьируемое число байт для представления индивидуальных символов (бит в байте может указывать, один или два байта применяются для хранения символа).
- **n** Каждый символ внешнего представления использует **n** байт.

Вызов **always_noconv()** возвращает **true**, если не требуется никакая конверсия между внутренним и внешним наборами символов; **false** — противоположный вывод. Ясно, что **always_noconv()==true** открывает возможность для реализации максимальной эффективности за счет отсутствия обращения к конверсионным функциям.

Вызов **cvt.max_length()** возвращает максимальное значение, которое **cvt.length(ss,p,q,n)** может вернуть для корректного набора аргументов.

Простейшей конверсией, которую я только могу придумать, является конверсия к символам верхнего регистра. Это почти самая простая версия **codecvt**, которая все же реально выполняет полезную службу:

```
class Cvt_to_upper : public codecvt<char,char,mbstate_t> { // преобразовать в верх. регистр
public:
    explicit Cvt_to_upper(size_t r = 0) : codecvt(r) { }

protected:
    // читает внешнее представление – пишет внутреннее:
    result do_in(State& s,
            const char* from, const char* from_end, const char*& from_next,
            char* to, char* to_end, char*& to_next
        ) const override;

    // читает внутреннее представление – пишет внешнее:
    result do_out(State& s,
            const char* from, const char* from_end, const char*& from_next,
            char* to, char* to_end, char*& to_next
        ) const override;

    result do_unshift(State&, E* to, E* to_end, E*& to_next) const override { return ok; }

    int do_encoding() const noexcept override { return 1; }
    bool do_always_noconv() const noexcept override { return false; }
```

```
    int do_length(const State&, const E* from, const E* from_end, size_t max) const override;
    int do_max_length() const noexcept override;      // максимально возможная длина
};

codecvt<char,char,mbstate_t>::result
Cvt_to_upper::do_out(State& s,
                     const char* from, const char* from_end, const char*& from_next,
                     char* to, char* to_end, char*& to_next) const
{
    return codecvt<char,char,mbstate_t>::do_out(s,from,from_end,from_next,to,to_end,to_next);
}

codecvt<char,char,mbstate_t>::result
Cvt_to_upper::do_in(State& s,
                    const char* from, const char* from_end, const char*& from_next,
                    char* to, char* to_end, char*& to_next) const
{
    // ...
}

int main() // тривиальный тест
{
    locale ulocale(locale(), new Cvt_to_upper);

    cin.imbue(ulocale);

    for (char ch; cin>>ch; )
        cout << ch;
}
```

Имеется **_byname**-версия (§39.4, §39.4.1) для **codecvt**:

```
template<class I, class E, class State>
class codecvt_byname : public codecvt<I,E,State> {
    // ...
};
```

39.4.7. Сообщения

Большинство пользователей предпочитают общаться с программами на своем родном языке. Стандартная библиотека предоставляет простой механизм для хранения специфичного для **locale** набора строк, из которых программист может составлять простые сообщения. По сути, **messages** представляют собой базу данных «только для чтения»:

```
class messages_base {
public:
    using catalog = /* целый тип, зависящий от реализации */; // тип идентификатора каталога
};

template<class C>
class messages : public locale::facet, public messages_base {
public:
    using char_type = C;
    using string_type = basic_string<C>;
    // ...
};
```

Интерфейс **messages** довольно прост:

Фасет messages<C> (§iso.22.4.7.1)
catalog open(const string& s, const locale& loc) const;
string_type get(catalog cat, int set, int id, const basic_string<C>& def) const;
void close(catalog cat) const;

Вызов **open(s,loc)** открывает «каталог» сообщений **s** для схемы локализации **loc**. Каталог это набор строк, организованный способом, зависящим от реализации, и доступный посредством функции **messages::get()**. При неудаче возвращается отрицательное значение. Каталог должен быть открыт до первой попытки чтения функцией **get()**.

Вызов **close(cat)** закрывает каталог, идентифицируемый посредством **cat**, и освобождает все ресурсы, ассоциированые с этим каталогом.

Вызов **get(cat,set,id,"foo")** ищет сообщение, идентифицируемое посредством **(set,id)** в каталоге **cat**. Если строка находится, то возвращается эта строка; иначе **get()** возвращает умолчательную строку (здесь это **string("foo")**).

Вот пример фасета **messages** в реализации, в которой каталог сообщений есть вектор наборов сообщений и сообщение есть строка:

```
struct Set {
    vector<string> msgs;
};

struct Cat {
    vector<Set> sets;
};

class My_messages : public messages<char> {
    vector<Cat>& catalogs;
public:
    explicit My_messages(size_t = 0) :catalogs{*new vector<Cat>} { }

    catalog do_open(const string& s, const locale& loc) const; // открыть каталог s
    string do_get(catalog cat, int s, int m, const string&) const; // взять сообщ-е (s,m) из cat

    void do_close(catalog cat) const
    {
        if (catalogs.size()<=cat)
            catalogs.erase(catalogs.begin()+cat);
    }

    ~My_messages() { delete &catalogs; }
};
```

У **messages** все функции-члены константные, так что структура данных каталога (это **vector<Set>**) хранится вне фасета.

Сообщение выбирается при указании каталога, набора (set) внутри каталога и строки сообщения внутри набора. Строка предоставляется в виде аргумента, чтобы быть использованной по умолчанию, если никакое сообщение не находится внутри каталога:

```
string My_messages::do_g et(catalog cat, int set, int id, const string& def) const
{
    if (catalogs.size()<=cat)
        return def;
    Cat& c = catalogs[cat];
    if (c.sets.size()<=set)
        return def;
```

```
    Set& s = c.sets[set];
    if (s.msgs.size()<=msg)
        return def;
    return s.msgs[id];
}
```

Открытие каталога включает в себя чтение текстуального представления с диска в **Cat** структуру. Здесь я выбрал тривиальное для чтения представление. Наборы разделяются посредстовм <<< и >>>, и каждое сообщение является строкой текста:

```
messages<char>::catalog My_messages::do_open(const string& n, const locale& loc) const
{
    string nn = n + locale().name();
    ifstream f(nn.c_str());
    if (!f) return -1;

    catalogs.push_back(Cat{});              // делаем каталог в памяти
    Cat& c = catalogs.back();

    for(string s; f>>s && s=="<<<"; ) {     // читаем Set
        c.sets.push_back(Set{});
        Set& ss = c.sets.back();
        while (getline(f,s) && s != ">>>")  // читаем сообщение
            ss.msgs.push_back(s);
    }
    return catalogs.size()-1;
}
```

Вот простой пример использования:

```
int main()
    // тривиальный тест
{
    if (!has_facet<My_messages>(locale())) {
        cerr << "no messages facet found in" << locale().name() << '\n';
        exit(1);
    }

    const messages<char>& m = use_facet<My_messages>(locale());
    extern string message_directory;  // здесь я храню сообщения

    auto cat = m.open(message_directory,locale());
    if (cat<0) {
        cerr << "no catalog found\n";
        exit(1);
    }

    cout << m.get(cat,0,0,"Missed again!") << endl;
    cout << m.get(cat,1,2,"Missed again!") << endl;
    cout << m.get(cat,1,3,"Missed again!") << endl;
    cout << m.get(cat,3,0,"Missed again!") << endl;
}
```

Если имеется каталог

```
<<<
hello
goodbye
>>>
<<<
yes
no
maybe
>>>
```

то программа выводит следующее

```
hello
maybe
Missed again!
Missed again!
```

39.4.7.1. Использование сообщений из других facet

Дополнительно к тому, что **messages** являются репозитарием зависимых от **locale** строк, применяемых для коммуникации с пользователями, их еще можно использовать при хранении строк для других **facet**. Например, фасет **Season_io** (§39.3.2) мог бы быть написан следующим образом:

```
class Season_io : public locale::facet {
    const messages<char>& m;  // директория сообщений
    messages_base::catalog cat; // каталог сообщений
public:
    class Missing_messages { };

    Season_io(size_t i = 0)
        : locale::facet(i),
        m(use_facet<Season_messages>(locale())),
        cat(m.open(message_directory,locale()))
    {
        if (cat<0)
            throw Missing_messages();
    }

    ~Season_io() { } // чтобы можно было уничтожать объекты Season_io (§39.3)

    const string& to_str(Season x) const; // строковое представление x

    bool from_str(const string& s, Season& x) const; // поместить Season соответ-ий s в x

    static locale::id id; // объект идентифицирующий фасет (§39.2, §39.3, §39.3.1)
};

locale::id Season_io::id; // определяем идентифицирующий объект

string Season_io::to_str(Season x) const
{
    return m->get(cat,0,x,"no-such-season");
}

bool Season_io::from_str(const string& s, Season& x) const
{
    for (int i = Season::spring; i<=Season::winter; i++)
        if (m->get(cat,0,i,"no-such-season") == s) {
            x = Season(i);
            return true;
        }
    return false;
}
```

Эта версия, базирующуюся на **messages**, отличается от оригинального решения (§39.3.2) тем, что реализатор набора **Season**-строк для нового **locale** должен быть способным добавить их в каталог для **messages**. Это нетрудно для того, кто может добавлять новый **locale** в среду исполнения. Но поскольку **messages** предоставляют лишь интерфейс

«только для чтения», добавление нового набора строк сезонов может оказаться вне пределов возможностей прикладного программиста.

Имеется **_byname**-версия (§39.4, §39.4.1) для **messages**:

```
template<class C>
class messages_byname : public messages<C> {
    // ...
};
```

39.5. Полезные интерфейсы

Помимо своей роли в **iostream** средства **locale** не так-то просто использовать. И тут нам на помощь приходят полезные вспомогательные интерфейсы, упрощающие нотацию и минимизирующие вероятность ошибок.

39.5.1. Классификация символов

Чаще всего фасет **ctype** используется для того, чтобы узнать, принадлежит ли символ к данной классификации. Для этого предназначен ряд функций:

Классификация символов в соответствии с locale (§iso.22.3.3.1)	
isspace(c,loc)	Является ли **c** пробелом в **loc**?
isblank(c,loc)	Является ли **c** разделителем **loc**?
isprint(c,loc)	Является ли **c** печатным символом?
iscntrl(c,loc)	Является ли **c** управляющим символом?
isupper(c,loc)	Является ли **c** символом верхнего регистра?
islower(c,loc)	Является ли **c** символом нижнего регистра?
isalpha(c,loc)	Является ли **c** буквой?
isdigit(c,loc)	Является ли **c** десятичной цифрой?
ispunct(c,loc)	Является ли **c** ничем из вышеперечисленного?
isxdigit(c,loc)	Является ли **c** шестнадцатеричной цифрой?
isalnum(c,loc)	**isalpha(c)** или **isdigit(c)**
isgraph(c,loc)	**isalpha(c)** или **isdigit(c)** или **ispunct(c)** (внимание: не пробел)

Эти функции легко реализуются с использованием **use_facet**. Например:

```
template<class C>
inline bool isspace(C c, const locale& loc)
{
    return use_facet<ctype<C>>(loc).is(space,c);
}
```

Одноаргументные версии этих функций (§36.2.1) используют текущую глобальную схему локализации языка С. За исключением редких случаев, в которых эти глобальные схемы локализации яызков С и С++ разнятся (§39.2.1), мы можем трактовать одноаргументные версии как двухаргументные, примененные к **locale()**. Например:

```
inline int isspace(int i)
{
    return isspace(i,locale());  // практически одно и то же
}
```

39.5.2. Конверсия символов

Конверсия символов между нижним и верхним регистрами могут быть чувствительны к **locale**:

Конверсия символов (§iso.22.3.3.2.1)	
c2= toupper(c,loc)	use_facet<ctype<C>>(loc).toupper(c)
c2= tolower(c,loc)	use_facet<ctype<C>>(loc).tolower(c)

39.5.3. Конвертирование строк

Конверсии кодов символов могут быть чувствительны к **locale**. Шаблон класса **wstring_convert** реализует конверсии между широкими строками и строками байт. Он позволяет вам определить фасет конвертирования кодов (вроде **codecvt**) для выполнения конверсий без какого-либо влияния на потоки или схемы локализации. Например, вы можете использовать фасет конверсии кодов **codecvt_utf8**, чтобы напрямую выводить мультибайтовую UTF-8 последовательность в **cout** без изменения **locale** этого **cout**:

```
wstring_convert<codecvt_utf8<wchar_t>> myconv;
string s = myconv.to_bytes(L"Hello\n");
cout << s;
```

Определение **wstring_convert** достаточно традиционное:

```
template<class Codecvt,
         class Wc = wchar_t,
         class Wa = std::allocator<Wc>,     // аллокатор широких символов
         class Ba = std::allocator<char>    // аллокатор байт
    >
class wstring_conver t {
public:
    using byte_string = basic_string<char, char_traits<char>, Ba>;
    using wide_string = basic_string<Wc, char_traits<Wc>, Wa>;
    using state_type = typename Codecvt::state_type;
    using int_type = typename wide_string::traits_type::int_type;
    // ...
};
```

Конструкторы у **wstring_convert** позволяют специфицировать фасет конверсии символов, начальное состояние конверсии и значения, которые будут использоваться в случае ошибок:

wstring_convert<Codecvt,Wc,Wa,Ba> (§iso.22.3.3.2.2)	
wstring_convert cvt {};	wstring_conver t cvt {new Codecvt};
wstring_convert cvt {pcvt,state}	cvt использует фасет конверсии *pcvt и состояние конверсии state
wstring_convert cvt {pcvt};	wstring_convert cvt {pcvt,state_type{}};
wstring_convert cvt {b_err,w_err};	wstring_convert cvt{}; использ. b_error и w_err
wstring_convert cvt {b_err};	wstring_convert cvt{}; использ. b_error
cvt.~wstring_convert();	Деструктор
ws=cvt.from_bytes(c)	ws содержит char c, конвертированный в Wc'ы
ws=cvt.from_bytes(s)	ws содержит char'ы из s, конвертированные в Wc'ы; s есть строка в С-стиле или string
ws=cvt.from_bytes(b,e)	ws содержит char'ы из [b:e), конвертированные в Wc'ы

wstring_conver t<Codecvt,Wc,Wa,Ba> (§iso.22.3.3.2.2)	
s=cvt.to_bytes(wc)	s содержит wc, конвертированный в chars'ы
s=cvt.to_bytes(ws)	s содержит Wc'ы из ws, конвертированные в char'ы; ws есть строка в С-стиле или basic_string<Wc>
s=cvt.to_bytes(b,e)	s содержит Wc'ы из [b:e), конвертированные в char'ы
n=cvt.converted()	n есть число эл-ов, конвертированных cvt
st=cvt.state()	st есть состояние cvt

Если конверсия к **wide_string** не увенчается успехом, то функции **cvt**, сконструированного со строкой **w_err**, вернут эту строку (строка сообщения об ошибке); в ином случае они сгенерируют исключение **range_error**.

Если конверсия к **byte_string** не увенчается успехом, то функции **cvt**, сконструированного с неумолчательной строкой **b_err**, вернут эту строку (строка сообщения об ошибке); в ином случае они сгенерируют исключение **range_error**.

Пример:

```
void test()
{
    wstring_convert<codecvt_utf8_utf16<wchar_t>> converter;

    string s8 = u8"This is a UTF-8 string";
    wstring s16 = converter.from_bytes(s8);
    string s88 = converter.to_bytes(s16);

    if (s8!=s88)
        cerr <"Insane!\n";
}
```

39.5.4. Конвертирование буферов

Можно использовать фасет конверсии кодов (§39.4.6) для прямой записи в буфер и чтения из буфера потока (§38.6):

```
template<class Codecvt,
         class C = wchar_t,
         class Tr = std::char_traits<C>
    >
class wbuffer_convert
    : public std::basic_streambuf<C,Tr> {
public:
    using state_type = typename Codecvt::state_type;
    // ...
};
```

wbuffer_convert<Codecvt,C,Tr> (§iso.22.3.3.2.3)	
wbuffer_convert wb {psb,pcvt,state};	wb конвертирует streambuf *psb, используя конвертер *pcvt начальное состояние конверсии state
wbuffer_convert wb {psb,pcvt};	wbuffer_convert wb {psb,pcvt,state_type{}};
wbuffer_convert wb {psb};	wbuffer_convert wb {psb,new Codecvt{}};
wbuffer_convert wb {};	wbuffer_convert wb {nullptr};
psb=wb.rdbuf()	psb есть буфер потока wb
psb2=wb.rdbuf(psb)	Сделать w потоковым буфером *psb; *psb2 есть предыдущий потоковый буфер wb
t=wb.state()	t есть состояние конверсии wb

39.6. Советы

[1] Нужно учитывать, что любая нетривиальная программа будет использоваться в странах с разными культурными особенностями; §39.1.

[2] Не следует полагать, что все используют тот же самый набор символов, что и вы; §39.1, §39.4.1.

[3] Предпочитайте использование **locale** вместо переписывания кода для ввода/вывода, зависящего от культурных традиций; §39.1.

[4] Применяйте **locale** для того, чтобы учитывать внешние (не C++) стандарты; §39.1.

[5] Можно трактовать **locale** как контейнер для **facet**; §39.2.

[6] Не помещайте строки именования **locale** в тексты программ; §39.2.1.

[7] Старайтесь сосредоточить изменяемые **locale** в одном-двух местах программы; §39.2.1.

[8] Минимизируйте использование глобальной форматирующей информации; §39.2.1.

[9] Старайтесть использовать учитывающие локализацию сравнения и сортировки; §39.2.2, §39.4.1.

[10] Делайте фасеты неизменяемыми; §39.3.

[11] Пусть **locale** управляет временем жизни фасетов; §39.3.

[12] Можно создавать свои собственные **facet**; §39.3.2.

[13] При написании функций ввода/вывода, чувствительных к локализации, не забывайте обрабатывать исключения, генерируемые пользовательскими (замещающими) функциями; §39.4.2.2.

[14] Если вам нужны разделители в числах, используйте **numput**; §39.4.2.1.

[15] Используйте простой тип **Money** для хранения денежных сумм; §39.4.3.

[16] Используйте простые пользовательские типы для хранения значений, требующих ввода/вывода, чувствительного к **locale**; §39.4.3.

[17] Фасет **time_put** можно использовать как для времени в стиле **<chrono>**, так и в стиле **<ctime>**; §39.4.4.

[18] Отдавайте предпочтение функциям классификации символов, в которых явно используется **locale**; §39.4.5, §39.5.

40

Численные расчеты

Целью вычислений является понимание, а не цифры.
— Р. В. Хэмминг

Для студентов цифры часто служат дорогой к пониманию.
— А. Ральстон

- Введение
- Числовые пределы
  Макросы числовых пределов
- Стандартные математические функции
- Комплексные числа
- Численные массивы: **valarray**
  Конструкторы и присваивания; Индексирование; Операции; Срезы; **slice_array**; Обобщенные срезы
- Обобщенные численные алгоритмы
  accumulate(); **inner_product()**; **partial_sum()** и **adjacent_difference()**; **iota()**
- Случайные числа
  «Движки»; Аппаратные генераторы случайных чисел; Распределения; Случайные числа в языке C
- Советы

40.1. Введение

Язык C++ при своем создании не ориентировался специально на численные расчеты. Но численные расчеты часто сопутствуют другим предметным областям — организации и доступу к базам данных, решению сетевых проблем, графике, имитационным программам, финансовому анализу; так C++ стал привлекательным средством для выполнения вычислений в более общих или более крупных контекстах. Численные расчеты сами прошли большой путь развития, начиная с простых циклов поверх векторов чисел с плавающей запятой. Там, где для расчетов требуются более сложные структуры данных, там язык C++ показывает себя во всей красе. В результате, в настоящее время C++ широко применяется для научных вычислений, в технических расчетах, в финансовых областях в рамках изощренных вычислений с нетривиальными структурами данных. Настоящий раздел книги имеет дело с частями стандартной библиотеки, которые поддерживают вычисления. При этом я совершенно не пытаюсь обучать численным методам. Эта тема сама по себе исключительно важная и интересная. Для ее изучения вам потребуется хороший учебный курс по численным методам, или хотя бы неплохой учебник по этой теме.

Помимо данной главы мы ранее в Главе 29 детально рассматривали важную тему, относящуюся к численным расчетам: матрицы размерности N.

40.2. Числовые пределы

Чтобы серьезно работать с числами, нам нужно знать подробнее о свойствах встроенных числовых типов. Чтобы максимально использовать возможности аппаратуры, такие свойства делают зависимыми от реализации, а не определяемыми исключительно в рамках языка программирования (§6.2.8). Нам нужно, например, знать, какое максимальное значение для типа **int** реализуется на нашем компьютере? Каково наименьшее значение для типа **float**? Какое число бит формирует тип **char**?

Ответы на эти вопросы дают специализации шаблона **numeric_limits**, представленного в файле **<limits>**. Например:

```
void f(double d, int i)
{
    char classification[numeric_limits<unsigned char>::max()];

    if (numeric_limits<unsigned char>::digits==numeric_limits<char>::digits ) {
        // тип char – беззнаковый
    }

    if (i<numeric_limits<short>::min() || numeric_limits<short>::max()<i) {
        // i не может храниться в short без потери точности
    }

    if (0<d && d<numeric_limits<double>::epsilon()) d = 0;

    if (numeric_limits<Quad>::is_specialized) {
        // доступна информация о числовых пределах для типа Quad
    }
}
```

Каждая специализация предоставляет надлежащую информацию по типу ее аргумента. Таким образом, общий шаблон **numeric_limits** — это просто способ удобно отобразить набор констант и **constexpr**-функций:

```
template<typename T>
class numeric_limits {
public:
    static const bool is_specialized = false; // есть информация для numeric_limits<T>?
    // ... неинтересные умолчательные установки ...
};
```

Реальная информация сосредоточена в специализациях. Каждая реализация предоставляет специализации **numeric_limits** для фундаментальных числовых типов, а не для типов вроде **void**, перечислений и типов стандартной библиотеки вроде **complex<double>**.

Для интегрального типа, такого как **char**, лишь небольшая часть информации представляет интерес. Вот **numeric_limits<char>** для реализации, в которой **char** занимает 8 бит и является знаковым:

```
template<>
class numeric_limits<char> {
public:
    static const bool is_specialized = true; // да, информация есть

    static const int digits = 7;              // число бит (исключая знак)
```

```
    static const bool is_signed = true;      // в данной реализации char знаковый
    static const bool is_integer = true;     // char – это интегральный тип

    static constexpr char min() noexcept { return -128; }  // наименьшее значение
    static constexpr char max() noexcept { return 127; }   // наибольшее значение

    // масса объявлений, не имеющих отношение к char
};
```

Здесь все функции **constexpr**, так что их можно использовать там, где требуются константные выражения, и они не вносят дополнительных накладных расходов при выполнении программы.

Большинство членов **numeric_limits** предназначены для описания чисел с плавающей запятой. Например, вот возможная реализация для **float**:

```
template<>
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2;       // основание степени (в данном случае, двойки)
    static const int digits = 24;     // количество (двоичных) цифр в мантиссе
    static const int digits10 = 9;    // количество десятичных цифр в мантиссе

    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;

    static constexpr float min() noexcept { return 1.17549435E-38F; } // наименьш. положительное
    static constexpr float max() noexcept { return 3.40282347E+38F; } // наибольш. положительное
    static constexpr float lowest() noexcept { return -3.40282347E+38F; }  // наименьш. значение

    static constexpr float epsilon() noexcept { return 1.19209290E-07F; }
    static constexpr float round_error() noexcept { return 0.5F; } // наибольш. ошибка округления

    static constexpr float infinity() noexcept { return /* некоторое значение */; }

    static constexpr float quiet_NaN() noexcept { return /* некоторое значение */; }
    static constexpr float signaling_NaN() noexcept { return /* некоторое значение */; }
    static constexpr float denorm_min() noexcept { return min(); }

    static const int min_exponent = -125;
    static const int min_exponent10 = -37;
    static const int max_exponent = +128;
    static const int max_exponent10 = +38;

    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;

    static const bool is_iec559 = true;   // отвечает IEC-559
    static const bool is_bounded = true;
    static const bool is_modulo = false;
    static const bool traps = true;
    static const bool tinyness_before = true;

    static const float_round_style round_style = round_to_nearest;
};
```

Отметим, что **min()** описывает минимальное положительное нормализованное число и что **epsilon** это наименьшее положительное число с плавающей запятой, так что **1+epsilon-1** больше **0**.

Определяя скалярный тип в стилистике встроенных типов, будет неплохо предоставить подходящую специализацию для **numeric_limits**. Например, при реализации типа **Quad** с учетверенной точностью будет разумным предоставить **numeric_limits<Quad>**. При реализации типа, не имеющего с математической точки зрения числового смысла, например **Dumb_ptr**, первичный шаблон **numeric_limits<Dumb_ptr<X>>** должен иметь член **is_specialized** равным **false**, указывающий на отсутствие информации на эту тему.

Конечно, в теории можно представить пользовательские типа с **numeric_limits**, каким-то разумным образом описывающие свойства типа, не имеющего ничего общего с числами с плавающей запятой. Но тогда лучше использовать общую технику описания свойств типа, а не специализировать **numeric_limits** свойствами, не относящимися к стандартным.

40.2.1. Макросы числовых пределов

Язык C++ унаследовал от языка C макросы, описывающие свойства целых. Они находятся в файле **<climits>**:

Макросы целочисленных пределов (_iso.diff.library_, сокращено)	
CHAR_BIT	Число бит в **char** (обычно 8)
CHAR_MIN	Наименьшее значение **char** (возможно отрицательное)
CHAR_MAX	Наибольшее значение **char** (обычно 127, если **char** знаковый, и 255, если **char** беззнаковый)
INT_MIN	Наименьшее значение **int**
LONG_MAX	Наибольшее значение **long**

Имеются также аналогично поименованные макросы для типов **signed char**, **long long** и т.д.

Аналогично, **<cfloat>** и **<float.h>** определяют макросы, описывающие свойства чисел с плавающей запятой:

Макросы чисел с плавающей запятой (_iso.diff.library_, сокращено)	
FLT_MIN	Наименьш. положительное знач-е **float** (напр., **1.175494351e-38F**)
FLT_MAX	Наибольш. знач-е **float** (напр., **3.402823466e+38F**)
FLT_DIG	Точность чисел после запятой **float** (напр., **6**)
FLT_MAX_10_EXP	Наибольная десятичная экспонента **float** (напр., **38**)
DBL_MIN	Наименьшее значение **double**
DBL_MAX	Наибольш. знач-е **double** (напр., **1.7976931348623158e+308**)
DBL_EPSILON	Наименьш. **double**, при котором **1.0+DBL_EPSILON!=1.0**

Имеются аналогично поименованные макросы для **long double**.

40.3. Стандартные математические функции

В файле **<cmath>** находятся стандартные математические функции:

Стандартные математические функции	
abs(x)	Абсолютное значение
ceil(x)	Наименьшее целое **>= x**
floor(x)	Наибольшее целое **<= x**
sqrt(x)	Квадратный корень; **x** не должен быть отрицательным
cos(x)	Косинус

Продолжение

Стандартные математические функции	
sin(x)	Синус
tan(x)	Тангенс
acos(x)	Арккосинус; рез-т не отрицательный
asin(x)	Арксинус; возвращает ближайший к 0 результат
atan(x)	Арктангенс
sinh(x)	Гиперболический синус
cosh(x)	Гиперболический косинус
tanh(x)	Гиперболический тангенс
exp(x)	Экспонента по основанию **e**
log(x)	Натуральный логарифм, по основанию **e**; **x** должен быть положительным
log10(x)	Логарифм по основанию 10

Имеются версии, принимающие аргументы типа **float**, **double**, **long double** и **complex** (§40.4). Для всех функций тип возврата соответствует типу аргумента.

Об ошибках информирует **errno** из **<cerrno>** значением **EDOM** для ошибок области вычислений вообще, и значением **ERANGE** для ошибок, связанных с диапазоном допустимых значений. Пример:

```
void f()
{
    errno = 0;      // очистка старого состояния, инициирующего ошибку
    sqrt(-1);
    if (errno==EDOM) cerr << "sqrt() not defined for negative argument";

    pow(numeric_limits<double>::max(),2);
    if (errno == ERANGE) cerr << "result of pow() too large to represent as a double";
}
```

По историческим причинам часть математических функций располагается в файле **<cstdlib>**, а не в **<cmath>**:

Математические функции из <cstdlib> (§iso.26.8)	
n2=abs(n)	Абсолютное значение; **n** есть **int**, **long** или **long long**; **n2** имеет тот же тип, что и **n**
n2=labs(n)	«Абсолютное значение **long**»; **n** и **n2** имеют тип **long**
n2=llabs(n)	«Абсолютное значение **long long**»; **n** и **n2** имеют тип **long long**
p=div(n,d)	**p=div(n,d)**
p=ldiv(n,d)	Разделить **n** на **d**; **p** есть {частное,остаток}; **n** и **d** имеют тип **long**
p=lldiv(n,d)	Разделить **n** на **d**; **p** есть {частное,остаток}; **n** и **d** имеют тип **long long**

Версии **l*()** находятся здесь потому, что язык C не поддерживает перегрузку функций. Возвратами функций **ldiv()** являются структуры **div_t**, **ldiv_t** и **lldiv_t**, соответственно. Структуры имеют члены **quot** (частное) и **rem** (остаток) для пользовательских типов.

Имеется отдельный ISO стандарт для специальных математических функций [C++Math,2010]. Реализации могут добавлять эти функции в **<cmath>**:

Специальные математические функции (опционально)			
assoc_laguerre()	**assoc_legendre()**	**beta()**	**comp_ellint_1()**
comp_ellint_2()	**comp_ellint_3()**	**cyl_bessel_i()**	**cyl_bessel_j()**
cyl_bessel_k()	**cyl_neumann()**	**ellint_1()**	**ellint_2()**
ellint_3()	**expint()**	**hermite()**	**laguerre()**
legendre()	**riemann_zeta()**	**sph_bessel()**	**sph_legendre()**
sph_neumann()			

Если вам эти функции не знакомы, то скорее всего они вам и не потребуются.

40.4. Комплексные числа

В стандартной библиотеке определяются комплексные числа типов **complex<float>**, **complex<double>** и **complex<long double>**. Тип **complex<Scalar>**, где **Scalar** есть некий тип, поддерживающий обычные арифметические операции, скорее всего работает, но не гарантируется, что будет переносимым.

```
template<typename Scalar>
class complex {
    // complex – пара скалярных величин, практически пара координат
    Scalar re, im;
public:
    complex(const Scalar & r = Scalar{}, const Scalar & i = Scalar{}) :re(r), im(i) { }

    Scalar real() const { return re; }   // вещественная часть
    void real(Scalar r) { re=r; }

    Scalar imag() const { return im; } // мнимая часть
    void imag(Scalar i) { im = i; }

    template<typename X>
        complex(const complex<X>&);

    complex<T>& operator=(const T&);
    complex& operator=(const complex&);
    template<typename X>
        complex<T>& operator=(const complex<X>&);

    complex<T>& operator+=(const T&);
    template<typename X>
        complex<T>& operator+=(const complex<X>&);

    // аналогично: -=, *=, /=
};
```

Стандартный тип **complex** не защищает от проблем с сужением значений:

```
complex<float> z1 = 1.33333333333333333;      // сужается
complex<double> z2 = 1.33333333333333333;     // сужается
z1=z2;                                        // сужается
```

Во избежание неожиданных эффектов сужения значений используйте {}-инициализацию:

```
complex<float> z3 {1.33333333333333333};   // ошибка: преобразование с сужением
```

В файле **<complex>** определяется также масса полезных операций:

Операции над complex	
z1+z2	Сложение
z1-z2	Вычитание
z1*z2	Умножение
z1/z2	Деление
z1==z2	Равенство
z1!=z2	Неравенство

Продолжение

Операции над complex	
norm(z)	**Квадрат abs(z)**
conj(z)	Сопряженное число: **{z.re,-z.im}**
polar(x,y)	Создать комплексное число по данным полярным координатам (ро,тета)
real(z)	Вещественная часть
imag(z)	Мнимая часть
abs(z)	Расстояние от (0,0): **sqrt(z.re*z.re+z.im*z.im)**; также известно, как ро
arg(z)	Угол от полож-ной вещественной оси: **atan2(z.im/z.re)**; также известен, как тета
out<<z	Вывод комплексного числа
in>>z	Ввод комплексного числа

Стандартные математические функции (§40.3) доступны и для комплексных функций. Заметьте, что тип **complex** не предоставляет операций **<** и **%**. Большие подробности приведены в §18.3.

40.5. Численные массивы: valarray

Многие численные алгоритмы опираются на относительно несложный одномерный вектор с элементами, имеющими числовой тип с плавающей запятой. Широко используются библиотеки, использующие этот тип, и в них реализуется специальная оптимизация, обязательная для многих задач. Тип **valarray** из **<valarray>** как раз и является таковым вектором — массивом из чисел с плавающей запятой. Он предоставляет обычные векторные математические операции для массивов плюс поддержку срезов.

Классы численных массивов (§iso.26.6.1)	
valarray<T>	Численный массив эл-ов типа T
slice	BLAS-образный срез (начало, длина, and и шаг); §40.5.4
slice_array<T>	Подмножество массива, описанное посредством среза; §40.5.5
gslice	Обобщенный срез, описывающий матрицу
gslice_array<T>	Подмножество матрицы, описанное посредством обобщ. среза; §40.5.6
mask_array<T>	Подмножество массива, описанное посредством маски; §40.5.2
indirect_array<T>	Подмножество массива, описанное посредством списка индексов; §40.5.2

Тип **valarray** в своей первоначальной идее призван был предоставить традиционные для языка Fortran средства обработки плотных многомерных массивов с агрессивной оптимизацией. Это достигается при активной помощи со стороны компилятора и вспомогательных средств, ассоциированных с типом **valarray**.

40.5.1. Конструкторы и присваивания

Конструкторы типа **valarray** позволяют выполнить инициализацию из вспомогательных типов числовых массивов и из одиночных значений:

Конструкторы valarray<T> (§iso.26.6.2.2)	
valarray va{};	**valarray** без элементов
valarray va {n};	**valarray** из **n** эл-ов со значением **T{}**; explicit
valarray va {t,n};	**valarray** из **n** эл-ов со значением **t**
valarray va {p,n};	**valarray** из **n** эл-ов со значениями, скопированными из **[p:p+n)**
valarray va {v2};	Перемещающий и копирующий конструкторы

Продолжение

Конструкторы valarray<T> (§iso.26.6.2.2)	
valarray va {a};	Создать **va** с эл-ами из **a**; а может быть **slice_array, gslice_array, mask_array** или **indirect_array**; число эл-ов есть число эл-ов **a**
valarray va {args};	Создать из **initializer_list {args}**; число эл-ов есть число эл-ов **{args}**
va.~valarray()	Деструктор

Например:

```
valarray<double> v0;                  // можно присвоить что-нибудь позже
valarray<float> v1(1000);             // 1000 элементов со значением float()==0.0F

valarray<int> v2(-1,2000);            // 2000 элементов со значением -1
valarray<double> v3(100,9.8064);      // ошибка: размер задан числом с плав. запятой

valarray<double> v4 = v3;             // v4 содержит v3.size() элементов

valarray<int> v5 {-1,2000};           // два элемента
```

В конструкторах с двумя аргументами значение идет перед числом элементов Это отличается от соглашения для стандартных контейнеров (§31.3.2).

Число элементов в аргументе **valarray** для копирующего конструктора определяет размер результирующего **valarray**.

Дополнительно к спискам инициализаторов поддерживается конструктор, который копирует элементы из встроенных массивов. Например:

```
void f(const int* p, int n)
{
    const double vd[] = {0,1,2,3,4};
    const int vi[] = {0,1,2,3,4};

    valarray<double> v1{vd,4};        // 4 элемента: 0,1,2,3
    valarray<double> v2{vi,4};        // ошибка типа: vi – это не указатель на double
    valarray<double> v3{vd,8};        // ошибка: слишком мало элементов для инициализации
    valarray<int> v4{p,n};            // лучше, если p указывает на как минимум n int'ов
}
```

Тип **valarray** и ассоциированные с ним средства разработаны с целью эффективных вычислений. Для этого реализатору типа **valarray** разрешено использовать любые схемы оптимизации, которые только физически возможны. Операции с **valarray** должны быть свободны от побочных эффектов (за исключением явных аргументов, конечно), для **valarray** не предоставляются никакие синонимы, а введение вспомогательных типов и вопрос об устранении временных объектов допускаются до тех пор, пока поддерживается базовая семантика. Нет никаких проверок корректности диапазонов. Элементы для **valarray** должны поддерживать умолчательную копирующую семантику (§8.2.6).

Присваивать можно другие **valarray**, скаляры или подмножества **valarray**:

Присваивания valarray<T> (§iso.26.6.2.3)	
va2=va	Копирующее присваивание: **va2.size()** становится**va.size()**
va2=move(va)	Перемещающее присваивание: **va** становится пустым
va=t	Скалярное присваивание: каждый эл-т **va** есть копия **t**

Продолжение

Присваивания valarray<T> (§iso.26.6.2.3) (§iso.26.6.2.3)	
va={args}	Присваивание из initializer_list {args}; число эл-ов va становится {args}.size()
va=a	Присваивание из a; a.size() должно быть равно va.size(); a может быть slice_array, gslice_array, mask_array, или indirect_array
va@=va2	v[i]@=va2[i] для каждого эл-а va; @ может быть /, %, +, -, ^, &, |, <<, или >>
va@=t	v[i]@=t для каждого эл-а va; @ может быть /, %, +, -, ^, &, |, <<, или >>

Для присваивания **valarray** должно быть совпадение размеров (элементы копируются в соответствующие позиции). В случае разных размеров результат копирования не определен.

Можно также присваивать скаляры. Например, **v=7** присваивает **7** каждому элементу **v** (имеющему тип **valarray**). Это, конечно, может оказаться сюрпризом для многих программистов и должен рассматриваться как случайный, но полезный вырожденный случай. Пример:

```
valarray<int> v {1,2,3,4,5,6,7,8};
v *= 2;    // v=={2,4,6,10,12,14,16}
v = 7;     // v=={7,7,7,7,7,7,7,7}
```

40.5.2. Индексирование

Индексирование применяется для выбора элемента **valarray** или для выбора подмножества его элементов:

Индексирование valarray<T> (§iso.26.6.2.4, §iso.26.6.2.5)	
t=va[i]	Индексирование: t есть ссылка на i-ый эл-т va; нет проверки диапазона
a2=va[x]	Подмножество: x есть slice, gslice, valarray<bool> или valarray<size_t>

Каждый **operator[]** возвращает подмножество элементов из **valarray**. Тип возврата (тип объекта, представляющего подмножество) зависит от типа аргумента.

Для константных аргументов результат содержит копии элементов. Для неконстантных аргументов результат содержит ссылки на элементы. Поскольку C++ напрямую не поддерживает массивы ссылок (нельзя написать **valarray(int&)**), то реализации призваны закамуфлировать это тем или иным способом. Это должно реализовываться эффективным образом. В любом случае индекс должен описывать, какие элементы возвращаются, а присваиваемый массив должен иметь соответствующую длину и тип элементов:

- Срез (**slice**) от **const valarray**:

```
valarray<T> operator[](slice) const;    // копии элементов
// ...
const valarray<char> v0 {"abcdefghijklmnop",16};
valarray<char> v1 {v0[slice(2,5,3)]};   // {"cfilo",5}
```

- Срез от неконстантного **valarray**:

```
slice_array<T> operator[](slice);       // ссылки на элементы
// ...
valarray<char> v0 {"abcdefghijklmnop",16};
valarray<char> v1 {"ABCDE",5};
v0[slice(2,5,3)] = v1;                  // v0=={"abAdeBghCjkDmnEp",16}
```

- **gslice** для **const valarray**:

```
valarray<T> operator[](const gslice&) const;     // копии элементов
// ...
const valarray<char> v0 {"abcdefghijklmnop",16};
const valarray<size_t> len {2,3};
const valarray<size_t> str {7,2};
valarray<char> v1 {v0[gslice(3,len,str)]};     // v1=={"dfhkmo",6}
```

- **gslice** для неконстантного **valarray**:

```
gslice_array<T> operator[](const gslice&); // ссылки на элементы
// ...
valarray<char> v0 {"abcdefghijklmnop",16};
valarray<char> v1 {"ABCDE",5};
const valarray<size_t> len {2,3};
const valarray<size_t> str {7,2};
v0[gslice(3,len,str)] = v1;          // v0=={"abcAeBgCijDlEnFp",16}
```

- **valarray<bool>** (маска) от **const valarray**:

```
valarray<T> operator[](const valarray<bool>&) const;     // копии элементов
// ...
const valarray<char> v0 {"abcdefghijklmnop",16};
const bool vb[] {false, false, true, true, false, true};
valarray<char> v1 {v0[valarray<bool>(vb, 6)]};   // v1=={"cdf",3}
```

- **valarray<bool>** от неконстантного **valarray**:

```
mask_array<T> operator[](const valarray<bool>&);    // ссылки на элементы
// ...
valarray<char> v0 {"abcdefghijklmnop", 16};
valarray<char> v1 {"ABC",3};
const bool vb[] {false, false, true, true, false, true};
v0[valarray<bool>(vb,6)] = v1;          // v0=={"abABeCghijklmnop",16}
```

- **valarray<size_t>** (набор индексов) от **const valarray**:

```
valarray<T> operator[](const valarray<size_t>&) const;     // ссылки на элементы
// ...
const valarray<char> v0 {"abcdefghijklmnop",16};
const size_t vi[] {7, 5, 2, 3, 8};
valarray<char> v1 {v0[valarray<size_t>(vi,5)]};          // v1=={"hfcdi",5}
```

- **valarray<size_t>** (набор индексов) от неконстантного **valarray**:

```
indirect_array<T> operator[](const valarray<size_t>&);     // ссылки на элементы
// ...
valarray<char> v0 {"abcdefghijklmnop",16};
valarray<char> v1 {"ABCDE",5};
const size_t vi[] {7, 5, 2, 3, 8};
v0[valarray<size_t>(vi,5)] {v1};          // v0=={"abCDeBgAEjklmnop",16}
```

Заметьте, что при индексировании с маской (**valarray<bool>**) получается **mask_array**, а при индексировании набором индексов (**valarray<size_t>**) получается **indirect_array**.

40.5.3. Операции

Основная цель **valarray** — поддержка вычислений, так что большой набор операций поддерживается непосредственно:

Функции-члены valarray<T> (§iso.26.6.2.8)	
va.swap(va2)	Обменять элементы va и va2; noexcept;
n=va.size()	n есть число элементов va
t=va.sum()	t есть сумма элементов va, вычисленная используя +=
t=va.min()	t есть наименьший эл-т va, найденный используя <
t=va.max()	t есть наибольший эл-т va, найденный используя <
va2=va.shift(n)	Линейный правый сдвиг эл-ов
va2=va.cshift(n)	Круговой правый сдвиг эл-ов
va2=va.apply(f)	Применить f: знач-е каждого эл-а va2[i] есть va[f(i)]
va.resize(n,t)	Сделать из va valarray с n эл-ов со значением t
va.resize(n)	va.resize(n,T{})

Проверка диапазонов отсутствует: результат попытки функции обратиться к элементу пустого **valarray** не определен.

Заметьте, что **resize()** не сохраняет никаких старых значений.

Операции над valarray<T> (§iso.26.6.2.6, §iso.26.6.2.7) v или v2, но не оба, могут быть скалярами; рез-т для арифм. операций есть valarray<T>	
swap(va,va2)	va.swap(va2)
va3=va@va2	Произвести @ над эл-ами va и va2, получая va3; @ может быть +, -, *, /, %, &, |, ^, << ,>>, &&, ||
vb=v@v2	Произвести @ над эл-ами v и v2, получая valarray<bool>; @ может быть ==, !=, <, <=, >, >=
v2=@(v)	Произвести @() над эл-ами v, получая v2; @ может быть abs, acos, asin, atan, cos, cosh, exp, log, log10
v3=atan2(v,v2)	Произвести atan2() над эл-ами v и v2
v3=pow(v,v2)	Произвести pow() над v и v2
p=begin(v)	p есть итератор произвольного доступа, указывающий на первый эл-т v
p=end(v)	p есть итератор произвольного доступа, указывающий на эл-т v, следующий за последним

Бинарные (двухоперандные) операции определены для двух **valarray** и для комбинации **valarray** с его скалярным типом. Скалярный тип трактуется как **valarray** правильного размера с каждым элементом, имеющим скалярное значение. Например:

```
void f(valarray<double>& v, valarray<double>& v2, double d)
{
    valarray<double> v3 = v*v2;     // v3[i] = v[i]*v2[i] для всех i
    valarray<double> v4 = v*d;      // v4[i] = v[i]*d для всех i
    valarray<double> v5 = d*v2;     // v5[i] = d*v2[i] для всех i
    valarray<double> v6 = cos(v);   // v6[i] = cos(v[i]) для всех i
}
```

Векторые операции приложимы ко всем элементам своего операнда способом, указуемым через * и **cos()** в показанных примерах. Естественно, операцию можно использовать лишь тогда, когда соответствующая операция приложима к скалярному типу. В противном случае компилятор выдаст ошибку при попытке специализировать операцию или функцию.

Там, где результат есть **valarray**, его длина равна длине операнда. При различающихся длинах результат двухоперандной операции не определен.

Эти операции возвращают новые объекты типа **valarray**, а не модифицируют свои операнды. Это конечно дорого, но это не так страшно при агрессивной оптимизации.

Например, если **v** есть **valarray**, то его можно масштабировать следующим орбразом: **v*=0.2**, или **v/=1.3**. То есть, применение скаляра к вектору означает его применение к каждому элементу вектора. Как всегда, ***=** есть более краткая запись для комбинации * и = (§18.3.1), которую компилятору легче оптимизировать.

Отметим, что операции, не имеющие дело с присваиваниями, создают новые **valarray**. Например:

```
double incr(double d) { return d+1; }

void f(valarray<double>& v)
{
    valarray<double> v2 = v.apply(incr);    // получаем инкрементированный valarray
    // ...
}
```

Здесь значение **v** не изменяется. К сожалению, **apply()** не принимает функциональный объект (§3.4.3, §11.4) в качестве аргумента.

Логические и циклические сдвиги, **shift()** и **cshift()**, возвращают новый **valarray** с элементами, соответственно сдвинутыми, а оригинальные элементы остаются нетронутыми. Например, циклический сдвиг **v2=v.cshitf(n)** производит **valarray** так, что **v2[i]==v[(i+n)%v.size()]**. Логический сдвиг **v3=v.shift(n)** производит **valarray** так, что **v3[n]** есть **v[i+n]**, если **i+n** является корректным индексом для **v**. Иначе, результат есть умолчательное значение элементов. Это подразумевает, что **shift()** и **cshift()** сдвигают влево при положительном аргументе, и сдвигают вправо при отрицательном аргументе. Например:

```
void f()
{
    int alpha[] = { 1, 2, 3, 4, 5 ,6, 7, 8 };
    valarray<int> v(alpha,8);          // 1, 2, 3, 4, 5, 6, 7, 8
    valarray<int> v2 = v.shift(2);     // 3, 4, 5, 6, 7, 8, 0, 0
    valarray<int> v3 = v<<2;           // 4, 8, 12, 16, 20, 24, 28, 32
    valarray<int> v4 = v.shift(-2);    // 0, 0, 1, 2, 3, 4, 5, 6
    valarray<int> v5 = v>>2;           // 0, 0, 0, 1, 1, 1, 1, 2
    valarray<int> v6 = v.cshift(2);    // 3, 4, 5, 6, 7, 8, 1, 2
    valarray<int> v7 = v.cshift(-2);   // 7, 8, 1, 2, 3, 4, 5, 6
}
```

Для типа **valarray** операции >> и << являются побитовыми сдвигающими операциями, а не сдвигами элементов или операциями ввода/вывода. Следовательно, **<<=** и **>>=** могут использоваться для сдвига битов внутри элементов интегральных типов. Например:

```
void f(valarray<int> vi, valarray<double> vd)
{
    vi <<= 2;      // vi[i]<<=2 для всех элементов vi
    vd <<= 2;      // ошибка: сдвиг не определен для значений с плав. запятой
}
```

Все операции и математические функции для **valarray** могут также применяться и к **slice_array** (§40.5.5), **gslice_array** (§40.5.6), **mask_array** (§40.5.2), **indirect_array** (§40.5.2) и к комбинации этих типов. Реализациям оставлено право конвертировать операнды, которые не есть **valarray**, к типу **valarray** перед выполнением требуемой операции.

40.5.4. Срезы

slice есть абстракция, которая позволяет нам эффективно манипулировать одномерным массивом (например, встроенным массивом, **vector** или **valarray**) как матрицей произ-

вольной размерности. Это ключевое понятие для векторов в языке Fortran и в языке математической библиотеки BLAS (Basic Linear Algebra Subprograms), что является базисом многих численных алгоритмов. В своей основе, **slice** это каждый **n**-ый элемент некоторой части массива:

```
class std::slice {
    // начальный индекс, длина и шаг
public:
    slice();                        // slice{0,0,0}
    slice(size_t start, size_t size, size_t stride);

    size_t start() const;           // индекс первого элемента
    size_t size() const;            // число элементов
    size_t stride() const;          // n-ый элемент находится по адресу start()+n*str ide()
};
```

Здесь *stride* — это расстояние (число элементов) между двумя элементами **slice**. Таким образом, **slice** осуществляет соответствие (отображение) неотрицательных чисел в индексы. Число элементов (**size()**) не влияет на отображение (адресацию), но позволяет нам найти конец последовательности. Такое отображение может использоваться для эмуляции двухмерных массивов внутри одномерного массива (например, **valarray**) общим, эффективным и достаточно простым способом. Рассмотрим матрицу 3x4 (три ряда, каждый по четыре элемента):

```
valarray<int> v {
    {00,01,02,03},  // строка 0
    {10,11,12,13},  // строка 1
    {20,21,22,23}   // строка 2
};
```

или графически:

00	01	02	03
10	11	12	13
20	21	22	23

Следуя обычным соглашениям языков C и C++, раскладка **valarray** в памяти такова, что сначала идут элементы строки смежным образом:

```
for (int x : v) cout << x << ' ';
```

Отсюда получается следующий результат:

```
0 1 2 3 10 11 12 13 20 21 22 23
```

или графически:

column 0: 0 4 8

00	01	02	03	10	11	12	13	20	21	22	23

row 0: 0 1 2 3

Ряд **x** описывается посредством **slice(x*4,4,1)**. То есть, первый элемент ряда **x** есть **x*4**-ый элемент вектора, следующий элемент ряда есть **(x*4+1)**-ый элемент и т. д., и в каждом ряде содержится **4** элемента. Например, **slice{0,4,1}** описывает первый ряд **v** (ряд **0**): **00**, **01**, **02**, **03**, а **slice{1,4,1}** описывает второй ряд (ряд **1**).

Столбец y определяется посредством **slice(y,3,4)**. То есть, первый элемент столбца y является y-ым элементом вектора, следующий элемент этого столбца есть **(y+4)**-ый элемент вектора и т. д., а в каждом столбце **3** элемента. Например, **slice{0,3,4}** описывает первый столбец (столбец **0**): **00**, **10**, **20**, а **slice{1,3,4}** описывает второй столбец (столбец **1**).

Кроме эмуляции двумерных массивов **slice** может описывать много других последовательностей. Это достаточно общий способ специфицирования простых последовательностей. Данное понятие будет позже изучено дополнительно в §40.5.6.

Можно думать о **slice** (о срезе) как о странном виде итераторов: **slice** позволяет описывать последовательность индексов для **valarray**. На этой основе мы можем построить итератор в STL-стиле:

```
template<typename T>
class Slice_iter {
    valarray<T>* v;
    slice s;
    size_t curr;    // индекс текущего элемента

    T& ref(size_t i) const { return (*v)[s.start()+i*s.stride()]; }
public:
    Slice_iter(valarray<T>* vv, slice ss, size_t pos =0)
        :v{vv}, s{ss}, curr{0} { }

    Slice_iter end() const { return {this,s,s.size()}; }

    Slice_iter& operator++() { ++curr; return *this; }
    Slice_iter operator++(int) { Slice_iter t = *this; ++curr; return t; }

    T& operator[](size_t i) { return ref(i); }  // индексация в стиле C
    T& operator()(size_t i) { return ref(i); }  // индексация в стиле Fortran
    T& operator*() { return ref(curr); }        // текущий элемент

    bool operator==(const Slice_iter& q) const
    {
        return curr==q.curr && s.stride()==q.s.stride() && s.start()==q.s.start();
    }

    bool operator!=(const Slice_iter& q ) const
    {
        return !(*this==q);
    }

    bool operator<(const Slice_iter& q) const
    {
        return curr<q.curr && s.stride()==q.s.stride() && s.start()==q.s.start();
    }
};
```

Так как **slice** имеет размер, мы даже можем выполнить проверку диапазона. Здесь я воспользовался **slice::size()** для того, чтобы определить функцию **end()** для предоставления итератора на элемент, следующий за последним элементом **slice**.

Так как **slice** может описывать строку или столбец, то **Slice_iter** позволяет выполнить траверс **valarray** по строке или по столбцу.

40.5.5. slice_array

Исходя из **valarray** и **slice** мы можем построить нечто, что выглядит и ведет себя как **valarray**, а на самом деле есть просто способ отсылки к подмножеству массива, описанного посредством среза:

slice_array<T> (§iso.26.6.5)	
slice_array sa {sa2};	Копирующий конструктор: sa ссылается на те же эл-ы, что и sa2
sa=sa2	Присвоить эл-т, на к-рый ссылается sa[i], каждому соответствующему эл-у, на к-рый ссылается sa2[i]
sa=va	Присвоить va[i] каждому соответствующему элементу, на к-рый ссылается sa[i]
sa=v	Присвоить v каждому эл-у, на к-рый ссылается sa
sa@=va	sa[i]@=va[i] для каждого эл-а sa; @ может быть /, %, +, -, ^, &, |, <<, или >>

Пользователь не может создавать **slice_array** напрямую. Вместо этого, он индексирует **valarray** для создания **slice_array** для заданного среза. Как только **slice_array** инициализирован, все ссылки на него косвенно идут на **valarray**, для которого он был создан. Например, мы можем создать что-то, что представляет каждый второй элемент массива следующим образом:

```
void f(valarray<double>& d)
{
   slice_array<double>& v_even = d[slice(0,d.size()/2+d.size()%2,2)];
   slice_array<double>& v_odd = d[slice(1,d.size()/2,2)];

   v_even *= v_odd; // перемножить элем-ты попарно и сохранить результат в четных элем-ах
   v_odd = 0;       // присвоить 0 каждому нечетному элементу d
}
```

slice_array можно копировать. Например:

```
slice_array<double> row(valarray<double>& d, int i)
{
     slice_array<double> v = d[slice(0,2,d.size()/2)];
     // ...
     return d[slice(i%2,i,d.size()/2)];
}
```

40.5.6. Обобщенные срезы

slice (§29.2.2, §40.5.4) может описывать строку или столбец n-мерного массива. Однако, иногда нам нужно извлечь подмассив, который не является строкой или столбцом. Например, мы могли бы извлечь матрицу 3x2 из верхнего-левого угла матрицы 4x3:

00	01	02
10	11	12
20	21	22
30	31	32

К сожалению, эти элементы не располагаются в памяти способом, который можно было бы описать единственным **slice**:

subarray:

0	1	2	3	4	5
00	01	02	10	11	12	20	21	22	30	31	32

«Обобщенным срезом» служит **gslice**, который содержит (почти) всю информацию из n срезов:

```
class std::gslice {
    // вместо одного шага и единственного slice, gslice содержит n шагов и n размеров
public:
    gslice();
    gslice(size_t sz, const valarray<size_t>& lengths, const valarray<size_t>& strides);

    size_t start()const;                    // индекс первого элемента
    valarray<size_t> size() const;          // число элементов измерения
    valarray<size_t> stride() const;        // шаг для index[0], index[1], ...
};
```

Добавочные значения позволяют **gslice** специфицировать отображение между **n** целыми и индексом, который должен использоваться для адресации элементов в массиве. Например, мы можем описать раскладку 3x2 матрицы парой (длина, шаг)-пар:

```
size_t gslice_index(const gslice& s, size_t i, size_t j) // max (i,j) для соответств. index
{
   return s.start()+i*s.stride()[0]+j*s.stride()[1];
}

valarray<size_t> lengths {2,3};     // 2 элемента в первом измерении
                                    // 3 элемента во втором измерении
valarray<size_t> strides {3,1};     // 3 – шаг первого индекса
                                    // 1 – шаг второго индекса

void f()
{
   gslice s(0,lengths,strides);

   for (int i=0; i<3; ++i)                          // строка
      for (int j=0; j<2; ++j)                       // столбец
         cout << "(" << i << "," << j << ")->" << gslice_index(s,i,j) << "; "; // соответствие
}
```

Вот вывод этой программы:

```
(0,0)->0; (0,1)->1; (1,0)->3; (1,1)->4; (2,0)->6; (2,1)->7
```

Таким образом, **gslice** с двумя (длина,шаг)-парами описывает подмассив двумерного массива, **gslice** с тремя (длина,шаг)-парами описывает подмассив трехмерного массива, и т.д. Используя **gslice** как индекс для **valarray**, получаем **gslice_array**, состоящий из элементов, описанных посредством **gslice**. Например:

```
void f(valarray<float>& v)
{
    gslice m(0,lengths,strides);
    v[m] = 0;       // обнуляем 0 to v[0],v[1],v[3],v[4],v[6],v[7]
}
```

gslice_array предлагает такой же набор членов, что и **slice_array** (§40.5.5). **gslice_array** является результатом использования **gslice** в качестве индекса для **valarray** (§40.5.2).

40.6. Обобщенные численные алгоритмы

В файле **<numeric>** стандартная библиотека предоставляет несколько обощенных численных алгоритмов в стиле нечисленных алгоритмов из **<algorithm>** (Глава 32). Эти алгоритмы представляют обобщенные версии обычных операций над последовательностями численных значений:

Численные алгоритмы (§iso.26.7) Эти алгоритмы принимают итераторы ввода	
x=accumulate(b,e,i)	x есть сумма i и эл-ов [b:e)
x=accumulate(b,e,i,f)	accumulate используя f вместо +
x=inner_product(b,e,b2,i)	x есть внутреннее произведение [b:e) и [b2:b2+(e-b)), то есть, сумма i и (*p1)*(*p2) для каждого p1 в [b:e) и соответствующего p2 в [b2:b2+(e-b))
x=inner_product(b,e,b2,i,f,f2)	inner_product используя f и f2 вместо + и *
p=partial_sum(b,e,out)	Элемент i из [out:p) есть сумма эл-ов [b:b+i]
p=partial_sum(b,e,out,f)	partial_sum используя f вместо +
p=adjacent_difference(b,e,out)	Элемент i из [out:p) есть (*b+i)-*(b+i-1) для i>0; если e-b>0, то *out есть *b
p=adjacent_difference(b,e,out,f)	adjacent_difference используя f вместо -
iota(b,e,v)	Для каждого эл-а в [b:e) присвоить ++v; таким образом, послед-ть принимает вид v+1, v+2, ...

Эти алгоритмы обобщают обычные операции, вроде вычисления суммы, позволяя применять их к любым видам последовательностей, используя применяемую операцию в качестве аргумента. Для каждого алгоритма обобщенная версия сопровождается версией, реализующей наиболее типичную операцию для данного алгоритма.

40.6.1. accumulate()

Простая версия **accumulate()** складывает элементы последовательности, используя операцию +:

```
template<typename In, typename T>
T accumulate(In first, In last, T init)
{
    for (; first!=last; ++first) // для всех элементов в [first:last)
        init = init + *first;   // плюс
    return init;
}
```

Ее можно использовать следующим образом:

```
void f(vector<int>& price, list<float>& incr)
{
    int i = accumulate(price.begin(),price.end(),0);      // аккумулируем в int
    double d = 0;
    d = accumulate(incr.begin(),incr.end(),d);            // аккумулируем в double

    int prod = accumulate(price.begin,price.end(),1,[](int a, int b) { return a*b; });
    // ...
}
```

Тип переданного начального значения определяет тип возврата.

Можно передавать начальное значение и операцию «комбинирования элементов» в качестве аргументов, так что **accumulate()** это не только про сложение.

Для **accumulate()** извлечение значения из структуры данных является общей операцией. Например:

```
struct Record {
    // ...
    int unit_price;
    int number_of_units;
};
```

```
long price(long val, const Record& r)
{
    return val + r.unit_price * r.number_of_units;
}

void f(const vector<Record>& v)
{
    cout << "Total value: " << accumulate(v.begin(),v.end(),0,price) << '\n';
}
```

Операции, похожие на **accumulate()**, имеют в иных сообществах программистов имена **reduce**, **reduction** и **fold**.

40.6.2. inner_product()

Накопление суммы из последовательности — это очень частая операция, но и накопление суммы из пары последовательностей тоже нередко встречается на практике:

```
template<typename In, typename In2, typename T>
T inner_product(In first, In last, In2 first2, T init)
{
    while (first != last)
        init = init + *first++ * *first2++;
    return init;
}

template<typename In, typename In2, typename T, typename BinOp, typename BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
    while (first != last)
        init = op(init,op2(*first++,*first2++));
    return init;
}
```

Как всегда, только начало второй входной последовательности передается в качестве аргумента. Предполагается, что вторая входная последовательность по крайней мере имеет не меньшую длину, чем первая:

```
valarray<double> operator*(const Matrix& m, valarray<double>& v)
{
    valarray<double> res(m.dim2());

    for (size_t i = 0; i<m.dim2(); i++) {
        auto& ri = m.row(i);
        res[i] = inner_product(ri,ri.end(),&v[0],double(0));
    }
    return res;
}

valarray<double> operator*(valarray<double>& v, const Matrix& m)
{
    valarray<double> res(m.dim1());

    for (size_t i = 0; i<m.dim1(); i++) {
        auto& ci = m.column(i);
        res[i] = inner_product(ci,ci.end(),&v[0],double(0));
    }
    return res;
}
```

40.6.3. partial_sum() и adjacent_difference()

Алгоритмы **partial_sum()** и **adjacent_difference()** противоположны друг другу и имеют дело с понятием инкрементных изменений.

Имея на входе последовательности **a**, **b**, **c**, **d** и т. д., алгоритм **adjacent_sum()** производит **a**, **b-a**, **c-b**, **d-c** и т.д.

Рассмотрим вектор температурных измерений. Можно трансформировать его в вектор температурных изменений следующим образом:

```
vector<double> temps;

void f()
{
    adjacent_difference(temps.begin(),temps.end(),temps.begin());
}
```

Например, показания **17**, **19**, **20**, **20**, **17** превращаются в **17**, **2**, **1**, **0**, **-3**.

И наоборот, **partial_sum()** позволяет вычислить конечный результат набора инкрементных разностей (изменений):

```
template<typename In, typename Out, typename BinOp>
Out partial_sum(In first, In last, Out res, BinOp op)
{
    if (first==last) return res;
    *res = *first;
    T val = *first;
    while (++first != last) {
        val = op(val,*first);
        *++res = val;
    }
    return ++res;
}

template<typename In, typename Out>
Out partial_sum(In first, In last, Out res)
{
    return partial_sum(first,last,res,plus);    // используем std::plus (§33.4)
}
```

Имея на входе последовательности **a**, **b**, **c**, **d** и т. д., **partial_sum()** производит **a**, **a+b**, **a+b+c**, **a+b+c+d** и т. д. Пример:

```
void f()
{
    partial_sum(temps.begin(),temps.end(),temps.begin());
}
```

Отметим способ, каким **partial_sum()** инкрементирует **res** перед присваиванием нового значения. Это позволяет **res** быть той же последовательностью, что и входная последовательность; **adjacent_difference()** ведет себя аналогично. Так,

```
partial_sum(v.begin(),v.end(),v.begin());
```

превращает последовательность **a**, **b**, **c**, **d** в **a**, **a+b**, **a+b+c**, **a+b+c+d**, и

```
adjacent_difference(v.begin(),v.end(),v.begin());
```

воспроизводит оригинальные значения. Так, **partial_sum()** превращает **17**, **2**, **1**, **0**, **-3** назад в **17**, **19**, **20**, **20**, **17**.

Для людей, которым кажется скучным делом анализировать приращения температурных показателей, я скажу, что анализ изменений на фондовой бирже и анализ приращений уровня моря используют те же самые две операции. Эти операции полезны для анализа любых последовательностей изменений.

40.6.4. iota()

Вызов **iota(b,e,n)** присваивает **n+i** i-тому элементу последовательности **[b:e)**. Пример:

```
vector<int> v(5);
iota(v.begin(),v.end(),50);
vector<int> v2 {50,51,52,53,54}

if (v!=v2)
    error("complain to your library vendor");
```

Странное имя **iota** происходит из языка APL, где оно считалось созвучным соответствующей греческой букве.

Не перепутайте **iota()** с функцией **itoa()** (§12.2.4).

40.7. Случайные числа

Случайные числа важны для многих приложений: эмуляции процессов, игр, криптографии, тестирования и т. д. В файле **<random>** стандартная библиотека определяет средства для генерации (псевдо-)случайных чисел. Эти случайные числа формируются как последовательности значений, производимые математическими алгоритмами, а не физическими процессами, когда такие числа были бы истинно случайными. Если реализация располагает такими истинно «случайными приборами» (генерирующими истинно случайные числа), то они будут представляться посредством **random_device** (§40.7.1).

Имеются четыре типа сущностей:

- *Однородный генератор случайных чисел* — это функциональный объект, который возвращает беззнаковые целые числа, так что каждое значение лежит в диапазоне возможных чисел и в идеале выбирается (возвращается) равновероятно.
- *«Движок»* (*engine*) — это однородный генератор случайных чисел, который может создаваться в умолчательном состоянии **E{}** или в состоянии, определяемым посредством **seed E{s}**.
- *Адаптор «движка» случайных чисел* — это «движок» случайных чисел, который принимает значения от другого «движка» случайных чисел, и применяет к ним алгоритм с целью получения последовательности с другими свойствами случайности.
- *Распределение случайных чисел* — это функциональный объект, который возвращает числа, распределенные в соответствии с ассоциированной функцией плотности вероятности $p(z)$ или в соответствии с дискретной функцией вероятности $P(zi)$.

Подробнее этот вопрос рассмотрен в §iso.26.5.1.

Генератор случайных чисел (исходный «движок») производит однородно распределенную последовательность значений, а распределение «сгибает» ее в требуемую форму. Если мы привяжем **normal_distribution** (нормальное распределение) к **default_random_engine** (умолчательный «движок» случайных чисел), то получим генератор, который производит нормальное распределение:

```
auto gen = bind(normal_distribution<double>{15,4.0},default_random_engine{});

for (int i=0; i<500; ++i) cout << gen();
```

Стандартная функция **bind()** делает функциональный объект, который активирует свой первый аргумент, получая свой второй аргумент (§33.5.1).

Применяя ASCII-графику (§5.6.3), я получаю:

```
3     **
4     *
5     *****
6     ****
7     ****
8     ******
9     ***********
10    ***************************
11    ************************
12    ************************************
13    ************************************************************
14    ***************************************************
15    *********************************************************
16    *********************************
17    ****************************************************
18    ************************************
19    **********************************
20    **************
21    ***********
22    ***********
23    *******
24    *****
25    ****
26    *
27    *
```

По большей части, большинству программистов требуется простое однородное распределение целых чисел или чисел с плавающей запятой в заданном диапазоне. Например:

```
void test()
{
      Rand_int ri {10,20};        // однородное распределение int в [10:20)
      Rand_double rd {0,0.5}; // однородное распределение double в [0:0.5)

      for (int i=0; i<100; ++i)
            cout << ri() << ' ';
      for (int i=0; i<100; ++i)
            cout << rd() << ' ';
}
```

К сожалению, **Rand_int** и **Rand_double** не являются стандартными классами, но их легко определить самим:

```
class Rand_int {
      Rand_int(int lo, int hi) : p{lo,hi} { }      // сохраняем параметры
      int operator()() const { return r(); }
private:
      uniform_int_distribution<>::param_type p;
      auto r = bind(uniform_int_distribution<>{p},default_random_engine{});
};
```

Я сохраняю параметры, используя стандартный синоним распределения **param_type** (§40.7.3), так что я могу применить **auto** во избежание необходимости именовать результат **bind()**.

Просто для разнообразия я использую отличную технику для **Rand_double**:

```
class Rand_double {
public:
    Rand_double(double low, double high)
        :r(bind(uniform_real_distribution<>(low,high),default_random_engine())) { }
    double operator()() { return r(); }
private:
    function<double()> r;
};
```

Одним из важных применений случайных чисел является их использования в алгоритмах сэмплирования. В таких алгоритмах нам надо сделать выборку некоторого размера из гораздо большей популяции. Вот алгоритм R (простейший алгоритм) из знаменитой работы [Vitter,1985]:

```
template<typename Iter, typename Size, typename Out, typename Gen>
Out random_sample(Iter first, Iter last, Out result, Size n, Gen&& gen)
{
    using Dist = uniform_int_distribution<Size>;
    using Param = typename Dist::param_type;

    // Наполняем хранилище и увеличиваем first:
    copy(first,n,result);
    advance(first,n);

    // Семплируем оставшиеся знач-я в [first+n:last), выбирая случайное
    // число r в диапазоне [0:k], и, если r<n, заменяя его.
    // k увеличивается с каждой итерацией, делая вероятность меньше.
    // Для итераторов с произвольным доступом, k = i-first
    // (при условии, что мы увеличиваем i, а не first).

    Dist dist;
    for (Size k = n; first!=last; ++first,++k) {
        Size r = dist(gen,Param{0,k});
        if(r < n)
            *(result + r) = *first;
    }
    return result;
}
```

40.7.1. «Движки»

Генератор равномерно распределенных случайных чисел — это функциональный объект, который производит почти однородно распределенную последовательность значений с типом **result_type**:

Генераторы равномерно распределенных случайных чисел: G<T> (§iso.26.5.1.3)	
G::result_type	Тип элемента в последовательности
x=g()	Операция **()**: **x** – следующий эл-т в последовательности
x=G::min()	**x** – наименьший эл-т, который может вернуть **g()**
x=G::max()	**x** – наибольший эл-т, который может вернуть **g()**

«Движок» случайных чисел есть генератор однородно распределенных случайных чисел с дополнительными свойствами, которые способствуют его широкому использованию:

«Движки» случайных чисел: E<T> (§iso.26.5.1.4)	
E e {};	Конструктор по умолчанию
E e {e2};	Копирующий конструктор
E e {s};	**e** будет в состоянии, определяемом зерном **s**
E e {g};	**e** будет в состоянии, определяемом вызовом **generate()** для последовательности зерен **g**
e.seed()	**e** будет в состоянии по умолчанию
e.seed(s)	**e** будет в состоянии, определяемом зерном **s**
e.seed(g)	**e** будет в состоянии, определяемом вызовом **generate()** для последовательности зерен **g**
e.discard(n)	Пропустить следующие **n** эл-тов последовательности
e==e2	Произведут ли **e** и **e2** совершенно одинаковые последовательности?
e!=e2	**!(e==e2)**
os<<e	Вывести представление **e** в **os**
is>>e	Прочитать представление «движка» ранее записанного с помощью **<<** из **is** в **e**

Здесь seed (зерно) — это значение из диапазона $[0:2^{32})$, которое используется для инициализации конкретного «движка». Последовательность зерен, **g**, есть объект, который предоставляет функцию **g.generate(b,e)**, которая при вызове заполняет интервал **[b:e)** вновь сгенерированными зернами (§iso.26.5.1.2).

Стандартные «движки» случайных чисел (§iso.26.5.3)	
default_random_engine	Псевдоним «движка» с широкой применяемостью и низкой ценой
linear_congruential_engine<UI,a,c,m>	$x_{i+1} = (ax_i + c) \bmod m$
mersenne_twister_engine<UI,w,n,m,r,a,u,d,s,t,c,l,f>	§iso.26.5.3.2
subtract_with_carry_engine<UI,w,s,r>	$x_{i+1} = (ax_i) \bmod b$, где $b = m^r - m^s + 1$, и $a = b - (b-1)/m$

Параметр **UI** для стандартного «движка» случайных чисел должен иметь тип беззнакового целого. Для **linear_congruential_engine<UI,a,c,m>**, если модуль **m** равен **0**, используется значение **numeric_limits<result_type>::max()+1**. Например, следующий код выводит индекс первого повторения числа:

```
map<int,int> m;
linear_congruential_engine<unsigned int,17,5,0> linc_eng;
for (int i=0; i<1000000; ++i)
    if (1<++m[linc_eng()]) cout << i << '\n';
```

Мне повезло; параметры были не такими уж плохими и я не получил дублирующих значений. Попробуйте **<unsigned int,16,5,0>** и увидите разницу. При прочих равных применяйте **default_random_engine** во всех случаях, кроме тех, когда у вас есть серьезные причины поступать по-другому.

Адаптор «движка» случайных чисел принимает «движок» в качестве аргумента и производит новый «движок» случайных чисел с иными параметрами случайности.

Стандартные адаптеры «движков» случайных чисел (§iso.26.5.4)	
discard_block_engine<E,p,r>	**E** есть «движок»; §iso.26.5.4.2
independent_bits_engine<E,w,UI>	Генерировать **w** бит в типе **UI**; §iso.26.5.4.3
shuffle_order_engine<E,k>	§iso.26.5.4.4

Например:

```
independent_bits_engine<default_random_engine,4,unsigned int> ibe;
for (int i=0; i<100; ++i)
    cout << '0'+ibe() << ' ';
```

Здесь получается 100 чисел в диапазоне [48:63] (['0':'0'+2^4–1)).

Для полезных движков определены несколько синонимов:

```
using minstd_rand0 = linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>;
using minstd_rand = linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>;
using mt19937 = mersenne_twister_engine<uint_fast32_t, 32,624,397,
        31,0x9908b0df,
        11,0xffffffff,
        7,0x9d2c5680,
        15,0xefc60000,
        18,1812433253>

using mt19937_64 = mersenne_twister_engine<uint_fast64_t, 64,312,156,
        31,0xb5026f5aa96619e9,
        29, 0x5555555555555555,
        17, 0x71d67fffeda60000,
        37, 0xfff7eee000000000,
        43, 6364136223846793005>;
using ranlux24_base = subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>;
using ranlux48_base = subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>;
using ranlux24 = discard_block_engine<ranlux24_base, 223, 23>;
using ranlux48 = discard_block_engine<ranlux48_base, 389, 11>;
using knuth_b = shuffle_order_engine<minstd_rand0,256>;
```

40.7.2. Аппаратные генераторы случайных чисел

Если реализации способны предоставить генераторы истинно случайных чисел, то этот источник представляется как однородный генератор случайных чисел **random_device**:

random_device (§iso.26.5.6)	
random_device rd {s};	The **string s** идентифицирует источник случайных чисел; зависит от реализации; explicit
d=rd.entropy()	**d** есть **double**; **d==0.0** для генератора псевдослучайных чисел

Трактуйте **s** как имя источника случайных чисел, вроде счетчика Гейгера, или просто файла, содержащего последовательность истинно случайных чисел. Энтропия (**entropy()**) определяется следующим образом:

$$S(P_0,\ldots,P_{n-1}) = -\sum_{i=0}^{i=n-1} P_i \log P_i$$

для устройства (прибора) с n состояниями, чьи ожидаемые вероятности есть P_0, ..., P_{n-1}. Энтропия — это оценка случайности, степень непредсказуемости генерируемых чисел. В отличие от термодинамики, высокая энтропия хороша для случайных чисел, поскольку при этом трудно предугадать последующее число. Формула отражает результат повторных выкидываний совершенного кубика с n-гранями.

random_device предназначен для использования в криптографических приложениях, но его применение в указанных приложениях без тщательной проверки будет верхом неосмотрительности.

40.7.3. Распределения

Распределение случайных чисел — это функциональный объект, который будучи вызван с аргументом-«движком», производит последовательность значений типа **result_type**:

Распределение случайных чисел D (§iso.26.5.1.6)	
D::result_type	Тип элемента **D**
D::param_type	Тип набора аргументов для создания **D**
D d {};	Конструктор по умолчанию
D d {p};	Создать из **param_type p**
d.reset()	Вернуть в состояние по умолчанию
p=d.param()	**p** – параметры **d** с типом **param_type**
d.param(p)	Вернуть в состояние, определенное **param_type p**
x=d(g)	**x** – результат работы **d** с генератором **g**
x=d(g,p)	**x** – результат работы **d** с генератором **g** и параметрами **p**
x=d.min()	**x** – наименьшее значение, к-рое может вернуть **d**
x=d.max()	**x** – наибольшее значение, к-рое может вернуть **d**
d==d2	**d** и **d2** производят одинаковые последовательности эл-тов?
d!=d2	**!(d==d2)**
os<<d	Пишет состояние **d** в **os**, чтобы его можно было прочитать с **>>**
is>>d	Прочитать ранее записанное с **<<** состояние из **is** в **d**

В последующих таблицах шаблонный аргумент **R** означает требование реального числа в математической формуле и **double** в качестве умолчания. А **I** означает требование целого с **int** по умолчанию.

Однородные распределения (§iso.26.5.8.2)			
Распределение	Предварительное условие	По умолчанию	Результат
uniform_int_distribution<I>(a,b)	$a \le b$ $P(i\|a,b) = 1/(b-a+1)$	(0,max)	$[a{:}b]$
uniform_real_distribution<R>(a,b)	$a \le b$ $p(x\|a,b) = 1/(b-a)$	(0.0,1.0)	$[a{:}b)$

Поле *precondition* (предусловие) специфицирует требования к аргументам-распределениям. Например:

```
uniform_int_distribution<int> uid1 {1,100};    // OK
uniform_int_distribution<int> uid2 {100,1};    // ошибка: a>b
```

Поле *default* специфицирует умолчательные аргументы. Например:

```
uniform_real_distribution<double> urd1 {};        // использ. a==0.0 and b==1.0
uniform_real_distribution<double> urd2 {10,20};   // использ. a==10.0 and b==20.0
uniform_real_distribution<> urd3 {};              // использ. double и a==0.0 and b==1.0
```

Поле *result* специфицирует диапазон результатов. Например:

```
uniform_int_distribution<> uid3 {0,5};
default_random_engine e;
for (int i=0; i<20; ++i)
    cout << uid3(e) << ' ';
```

Диапазон для **uniform_int_distribution** закрытый, и мы видим шесть возможных значений:

```
2 0 2 5 4 1 5 5 0 1 1 5 0 0 5 0 3 4 1 4
```

Для **uniform_real_distribution**, как и для всех других распределений с типом результата в виде числа с плавающей запятой, диапазон полуоткрытый.

Распределения Бернулли отражают последовательности подбрасываний монет с варьирующейся степенью нагрузки:

Распределения Бернулли (§iso.26.5.8.3)

Распределение	Предварительное условие	По умолчанию	Результат
bernoulli_distribution(p)	$0<=p<1$ $P(b\|p) = \begin{cases} p & \text{если } b = \text{true} \\ 1-p & \text{если } b = \text{false} \end{cases}$	(0.5)	{true,false}
binomial_distribution<I>(t,p)	$0 \le p \le 1$ и $0 \le t$ $P(i\|t,p) = \binom{t}{i} p^i (1-p)^{t-i}$	(1,0.5)	[0:∞)
geometric_distribution<I>(p)	$0 < p < 1$ $P(i\|p) = p(1-p)^i$	(0.5)	[0:∞)
negative_binomial_distribution<I>(k,p)	$0 < p < 1$ и $0 < k$ $P(i\|k,p) = \binom{k+i-1}{i} p^k (1-p)^i$	(1,0.5)	[0:∞)

Распределение Пуассона отражает вероятность заданного числа событий, происходящих в фиксированном интервале времени (или пространства):

Распределение Пуассона (§iso.26.5.8.4)

Распределение	Предварительное условие	По умолчанию	Результат
poisson_distribution<I>(m)	$0 < m$ $P(i\|\mu) = \frac{e^{-\mu}\mu^i}{i!}$	(1.0)	[0:∞)
exponential_distribution<R>(lambda)	$1 < lambda$ $p(x\|\lambda) = \lambda e^{-\lambda x}$	(1.0)	(0:∞)
gamma_distribution<R,R>(alpha,beta)	$0 < \alpha$ и $0 < \beta$ $p(x\|\alpha,\beta) = \frac{e^{-x/\beta}}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1}$	(1.0,1.0)	(0:∞)
weibull_distribution<R>(a,b)	$0 < al$ и $0 < b$ $p(x\|a,b) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} \exp\left(-\left(\frac{x}{b}\right)^a\right)$	(1.0,1.0)	[0:∞)
extreme_value_distribution<R>(a,b)	$0 < b$ $p(x\|a,b) = \frac{1}{b} \exp\left(\frac{a-x}{b} - \exp\left(\frac{a-x}{b}\right)\right)$	(0.0,1.0)	R

Нормальные распределения отображают вещественные значения в вещественные значения. Простейшее из них имеет форму колокола, что означает симметричное распределение значений около максимума (среднего значения) с расстоянием элементов от максимума, контролируемого стандартным отклонением:

Нормальные Распределения (§iso.26.5.8.5)

Распределение	Предварительное условие	По умолчанию	Результат
normal_distribution<R>(m,s)	$0 < s$	(0.0,1.0)	R
	$p(x\|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$		
lognormal_distribution<R>(m,s)	$0 < s$	(0.0,1.0)	>0
	$p(x\|m,s) = \frac{1}{sx\sqrt{2\pi}} \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right)$		
chi_squared_distribution<R>(n)	$0 < n$	(1)	>0
	$p(x\|n) = \frac{x^{(n/2)-1} e^{-x/2}}{\Gamma(n/2)2^{n/2}}$		
cauchy_distribution<R>(a,b)	$0 < b$	(0.0,1.0)	R
	$p(x\|a,b) = \left(\pi b \left(1 + \left(\frac{x-a}{b}\right)^2\right)\right)^{-1}$		
fisher_f_distribution<R>(m,n)	$0 < m$ и $0 < n$	(1,1)	>=0
	$p(x\|m,n) = \frac{\Gamma((m+n)/2)}{\Gamma(m/2)\Gamma(n/2)} \left(\frac{m}{n}\right)^{m/2} x^{(m/2)-1} \left(1 + m\frac{x}{n}\right)^{-(m+n)/2}$		
student_t_distribution<R>(n)	$0 < n$	(1)	R
	$p(x\|n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2}{\Gamma} (n/2)\left(1 + \frac{x^2}{n}\right)^{(n+1)/2}$		

Чтобы прочувствовать эти распределения, взгляните на их графические представления для разных параметров. Их легко сгенерировать самим или найти в интернете.

Сэмплированные распределения отображают целые в заданный диапазон согласно функции плотности вероятности **P**:

Сэмплированные распределения (§iso.26.5.8.6)

Распределение	Предварительное условие	По умолчанию	Результат
discrete_distribution<I>{b,e}	$0<=b[i]$	нет	**[0:e-b)**
	$P(i\|p_0,\cdots p_{n-1}) = p_i$		
	Послед-ть **[b:e)** предоставляет веса w_i так, что		
	$p_i = w_i/S$ и $0 < S = w_0 + \cdots + w_{n?1}$		
	где n = **e-b**		

Продолжение

Сэмплированные Распределения (§iso.26.5.8.6)			
Распределение	Предварительное условие	По умолчанию	Результат
discrete_distribution<I>(Ist)		**discrete_distribution<I>(Ist.begin(),Ist.end())**	
discrete_distribution<I>(n,min,max,f)		**discrete_distribution<I>(b,e)** где I-тый эл-т из [**b:e**) получается с помощью **f(min+I*(max?min)/n +(max-min)/(2*n))**	
piecewise_constant_distribution<R>{b,e ,b2,e2}	**b[I]<b[I+1]**	нет	[***b:*(e-1)**) $P(x\|x_0,\cdots x_n,\rho_0\cdots\rho_n)$
piecewise_linear_distribution<R>{b,e ,b2,e2}	**b[I]<b[I+1]**	нет	[***b:*(e-1)**) $P(x\|x_0,\cdots x_n,\rho_0\cdots\rho_n)$ $= p_i\,\frac{b_{i+1}-x}{b_{i+1}-b_i} + \rho_i\,\frac{x-b_i}{b_{i+1}-b_i}$ $b_i < b_{i+1}$ для всех b_i в [**b:e**) $\rho_i = w_i/S$ где $S = \frac{1}{2}\sum_{i=0}^{n-1}(w_i + w_{i+1})(b_{i+1} - b_i)$ [**b:e**] – границы интервалов [**b2:e2**] – веса

40.7.4. Случайные числа в языке C

В файлах **<cstdlib>** и **<stdlib.h>** стандартная библиотека предоставляет простую основу для генерации случайных чисел:

```
#define RAND_MAX implementation_defined /* большое положительное число */

int rand();                    // псевдослучайное число между 0 и RAND_MAX
void srand(unsigned int i);    // инициализирует генератор случайных чисел числом
```

Реализовать хороший генератор случайных чисел не легко, так что не во всех системах есть хороший **rand()**. В частности, младшие биты случайных чисел находятся под подозрением, и **rand()%n** не является хорошим (и переносимым) способом генерирования случайных чисел между **0** и **n-1**. Часто приемлемого результата можно добиться посредством **int((double(rand())/RAND_MAX)*n)**. Для серьезных приложений генераторы на базе **uniform_int_distribution** (§40.7.3) дадут более надежные результаты.

Вызов **srand()** начинает новую последовательность случайных чисел из зерна, **s**, полученного в виде аргумента. При отладке важно, чтобы случайные числа для заданного зерна повторялись при повторных запусках. Но чтобы игры были непредсказуемыми, нужно выбирать зерна из окружения программы. Часто некоторые биты из часов реального времени хорошо подходят в роли зерна.

40.8. Советы

[1] Математические проблемы — это тонкая материя. Если вы сами не уверены в своих силах, обратитесь к экспертам в этой области, а также экспериментируйте; §29.1.

[2] Используйте конкретные числовые типы, пригодные в соответствующих областях применения; §40.2.

[3] Используйте **numeric_limits** для того, чтобы быть уверенным в правильном выборе числовых типов; §40.2.
[4] Специализируйте **numeric_limits** для пользовательских числовых типов; §40.2.
[5] Используйте **numeric_limits** вместо соответствующих макросов; §40.2.1.
[6] Для комплексных расчетов используйте **std::complex**; §40.4.
[7] Используйте {}-инициализацию, чтобы избежать сужения значений; §40.4.
[8] Используйте **valarray** для математических вычислений в тех случаях, когда важна производительность, а не гибкость в плане типов элементов и их операций; §40.5.
[9] Выражайте операции над частями массивов в терминах срезов, а не циклов; §40.5.5.
[10] Срезы являются общей полезной абстракцией для доступа к компактным данным; §40.5.4, §40.5.6.
[11] Перед тем, как написать цикл по элементам последовательности для вычисления значения, рассмотрите возможность использования **accumulate()**, **inner_product()**, **partial_sum()** и **adjacent_difference()**; §40.6.
[12] Привязывайте «движок» случайных чисел к распределению, чтобы окончательно сформировать генератор случайных чисел; §40.7.
[13] Удостоверьтесь, что ваши случайные числа в достаточной мере случайны; §40.7.1.
[14] Если вам требуются истинно случайные числа, используйте **random_device**; §40.7.2.
[15] Используйте классы случайных чисел с заданными распределениями вместо **rand()**; §40.7.4.

41

Параллелизм

Старайтесь все делать как можно проще,
проще как только возможно,
но не проще этого.
— Альберт Эйнштейн

- Введение
- Модель памяти

  Память; Реорганизация машинных команд; Упорядочение памяти; Гонка за данными
- Атомарность

  Типы **atomic**; Флаги и «ограждения»
- **volatile**
- Советы

41.1. Введение

Параллелизм — параллельное исполнение нескольких задач одновременно, широко используется для улучшения вычислительной пропускной способности (путем использования нескольких процессоров для единственной ветки вычислений) или для уменьшения отклика на действия пользователя (путем предоставления «красной дорожки» одним частям кода в ущерб другим).

Поддержка параллелизма стандартом C++ рассмотрена в кратком учебном стиле в §5.3. В настоящем разделе и в следующем мы рассмотрим этот вопрос детальнее.

Назовем вычислительную активность, реализуемую параллельно другим активностям, *задачей*. *Потоком* назовем представление системного уровня для поддержки выполнения задачи. Класс **thread** (§42.2) из стандартной библиотеки выполняет задачу. Поток может разделять адресное пространство с другими потоками. Таким образом, все потоки могут обращаться к одним и тем же объектам в памяти. Грандиозная задача программиста в многопоточном окружения состоит в управлении координацией такого доступа.

Поддержка параллелизма стандартной библиотекой состоит в следующем:

- *Модель памяти*: система гарантий для одновременного доступа к памяти (§41.2), которая обеспечивает простой стандартный доступ к памяти, достаточно очевидный для каждого программиста.
- *Программирование без блокировок*: четко структурированные низкоуровневые механизмы для избежания гонки за данными (§41.3).

- *Библиотека потоков*: набор компонентов, поддерживающих традиционную модель программирования параллелизма системного уровня на базе блокировки потоков, таких как **thread**, **condition_variable** и **mutex** (§42.2).
- Поддержка библиотекой понятия *задачи*: ряд средств параллельного программирования пользовательского уровня: **future**, **promise**, **packaged_task** и **sync()** (§42.4).

Эти вопросы мы упорядочиваем в направлении от самых фундаментальных и низкоуровневых к высокоуровневым средствам пользовательского уровня. Модель памяти достаточно традиционна для многих стилей программирования. Ради высокой продуктивности и уменьшения вероятности ошибок лучше работать на высоком уровне. К примеру, следует предпочесть **future**, а не **mutex** для обмена информацией, и **mutex** вместо **atomic** во всех случаях, кроме простейших счетчиков и тому подобных вещей. Всюду, где только возможно, оставьте сложные проблемы для реализаторов стандартной библиотеки.

В контексте стандартной библиотеки языка C++ *блокировка* (lock) это **mutex** (взаимно исключающая переменная) и все абстракции, строящиеся поверх **mutex** для гарантирования взаимно исключающего доступа к ресурсу или для синхронизации продвижения нескольких параллельных задач.

Вопросы, связанные с *процессами*, то есть с потоками исполнения, имеющими изолированные адресные пространства и взаимодействующими друг с другом посредством специальных механизмов межпроцессной коммуникации [Tanenbaum,2007], в нашей книге не рассматриваются. Мы сосредотачиваемся на совместном владении данными при параллельном исполнении потоков, и я считаю, что здесь не обязательно контролировать этот доступ непосредственно прикладным программистам.

До тех пор, пока вы не передаете иным потокам указатели на данные, эти данные свободны от проблем. Это, кстати, касается и глобальных данных, которые лучше избегать и по указанной причине.

Настоящая глава не является исчерпывающим руководством по параллельному программированию и даже не рассматривает в полной мере все средства стандартной библиотеки, связанные с параллельным программированием. Она посвящена следующему:

- Перечню проблем, с которыми сталкивается программист при параллельном программировании на системном уровне.
- Достаточно подробному рассмотрению средств стандартной библиотеки согласно стандарту параллельного программирования.
- Введению в вопрос использования базовых средств стандартной библиотеки на уровне потоков и блокировок.

Настоящая глава не предназначена для

- Детального рассмотрения программирования без блокировок.
- Глубокого обучения параллельному программированию и сопутствующих методик и технологий.

Параллельное программирование было популярной темой исследований и применялось на практике в течение последних 40 лет, так что не удивительно, что этой теме посвящена масса специализированной литературы (например, по вопросам, связанным с C++, смотри [Wilson,1996]). В частности, много источников посвящены представлению POSIX-потоков; их можно использовать в качестве примеров, которые легко усовершенствовать с помощью средств стандартной библиотеки.

На контрасте со средствами в стиле POSIX и многими старыми библиотеками C++ для поддержки потоков, средства стандартной библиотеки безопасны по типам. Кроме того, мы можем формировать задачи как функциональные объекты (например, как лямбды) и передавать их потокам без применения явных приведений. И нет никаких причин

по изобретению новых соглашений о передаче сообщений об ошибках из одного потока в другой; **future** (§5.3.5.1, §42.4.4) легко справляется с этим. Поскольку параллельное программирование не является простой темой в самом общем случае и с учетом того, что код для работы разных потоков часто пишется в разное время разными программистами, я считаю проблему безопасности по типам и стандарт обработки ошибок более важными, чем это имеет место при написании однопоточных программ.

41.2. Модель памяти

Многие механизмы параллельности языка C++ предоставляются как компоненты стандартной библиотеки. Эти компоненты полагаются на набор гарантий языка C++, известных как *модель памяти*. Модель памяти является результатом дискуссий между разработчиками машинных архитектур и разработчиками компиляторов на предмет того, как лучше представить аппаратуру компьютера. Модель памяти специфицирована в ISO C++ стандарте и она должна позволить большинству программистов не думать о деталях современной компьютерной архитектуры.

Чтобы вникнуть в данный вопрос, отметим, что операции над объектами в памяти никогда напрямую не выполняются прямо в памяти. Вместо этого, объекты (или их части) могут загружаться в регистры процессора, там модифицироваться и после этого переписываться назад в память. Кроме того, есть еще и кэш-память, стоящая между памятью и регистрами. Рассмотрим простейший пример инкрементирования целой переменной **x**:

```
// прибавим 1 к x:
    load x into cache element Cx
    load Cx into register Rx
    Rx=Rx+1;
    store Rx back into Cx
    store Cx back into x
```

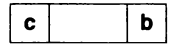
К памяти могут иметь одновременный (параллельный) доступ несколько потоков, и кэш-память может разделяться между потоками, выполняющимися на одинаковых или разных процессорных единицах (называемых процессорами, ядрами или гиперпотоками выполнения). Здесь открывается масса возможностей для простой операции (вроде «добавить единицу к **x**») стать некорректной. Для тех из читателей, которые жаждут разобраться с этими вопросами подробнее, я рекомендую Приложение C из работы [McKenney,2012].

41.2.1. Память

Рассмотрим ситуацию с двумя глобальными переменными **b** и **c**:

```
// поток 1:                    // поток 2:
    char c = 0;                    char b = 0;
    void f()                       void g()
    {                              {
        c = 1;                         b = 1;
        int x = c;                     int y = b;
    }                              }
```

Теперь **x==1** и **y==1**, как это ожидают многие. Почему это стоит обсуждать? А что будет, если компоновщик (линкер) расположит **c** и **b** в одном и том же слове памяти и компьютер не может загрузить и хранить что-либо в объеме памяти, меньшим слова:



Без четкого определения разумной модели памяти поток 1 мог бы прочесть слово, содержащее **b** и **c**, изменить **c** и записать слово назад в память. В то же время, поток 2 мог проделать тоже самое над **b**. И теперь вопрос: какой поток прочтет слово первым и какой поток будет писать результат назад в память последним? От этого будет зависеть результат. Можно в результате получить **10**, **01** или **11** (но не **00**). Модель памяти спасает нас от этого хаоса: мы получим **11**. Причина, по которой **00** невозможно, состоит в том, что инициализации **b** и **c** выполняются раньше, чем стартуют потоки.

Модель памяти C++ гарантирует, что два потока исполнения могут модифицировать и выполнять доступ к разным локациям (местоположениям) в памяти без паразитного влияния друг на друга. Это то, что мы наивно ожидаем. Это задача компилятора защищать нас от тонких моментов в работе современных процессоров. Фактически мы программируем на «машине», представленной комбинацией аппаратуры и очень низкоуровневого кода.

Битовые поля (§8.2.7) предоставляют доступ к частям машинных слов. Если **b** и **c** являются двумя полями одного и того же слова, на большинстве аппаратуры нет возможности избежать проблем (гонка за данными) с кодом вроде показанного выше примера «**b**-и-**c**» без использования той или иной формы блокировок. В итоге, язык определяет «локацию в памяти» как единицу памяти, для которой интуитивное поведение гарантируется (за исключением индивидуальных битовых полей).

Локация памяти есть либо объект арифметического типа (§6.2.1), либо указатель или максимальная последовательность смежных битовых полей, все ненулевого размера. Например:

```
struct S {
    char a;                 // локация #1
    int b:5;                // локация #2
    unsigned c:11;
    unsigned :0;            // внимание: :0 – "особый случай"  (§8.2.7)
    unsigned d:8;           // локация #3
    struct { int ee:8; } e; // локация #4
};
```

Здесь **S** имеет в точности четыре отдельные локации (метоположения) в памяти. Не пытайтесь обновлять битовые поля **b** и **c** из разных потоков без явной синхронизации.

Из представленных выше объяснений вы можете заключить, что если **x** и **y** одного типа, то **x=y** гарантирует результат, что **x** будет копией **y**. Это так, но только если нет гонки за данными (§41.2.4) и **x** и **y** представляют локации в памяти. В противном случае вы должны принять меры, например путем явной синхронизации мест памяти с разделяемым доступом (§41.3, §42.3.1).

41.2.2. Реорганизация машинных команд

Ради увеличения эффективности компиляторы, оптимизаторы и даже аппаратура реорганизуют машинные команды. Рассмотрим следующее:

```
// поток 1:
    int x;
    bool x_init;

    void init()
    {
        x = initialize();       // в initialize() не используется x_init
        x_init = true;
        // ...
    }
```

В этом куске кода нет явных указаний на то, что присваивание **x** выполняется до присваивания **x_init**. Оптимизатор кода (или планировщик выполнения машинного кода) может решить повысить скорость программы, выполняя **x_init=true** первым.

Мы, вероятно, планировали использовать **x_init** для ответа на вопрос, инициализирован ли **x** посредством **initialize()**, или нет. Но явно мы этого не сказали, так что ни компилятор, ни оптимизатор, ни аппаратура об этом ничего не знают.

Добавим в программу еще один поток:

```
// поток 2:
    extern int x;
    extern bool x_init;

    void f2()
    {
        int y;
        while (!x_init)        // дождаться завершения инициализации, если нужно
            this_thread::sleep_for(milliseconds{10});
        y = x;
        // ...
    }
```

Теперь у нас проблема: поток 2 может не ждать и присвоить **y** неинициализированный **x**.

Даже если поток 1 не будет устанавливать значения **x_init** и **x** в неправильном порядке, то все равно есть проблема. В потоке 2 нет присваиваний для **x_init**, так что оптимизатор может решить вынести оценку **!x_init** за цикл и цикл 2 никогда не «заснет» или «заснет» навсегда.

41.2.3. Упорядочение памяти

Время, требуемое для перевода значения из памяти в кэш-память, а затем из кэша в регистр, может быть по процессорным меркам довольно большим. В лучшем случае, 500 машинных команд (инструкций) могут выполниться за это время, и еще 500 инструкций до того, как значение вернется на свое место в памяти. Конечно, число 500 есть просто оценка возможностей, но за последнее время это число стабильно растет. Если нет спешки в загрузке и хранении в памяти некоторого значения, то в программах, оптимизированных в отношении ввода/вывода, вышеуказанная оценка может достигать многих тысяч циклов выполнения машинных инструкций. Это одна из особенностей современной архитектуры, позволяющая добиться исключительной производительности, но это также открывает возможности для недоразумений, когда разные потоки справляются о значениях в разных местах памяти в разные моменты времени. Но не нужно также забывать, что мое упрощенное изложение не затрагивает многоуровневых кэшей; многие популярные архитектуры имеют трехуровневый кэш. Приведем графическую иллюстрацию для случая архитектуры с двухуровневым кэшем, где каждое ядро имеет собственный кэш второго уровня, пара ядер разделяет кэш первого уровня, и все ядра совместно разделяют память:



Упорядочение памяти — это термин, отражающий следующий вопрос: «что программист думает о том, как именно поток видит значение в памяти?». Простейшее упорядочение памяти часто называют *последовательно согласованным*. В такой модели памяти любой поток видит результаты каждой операции, выполняемой в одном и том же порядке. Порядок здесь таков, как если бы все инструкции выполнялись последовательно в единственном потоке. Поток по-прежнему может переупорядочивать операции, но в точках, где иные потоки могут справляться о значении переменной, набор операций, выполненных до этого, и значение рассматриваемой переменной должны быть однозначно определенными и одними и теми же для всех потоков. Операция, справляющаяся о значении, и навязывающая согласованный взгляд на локацию в памяти, называется *атомарной операцией* (см. §41.3). Простое чтение или запись не накладывают порядка.

Возможно много последовательно согласованных порядков для заданного набора потоков. Рассмотрим следующее:

```
// поток 1:                    // поток 2:
    char c = 0;                    char b = 0;
    extern char b;                 extern char c;
    void f1()                      void f2()
    {                              {
        c = 1;                         b = 1;
        int x = b;                     int y = c;
    }                              }
```

В предположении, что инициализация **c** и **b** выполняется статически (перед стартом любого потока), имеется три возможных порядков выполнения:

```
c = 1;      b = 1;      c = 1;
x = b;      y = c;      b = 1;
b = 1;      c = 1;      x = b;
y = c;      x = b;      y = c;
```

Результатами будут **01**, **10** и **11**, соответственно. Невозможен лишь результат **00**. Для получения предсказуемого результата нужна какая-либо форма синхронизации доступа к разделяемым переменным.

Последовательно согласованный порядок удобен программистам, но на некоторых машинных архитектурах накладывает существенные дополнительные расходы на синхронизацию, которую можно устранить посредством ослабления правил. Например, два потока, выполняющиеся на разных ядрах, могут начать чтение **x** и **y** до записи в **a** и **b**, или по крайней мере до завершения записи. Это может привести к результату **00**. Ослабленные модели памяти позволяют такое.

41.2.4. Гонка за данными

Из этих примеров любой разумный человек может сделать вывод о том, что нужно быть исключительно внимательным и осторожным при программировании потоков. Но как это сделать? Во-первых, нужно избегать *гонки за данными*. Два потока вступают в гонку за данными, если оба хотят получить доступ к местоположению в памяти (как это описано в §41.2.1) одновременно, и по крайней мере один нуждается в записи. Если два потока вступают в гонку за данными, то язык в этом случае ничего не гарантирует: поведение программы не определено. Последствия от гонки за данными могут быть (как показано в §41.2.2) катастрофическими. Оптимизаторы могут реорганизовать код на основе представлений о значениях: в итоге код по мере выполнения может затрагивать данные, не связанные логически с этой задачей.

Вот несколько способов избежать гонки за данными:

- Постарайтесь повнимательнее проанализировать код и вставить в некоторые места кода блокировки. Это возможно наиболее популярный метод, но и чреватый ошибками.
- Найдите программу, которая анализирует код на предмет выявления потенциальных гонок за данными. Такие программы редки, а те из них, которые более-менее гарантируют успех, пока что не вышли за пределы исследовательских проектов.
- Проектируйте ваш код таким образом, чтобы потоки взаимодействовали исключительно через интерфейсы «положил-и-взял», так что потокам не нужно одновременно манипулировать одним и тем же местоположением в памяти (§5.3.5.1, §42.4).
- Используйте более высокоуровневые библиотеки, которые делают разделение данных между потоками неявным или хорошо управляемым. В качестве примеров можно привести параллельные реализации алгоритмов в библиотеках, инструменты, базирующиеся на директивах (например, OpenMP) и транзакционную память.

Можно сказать, что оставшаяся часть настоящей главы представляет собой подход снизу-вверх в направлении последней возможности. В процессе рассмотрения мы коснемся инструментов, нужных для поддержки всех способов избежать гонки за данными.

Почему программистам приходится мучиться со всей этой сложностью (альтернатива состоит в написании простых последовательных программ, имеющих минимальные возможности существования гонки за данными)? Я дам два ответа на поставленный вопрос:

1. Так устроен мир. Сложности машинных архитектур объективны и языки системного программирования, к которым относится и C++, должны предоставить реальные средства для программистов, работающих в этом реальном мире. Программисты должны справляться с низкоуровневыми деталями, чтобы соответствовать требованиям заказчиков в плане высокой производительности.
2. Комитет по стандартизации C++ серьезно рассматривал этот вопрос. Мы были не против предоставить модель памяти, которая была бы улучшенной версией того, что предоставляют языки Java и C#. Это бы помогло и комитету, и программистам сэкономить кучу времени. Но такая идея была заветирована реализаторами операционных систем и виртуальных машин: от них прозвучало требование предоставить именно то, что в итоге стандарт и предоставил. Иначе замедление в производительности выражалось бы фактором 2 или более.

По счастью, большинству программистов не приходится работать на самом низком уровне близости к аппаратуре. Большинству программистов нет нужды знакомиться с моделью памяти, а к реорганизации машинных кодов они могут относиться как к забавной экзотике.

Пишите свободный от гонки за данными код и не заморачивайтесь с организацией памяти (§41.3); тогда модель памяти будет гарантировать вам то интуитивно ожидаемое поведение, какое вам кажется естественным. Это лучше, чем последовательная согласованность.

Лично я воспринимаю тематику, связанную с машинной архитектурой, восхитительной (см. [Hennesey,2011], [McKenney,2012]), но большинству разумных программистов нет нужды опускаться столь низко в сторону особенностей машинной аппаратуры. Оставьте это экспертам, а сами наслаждайтесь высокоуровневыми средствами, которые разработаны экспертами для вас.

41.3. Атомарность

Стиль программирования без блокировок — это набор приемов написания параллельных программ без использования явных блокировок. Вместо этого, программист полагается на примитивные операции (непосредственно поддерживаемые аппаратурой), помогающие избежать гонки за данными (§41.2.4) для небольших объектов (в типичном случае размером в одно-два машинных слова). Примитивные операции, не подверженные гонке за данными, часто называют *атомарными операциями*. Их можно использовать в реализации более высокоуровневых механизмов, таких как блокировки, потоки и структуры, свободные от блокировок.

За редким исключением простых атомарных счетчиков, тема программирования без блокировок является прерогативой специалистов. Кроме понимания механизмов языка здесь требуется знание машинных архитектур и деталей конкретных реализаций. Не стоит спешить использовать материал данного раздела сразу же после его прочтения. Для каждой атомарной операции гарантируется, что каждый поток в конце концов не будет блокироваться, даже если иные потоки конкурируют за доступ к атомарному объекту. Технологии без блокировок обеспечивают большее быстродействие, чем программы, использующие блокировки.

Стандартные атомарные типы и операции помогают писать существенно более переносимый код, чем это получается при использовании традиционных приемов программирования без блокировок.

Операция синхронизации — это то, что определяет, когда поток может увидеть результаты работы другого потока. Между операциями синхронизации компилятор и процессор могут переупорядочивать машинный код до тех пор, пока не нарушаются семантические правила языка. В итоге, все это касается исключительно и только производительности. Операция синхронизации на одном или более местоположении в памяти относится к операции потребления, к операции захвата (запроса на синхронизацию) и к операции освобождения (§iso.1.10).

- Для *операции захвата* другие процессоры будут видеть ее эффект до эффектов от последующих операций.
- Для *операции освобождения* другие процессоры будут видеть эффекты от всех предшествующих операций до эффекта от данной операции.
- *Операция потребления* есть более слабая форма операции захвата. Для нее другие процессоры будут видеть ее эффект до эффектов последующих операций, за исключением эффектов, не зависящих от значения операции потребления, которое могло существовать до операции потребления.

Атомарная операция гарантирует, что состояние памяти такое, какое требуется для указанного порядка памяти (§41.2.2). По умолчанию, порядок памяти есть **memory_order_seq_cst** (последовательно согласованный; §41.2.2). Вот список стандартных порядков памяти (§iso.29.3):

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

Здесь перечислители представляют:

- **memory_order_relaxed**: никакие операции не упорядочивают память.
- **memory_order_release**, **memory_order_acq_rel** и **memory_order_seq_cst**: операция сохранения выполняет операцию освобождения над затрагиваемым местоположением в памяти.
- **memory_order_consume**: операция загрузки выполняет операцию потребления над затрагиваемым местоположением в памяти.
- **memory_order_acquire**, **memory_order_acq_rel** и **memory_order_seq_cst**: операция загрузки выполняет операцию захвата на затрагиваемом местоположении в памяти.

Для примера рассмотрим (§iso.29.3) атомарные загрузки и сохранения (§41.3.1) для отражения модели ослабленной памяти:

```
// поток 1:
    r1 = y.load(memory_order_relaxed);
    x.store(r1,memory_order_relaxed);

// поток 2:
    r2 = x.load(memory_order_relaxed);
    y.store(42,memory_order_relaxed);
```

Здесь разрешается произвести результат **r2==42**, как будто время идет в обратном направлении в потоке 2. Таким образом, ослабленная модель памяти **memory_order_relaxed** разрешает такой порядок выполнения:

```
    y.store(42,memory_order_relaxed);
    r1 = y.load(memory_order_relaxed);
    x.store(r1,memory_order_relaxed);
    r2 = x.load(memory_order_relaxed);
```

За разъяснениями обратитесь к специализированной литературе, например [Boehm, 2008] и [Williams, 2012].

От специфики архитектуры зависит, имеет ли данная модель упорядочения памяти смысл. Ясно, что ослабленная модель памяти вряд ли будет напрямую использоваться в прикладном программировании. Это даже более специфическая задача, чем реализация стиля пррограммирования без блокировок. Такую задачу по частям решают отдельные блоки ядра операционной системы, драйверы устройств и реализации виртуальных машин. Если два потока не разделяют напрямую некоторые данные, то для ряда машинных архитектур посредством модели ослабленной памяти удается добиться существенной прибавки в производительности за счет примитивов передачи сообщений (**future** и **promise**; §42.4.4).

Чтобы позволить машинным архитектурам достигать существенной оптимизации при модели ослабленной памяти, стандартная библиотека предоставляет атрибут **[[carries_dependency]]** для того, чтобы распространить зависимость от упорядочения памяти поверх функциональных вызовов (§iso.7.6.4). Например:

```
[[carries_dependency]] struct foo* f(int i)
{
    // позволяет вызвавшему использовать memory_order_consume над результатом:
    return foo_head[i].load(memory_order_consume);
}
```

Также можно применять атрибут **[[carries_dependency]]** к аргументам функций, и есть функция **kill_dependency()** для остановки распространения таких зависимостей.

Про модель памяти C++ можно сказать (устами Lawrence Crowl), что это наиболее сложная особенность параллелизма. Ее стоит использовать

- если для вашей машины это имеет смысл
- если у вас величайшая пропускная способность для преимущественно чтения атомарных структур данных и
- если вы хотите потратить пару недель на тестирование вашего кода

В общем, это территория экспертов.

Будем считать, что вас предупредили.

41.3.1. Атомарные типы

Атомарные типы являются специализациями шаблона **atomic**. Операции над объектами типа **atomic** являются *атомарными*. То есть, они выполняются одним потоком без помех со стороны других потоков.

Операции над этими типами весьма простые: загрузка и хранение, обмен значениями, инкрементирование и т. д. на простом объекте (обычно соответствующем одному местоположению в памяти; §41.2.1). Объекты должны быть простыми, иначе аппаратура не сможет напрямую работать с ними.

Следующая таблица призвана дать первое впечатление. Если явно не оговорено обратное, то упорядочение памяти есть **memory_order_seq_cst** (последовательно упорядочена):

atomic<T> (§iso.29.5) **x.val** есть значение атомарной **x**; все функции **noexcept**	
atomic x;	**x** – не инициализирована
atomic x {};	Конструктор по умолчанию: **x.val=T{}**; constexpr
atomic x {t};	Конструктор: **x.val=t**; constexpr
x=t	Присваивание **T**: **x.val=t**
t=x	Неявное преобразование к **T**: **t=x.val**
x.is_lock_free()	Действия над **x** без блокировок?
x.store(t)	**x.val=t**
x.store(t,order)	**x.val=t**; порядок памяти **order**
t=x.load()	**t=x.val**
t=x.load(order)	**t=x.val**; порядок памяти **order**
t2=x.exchange(t)	Поменять местами значения **x** и **t**; **t2** – предыдущее значение **x**
t2=x.exchange(t,order)	Поменять местами значения **x** и **t**; порядок памяти **order**; **t2** – предыд. значение **x**
b=x.compare_exchange_weak(rt,t)	Если **b=(x.val==r t)**, **x.val=t**, иначе **rt=x.val**; **rt – T&**
b=x.compare_exchange_weak(rt,t,o1,o2)	**b=x.compare_exchange_weak(rt,t)**; использовать порядок памяти **o1**, когда **b==true**; использовать порядок памяти **o2**, когда **b==false**
b=x.compare_exchange_weak(rt,t,order)	**b=x.compare_exchange_weak(rt,t)**; использ. **order** порядок памяти (см. §iso.29.6.1[21])
b=x.compare_exchange_strong(rt,t,o1,o2)	Как **b=x.compare_exchange_weak(rt,t,o1,o2)**
b=x.compare_exchange_strong(rt,t,order)	Как **b=x.compare_exchange_weak(rt,t,order)**
b=x.compare_exchange_strong(rt,t)	Как **b=x.compare_exchange_weak(rt,t)**

У типов **atomic** нет копирующих или перемещающих операций. Операция присваивания и конструктор принимают значения содержащегося типа **T** и выполняют доступ к содержащемуся значению.

По умолчанию **atomic** (без явного {}) неинициализирован, что согласуется со стандартной библиотекой языка С.

Имеется операция **is_lock_free()**, так что можно проверять, являются ли эти операции свободными от блокировок или же они были реализованы с применением блокировок. На всех основных реализациях **is_lock_free()** возвращает **true** для интегральных типов и типов указателей.

Средства **atomic** разработаны для типов, которые отображаются в простые встроенные типы. Следует ожидать, что **atomic<T>** реализованы с блокировками в случае, если объекты типа **T** большие. Шаблонный аргумент типа **T** должен копироваться тривиально (не должен иметь пользовательских копирующих операций).

Инициализация **atomic** переменной не является атомарной, так что при ее выполнении возможны гонки за данными с доступом из другого потока (§iso.29.6.5). Стоит отметить, что вызвать гонку за данными во время инициализации не такая уж и простая задача. Как всегда, старайтесь делать инициализацию нелокальных объектов как можно более простой и выполняйте ее константными выражениями (никакой гонки за данными до начала выполнения программы не может быть).

Простая **atomic** переменная близка к идеалу для счетчика ссылок, вроде подсчета использования разделяемых структур данных. Например:

```
template<typename T>
class shared_ptr {
public:
    // ...
    ~shared_ptr()
    {
        if (--*puc) delete p;
    }
private:
    T*p;                    // указатель на разделяемый объект
    atomic<int>* puc;       // указатель на счетчик ссылок
};
```

Здесь ***puc** имеет тип **atomic** (аллоцируется где-то посредством **shared_ptr** конструктора), так что операция декремента (--) атомарная и новое значение корректно отображается в **thread**, уничтожающем **shared_ptr**.

Первый аргумент операции «сравнить-и-обменять» (**rt** в таблице) является ссылкой для того, чтобы объект, на который ссылаются, мог быть обновлен, если операции не удается обновить свою цель (это **x** в таблице).

Разница между **compare_exchange_strong()** и **compare_exchange_weak()** состоит в том, что слабая (weak) версия может провалиться по неожиданным причинам. Например, особенности аппаратуры или реализации **x.compare_exchange_weak(rt,t)** могут привести к неудаче, даже если **x.val==rt**. Допущение такой неудачи позволяет реализовать **compare_exchange_weak()** на архитектурах, где будет затруднительно или дорого реализовать **compare_exchange_strong()**.

Классический цикл «сравнить-и-обменять» можно написать так:

```
atomic<int> val = 0;
// ...
int expected = val.load();              // читаем текущее значение
do {
    int next = fct(expected);           // подсчитываем новое значение
} while (!val.compare_exchange_weak(expected,next));   // пишем next в val или в expected
```

Атомарная **val.compare_exchange_weak()(expected,next)** читает текущее значение **val** и сравнивает его с **expected**; если есть равенство, то **next** записывается в **val**. Если другой поток записал что-то в **val** пока мы читали его для приготовления к изменению, нужно повторить попытку снова. В новой попытке мы используем новое значение **expected**, полученное от **compare_exchange_weak()**. В конце концов, ожидаемое (expected) значение будет записано. Значение **expected** есть «текущее значение **val**, как его видит этот поток». Итак, поскольку **expected** обновляется до текущего значения каждый раз, как выполняется **compare_exchange_weak()**, мы никогда не получим бесконечного цикла.

Операции вроде **compare_exchange_strong()** широко известны как операции «сравнить-и-обменять». Имеется серьезная проблема с такими операциями (в любом языке и на любых машинах). Рассмотрим добавление узла в голову очень простого односвязного списка, свободного от блокировок, если значение **data** меньше, чем **data** в голове:

```
extern atomic<Link*> head;              // разделяемая голова списка

Link* nh = new Link(data,nullptr);      // подготовим узел для вставки
Link* h = head.load();                  // читаем разделяемую голову списка
do {
    if (h->data<data) break;            // если так, добавить в другом месте
    nh->next = h;                       // элемент next – предыдущая голова
} while (!head.compare_exchange_weak(h,nh));    // пишем nh в head или в h
```

Это упрощенная версия кода, который внедряет **data** в надлежащую позицию в упорядоченном связанном списке. Читаем **head**, используем его в **next** в новом **Link**, а затем пишем указатель на новый **Link** в **head**. Делаем это раз за разом, пока другой поток не сподобится изменить **head**, пока я добиваюсь, чтобы был готов **nh**.

Изучим код детальнее. Назовем **A** то значение **head**, которое я читаю. Если никакой другой поток не поменял значение **head** перед тем, как я выполняю **compare_exchange_weak()**, то он находит **A** в **head** и успешно заменяет его на мой **nh**. Если другой поток меняет **head** на **B** после того, как я прочитал **A**, то мой **compare_exchange_weak()** потерпит неудачу и я снова пойду в цикле за новым чтением **head**.

Кажется, что все в порядке. Что может пойти не так? После того, как я прочитал значение **A**, другой поток изменил величину **head** в **B** и повторно использовал **Link**. Затем, некоторый поток использует узел **A** и повторно внедрит его в голову (**head**) списка. Теперь мой **compare_exchange_weak()** найдет **A** и выполнит обновление. Однако, список изменился; значение **head** изменилось от **A** к **B**, и снова к **A**. Это во многих случаях может быть существенным, но в нашем упрощенном примере **A->data** могло измениться, так что критически важное сравнение **data** может оказаться ложным. Такие ошибки очень трудно обнаружить. Есть способы справиться с такими проблемами [Dechev,2010]. Здесь я рассмотрел пример для того, чтобы подчеркнуть тонкости программирования без блокировок.

Интегральные атомарные типы предлагают атомарные арифметические и битовые операции:

atomic<T> для интегральных T (§iso.29.6.3) **x.val** есть значение атомарной **x**; все функции **noexcept**	
z=x.fetch_add(y)	**x.val+=y**; **z** – предыдущий **x.val**
z=x.fetch_add(y,order)	**z=x.fetch_add(y)**; использ. **order**
z=x.fetch_sub(y)	**x.val-=y**; **z** – предыдущий **x.val**
z=x.fetch_sub(y,order)	**z=x.fetch_sub(y)**; использ. **order**
z=x.fetch_and(y)	**x.val&=y**; **z** – предыдущий **x.val**

Продолжение

atomic<T> для интегральных T (§iso.29.6.3) x.val есть значение атомарной x; все функции noexcept	
z=x.fetch_and(y,order)	z=x.fetch_and(y); использ. order
z=x.fetch_or(y)	x.val|=y; z – предыдущий x.val
z=x.fetch_or(y,order)	z=x.fetch_or(y); использ. order
z=x.fetch_xor(y)	x.val^=y; z – предыдущий x.val
z=x.fetch_xor(y,order)	z=x.fetch_xor(y); использ. order
++x	++x.val; вернуть x.val
x++	x.val++; вернуть предыдущий x.val
--x	--x.val; вернуть x.val
x--	x.val--; вернуть предыдущий x.val
x+=y	x.val+=y; вернуть x.val
x-=y	x.val-=y; вернуть x.val
x&=y	x.val&=y; вернуть x.val
x|=y	x.val|=y; вернуть x.val
x^=y	x.val^=y; вернуть x.val

Рассмотрим популярную идиому «блокировки с двойной проверкой». Основная идея состоит в том, что если инициализация x должна выполняться под блокировкой, то вряд ли вы захотите платить за опрос блокировки каждый раз, когда вы выполняете доступ к x, чтобы узнать о завершении инициализации. Вместо этого, вы блокируете и выполняете инициализацию только в том случае, если переменная x_init равна false:

```
X x;                                  // нужна блокировка, чтобы инициализировать X
mutex lx;                             // mutex которым блокируют x при инициализации
atomic<bool> x_init {false};          // atomic, применяемый для уменьшения числа блокировок

void some_code()
{
    if (!x_init) {          // делаем, если x не инициализирована
        lx.lock();
        if (!x_init) {      // делаем, если x по прежнему не инициализирована
            // ... инициализируем  x ...
            x_init = true;
        }
        lx.unlock();
    }
    // ... используем x ...
}
```

Если бы x_init не был atomic, то переупорядочение машинных команд могло бы передвинуть инициализацию x перед очевидно неуместной проверкой x_init (см. §41.2.2). Мы предотвращаем это, сделав x_init атомарным.

Выражение !x_init полагается на неявное приведение от atomic<T> к T.

Этот код можно еще больше упростить, если применить технику «получение ресурса есть инициализация» (§42.3.1.4).

Идиома блокировки с двойной проверкой представлена в стандартной библиотеке посредством once_flag и call_once() (§42.3.3), так что вам не нужно писать такой код явным образом.

Стандартная библиотека также поддерживает atomic указатели:

atomic<T*> для указателей (§iso.29.6.4) x.val есть значение атомарной x; все функции noexcept	
z=x.fetch_add(y)	x.val+=y; z – предыдущий x.val
z=x.fetch_add(y,order)	z=x.fetch_add(y); использ. order
z=x.fetch_sub(y)	x.val-=y; z – предыдущий x.val
z=x.fetch_sub(y,order)	z=x.fetch_sub(y); использ. order
++x	++x.val; вернуть x.val
x++	x.val++; вернуть предыдущий x.val
--x	--x.val; вернуть x.val
x--	x.val--; вернуть предыдущий x.val
x+=y	x.val+=y; вернуть x.val
x-=y	x.val-=y; вернуть x.val

Ради совместимости со стандартной библиотекой языка C типы функций-членов atomic имеют следующие эквиваленты:

atomic_* operations (§iso.29.6.5) Все функции noexcept	
atomic_is_lock_free(p)	Атомарны ли объекты типа *p?
atomic_init(p,v)	Инициализировать *p знач-ем v
atomic_store(p,v)	Поместить v в *p
x=atomic_load(p)	Присвоить *p в x
x=atomic_load(p)	Загрузить *p в x
b=atomic_compare_exchange_weak(p,q,v)	Сравнить и обменять *p и *q; b=(*q==v) ...
еще примерно 70 функций ...	

41.3.2. Флаги и «ограждения»

Кроме поддержки атомарных типов, стандартная библиотека предлагает два низкоуровневых средства синхронизации: атомарные флаги и «ограждения». Их цель состоит в том, чтобы реализовать самые низкоуровневые средства атомарности, такие как атомарные типы и спин-блокировки. Это единственные механизмы, свободные от блокировок, которые гарантированно поддерживаются на всех основных платформах.

По-существу, у программистов нет нужды использовать флаги и «ограждения».

41.3.2.1. Флаги

Простейшим атомарным типом является atomic_flag и он единственный, для которого гарантируются операции в каждой реализации. atomic_flag представляет один бит информации. При необходимости остальные атомарные типы могут реализовываться, используя atomic_flag.

Два возможных значения для atomic_flag есть set и clear.

atomic_flag (§iso.29.7) Все функции noexcept	
atomic_flag fl;	Значение fl не определено
atomic_flag fl {};	Умолчательный констр-р: знач-е fl есть 0
atomic_flag fl {ATOMIC_FLAG_INIT};	Инициализировать fl знач-ем clear
b=fl.test_and_set()	Установить fl, b – старое значение fl
b=fl.test_and_set(order)	Установить fl, b – старое значение fl; использ. порядок памяти order

Продолжение

atomic_flag (§iso.29.7) Все функции noexcept	
fl.clear()	Очистить fl
fl.clear(order)	Очистить fl; использ. порядок order
b=atomic_flag_test_and_set(flp)	Установить *flp; b старое значение *flp
b=atomic_flag_test_and_set_explicit(flp,order)	Установить *flp; b старое значение *flp; использ. порядок order
atomic_flag_clear(flp)	Очистить *flp
atomic_flag_clear_explicit(flp,order)	Очистить *flp; использ. порядок order

Возвраты типа **bool** есть **true** для **set** и **false** для **clear**.

Использование {} для инициализации **atomic_flag** кажется имеет смысл. Однако, нет уверенности в том, что **0** представляет clear. По слухам, имеются машины, на которых **1** представляет clear. Очистка (clearing) посредством **ATOMIC_FLAG_INIT** является единственным переносимым и надежным способом инициализации **atomic_flag**. Макрос **ATOMIC_FLAG_INIT** предоставляется реализациями.

Можно представлять себе **atomic_flag** как простую спин-блокировку:

```
class spin_mutex {
    atomic_flag flag = ATOMIC_FLAG_INIT;
public:
    void lock() { while(flag.test_and_set()); }
    void unlock() { flag.clear(); }
};
```

Отметим, что спин-блокировки могут обходиться очень дорого.

41.3.2.2. «Ограждения»

«Ограждения», также известные как *барьеры памяти*, это операции, которые ограничивают переупорядочение машинных команд конкретным упорядочением памяти (§41.2.3). Больше операции «ограждения» ничего не делают. Можно сказать, что они просто замедляют работу программы до безопасной скорости, позволяя иерархии памяти достичь хорошо определенного состояния.

«Ограждения» (§iso.29.8) Все функции noexcept	
atomic_thread_fence(order)	Применить порядок памяти order
atomic_signal_fence(order)	Применить порядок памяти order к потоку и обработчику сигнала, исполняемого на этом потоке

«Ограждения» используются в комбинации с **atomic** (нужны для того, чтобы видеть эффект от «ограждений»).

41.4. volatile

Спецификатор **volatile** используется для того, чтобы отметить факт, что объект может модифицироваться чем-то внешним по отношению к потоку управления. Пример:

```
volatile const long clock_register; // обновляется системным таймером
```

Спецификатор **volatile** сообщает компилятору, чтобы он не оптимизировал (не выбрасывал) очевидно избыточные чтения и записи. Например:

```
auto t1 {clock_register};
// ... здесь clock_register не используется ...
auto t2 {clock_register};
```

Если бы **clock_register** не был **volatile**, то компилятор с полным правом исключил бы одно из чтений и предположил, что **t1==t2**.

Не следует применять **volatile** иначе как в низкоуровневом коде, непосредственно взаимодействующим с аппаратурой.

Не следует думать, что **volatile** имеет какой-то специальный смысл в модели памяти. Это не так. И это (в отличие от некоторых более поздних языков) не механизм синхронизации. Для синхронизации используйте **atomic** (§41.3), **mutex** (§42.3.1) или **condition_variable** (§42.3.4).

41.5. Советы

[1] Используйте параллелизм для улучшения отзывчимости программы или для увеличения пропускной способности; §41.1.

[2] Работайте на самом высоком уровне абстракций, какой вы только можете позволить; §41.1.

[3] Предпочитайте **packaged_task** и **future** перед использованием **thread** и **mutex**; §41.1.

[4] Предпочитайте **mutex** и **condition_variable**, а не непосредственное использование **atomic** (только для простых счетчиков); §40.2.

[5] Старайтесь избегать явно разделяемых данных всюду, где только возможно; §41.1.

[6] Рассматривайте процессы как альтернативу потокам; §41.1.

[7] Стандартные средства параллелизма безопасны по типам; §41.1.

[8] Модель памяти существует для того, чтобы спасти большинство программистов от необходимости думать о машинных архитектурах; §41.2.

[9] Модель памяти представляет память так, как она воспринимается по наивности; §41.2.

[10] Отдельные потоки, выполняющие доступ к отдельным битовым полям структур, могут мешать друг другу; §41.2.

[11] Избегайте гонки за данными; §41.2.4.

[12] Атомарность помогает осуществлять программирование без блокировок; §41.3.

[13] Программирование без блокировок помогает избежать мертвых взаимоблокировок и гарантирует, что любой поток не зависнет; §41.3.

[14] Оставьте программирование без блокировок для экспертов; §41.3.

[15] Оставьте модель ослабленной памяти для экспертов; §41.3.

[16] **volatile** говорит компилятору, что значение объекта может быть изменено чем-то, что не является частью программы; §41.4.

[17] **volatile** в C++ не является механизмом синхронизации; §41.4.

42

Потоки и задачи

Сохраняйте спокойствие и идите вперед
— Английская поговорка

- Введение
- Потоки

  Идентификация; Конструирование; Деструктор; Пространство имен **this_thread**; Уничтожение **thread**; Переменные **thread_local**
- Устранение гонки за данными

  Мьютексы; Множественные блокировки; Условные переменные
- Параллелизм на базе задач

  future и **promise**; **promise**; **packaged_task**; **shared_future**; **async()**; Пример параллельной **find()**
- Советы

42.1. Введение

Параллелизм — выполнение нескольких задач одновременно — широко используется для увеличениея вычислительной пропускной способности (используя неколько процессоров для одной и той же задачи) или для улучшения отзывчивости программы (одна часть программы продвигается, в то время как остальные части ждут ответа от пользователя).

Поддержка стандартом C++ параллелизма рассмотрена в простой манере в §5.3. В настоящей главе (и в предыдущей) этот вопрос рассматривается более систематически.

Назовем вычислительную активность, потенциально выполняемую параллельно другим активностям, *задачей*. *Поток* — это представление средств системного уровня для выполнения задачи. Поток реализуется типом **thread**, который выполняет задачу и разделяет адресное пространство с другими потоками. То есть, все потоки в одном адресном пространстве имеют доступ к одним и тем же местоположениям в памяти. Важнейшей задачей программиста будет обеспечить такой доступ в разумной манере.

42.2. Потоки

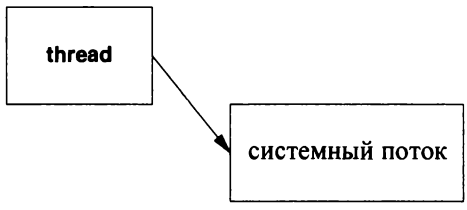
Поток является абстракцией компьютерной аппаратуры для вычислительной активности. В языке C++ тип **thread** из стандартной библиотеки обеспечивает точное соответствие потокам уровня операционной системы. На компьютерах с несколькими процессор-

ными единицами (ядрами) потоки позволяют эффективно задействовать все эти единицы. Все потоки задействуют одно и то же адресное пространство. Во избежание этого нужно использовать процессы. Потоки располагают отдельными стеками, так что локальные переменные не подвержены гонке за данными. Нужно остерегаться контекстных захватов по ссылке в лямбдах (§11.4.3). Намеренное и аккуратное разделение стековой памяти полезно, например, для передачи частей локального массива параллельной задаче сортировки.

Если поток не может продолжать работать (например, он встретил **mutex**, которым владеет другой **thread**), то говорят, что он блокирован или что он спит.

thread (§iso.30.3.1)	
id	Тип идентификатора **thread**
native_handle_type	Тип системного дескриптора потока; зависит от реализации (§iso.30.2.3)
thread t {};	Конструктор по умолчанию: создает **thread**, который (еще) не имеет задачи;
thread t {t2};	Перемещающий конструктор; noexcept
thread t {f,args};	Конструктор: выполняет **f(args)** на новом **thread**; explicit
t.~thread();	Деструктор: если **t.joinable()**, то **terminate()**; иначе – ничего не делает
t=move(t2)	Перемещение: если **t.joinable()**, то **terminate()**; noexcept
t.swap(t2)	Поменять местами значения **t** и **t2**; noexcept
t.joinable()	Имеется ли связанный с **t** поток исполнения? **t.get_id()!=id{}**?; noexcept
t.join()	Присоединяет **t** к текущему потоку (**thread**); то есть, блокирует текущий поток, пока **t** не завершится; генерирует **system_error**, если обнаружена взаимоблокировка (напр., **t.get_id()==this_thread::get_id()**); генерирует **system_error** если **t.id==id{}**
t.detach()	Гарантирует, что **t** не представляет никакой поток в системе; генерирует **system_error** если **t.id!=id{}**
x=t.get_id()	**x** содержит **id** потока **t**; noexcept
x=t.native_handle()	**x** – системный дескриптор **t** (тип **native_handle_type**)
n=hardware_concurrency()	**n** – число аппаратных процессорных единиц (**0** значит «не знаю»); noexcept
swap(t,t2)	**t.swap(t2)**; noexcept

Тип **thread** представляет системный поток, возможно на выделенной аппаратуре:



Как следствие, **thread** допускает перемещение, но не копирование.

Если **thread** стал источником перемещения, то после этого он больше не представляет поток выполнения команд. В частности, к нему нельзя применять **join()**.

Операция **thread::hardware_concurrency()** сообщает о числе задач, которые могут одновременно выполняться с аппаратной поддержкой. Точный смысл операции зависит от компьютерной архитектуры, но обычно это число меньше, чем число потоков, предлагаемое операционной системой (например, за счет мультиплексирования по времени), а иногда больше, чем число процессоров или ядер. К примеру, мой небольшой двухъядерный ноутбук сообщает о четырех аппаратных потоках (используется *hyper-threading*).

42.2.1. Идентификация

Каждый поток исполнения имеет уникальный идентификатор, представимый значением типа **thread::id**. Если **thread** не представляет собой реальный поток исполнения команд, то его **id** есть умолчательный **id{}**. Для **thread t** его **id** находится вызовом **t.get_id()**.

Для текущего **thread** его **id** находится посредством **this_thread::get_id()** (§42.2.6).

Для **thread** его **id** равен **id{}** в случае, если

- нет связанной с ним задачи,
- он завершен,
- он был перемещен или
- он был подвергнут операции **detach()** (отсоединен).

У каждого потока есть **id**, но системный поток может продолжать работать даже, если у него нет **id** (например, после вызова **detach()**).

Можно копировать **thread::id** и эти идентификаторы можно сравнивать обычными операциями сравнения (==, < и т. д.), можно выводить операцией << и их можно хэшировать специализацией **hash<thread::id>** (§31.4.3.4). Например:

```
void print_id(thread& t)
{
    if (t.get_id()==id{})
        cout << "t not joinable\n";
    else
        cout << "t's id is " << t.get_id() << '\n';
}
```

Заметьте, что **cout** является глобальным разделяемым объектом, так что такие операторы вывода не дают гарантии того, что символы в выводе находятся в распознаваемой последовательности, если только вы не уверены, что никаких два потока не пишут в **cout** одновременно (§iso.27.4.1).

42.2.2. Конструирование

Конструктор **thread** принимает задачу для выполнения и аргументы, нужные для этой задачи. Число и тип аргументов должны соответствовать тому, что требуется задаче. Пример:

```
void f0();      // нет аргументов
void f1(int);   // один аргумент типа int

thread t1 {f0};
thread t2 {f0,1};                   // ошибка: слишком много аргументов
thread t3 {f1};                     // ошибка: слишком мало аргументов
thread t4 {f1,1};
thread t5 {f1,1,2};                 // ошибка: слишком много аргументов
thread t3 {f1,"I'm being silly"};   // ошибка: неправильный тип аргумента
```

После конструирования **thread** начинает выполнять задачу сразу, как только среда выполнения получит для этого необходимые системные ресурсы. Можно трактовать это как «немедленно». Отдельной операции «запустить **thread** в работу» нет.

Если вам нужно построить набор задач и связать их воедино (например, чтобы они выполняли коммуникацию посредством очередей сообщений), вы сначала определяете задачи как функциональные объекты, а затем — когда они все готовы к работе — стартуете потоки **thread**. Например:

```
template<typename T>
class Sync_queue<T> { // очередь, предоставляющая put() и get() без гонок за данными (§42.3.4)
    // ...
};

struct Consumer {
    Sync_queue<Message>& head;
    Consumer(Sync_queue<Message>& q) :head(q) {}
    void operator()();    // берем сообщения из головы
};

struct Producer {
    Sync_queue<Message>& tail;
    Consumer(Sync_queue<Message>& q) :tail(q) {}
    void operator()();    // добавляем сообщения в хвост
};

Sync_queue<Message> mq;
Consumer c {mq};          // создаем задачи и "связывем их воедино"
Producer p {mq};

thread pro {p};           // наконец: запускаем потоки
thread con {c};

// ...
```

Попытка создания нескольких **thread** путем установки соединений между задачами, которые будут выполняться набором **thread**, является сложной проблемой, чреватой ошибками.

Конструкторы **thread** являются шаблонами с переменным числом параметров (§28.6). Из этого следует, что чтобы передать ссылку на конструктор **thread**, мы должны использовать связку ссылок (§33.5.1). Например:

```
void my_task(vector<double>& arg);

void test(vector<double>& v)
{
    thread my_thread1 {my_task,v};              // oops: передает копию v
    thread my_thread2 {my_task,ref(v)};         // OK: передает v по ссылке
    thread my_thread3 {[&v]{ my_task(v); }};    // OK: избегаем проблемы с ref()
    // ...
}
```

Проблема здесь в том, что шаблон с переменным числом параметров использует **bind()** (или какой-либо эквивалентный механизм), так что ссылка по умолчанию разыменовывается и результат копируется. В итоге, если **v** был **{1,2,3}** и **my_task** инкрементирует элементы, то **thread1** никогда не повлияет на **v**. Заметьте, что все три **thread** вовлечены в гонку за данными на переменной **v**; это пример про соглашения о вызовах, а не пример хорошего стиля программирования.

Сконструированный по умолчанию **thread** преимущественно полезен в качестве цели для перемещения. Пример:

```
vector<thread> worker(1000); // 1000 потоков по умолчанию

for (int i=0; i!=worker.size(); ++i) {
    // ... вычисляем аргумент для worker[i] и создаем поток tmp ...
    worker[i] = move(tmp);
}
```

Перемещение задачи из одного **thread** в другой не влияет на его выполнение; просто изменяется то, на что **thread** ссылается.

42.2.3. Деструктор

Очевидно, что деструктор **thread** уничтожает объект **thread**. Чтобы воспрепятствовать тому, что системный поток переживет свой **thread**, деструктор вызывает **terminate()** для завершения программы, если **thread** есть **joinable()** (то есть, если **get_id()!=id{}**). Например:

```
void heartbeat()
{
    while(true) {
        output(steady_clock::now());
        this_thread::sleep_for(second{1}); // §42.2.6
    }
}

void run()
{
    thread t {heartbeat};
}   // terminate(), поскольку heartbeat() по прежнему исполняется на момент уничтожения t
```

Если вам действительно нужно, чтобы системный поток продолжал работать вне времени жизни его **thread**, то обратитесь к §42.2.5.

42.2.4. join()

t.join() говорит текущему **thread** не приступать к работе, пока **t** не завершится. Пример:

```
void tick(int n)
{
    for (int i=0; i!=n; ++i) {
        this_thread::sleep_for(second{1}); // §42.2.6
        output("Alive!");
    }
}

int main()
{
    thread timer {tick,10};
    timer.join();
}
```

Тут будет десять раз выведено **Alive!** с интервалами около 1 сек. Если опустить **timer.join()**, то программа остановится перед тем, как **tick()** что-либо выведет. Функция **join()** заставит основную программу ждать, пока **timer()** не завершится.

Как описывается в §42.2.3, попытка выполнять **thread** вне области его видимости (более точно — после вызова деструктора) без вызова **detach()** рассматривается как фатальная

(для программы) ошибка. Но мы можем просто забыть вызвать **join()** для **thread**. Когда мы смотрим на **thread** как на ресурс, неплохо вспомнить про парадигму «захват ресурса есть инициализация» (§5.2, §13.3). Рассмотрим простую тестовую программу:

```
void run(int i, int n)      // внимание: действительно плохой код
{
    thread t1 {f};
    thread t2;
    vector<Foo> v;
    // ...
    if (i<n) {
        thread t3 {g};
        // ...
        t2 = move(t3);      // переместим t3 во внешнюю область видимости
    }
    v[i] = Foo{};   // может сгенерировать исключение
    // ...
    t1.join();
    t2.join();
}
```

Здесь я допустил несколько неприятных ошибок. В частности:

- Можно никогда не достичь двух **join()** в конце. В этом случае деструктор для **t1** завершит программу.
- Можно достичь двух **join()** в конце без выполнения перемещения **t2=move(t3)**. В этом случае **t2.join()** завершит программу.

При таком способе использования **thread** нам нужен деструктор, который неявно вызовет **join()**. Например:

```
struct guarded_thread : thread {
    using thread::thread;     // §20.3.5.1
    ~guarded_thread() { if (t.joinable()) t.join(); }
};
```

К сожалению, **guarded_thread** не является классом стандартной библиотеки, но в лучших традициях парадигмы «захват ресурса есть инициализация» **guarded_thread** делает наш код короче и менее подверженным ошибкам. Например:

```
void run2(int i, int n)     // простой пример использования guarded_thread
{
    guarded_thread t1 {f};
    guarded_thread t2;
    vector<Foo> v;
    // ...
    if (i<n) {
        thread t3 {g};
        // ...
        t2 = move(t3);      // переместим t3 во внешнюю область видимости
    }
    v[i] = Foo{};           // может сгенерировать исключение
    // ...
}
```

Но почему бы деструктору **thread** просто не вызвать **join()**? Существует долгая традиция позволить жить системным потокам вечно или же пусть они сами решают, когда им завершаться. Если бы это работало, то **timer()**, выполняющий **tick()** (§42.2.2) был бы примером такого потока. У потоков, мониторящих структуры данных, есть много полезных примеров. Такие потоки (и процессы) часто называют *демонами*. Еще одним примером

использования отсоединенных потоков является их применение для инициации потока, который выполнит задачу, после чего о нем можно забыть; «приборка в хозяйстве» в таком случае оставляется за операционной системой.

42.2.5. detach()

Если неумышленно позволить **thread** выполняться после работы деструктора, то это рассматривается как грубая ошибка. Если же вы осознанно хотите позволить системному потоку пережить **thread** (дескриптор), используйте **detach()**. Например:

```
void run2()
{
    thread t {heartbeat};
    t.detach();      // позволит heartbeat работать независимо
}
```

У меня есть философские проблемы относительно открепленных потоков. Имея выбор, я бы предпочел:

- точно знать, какие потоки работают,
- иметь возможность выявить, какие потоки продвигаются так, как ожидалось,
- быть в состоянии проверять, действительно ли потоки, которые вроде бы уничтожили себя, на самом деле это сделали,
- знать, безопасно ли использовать результаты работы потока,
- быть уверенным в том, что все ресурсы, связанные с потоком, освобождены надлежащим образом, и
- быть уверенным, что поток не пытается получить доступ к объектам из области видимости, в которой он был создан, после того, как эта область видимости стала неактивной (была уничтожена).

Если только я не выхожу за границы стандартной библиотеки (например, используя **get_native_handle()** и иные системные средства), я не могу все перечисленное иметь для отсоединенных потоков. Кроме того, как я могу отлаживать программу, в которой поведение отсоединенных потоков не поддается непосредственному наблюдению? Что будет, если отсоединенный поток владеет указателем на нечто из области видимости, в которой он был создан? Это может приводить к испорченным данным, краху системы и к проблемам с безопасностью. И все равно, отсоединенные потоки все-таки могут использоваться и даже отлаживаться. Люди десятилетиями их используют, но люди применяют разрушительные средства на протяжении столетий. Все-таки, имея выбор, я бы предпочел не использовать **detach()**.

Отметим, что **thread** предоставляет перемещающее присваивание и перемещающий конструктор. Это позволяет **thread** мигрировать из области видимости, в которой он конструировался, что является альтернативой для **detach()**. Во избежании потери связи с потоками, мы можем осуществлять их миграцию в основной модуль программы, осуществлять доступ к ним посредством **unique_ptr** или **shared_ptr**, или помещать их в контейнер (вроде **vector<thread>**). Пример:

```
vector<thread> my_threads; // держим здесь потоки, вместо отсоединения

void run()
{
    thread t {heartbeat};
    my_threads.push_back(move(t));
```

```
    // ...
    my_threads.emplace_back(tick,1000);
}

void monitor()
{
    for (thread& t : my_threads)
        cout << "thread " << t.get_id() << '\n';
}
```

В более реалистичном примере я бы связал некоторую информацию с каждым **thread** в **my_threads**. Возможно, я бы запустил **monitor** как задачу.

Если вы все-же обязаны использовать **detach()**, то удостоверьтесь, что поток не ссылается на переменные в своей области видимости. Например:

```
void home()    // не делайте этого
{
    int var;
    thread disaster{[&]{ this_thread::sleep_for(second{7.3});++var; }}
    disaster.detach();
}
```

Несмотря на предупреждающий комментарий, этот код выглядит невинным. На самом деле, это не так: системный поток, активированный посредством **disaster()**, будет «вечно» писать на адрес, где **var** из **home()** был аллоцирован, выполняя порчу данных, которые позже могли аллоцироваться по этому адресу. Такую ошибку трудно обнаружить, поскольку она слабо связана с кодом, в котором себя проявляет, а повторные запуски программы будут приводить к разным результатам (некоторые запуски вообще могут быть свободными от симптомов).

Отметим, что корень проблемы состоит в нарушении хорошо известного принципа «Не передавать указатель на локальный объект во вне его области видимости» (§12.1.4). Однако, в лямбдах можно легко (и почти невидимо) создать указатель на локальную переменную: **[&]**. По-счастью, мы можем вызвать **detach()**, чтобы **thread** покинул свою область видимости; не делайте это без важной причины, или делайте это после внимательного изучения того, что его задача может сделать.

42.2.6. Пространство имен this_thread

Операции на текущем потоке определены в пространстве имен **this_thread**:

Пространство имен this_thread (§iso.30.3.1)	
x=get_id()	**x** есть **id** текущего **thread**; noexcept
yield()	Позволить планировщику выполнять другой **thread**; noexcept
sleep_until(tp)	Заставить текущий **thread** «заснуть» до **time_point tp**
sleep_for(d)	Заставить текущий **thread** «заснуть» на **duration d**

Чтобы получить идентификатор текущего **thread**, вызовите **this_thread::get_id()**. Пример:

```
void helper(thread& t)
{
    thread::id me {this_thread::get_id()};
    // ...
    if (t.get_id()!=me) t.join();
    // ...
}
```

Аналогично, мы можем использовать **this_thread::sleep_until(tp)** и **this_thread::sleep_for(d)**, чтобы заставить текущий **thread** «заснуть».

Функция **this_thread::yield()** используется для того, чтобы дать другому **thread** шанс продвинуться в выполнении. Текущий **thread** не блокирован, так что он должен будет когда-то продвинуться в исполнении, если только какой-нибудь другой **thread** не предпринимает специальных мер по недопущению этого. Таким образом, **yield()** полезен для ожидания атомарных событий изменения состояния и для кооперации мультипотоков. Обычно, лучше использовать **sleep_for(n)**, а не вызывать **yield()**. Аргумент вызова **sleep_for()** дает планировщику шанс для осуществления рационального выбора потоков для их выполнения. Следует рассматривать **yield()** как средство оптимизации в очень редких и специальных случаях.

Все основные реализации делают **thread** вытесняемыми: реализации могут переключать одну задачу на другую так, чтобы гарантировать всем потокам разумный темп продвижения. Однако, по историческим причинам вытеснение лишь приветствуется, но не требуется по стандарту (§iso.1.10).

Обычно программисты не вовлекаются в путаницу с системными часами. Но если часы перезапускаются (например, если они отклонились от правильного времени), то это влияет на **wait_until()**, но не на **wait_for()**. То же относится к **wait_until()** и **wait_for()** для **timed_mutex** (§42.3.1.3).

42.2.7. Уничтожение thread

Одна важная операция отсутствует в **thread**. Не существует простого и стандартного способа сказать запущенному **thread** о том, что мы потеряли интерес к выполняемой им задаче, так что не будет ли он так любезен прекратить выполняться и освободить все занятые им ресурсы. Например, если я запустил параллельный **find()** (§42.4.7), то мне часто будет требоваться оповестить все еще активные задачи о требовании остановиться, так как я нашел то, что мне требовалось. Отсутствие такой операции (с потенциальными именами *kill*, *cancel* или *interrupt*) объясняется многими причинами, в том числе историческими.

При необходимости прикладные программисты должны написать собственные процедуры выполнения указанной задачи. Например, многие задачи периодически (в цикле) опрашивают пользователя на предмет «Не следует ли завершить работу» и при положительном ответе просят соответствующий **thread** освободить ресурсы и остановиться. Если нет такого опрашивающего цикла, то приложение может периодически проверять «нужную» переменную, чтобы удостовериться в необходимости продолжать работу.

Ясно, что трудно специфицировать завершающую операцию в общем виде, чтобы она могла быть реализована на всех системах, но в конкретных условиях нетрудно реализовать самим все необходимые действия.

42.2.8. Переменные thread_local

Как говорит это имя, переменная **thread_local** является локальным объектом, которым владеет **thread** и который недоступен из других потоков, если только владелец не дал им указатель на этот объект (по неосторожности). В этом **thread_local** напоминает локальные переменные, но последние имеют время жизни и доступ к ним лимитированными областью видимости внутри функции, в то время как **thread_local** разделяется между всеми функциями потока и живет столь же долго, что и **thread**. Объект **thread_local** может быть **extern**.

В большинстве случаев иметь объекты локальными (в стеке) предпочтительно в случае их разделения; но при этом создаются и логические проблемы глобальных переменных. Как всегда, можно использовать пространства имен, чтобы ограничить проблемы

с нелокальными данными. Стоит отметить, что на многих системах размер памяти под хранение в стеке лимитирован, так что **thread_local** становится важным для задач, требующих больших объемов неразделяемых данных.

Говорят, что **thread_local** имеет время жизни хранилища потока (§iso.3.7.2). Каждый **thread** имеет свою собственную копию своих переменных **thread_local**. Инициализация **thread_local** выполняется до их первого использования (§iso.3.2). Будучи сконструированными, они уничтожаются тогда, когда **thread** выходит из области видимости.

Важным случаем применения **thread_local** является необходимость для **thread** явным образом хранить кэш данных для эксклюзивного доступа. Это усложняет логику программы, но на машинах с разделяемыми кэшами может привести к значительному выигрышу в производительности. Также это может снизить стоимость блокировок путем передачи данных большими порциями.

В общем случае, нелокальная память является проблемой для параллельного программирования, поскольку часто не тривиально определить, разделена ли она, и тем самым является источником гонки за данными. В частности, статические члены класса могут стать большой проблемой, поскольку они скрыты от пользователей класса и можно легко пропустить потенциальную гонку за данными. Рассмотрим дизайн **Map** с умолчательными значениями для типов:

```
template<typename K, typename V>
class Map {
public:
    Map();
    // ...
    static void set_default(const K&,V&);    // умолч. знач-е для всех Map типа Map<K,V>
private:
    static pair<const K,V> default_value;
};
```

Почему пользователь будет опасаться гонки за данными на двух разных объектах **Map**? Потому что пользователь, который выделил **set_default()** среди других членов, может этого опасаться, но ведь **set_default()** можно и не заметить (§16.2.12).

Стало популярным использовать статические (один экземпляр на класс) значения. Примерами служат умолчательные значения, счетчики использования, кэши, ответы на частые вопросы и многие другие случаи. Когда это применяется в параллельной системе, мы сталкиваемся с классической проблемой:

```
// где-то в потоке thread 1:
    Map<string,int>::set_default("Heraclides",1);

// где-то в потоке thread 2:
    Map<string,int>::set_default("Zeno",1);
```

Здесь имеется потенциальная гонка за данными: который **thread** первым запустит на выполнение **set_default()**?

Здесь нам поможет **thread_local**:

```
template<typename K, typename V>
class Map {
    // ...
private:
    static thread_local pair<const K,V> default_value;
};
```

Теперь потенциальная гонка за данными устранена. Однако, в то же самое время, нет единого **default_value**, разделяемого между всеми пользователями. В этом примере поток

1 никогда не увидит эффекта от **set_default()** в потоке 2. Почти как всегда, это не планировалось в оригинальном коде, так что добавлением **thread_local** мы просто заменили одну проблему на другую. Не стоит считать **thread_local** панацеей.

Переменная из пространства имен, локальная статическая переменная и статический член класса могут объявляться **thread_local**. Как и для статических переменных, конструирование локальных переменных **thread_local** защищено «ключом первого раза» (§42.3.3). Порядок конструирования переменных **thread_local** не определен, так что старайтесь сохранять конструирование разных переменных **thread_local** независимыми от этого порядка, а также применяйте инициализацию на этапе компиляции или на этапе выполнения. Как и простые статические переменные, переменные **thread_local** по умолчанию инициализируются нулем (§6.3.5.1).

42.3. Устранение гонки за данными

Простейший способ избежать гонки за данными состоит в том, чтобы не разделять данные между потоками. Старайтесь хранить важные данные в локальных переменных, в свободной памяти, не разделяемой с другими потоками, или в памяти **thread_local** (§42.2.8). Не передавайте указатели на такие данные в другие потоки. Если же требуется, чтобы эти данные обрабатывались другим **thread** (например, параллельная сортировка), передавайте указатели на конкретные части данных и не касайтесь этих частей до завершения задачи.

Эти простые правила базируются на идее избежать одновременного доступа к данным, так что не требуется их блокировка, что ведет к максимально эффективным программам. Там, где эти правила не проходят, например в случае необходимости разделять множество данных, то следует использовать ту или иную форму блокировок:

Мьютекс (семафор; взаимно исключающая переменная): это объект, служащий для представления эксклюзивных прав доступа к ресурсу. При доступе к ресурсу захватите мьютекс, выполните доступ и после этого освободите мьютекс (§5.3.4, §42.3.1).

Условные переменные: это переменные, используемые **thread** для ожидания события, генерируемого другим **thread** или таймером (§5.3.4.1, §42.3.4).

Строго говоря, условные переменные не предотвращают гонку за данными. Они, скорее, спасают нас от разделяемых данных, которые могут стать источником гонки за данными.

42.3.1. Мьютексы

Мьютекс — это объект типа **mutex**, который используется для представления эксклюзивного доступа к некоторому ресурсу. Он может использоваться для предотвращения гонки за данными и для синхронизации доступа к данным, разделяемым между множественными потоками (**thread**).

Классы мьютекс (§iso.30.4)	
mutex	Нерекурсивный мьютекс; **thread** будет заблокирован, если захочет захватить уже захваченный **mutex**
recursive_mutex	Мьютекс, который может повторно захватываться одним **thread**
timed_mutex	Нерекурсивный мьютекс с функциями ожидания захвата в течение ограниченного времени
recursive_timed_mutex	Рекурсивный мьютекс временными функциями
lock_guard<M>	Страж lock_guard для мьютекса **mutex M**
unique_lock<M>	Страж unique_lock для мьютекса **mutex M**

Простой **mutex** — это небольшой и наибыстрейший мьютекс. Помимо добавления функциональности другие мьютексы влекут за собой некоторые дополнительные накладные расходы, которые могут оказаться (или не оказаться) существенными на конкретной машине.

В каждый момент времени только один **thread** может владеть мьютексом:

- *Захватить* мьютекс означает получить эксклюзивное владение им; операция захвата может блокировать **thread**, выполняющий эту операцию.
- Освободить мьютекс означает отказаться от эксклюзивного владения; эта операция позволяет другому **thread** впоследствии захватить этот мьютекс. Таким образом, операция освобождения разблокирует ожидающие потоки.

Если несколько потоков блокированы на мьютексе, то системный планировщик сам выбирает поток, который будет разблокирован, причем некоторый «несчастливый» **thread** может никогда и не возобновить работу. Планировщики, которые дают ожидающим потокам одинаковые шансы на возобновление работы, называют «честными». Например, планировщик может всегда выбирать **thread** с наибольшим **thread::id**, тем самым обрекая на голод **thread** с меньшим **id**. Стандарт не гарантирует «честность» планировщиков, но на практике почти все из них более-менее «честные». Это означает, что вероятность того, что какой-то поток будет вечно «голодать», ничтожно мала. Можно для планировщика, например, придерживаться случайной выборки потоков (из тех, что заблокированы), подлежащих выполнению.

Сам по себе, мьютекс ничего не делает. Вместо этого, мьютекс используется для представления чего-то иного. Мы используем владение мьютексом для представления прав на манипулирование ресурсом вроде объекта, структуры данных или же устройства ввода/вывода. Например, мы может определить **cout_mutex** для представления права использовать **cout** из потока:

```
mutex cout_mutex; // представляет право использовать cout

template<typename Arg1, typename Arg2, typename Arg3>
void write(Arg1 a1, Arg2 a2 = {}, Arg3 a3 = {})
{
    thread::id name = this_thread::get_id();

    cout_mutex.lock();
    cout << "From thread " << name << " : " << a1 << a2 << a3;
    cout_mutex.unlock();
}
```

Если все потоки используют **write()**, то мы получим вывод, надлежаще разделенный для всех потоков. «Засада» в том, что все потоки должны использовать мьютекс так, как это задумывалось. Связь между мьютексом и ресурсом неявная. В примере с **cout_mutex** поток, использующий **cout** явным образом, может попортить вывод. Стандарт гарантирует, что переменная **cout** защищена от порчи, но нет защиты от вывода из других потоков при перемешивании вывода.

Заметьте, что я блокировал мьютекс только на один оператор, нуждающийся в блокировке. Для минимизации времени в состоянии блокировки, нужно удерживать блокировку лишь там, где это действительно нужно. Секция кода, защищенная блокировкой, часто называют *критической секцией*. Таким образом, следует минимизировать размер критических секций.

Мьютексы стандартной библиотеки реализуют *семантику эксклюзивного владения*. В одно и то же время лишь один поток имеет эксклюзивный доступ к ресурсу. Известны

и другие типы мьютексов, например мьютексы множественного чтения-эксклюзивной записи, но они не предоставляются стандартной библиотекой.

42.3.1.1. mutex и recursive_mutex

Класс **mutex** предлагает простой набор операций:

mutex (§iso.30.4.1.2.1)	
mutex m {};	Конструктор по умолчанию: **m** не принадлежит ни одному **thread**; constexpr; noexcept
m.~mutex()	Деструктор: поведение неопределено, если мьютекс захвачен
m.lock()	Захватить **m**; блокирует, покуда не захвачен
m.try_lock()	Попробовать захватить **m**; успешен ли захват?
m.unlock()	Освободить **m**
native_handle_type	Системный тип мьютекса, зависящий от реализации
nh=m.native_handle()	**nh** – системный дескриптор для мьютекса **m**

Мьютекс нельзя копировать или перемещать. Думайте о **mutex** как о ресурсе, а не о дескрипторе ресурса. Фактически, в типичном случае **mutex** реализуется в виде дескриптора к системному ресурсу, но поскольку этот системный ресурс невозможно разделять, копировать и перемещать, то не стоит представлять его отдельной сущностью.

Типовое использование **mutex** довольно простое:

```
mutex cout_mutex; // инициализируется, как "не принадлежащий ни одному потоку"

void hello()
{
    cout_mutex.lock();
    cout << "Hello, ";
    cout_mutex.unlock();
}

void world()
{
    cout_mutex.lock();
    cout << "World!";
    cout_mutex.unlock();
}

int main()
{
    thread t1 {hello};
    thread t2 {world};

    t1.join();
    t2.join();
}
```

Вывод этой программы будет такой:

```
Hello, World!
```

или такой

```
World! Hello,
```

Операция **try_lock()** используется тогда, когда у нас есть полезная работа, пока другой поток использует ресурс. Рассмотрим, к примеру, генератор работ, составляющий запросы на работы для других задач и помещающий их в очередь работ:

```
extern mutex wqm;
extern list<Work> wq;

void composer()
{
    list<Work> requests;

    while (true) {
        for (int i=0; i!=10; ++i) {
            Work w;
            // ... составить рабочий запрос ...
            requests.push_back(w);
        }
        if (wqm.try_lock()) {
            wq.splice(requests);        // splice() запросы в список (§31.4.2)
            wqm.unlock();
        }
    }
}
```

Когда некоторый серверный поток изучает **wq**, **composer()** занят выполнением другой работы вместо пустого ожидания.

Применяя блокировки, нужно учитывать возможность «мертвых блокировок», когда мы ждем поток, который никогда не освободит мьютекс. Простейшая форма «мертвой блокировки» требует лишь одну блокировку и один **thread**. Рассмотрим вариант операции вывода, безопасной по потокам:

```
template<typename Arg, typename... Args>
void write(Arg a, Args tail...)
{
    cout_mutex.lock();
    cout << a;
    write(tail...);
    cout_mutex.unlock();
}
```

Теперь если **thread** вызывает **write("Hello,","World!")**, то он намертво заблокирует сам себя при рекурсивном обращении к **tail**.

Рекурсивные и взаимно рекурсивные вызовы встречаются достаточно часто, чтобы стандарт предоставил решение. **recursive_mutex** подобен обычному **mutex**, за исключением того, что только единственный **thread** может повторно захватывать его. Например:

```
recursive_mutex cout_mutex; // зам-а на recursive_mutex, чтобы избеж. "мертвой блокировки"

template<typename Arg, typename... Args>
void write(Arg a, Args tail...)
{
    cout_mutex.lock();
    cout << a;
    write(tail...);
    cout_mutex.unlock();
}
```

Здесь рекурсивный вызов **write()** корректно обрабатывается посредством **cout_mutex**.

42.3.1.2. Ошибки mutex

Попытки манипулировать мьютексом могут породить ошибки. В таком случае операции генерируют исключение **system_error**. Некоторые из возможных ошибок отражают ситуации в нижележащей системе:

Ошибки мьютекс (§iso.30.4.1.2)	
resource_deadlock_would_occur	Может быть «мертвая блокировка»
resource_unavailable_try_again	Тот или иной системный дескриптор недоступен
operation_not_permitted	**thread** не имеет права произвести операцию
device_or_resource_busy	Тот или иной системный дескриптор уже занят
invalid_argument	Плохой аргумент конструктора

Например:

```
mutex mtx;
try {
        mtx.lock();
        mtx.lock();      // попытка заблокировать второй раз
}
catch (system_error& e) {
        mtx.unlock();
        cout << e.what() << '\n';
        cout << e.code() << '\n';
}
```

При этом я получил следующий вывод:

```
device or resource busy
generic: 16
```

Это аргумент в пользу использования **lock_guard** или **unique_lock** (§42.3.1.4).

42.3.1.3. timed_mutex и recursive_timed_mutex

Простейший **mtx.lock()** является безусловным. Если мы не хотим блокироваться, мы можем использовать **mtx.try_lock()**, но если не удается захватить **mtx**, то часто нам следует подождать, а потом попытаться снова. Поддержку этому осуществляют **timed_mutex** и **recursive_timed_mutex**:

timed_mutex (§iso.30.4.1.3.1)	
timed_mutex m {};	Конструктор по умолчанию; **m** не захвачен; constexpr; noexcept
m.~timed_mutex()	Деструктор: поведение неопределено, если мьютекс захвачен
m.lock()	Захватить **m**; блокирует, покуда не захвачен
m.try_lock()	Попробовать захватить **m**; успешен ли захват?
m.try_lock_for(d)	Попробовать захватить **m** в течение не более **duration d**; успешен ли захват?
m.try_lock_until(tp)	Попробовать захватить **m** самое позднее до **time_point tp**; успешен ли захват?
m.unlock()	Освободить **m**
native_handle_type	Системный тип мьютекса, зависящий от реализации
nh=m.native_handle()	**nh** – системный дескриптор для мьютекса

Интерфейс **recursive_timed_mutex** аналогичен интерфейсу **timed_mutex** (как интерфейс **recursive_mutex** идентичен интерфейсу **mutex**).

Для **this_thread** мы можем выполнить **sleep_until(tp)** для **time_point** и **sleep_for(d)** для **duration** (§42.2.6). В общем случае мы можем выполнить **m.try_lock_until(tp)** или **m.try_lock_for(d)** для **timed_mutex m**. Если момент времени **tp** находится перед текущим моментом времени или если **d** меньше или равен нулю, то операция эквивалентна простому **try_lock()**.

Для примера рассмотрим обновление выходного буфера новым изображением (в видеоигре или в программе визуализации):

```
extern timed_mutex imtx;
extern Image buf;

void next()
{
    while (true) {
        Image next_image;
        // ... вычисляем ...

        if (imtx.try_lock(milliseconds{100})) {
            buf = next_image;
            imtx.unlock();
        }
    }
}
```

Здесь предположение состоит в том, что если изображение не может обновиться достаточно быстро (здесь за 100 милисекунд), то пользователь предпочтет более позднее изображение. Далее, предполагается, что пропуск изображения в последовательности обновляемых изображений трудно заметить, так что нет необходимости в более сложных решениях.

42.3.1.4. lock_guard и unique_lock

Блокировка есть ресурс, так что мы не должны забывать освобождать его. То есть, каждый **m.lock()** должен иметь соответствующий **m.unlock()**. Имеются обычные возможности ошибиться; например:

```
void use(mutex& mtx, Vector<string>& vs, int i)
{
    mtx.lock();
    if (i<0) return;
    string s = vs[i];
    // ...
    mtx.unlock();
}
```

Здесь **m.unlock()** на месте, но если **i<0** или если **i** вне диапазона у **vs**, а **vs** имеет проверку диапазона, то поток выполнения никогда не доберется до **mtx.unlock()** и **mtx** может быть заблокирован навсегда.

Стандартная библиотека предоставляет классы (следующие парадигме «захват ресурса есть инициализация») **lock_guard** и **unique_lock** для решения таких проблем.

Простой **lock_guard** является простейшим и быстрейшим стражем (guard). За небольшую плату дополнительная функциональность предоставляется посредством **unique_ptr**.

lock_guard<M> (§iso.30.4.2) m есть блокируемый объект	
lock_guard lck {m};	lck захватывает m; explicit
lock_guard lck {m,adopt_lock_t};	lck удерживает m; предполагается, что текущий thread уже захватил m; noexcept
lck.~lock_guard()	Деструктор: вызывает unlock() для удерживаемого мьютекса

Например:

```
void use(mutex& mtx, vector<string>& vs, int i)
{
    lock_guard<mutex> g {mtx};
    if (i<0) return;
    string s = vs[i];
    // ...
}
```

Деструктор lock_guard выполняет требуемую операцию unlock() над своими аргументами.

Как всегда, нужно удерживать блокировку не дольше необходимого, так что lock_guard не является оправданием для удержания блокировки вплоть до конца большой области видимости в случае, если нам нужна блокировка лишь на небольшой ее части. Ясно, что проверка i не нуждается в блокировке, и поэтому мы может выполнить ее раньше:

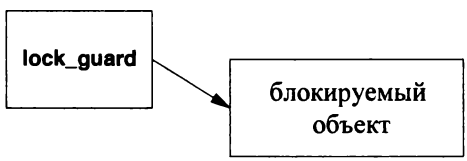
```
void use(mutex& mtx, vector<string>& vs, int i)
{
    if (i<0) return;
    lock_guard<mutex> g {mtx};
    string s = vs[i];
    // ...
}
```

Далее, представим, что нам нужно блокировать чтение v[i]. Тогда можно поместить lock_guard в небольшой блок кода:

```
void use(mutex& mtx, vector<string>& vs, int i)
{
    if (i<0) return;
    string s;
    {
        lock_guard<mutex> g {mtx};
        s = vs[i];
    }
    // ...
}
```

Минимизация размеров критических секций полезна практически всегда. Помимо прочего, это вынуждает нас мыслить о блокировке, как о локальном действии.

Так, lock_guard, а также unique_lock, являются дескрипторами ресурса («стражами») для объекта, который блокируется для удержания владения и разблокируется освобождением ресурса.



Такие объекты называют *блокируемыми объектами*.

lock_guard это очень простой класс, у которого нет интересных операций. Он реализует парадигму «захват ресурса есть инициализация» для **mutex**. Для того, чтобы получить сам объект, содержащий **mutex**, мы используем **unique_lock**:

unique_lock<M> (§iso.30.4.2) **m** – блокируемый объект	
unique_lock lck {};	Конструктор по умолчанию: **lck** не содержит мьютекс;
unique_lock lck {m};	**lck** захватывает **m**; explicit
unique_lock lck {m,defer_lock_t};	**lck** содержит **m**, но не захватывает его
unique_lock lck {m,tr y_to_lock_t};	**lck** содержит **m** и выполняет **m.try_lock()**; если захват успешен, **lck** владеет **m**; иначе – нет
unique_lock lck {m,adopt_lock_t};	**lck** содержит **m**; предполагается, что текущий **thread** уже захватил **m**
unique_lock lck {m,tp};	**lck** содержит **m** и вызывает **m.try_lock_until(tp)**; если захват успешен, **lck** владеет **m**; иначе – нет
unique_lock lck {m,d};	**lck** содержит **m** и вызывает **m.try_lock_for(d)**; если захват успешен, **lck** владеет **m**; иначе – нет
unique_lock lck {lck2};	Перемещающий конструктор: **lck** содержит мьютекс из **lck2** (если такой был); **lck2** не содержит мьютекс
lck.~unique_lock()	Деструктор: вызывает **unlock()** у мьютекса (если есть)
lck2=move(lck)	Перемещение: **lck** содержит мьютекс из **lck2** (если такой был); **lck2** не содержит мьютекс
lck.lock()	**m.lock()**
lck.try_lock()	**m.try_lock()**; успешен ли захват?
lck.try_lock_for(d)	**m.try_lock_for(d)**; успешен ли захват?
lck.try_lock_until(tp)	**m.try_lock_until(tp)**; успешен ли захват?
lck.unlock()	**m.unlock()**
lck.swap(lck2)	Обменивает блокир. объекты **lck** и **lck2**; noexcept
pm=lck.release()	**lck** более не владеет ***pm**; noexcept
lck.owns_lock()	Владеет ли **lck** блокируемым объектом? noexcept
bool b {lck};	Преобразование в **bool**; **b==lck.owns_lock()**; explicit; noexcept
pm=lck.mutex()	***pm** – владеемый блокир. объект, если есть; иначе **pm=nullptr**; noexcept
swap(lck,lck2)	**lck.swap(lck2)**; noexcept

Очевидно, что временные операции разрешены только в том случае, если мьютекс есть **timed_mutex** или **recursive_timed_mutex**.

Например:

```
mutex mtx;
timed_mutex mtx2;

void use()
{
    unique_lock<defer_lock_t,mutex> lck {mtx};
    unique_lock<defer_lock_t,timed_mutex> lck2 {mtx2};

    lck.try_lock_for(milliseconds{2});    // ошибка: mutex не имеет функции try_lock_for()
    lck2.try_lock_for(milliseconds{2});   // OK
```

```
    lck2.try_lock_until(steady_clock::now()+milliseconds{2});
    // ...
}
```

Если вы предоставляете **duration** или **time_point** в качестве второго аргумента для **unique_lock**, то конструктор выполнит соответствующую try-операцию. Операции **owns_lock()** позволяют проверить, успешна ли операция захвата. Например:

```
timed_mutex mtx2;

void use2()
{
    unique_lock<timed_mutex> lck2 {mtx2,milliseconds{2}};
    if (lck2.owns_lock()) {
        // захват успешен:
        // ... делаем что-то ...
    }
    else {
        // время вышло:
        // ... делаем что-нибудь еще ...
    }
}
```

42.3.2. Множественные блокировки

Нередко для решения задачи приходится захватывать несколько ресурсов. К сожалению, захват двух блокировок может приводить к «мертвой блокировке». Пример:

```
mutex mtx1; // защищает один ресурс
mutex mtx2; // защищает другой ресурс

void task(mutex& m1, mutex& m2)
{
    unique_lock<mutex> lck1 {m1};
    unique_lock<mutex> lck2 {m2};
    // ... используем ресурсы ...
}

thread t1 {task,ref(mtx1),ref(mtx2)};
thread t2 {task,ref(mtx2),ref(mtx1)};
```

Здесь **ref()** это связка ссылок **std::ref()** из файла **<functional>** (§33.5). Она требуется для передачи ссылки шаблону с переменным числом параметров (конструктору **thread**; §42.2.2). Мьютекс нельзя копировать или перемещать, так что я должен передавать его по ссылке (или использовать указатель).

Измените имена **mtx1** и **mtx2** на что-нибудь, что не указывало бы на порядок исполнения, и отделите определения **t1** и **t2** друг от друга в исходном тексте программы, после чего не будет так уж очевидно, что реален шанс на мертвую блокировку с **t1**, владеющим **mtx**, и с **t2**, владеющим **mtx2**: каждый из них пытается захватить второй мьютекс навечно.

Блокировки (§iso.30.4.2) **locks** – последовательность одного или более блокируемых объектов **lck1**, **lck2**, **lck3**, ...	
x=try_lock(locks)	Попробовать захватить все эл-ы **locks**; блокировки захватываются по порядку; **x=-1**, если все блокировки захвачены; иначе, **x=n**, где n – блокировка, к-рая не могла быть захвачена и все блокировки остаются незахваченными
lock(locks)	Захватить все эл-ы **locks**; не допускает «мертвых блокировок»

Действительный алгоритм работы **try_lock()** не специфицирован, но он вполне может быть таким:

```
template <typename M1, typename... Mx>
int try_lock(M1& mtx, Mx& tail...)
{
    if (mtx.try_lock()) {
        int n = try_lock(tail...);
        if (n == -1) return -1;    // все объекты заблокированы
        mtx.unlock();  // возвращаемся
        return n+1;
    }
    return 1; // не удалось заблокировать mtx
}
template <typename M1>
int try_lock(M1& mtx)
{
    return (mtx.try_lock()) ? -1 : 0;
}
```

С этим **lock()** ошибочный **task()** исправляется и упрощается:

```
void task(mutex& m1, mutex& m2)
{
    unique_lock lck1 {m1,defer_lock_t};
    unique_lock lck2 {m1,defer_lock_t};
    lock(lck1,lck2);
    // ... используем ресурсы ...
}
```

Отметим, что прямое применение **lock()** к мьютексам, **lock(m1,m2)**, а не **unique_lock**, оставляет программиста с обязательством явным образом освободить **m1** и **m2**.

42.3.3. call_once()

Часто мы хотим инициализировать объект без того, чтобы попасть в гонку за данными. Тип **once_flag** и функция **call_once()** предлагают эффективный низкоуровневый и простой инструмент:

call_once (§iso.30.4.2)	
once_flag fl {};	Конструктор по умолчанию: **fl** еще не вызвана
call_once(fl,f,args)	Вызвать **f(args)**, если **fl** еще не была использована

Например:

```
class X {
public:
    X();
    // ...
private:
    // ...
    static once_flag static_flag;
    static Y static_data_for_class_X;
    static void Init();
};

X::X()
{
    call_once(static_flag,Init());
}
```

Инициализация локальной статической переменной на этапе выполнения реализуется посредством **call_once()** или другим механизмом, аналогичным **call_once()**. Рассмотрим следующее:

```
Color& default_color()    // пользовательский код
{
    static Color def { read_from_environment("background color") };
    return def;
}
```

Это можно реализовать так:

```
Color& default_color()    // сгенерированный код
{
    static Color def;
    static_flag __def;
    call_once(__def,read_from_environment,"background color");
    return def;
}
```

Я использую префикс в виде двойного подчеркивания (§6.3.3) для того, чтобы подчеркнуть, что эта версия представляет код, сгенерированный компилятором.

42.3.4. Условные переменные

Условные переменные являются средством коммуникации между потоками. Поток **thread** может ожидать на **conditional_variable** возникновения некоторого события, например достижения временного интервала или завершения другого **thread**.

condition_variable (§iso.30.5) **lck** должен быть **unique_lock<mutex>**	
condition_variable cv {};	Конструктор по умолчанию: генерирует **system_error** если какой-то системный ресурс не может быть получен
cv.~condition_variable()	Деструктор: ни один **thread** не может быть в состоянии ожидания и не быть оповещенным
cv.notify_one()	Разблокировать один ожидающий **thread** (если есть); noexcept
cv.notify_all()	Разблокировать все ожидающие **threads**; noexcept
cv.wait(lck)	**lck** должен принадлежать вызывающему **thread**; атомарно вызывает **lck.unlock()** и блокирует; разблокирует по оповещении или «неожиданно»; по разблокировании вызывает **lck.lock()**
cv.wait(lck,pred)	**lck** должен принадлежать вызывающему **thread**; **while (!pred()) wait(lock);**
x=cv.wait_until(lck,tp)	**lck** должен принадлежать вызывающему **thread**; атомарно вызывает **lck.unlock()** и блокирует; разблокирует по оповещении или таймауту **tp**; по разблокировании вызывает **lck.lock()**; **x=timeout** если время вышло; иначе **x=no_timeout**
b=cv.wait_until(lck,tp,pred)	**while (!pred()) if (wait_until(lck,tp)==cv_status::timeout); b=pred()**
x=cv.wait_for(lck,d)	**x=cv.wait_until(lck,steady_clock::now()+d)**
b=cv.wait_for(lck,d,pred)	**b=cv.wait_until(lck,steady_clock::now()+d,move(pred))**
native_handle_type	См. §iso.30.2.3
nh=cv.native_handle()	**nh** – системный дескриптор для **cv**

Условные переменные могут (или не могут) полагаться на системные ресурсы, так что конструктор может не сработать из-за отсутствия ресурса. Однако, как и **mutex**, **conditional_variable** нельзя копировать и перемещать, так что лучше мыслить о таких переменных как о ресурсе как таковом, а не о дескрипторе ресурса.

Когда **conditional_vatiable** уничтожается, все ожидающие потоки (если они есть) должны уведомляться об этом (то есть их нужно пробудить), или же они будут ждать вечно.

Статус, возвращаемый **wait_until()** и **wait_for()** определяется следующим образом:

```
enum class cv_status { no_timeout, timeout };
```

Для **conditional_variable** **unique_lock** используется функциями **wait** для предотвращения потери пробуждений из-за состязания на списке ожидающих **thread** у **unique_lock**.

Простая **wait(lck)** — это низкоуровневая операция, которую следует использовать с особой осторожностью и, в основном, при реализации более высоких абстракций; в ней пробуждения могут возникать «неожиданно», так как система может решить возобновить поток у **wait()**, даже если никакой другой поток не затребовал это.

Допущение «неожиданных» пробуждений позволяет упростить реализацию условных переменных (**conditional_variable**) на некоторых системах. Всегда вызывайте **wait()** в цикле. Например:

```
while (queue.empty()) wait(queue_lck);
```

Этот цикл отражает суть ожидания с условием, так что всегда предпочитайте его безусловному **wait()**.

Поток может ожидать в течение заданного времени:

```
void simple_timer(int delay)
{
    condition_variable timer;
    mutex mtx;                                  // мьютекс, охраняющий timer
    auto t0 = steady_clock::now();
    unique_lock<mutex> lck(mtx);                // захватить mtx
    timer.wait_for(lck,milliseconds{delay});    // освободить и перезахватить mtx
    auto t1 = steady_clock::now();
    cout << duration_cast<milliseconds>(t1-t0).count() << "milliseconds passed\n";
} // неявно освобождает mtx
```

Можно сказать, что это реализация **this_thread::wait_for()**. Против гонки за данными **wait_for()** защищается мьютексом. Функция **wait_for()** освобождает **mutex** во время засыпания и снова захватывает его, когда ее **thread** разблокируется. Наконец, **lck** (неявно) освобождает **mutex** в конце своей области видимости.

Другой простой пример использования **conditional_variable** состоит в управлении потоком сообщений от источника к потребителю:

```
template<typename T>
class Sync_queue {
public:
    void put(const T& val);
    void put(T&& val);
    void get(T& val);
private:
    mutex mtx;
    condition_variable cond;
    list<T> q;
};
```

По идее, **put()** и **get()** не должны встать друг у друга на пути. Поток, выполняющий **get()**, будет спать до тех пор, пока для него не появится значение в очереди сообщений.

```
template<typename T>
void Sync_queue::put(const T& val)
{
    lock_guard<mutex> lck(mtx);
    q.push_back(val);
    cond.notify_one();
}
```

Здесь захватывается **mutex** очереди, добавляется значение в конец очереди, вызывается **notify_one()** для пробуждения возможно заблокированного потребителя, и неявно освобождается **mutex**. Я представил rvalue-версию **put()**, чтобы можно было передавать объекты типов, допускающих перемещение, а не копирование, вроде **unique_ptr** (§5.2.1, §34.3.1) и **packaged_task** (§42.4.3).

Функция **get()** несколько более сложная, потому что она должна блокировать свой **thread**, только в том случае, если **mutex** предотвращает доступ или если очередь пустая:

```
template<typename T>
void Sync_queue::get(T& val)
{
    unique_lock<mutex> lck(mtx);
    cond.wait(lck,[this]{ return !q.empty(); });
    val=q.front();
    q.pop_front();
}
```

Код, вызывающий **get()**, остается блокированным до тех пор, пока **Sync_queue** не перестанет быть пустым.

Я использовал **unique_lock**, а не простой **lock_guard**, потому что **lock_guard** сильно упрощен и не предоставляет операций, необходимых для разблокирования и повторного блокирования **mutex**.

Я использовал **[this]**, чтобы лямбды могли иметь доступ к объекту **Sync_queue** (§11.4.3.3).

Я выполнил возврат из **get()** посредством аргумента-ссылки, чтобы тип элемента с копирующим конструктором, генерирующим исключение, не вызвал проблем. Это стандартная техника (как например, STL адаптер **stack** предоставляет **pop()** и контейнеры предоставляют **front()**). Общий вариант **get()**, выполняющий стандартный возврат, в принципе возможен, но он не такой простой. За примером обратитесь к **future<T>::get()** (§42.4.4).

Пара источник-потребитель весьма проста:

```
Sync_queue<Message> mq;

void producer()
{
    while (true) {
        Message m;
        // ... наполняем m ...
        mq.put(m);
    }
}

void consumer()
{
    while (true) {
        Message m;
        mq.get(m);
        // ... используем m ...
    }
}
```

```
thread t1 {producer};
thread t2 {consumer};
```

Применение **conditional_variable** спасает потребителя от явной обработки случая выхода за пределы необходимой работы. Если бы мы просто использовали **mutex** для контроля доступа к **Sync_queue**, то пользователю пришлось бы то и дело осуществлять пробуждение, смотреть за работой на очереди и думать, что делать, когда очередь пустая.

Я копирую значения в и из списка **list**, который применяется для хранения элементов очереди. Копирование для типа элемента может привести к генерации исключения, но если это произойдет, то **Sync_queue** останется неизменным и функция **put()** или **get()** не увенчаются успехом.

Само по себе, **Sync_queue** не является разделяемой структурой данных, так что мы не используем для нее отдельный **mutex**; только **put()** и **get()** нуждаются в защите от гонки за данными.

В ряде случаев простой **Sync_queue** имеет фатальный недостаток: что если потребителю придется ждать вечно из-за того, что производитель (источник) перестанет добавлять значения? Что, если потребителю нужно выполнять другую работу, так что он не может ждать слишком долго? Для конкретных случаев имеются специфические ответы на эти вопросы, но в общем случае можно использовать таймаут для **get()**, то есть специфицировать максимальное время ожидания:

```
void consumer()
{
    while (true) {
        Message m;
        mq.get(m,milliseconds{200});
        // ... используем m ...
    }
}
```

Чтобы это сработало, мы должны добавить второй **get()** для **Sync_queue**:

```
template<typename T>
void Sync_queue::get(T& val, steady_clock::duration d)
{
    unique_lock<mutex> lck(mtx);
    bool not_empty = cond.wait_for(lck,d,[this]{ return !q.empty(); });
    if (not_empty) {
        val=q.front();
        q.pop_front();
    }
    else
        throw system_error{"Sync_queue: get() timeout"};
}
```

Используя таймаут, мы должны решить, что делать после окончания ожидания: мы получили данные или же вышло время? На самом деле, мы не беспокоимся о таймауте, а просто используем возврат **wait_for()** (true или false). Но я предпочитаю оповещать о неудаче функции **get()** и выходе по таймауту с помощью генерации исключения. Иначе нужно было бы возвращать тип **bool**.

Эквивалентная модификация функции **put()** будет ждать, пока потребитель не поработает с длинной очередью, но не будет ждать слишком долго:

```
template<typename T>
void Sync_queue::put(T val, steady_clock::duration d, int n)
{
    unique_lock<mutex> lck(mtx);
```

```
        bool not_full = cond.wait_for(lck,d,[this]{ return q.size()<n; });
        if (not_full) {
            q.push_back(val);
            cond.notify_one();
        }
        else {
            cond.notify_all();
            throw system_error{"Sync_queue: put() timeout"};
        }
    }
```

Альтернативный возврату **bool** вариант для **put()** состоит в стимулировании потребителя явно обрабатывать оба случая. Во избежание некоторых внутренних проблем с таким решением, я все же предпочел сигнализировать о неудаче с помощью генерации исключения.

Я выбрал **notify_all()**, если очередь полностью заполняется. Выбор между **notify_all()** и **notify_one()** зависит от характера приложения и не всегда очевиден. Оповещение одного лишь **thread** сериализует доступ к очереди и минимизирует пропускную способность в случае с несколькими потенциальными потребителями. С другой стороны, оповещение всех ожидающих потоков способно пробудить несколько **thread**, что вызовет соревнование на мьютексе и возможное пробуждение потоков только для того, чтобы выявить пустую очередь (опустошенную другими потоками). В общем случае лучше провести измерения.

42.3.4.1. condition_variable_any

condition_variable оптимизирован для **unique_lock<mutex>**. А **condition_variable_any**, хоть функционально и эквивалентен **condition_variable**, но может использовать любой блокируемый объект для своих операций:

condition_variable_any (§iso.30.5.2)
lck может быть любым блокируемым для требуемых операций
... аналогично **condition_variable** ...

42.4. Параллелизм на базе задач

До сих пор данная глава фокусировалась на низкоуровневых механизмах выполнения параллельных задач: упор был сделан на типе **thread**, на усилиях избежать гонки за данными и на синхронизации потоков. Для многих параллельных задач я считаю перечисленные усилия мешающими сосредоточиться на прикладной сути задач. Сейчас мы уточним понятие простого типа задачи: это задача, которая, получая аргументы, делает одну работу и производит один результат.

Для поддержки такой модели параллелизма на базе задач стандартная библиотека предлагает:

Поддержка задач (§iso.30.6.1)	
packaged_task<F>	Пакетировать объект-функцию **F** в качестве задачи
promise<T>	Тип объекта, в к-рый помещают один результат типа **T**
future<T>	Тип объекта, из которого берут один результат типа **T**
shared_future<T>	**future** future, из к-рого читают результат типа **T** несколько раз
x=async(policy,f,args)	Запустить **f(args)** на выполнение в соответствии с **policy**
x=async(f,args)	Запустить в соответствии с политикой по умолчанию: **x=async(launch::async|launch::deferred,f,args)**

Здесь демонстрируется множество деталей, которые редко когда касаются прикладного программиста. Следует фокусироваться на фундаментальной простоте модели на базе задач. Большинство усложняющих деталей нужны лишь для поддержки редких случаев, таких как полное сокрытие уровня потоков и блокировок.

Стандарная поддержка модели задач библиотекой — это лишь одна сторона того, что можно делать с моделью параллелизма на базе задач. Часто нам нужно реализовать некоторый набор небольших задач и позволить системе самой позаботиться о том, как это проецируется на возможности аппаратуры, и как избежать гонки за данными, как избежать «неожиданных» пробуждений, излишних ожиданий и т. д.

Главное достоинство этих средств — простота их использования прикладными программистами. В последовательной программе мы обычно пишем что-нибудь в таком роде:

```
res = task(args);    // выполнить задачу с данными аргументами и получить результат
```

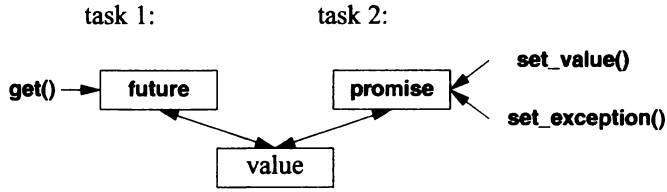
А вот параллельная версия:

```
auto handle = async(task,args);    // выполняем задачу с данными аргументами
// ... делаем что-нибудь еще ...
res = handle.get()                 // получаем результат
```

По умолчанию следует применять простейшие варианты и резервировать более сложные решения для тех случаев, когда вы абсолютно уверены в их необходимости.

42.4.1. future и promise

Как было сказано в §5.3.5, взаимодействие между задачами обрабатывается парой **future/promise**. Задача помещает свой результат в **promise**, а задача, нуждающаяся в этом результате, извлекает результат из соответствующего **future**:



На этой диаграмме *value* (*значение*) известно как разделенное состояние (§iso.30.6.4). В дополнение к возвращаемому значению или исключению, оно содержит информацию, необходимую двум потокам для безопасного обмена информацией. По минимуму, разделенное состояние должно быть способно содержать:

- *Значение* соответствующего типа или исключение. Для **future**, возвращающего **void**, значение это «ничто».
- *Бит готовности* для индикации того, что именно имеется для извлечения посредством **future** — значение или исключение.
- *Задача* для выполнения при вызове **get()** у **future** для работы, запущенной **async()** с политикой **deferred** (§42.4.6).
- *Счетчик использования* для того, чтобы состояние разделения могло уничтожаться тогда и только тогда, когда последний потенциальный пользователь откажется от доступа. В частности, если хранимое значение имеет тип класса с деструктором, то он вызывается при обнулении счетчика.
- *Взаимно исключающие данные*, позволяющие разблокировать любой **thread**, находящийся в состоянии ожидания (например, **condition_variable**).

Реализации могут выполнять следующие действия на разделенном состоянии:

- *Конструирование*: возможно с пользовательским аллокатором.
- *Привести в готовность*: установить бит готовности и разблокировать ожидающие потоки.
- *Освободить*: уменьшить счетчик использования и уничтожить разделенное состояние, если это был последний пользователь.
- *Покинуть*: если значение или исключение нельзя поместить в разделенное состояние посредством **promise** (например, **promise** уничтожено), то исключение **future_error** с условием ошибки **broken_promise** сохраняется в разделенном состоянии, и разделенное состояние переводится в состояние готовности.

42.4.2. promise

promise является дескриптором к разделенному состоянию (§42.4.1). Это место, где задача может разместить свой результат, подлежащий извлечению посредством **future** (§42.4.4).

promise<T> (§iso.30.6.5)	
promise pr {};	Конструктор по умолчанию: **pr** имеет разделенное состояние, которе еще не готово
promise pr {allocator_arg_t,a};	Создать **pr**; использует аллокатор **a**, чтобы создать неготовое разделенное состояние
promise pr {pr2};	Перемещающий конструктор: **pr** получает состояние **pr2**; **pr2** более не имеет разд. состояния; noexcept
pr.~promise()	Деструктор: покинуть разд. состояние; сохранить в нем исключение **broken_promise**
pr2=move(pr)	Перемещение: **pr2** получает состояние **pr**; **pr** более не имеет разд. состояния; noexcept
pr.swap(pr2)	Обменять значения **pr** и **pr2**; noexcept
fu=pr.g et_future()	**fu** есть **future**, соответствующее **pr**
pr.set_value(x)	Результат задачи есть значение **x**
pr.set_value()	Установить рез-т задачи для **future** типа **void**
pr.set_exception(p)	**p** есть **exception_ptr**, указывающий на результат задачи в виде исключения
pr.set_value_at_thread_exit(x)	Результат задачи есть значение **x**; результат не готов до выхода **thread**
pr.set_exception_at_thread_exit(p)	**p** есть **exception_ptr**, указывающий на результат задачи в виде исключения; результат не готов до выхода **thread**
swap(pr,pr2)	**pr.swap(pr2)**; noexcept

Для **promise** нет копирующих операций.

Функция **set** генерирует исключение **future_error**, если значение или исключение уже установлены.

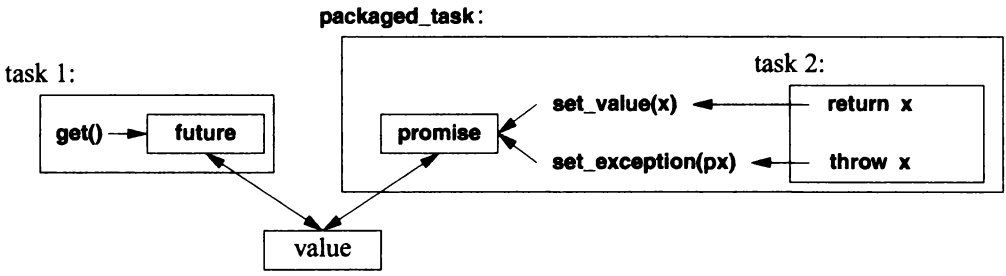
Возможна передача единственного значения результата посредством **promise**. Это может быть воспринято как ограничение, но нужно помнить, что значение перемещается в и из разделенного состояния, а не копируется, так что можно передать задешево целую коллекцию объектов. Например:

```
promise<map<string,int>> pr;
map<string,int>> m;
// ... заполняем m миллионом пар <string,int> ...
pr.set_value(m);
```

Задача может извлечь **map** из соответствующего **future** практически за нулевую плату.

42.4.3. packaged_task

packaged_task (пакетированная задача) содержит задачу и пару **future/promise**:



Мы передаем **packaged_task** задачу (функцию или функциональный объект) для исполнения. Когда наша задача исполнит **return x**, это вызовет **set_value()** на **packaged_task** у **promise**. Аналогично, **throw x** вызывает **set_exception(px)**, где **px** это исключение **exception_ptr** для **x**. В своей основе, **packaged_task** выполняет задачу, **f(args)**, примерно так:

```
try {
    pr.set_value(f(args)); // предполагаем, что promise назван pr
}
catch(...) {
    pr.set_exception(current_exception());
}
```

У **packaged_task** имеется традиционный набор операций:

packaged_task<R(ArgTypes...)> (§iso.30.6.9)	
packaged_task pt {};	Конструктор по умолчанию: **pt** не содержит задачи; noexcept
packaged_task pt {f};	Создать **pt**, содержащий **f**; **f** перемещается в **pt**; использовать аллокатор по умолчанию; explicit
packaged_task pt {allocator_arg_t,a,f};	Создать **pt**, содержащий **f**; **f** перемещается в **pt**; использовать аллокатор **a**; explicit
packaged_task pt {pt2};	Перемещающий конструктор: **pt** получает состояние **pt2**; **pt2** более не имеет задачи;
pt=move(pt2)	Перемещение: **pt** получает состояние **pt2**; уменьшить счетчик использования для предыд. разд. состояния **pt**; **pt2** более не имеет задачи; noexcept
pt.~packaged_task();	Деструктор: покинуть разд. состояние
pt.swap(pt2)	Обменять значения **pt** и **pt2**; noexcept
pt.valid()	Имеет ли **pt** разд. состояние? Имеет, если задача была дана и не перемещена куда-то еще; noexcept
fu=pt.get_future()	**fu** есть **future** для **promise** из **pt**; генерирует **future_error**, если вызвана дважды

Продолжение

packaged_task<R(ArgTypes...)> (§iso.30.6.9)	
pt()(args)	Выполнить **f(args)**; **return x** в **f()** вызывает **set_value(x)** у **promise** из **pt**, **throw x** в **f()** вызывает **set_exception(px)** у **promise** из **pt**; **px** есть **exception_ptr** для **x**
pt.make_ready_at_exit(args)	Вызвать **f(args)**; результат недоступен до завершения **thread**
pt.reset()	Вернуть в исходное состояние; покинуть старое состояние
swap(pt,pt2)	**pt.swap(pt2)**
uses_allocator<PT,A>	**true_type**, если **PT** использует тип аллокатора **A**

packaged_task можно перемещать, но не копировать. Сам **packaged_task** может копировать свою задачу, и предполагается, что скопированная задача приводит к такому же результату, что и исходная. Это важно, поскольку задача может быть перемещена со своим **packaged_task** на стек нового **thread**.

Покинуть разделенное состояние (посредством деструктора или перемещения) означает привести его в готовность. Если нет хранящегося значения или исключения, то хранится указатель на **future_error** (§42.4.1).

Преимущество **make_ready_at_exit()** состоит в том, что результат не доступен до тех пор, пока деструкторы для переменных **thread_local** не будут выполнены.

Отсутствует операция **get_promise()** в пару к **get_future()**. Использование **promise** контролируется исключительно через **packaged_task**.

В простом примере нет необходимости ни в каких **thread**. Сначала определяет простую задачу:

```
int ff(int i)
{
    if (i) return i;
    throw runtime_error("ff(0)");
}
```

Теперь упаковываем эту функцию в **packaged_task** и вызываем ее:

```
packaged_task<int(int)> pt1 {ff};    // храним ff в pt1
packaged_task<int(int)> pt2 {ff};    // храним ff в pt2

pt1(1);    // пусть pt1 вызовет ff(1);
pt2(0);    // пусть pt2 вызовет ff(0);
```

Кажется, что пока что ничего не произошло. В частности, мы не видим исключения, вызванного **ff(0)**. Фактически, **pt1(1)** выполнил **set_value(1)** на **promise**, прикрепленном к **pt1**, и **pt1(0)** выполнил **set_exception(px)** на **promise**, прикрепленном к **pt2**; **px** есть **exception_ptr** для **runtime_error("ff(0)")**.

Позднее мы можем попытаться извлечь результаты. Операция **get_future()** используется для того, чтобы добраться до **future**, в который пакетированный поток отложит результат своей задачи.

```
auto v1 = pt1.get_future();
auto v2 = pt2.get_future();

try {
    cout << v1.get() << '\n';  // будет вывод
    cout << v2.get() << '\n';  // будет исключение
}
```

```
catch (exception& e) {
    cout << "exception: " << e.what() << '\n';
}
```

Вывод программы будет такой:

```
1
exception: ff(0)
```

Можно было бы получить тот же самый эффект, просто написав

```
try {
    cout << ff(1) << '\n';      // будет вывод
    cout << ff(0) << '\n';      // будет исключение
}
catch (exception& e) {
    cout << "exception: " << e.what() << '\n';
}
```

Но суть в том, что версия с **packaged_task** работает в точности, как будто выполняется обычный вызов функции, даже если вызовы задачи (здесь **ff**) и вызовы функций **get()** находятся в разных **thread**. При этом можно концентрироваться на специфике прикладной задачи, а не на том, чтобы постоянно думать о потоках (**thread**) и блокировках.

Можно перемещать **future**, **packaged_task** или обоих. По ходу дела **packaged_task** активируется и его задача откладывает свой результат в **future**, и при этом не нужно знать, какой именно поток исполняет все это и какой поток примет результат. Это простое и общее решение.

Рассмотрим **thread**, обрабатывающий серию запросов. Это может быть GUI-поток, или поток, располагающий доступом к специализированной аппаратуре, или же сервер, сериализующий доступ к ресурсу через очередь. Можно реализовать это посредством очереди сообщений (§42.3.4), или же можно передать задачи, подлежащие выполнению:

```
using Res = /* тип результата для сервера */;
using Args = /* типы аргументов для сервера */;

using PTT = Res(Args);

Sync_queue<packaged_task<PTT>> server;

Res f(Args);                                    // функция: сделать что-то
struct G {
    Res operator()(Args);                       // объект-функция: сделать что-то
    // ...
};
auto h = [=](Args a) { /* сделать что-то */ };      // лямбда

packaged_task<PTT> job1(f);
packaged_task<PTT> job2(G{});
packaged_task<PTT> job3(h);

auto f1 = job1.get_future();
auto f2 = job2.get_future();
auto f3 = job3.get_future();

server.put(move(job1));
server.put(move(job2));
server.put(move(job3));

auto r1 = f1.get();
auto r2 = f2.get();
auto r3 = f3.get();
```

Серверный **thread** получит набор нескольких **packaged_task** от очереди на сервере и выполнит их в удобной последовательности. В типичном варианте, задачи переносят данные из вызывающего контекста.

Задачи пишутся по существу так же, как функции, функциональные объекты и лямбды. Сервер вызывает задачи как функции обратного вызова (callback functions). Серверу проще использовать **packaged_task**, чем обычные функции, поскольку об обработке их исключений уже позаботились.

42.4.4. future

future это дескриптор разделенного состояния (§42.4.1). Это то место, где задача может извлечь результат, отложенный **promise** (§42.4.2).

future<T> (§iso.30.6.6)	
future fu {};	Конструктор по умолчанию: нет разделенного состояния; noexcept
future fu {fu2};	Перемещающий конструктор: **fu** получает разд. состояние **fu2**, если есть; **fu2** более не имеет разделенного состояния; noexcept
fu.~future()	Деструктор: освобождает разделенное состояние, если есть
fu=move(fu2)	Перемещ. присваивание: **fu** получает разд. состояние **fu2**, если есть; **fu2** более не имеет разд. состояния; освобождает старое разд. состояние **fu**, если есть
sf=fu.share()	Переместить знач-е **fu** в **shared_future sf**; **fu** более не имеет разд. состояния
x=fu.get()	Значение **fu** перемещается в **x**; если в **fu** хранилось искл-ние, генерировать его; **fu** более не имеет разд. состояния; не вызывайте **get()** дважды
fu.get()	Для **future<void>**: как **x=fu.get()**, но не перемещает никакого значения
fu.valid()	Действителен ли **fu**? то есть, имеет ли **fu** разд. состояние? noexcept
fu.wait()	Блокировать, покуда не появится значение
fs=fu.wait_for(d)	Блокировать, покуда не появится значение или на **duration d**; **fs** равно **ready** при наличии значения, **timeout** при истечении таймаута или **deferred**, если исполнение отложено
fs=fu.wait_until(tp)	Блокировать, покуда не появится значение или до **time_point tp**; **fs** равно **ready** при наличии значения, **timeout** при истечении таймаута или **deferred**, если исполнение отложено

future содержит уникальное значение и не допускает копирующих операций.

Значение, если оно имеется, *перемещается* из **future**. Так что **get()** можно вызвать лишь единожды. Если вам нужно читать результат несколько раз (например, разными задачами), используйте **shared_future** (§42.4.5).

Если вы попытаетесь дважды применить **get()**, то вы столкнетесь с неопределеным поведением. Более того, нельзя сказать, что будет, если вы воспользуетесь любой операцией, кроме самого первого вызова **get()**, вызова **valid()** или активизации деструктора в случае не-**valid()**. Стандарт в таких случаях рекомендует реализациям генерировать исключение **future_error** с условием ошибки **future_errc::no_state**.

Если тип значения **future<T>**, **T**, есть **void** или ссылка, то к **get()** применяются специальные правила:

- **future<void>::get()** не возвращает значения: она просто осуществляет возврат управления или генерирует исключение.
- **future<T&>::get()** возвращает **T&**. Ссылка это не объект, так что библиотека должна передать что-то другое, вроде **T***, а **get()** конвертирует это обратно в **T&**.

Узнать статус **future** можно вызовом **wait_for()** и **wait_until()**:

enum class future_status	
ready	**future** имеет значение
timeout	Истек таймаут операции
deferred	Выполнение задачи **future** отложено до вызова **get()**

Вот возможные ошибки операций с **future**:

Ошибки future: future_errc	
broken_promise	**promise** покинул состояние до получения значения
future_already_retrieved	Второй вызов **get()** у **future**
promise_already_satisfied	Второй вызов **set_value()** или **set_exception()** у **promise()**
no_state	Операция обратилась к разд. состоянию **promise** до того, как оно было создано (напр., **get_future()** или **set_value()**)

Кроме того, операция **shared_future<T>::get()** над значением типа **T** может генерировать исключение.

Взглянув на таблицу для **future<T>**, я обнаружил, что я пропустил две полезные функции:

- **wait_for_all(args)**: ждать до тех пор, пока каждый **future** в **args** не получит значения.
- **wait_for_any(args)**: ждать до тех пор, пока один **future** в **args** не получит значения.

Я могу легко реализовать версию **wait_for_all()**:

```
template<typename T>
vector<T> wait_for_all(vector<future<T>>& vf)
{
    vector<T> res;
    for (auto& fu : vf)
        res.push_back(fu.get());
    return res;
}
```

Этот код достаточно легко использовать, но есть «заноза»: если я ожидаю десять **future**, то я рискую десять раз блокировать мой **thread**. В идеале нужно, чтобы **thread** блокировался и разблокировался один раз. Но в большинстве случаев представленная реализация **wait_for_all()** достаточно хороша: если некоторые **thread** выполняются долго, то лишние ожидания не будут критичными. С другой стороны, если все потоки выполняются быстро, то они вероятно завершатся после первого ожидания.

Реализация **wait_for_any()** менее простая. Сначала нам нужно найти способ проверки того, что готов ли **future**? К счастью, это делается при помощи **wait_for()**. Например:

```
future_status s = fu.wait_for(seconds{0});
```

Использование **wait_for(seconds{0})** скажет нам, почему произошло возобновление, и эта функция выполняет проверку готовности перед тем, как подвесить выполнение. Чаще всего (но это не гарантировано стандартом) **wait_for(seconds{0})** выполняет возврат немедленно, а не после попытки подвесить на нулевое время.

Отталкиваясь от **wait_for()**, можно написать:

```
template<typename T>
int wait_for_any(vector<future<T>>& vf, steady_clock::duration d)
    // вернуть индекс готового future
    // если ни один future не готов, ждать в течение d, потом попробовать снова
{
```

```
        while(true) {
            for (int i=0; i!=vf.size(); ++i) {
                if (!vf[i].valid()) continue;
                switch (vf[i].wait_for(seconds{0})) {
                case future_status::ready:
                    return i;
                case future_status::timeout:
                    break;
                case future_status::deferred:
                    throw runtime_error("wait_for_all(): deferred future");
                }
            }
            this_thread::sleep_for(d);
        }
    }
```

Я посчитал **deferred** (§42.4.6) ошибкой для моего случая.

Отметим проверку на **valid()**. Вызов **wait_for()** для недействительного **future** (например, если для **future** вы уже выполнили **get()**) приведет к трудно идентифицируемой ошибке. В лучшем случае вы можете надеяться на некоторое исключение.

Как и в реализации **wait_for_all()**, в данной реализации есть «заноза»: в идеале клиент вызова **wait_for_all()** не должен пробуждаться лишь для того, чтобы узнать, имеются ли выполненные задачи, и должен разблокироваться немедленно после выполнения одной задачи. Наша простая реализация только аппроксимирует такое поведение. Для больших **d** бесполезные пробуждения маловероятны, но могут без необходимости привести к долгому ожиданию.

Функции **wait_for_all()** и **wait_for_any()** являются полезными строительными блоками параллельных алгоритмов. Я использую их в §42.4.6.

42.4.5. shared_future

Результирующее значение **future** можно прочитать только один раз. Поэтому, если вы хотите повторно читать это значение многочисленными потребителями, то вы должны скопировать его, а затем читать эту копию. Это то, что делает **shared_future**. Любой полезный **shared_future** прямо или косвенно инициализируется перемещением значения из **future** с тем же самым типом возврата.

shared_future<T> (§iso.30.6.7)	
shared_future sf {};	Конструктор по умолчанию: нет разд. состояния;
shared_future sf {fu};	Конструктор: переместить значение из **future fu**; **fu** более не имеет состояния; noexcept
shared_future sf {sf2};	Копирующий и перемещающий конструкторы; перемещ. конструктор имеет спецификатор noexcept
sf.~future()	Деструктор: освободить разд. состояние, если имеется
sf=sf2	Копирование присваиванием
sf=move(sf2)	Перемещение присваиванием; noexcept
x=sf.get()	Значение **sf** копируется в **x**; если в **fu** хранилось исключение, генерировать его
sf.get()	Для **shared_future<void>**: подобно **x=sf.get()** но не копирует никакое значение
sf.valid()	Имеет ли **sf** разделенное состояние? noexcept
sf.wait()	Блокировать, покуда не появится значение

Продолжение

shared_future<T> (§iso.30.6.7)	
fs=sf.wait_for(d)	Блокировать, покуда не появится значение или на duration d; fs равно ready при наличии значения, timeout при истечении таймаута или deferred, если исполнение отложено
fs=sf.wait_until(tp)	Блокировать, покуда не появится значение или до time_point tp; fs равно ready при наличии значения, timeout при истечении таймаута или deferred, если исполнение отложено

Очевидно, что **shared_future** очень похож на **future**. Ключевое различие состоит в том, что **shared_future** перемещает свое значение в то место, где его можно повторно читать и разделять. Для **future<T>**, специальные правила применяются к **get()**, когда тип у **future<T>**, **T**, есть **void** или ссылка:

- **shared_future<void>::get()** не возвращает значения: она просто осуществляет возврат управления или генерирует исключение.
- **shared_future<T&>::get()** возвращает **T&**. Ссылка это не объект, так что библиотека должна передать что-то другое, вроде **T***, а **get()** конвертирует это обратно в **T&**.
- **shared_future<T>::get()** возвращает **const T&**, когда **T** не ссылка.

Если только возвращаемый объект не ссылка, то он **const**, так что к нему можно безопасно осуществлять доступ из разных **thread** без синхронизации. Если же возвращаемый объект есть не-**const**-ссылка, то вам придется использовать некую форму осуществления взаимно исключающего доступа во избежание гонки за данными на ссылочном объекте.

42.4.6. async()

Имея **future** и **promise** (§42.4.1) и **packaged_task** (§42.4.3), мы можем писать простые задачи, не беспокоясь слишком много о потоках (**thread**). В таком случае **thread** это нечто, чему вы даете задачу для выполнения. Но все равно нам нужно решить, сколько требуется потоков, и на каком **thread** лучше выполнить задачу — на текущем или на каком-то ином. Эти задачи можно делегировать «пускачу потоков» (thread launcher), то есть функции, которая решает, создать ли новый **thread**, или повторно использовать **thread** или просто выполнить задачу на текущем **thread**.

Асинхронный запуск задач: async<F,Args>() (§iso.30.6.8)	
fu=async(policy,f,args)	Исполнить f(args) в соответствии с политикой policy
fu=async(f,args)	fu=async(launch::async|launch::deferred,f,args)

Функция **async()** является простым интерфейсом к «пускачу потоков» неизвестной внутренней изощренности. Вызов **async()** возвращает **future<R>**, где **R** это тип результата задачи. Пример:

```
double square(int i) { return i*i; }

future<double> fd = async(square,2);
double d = fd.get();
```

Если **thread** запущен для выполнения **square(2)**, то мы имеем рекордно медленный способ вычисления выражения **2*2**. Нотация может быть упрощена применением **auto**:

```
double square(int i) { return i*i; }

auto fd = async(square,2);
auto d = fd.get();
```

В принципе, тот, кто вызывает **async()**, может предоставить массу информации в помощь реализации **async()**, чтобы решить, запускать ли новый **thread**, а не исполнять задачу на текущем **thread**. Например, программист мог бы дать «пускачу» некий намек на то, как долго будет выполняться задача. Однако ж, в текущем стандарте есть только две политики:

Политики запуска: launch	
async	Выполнить задачу, как будто новый **thread** был запущен для этого
deferred	Выполнить задачу при вызове **get()** у **future** задачи

Обратите внимание на это «*как если бы*» (*as if*). Пускач имеет все права (на его усмотрение) в момент, когда он принимает решение запускать новый **thread** или нет. К примеру, так как умолчательная политика есть **async|deferred** (**async** или **deferred**), то выполнение сводится к **fd.get()**, вызывающей **square(2)**. Я могу даже предположить, что оптимизатор может сократить весь этот фрагмент до

```
double d = 4;
```

Конечно, нельзя быть уверенным в том, что реализация **async()** будет оптимизирована до такого тривиального результата. Оптимизиторы полагаются на реалистичные примеры, в которых выполняется заметный объем вычислений, так что запуск нового потока или использование по-новой существующего может базироваться на изучении конкретных обстоятельств.

Под использованием существующего потока я подразумеваю поток из ранее созданного пула потоков, которые **async()** может однажды создать и потом использовать их снова и снова. В зависимости от системной реализации потоков такое решение может привести к значительному снижению стоимости запуска задачи в **thread**. Но нужно быть осторожным в том, чтобы повторно используемый поток не зависел от потенциально остаточных данных от старой задачи, выполнявшейся на данном **thread**, и что тот поток не сохранил указатели на свой стек или данные **thread_local** (§42.2.8) в нелокальном хранилище. Такие данные могут привести к нарушению безопасности.

В качестве реалистичного примера использования **async()** можно привести пример запуска задачи для сбора входных данных от пользователя:

```
void user()
{
    auto handle = async([](){ return input_interaction_manager(); });
    // ...
    auto input = handle.get();
    // ...
}
```

Такая задача часто требует некоторых данных от клиента (того, кто вызывает). Я использовал лямбду для того, чтобы сделать очевидным факт, что я могу передавать аргументы или разрешить доступ к локальным переменным. Когда используете лямбду для специцифирования задачи, не забудьте захватить локальные переменные по ссылке. Это может привести к гонке за данными или доступу к кэшу двумя **thread**, имеющими доступ к одному и тому же кадру стека. Также следует помнить, что захват членов объекта посредством **[this]** (§11.4.3.3) предполагает, что члены объекта достигаются косвенно (через **this**), а не копируются, так что объект подвержен гонке за данными, если только вы не уверены в обратном. В случае сомнений просто копируйте (передача или захват по значению, **[=]**).

Можно использовать политику **launch::deferred** для начальной отладки. Это устранит ошибки, связанные с параллелизмом до тех пор, пока я не исправлю обычные последовательные ошибки. И можно всегда вернуться к **launch::deferred** для того, чтобы увериться в том, что ошибки действительно связаны с параллелизмом.

Со временем могут появиться и другие политики запуска, и разные реализации могут предложить свои политики. В итоге, можно будет улучшать производительность программы простой заменой политики, а не изменять программу, перерабатывая тонкие детали логики ее исполнения. Это хорошее следствие фундаментальной простоты модели на базе задач (§42.4).

Выбор умолчательной политики из **launch::async|launch::deferred** может быть проблемой на практике. Реализация может решить, что «нет параллелизма» это хорошая идея, и всегда использовать **launch::deferred**. Если у вас программа подозрительно ведет себя так же, как в однопотоковом случае, то лучше явно задать политику запуска.

42.4.7. Пример параллельной find()

Функция **find()** выполняет линейный поиск в последовательности. Предположим, что у вас имеются миллионы записей, которые не так-то легко отсортировать, и **find()** становится правильным алгоритмом для поиска. Но это может быть очень медленным, так что вместо того, чтобы начинать поиск с самого начала и выполнять его до конца, можно запустить 100 функций **find()**, каждую на одной сотой части последовательности.

Представим данные как **vector** с элементами типа **Record**:

```
extern vector<Record> goods;      // данные, в которых ведется поиск
```

Индивидуальная (последовательная) задача — это просто использование алгоритма **find_if()** из стандартной библиотеки:

```
template<typename Pred>
Record* find_rec(vector<Record>& vr, int first, int last, Pred pr)
{
    vector<Record>::iterator p = std::find_if(vr.begin()+first,vr.begin()+last,pr);
    if (p == vr.begin()+last)
        return nullptr; // в конце: запись не найдена
    return &*p;         // найдена: возвращаем указатель на элемент
}
```

Еще нужно решить, какое будет «зерно гранулирования» параллелизма, то есть нужно задать число записей, которые подвергнутся последовательному поиску.

```
const int grain = 50000;      // число записей для линейного поиска
```

На самом деле, корректный выбор размера зерна зависит от массы факторов, как то: особенности аппаратуры, реализация библиотеки, особенности данных и алгоритма.

Функция **pfind()** («параллельный find») просто выполняет определенное число вызовов **async()**, вытекающее из размера **grain** и числа элементов типа **Record**. После чего получает результаты:

```
template<typename Pred>
Record* pfind(vector<Record>& vr, Pred pr)
{
    assert(vr.size()%grain==0);

    vector<future<Record*>> res;

    for (int i = 0; i!=vr.size(); i+=grain)
        res.push_back(async(find_rec<Pred>,ref(vr),i,i+grain,pr));
```

```
    for (int i = 0; i!=res.size(); ++i)     // ищем результат в векторе future
        if (auto p = res[i].get())          // задача нашла совпадение?
            return p;

    return nullptr; // совпадение не найдено
}
```

Наконец, можно запускать поиск:

```
void find_cheap_red()
{
    assert(goods.size()%grain==0);

    Record* p = pfind(goods,
            [](Record& r) { return r.price<200 && r.color==Color::red; });
    cout << "record "<< *p << '\n';
}
```

Первая версия параллельной **find()** сначала запускает массу задач, а потом начинает ждать их по-порядку. Как и **std::find_if()**, она выдает первый элемент, соответствующий предикату; то есть она находит подходящий элемент с наименьшим индексом. Это хорошо, но:

- Можно финишировать (закончить с ожиданием массы задач), не найдя ничего (или только последняя задача что-нибудь найдет).
- Можно выбросить массу информации, которая могла бы быть полезной (например, тысячу позиций, соответствующих образцу).

Первая проблема не так страшна, как кажется. Предположим, что запуск **thread** ничего не стоит и что мы располагаем столькими же процессорными единицами (процессорами или или ядрами), сколько запускается задач — в таком случае мы затратим время такое же, как и на обработку одной задачи. То есть мы затратим столько же времени, сколько нужно на изучение 50000 записей, а не миллионов. Если у нас есть **N** процессорных единиц, то мы будем получать стопки результатов для **N*50000** записей. Если мы не найдем ничего вплоть до последнего сегмента **vector**, то затраченное время будет примерно равным **vr.size()/(N*grain)**.

Вместо того, чтобы по-порядку ждать каждую задачу, можно было бы попытаться заглядывать в результаты в порядке завершения задач. То есть можно было бы использовать **wait_for_any()** (§42.4.4). Например:

```
template<typename Pred>
Record* pfind_any(vector<Record>& vr, Pred pr)
{
    vector<future<Record*>> res;

    for (int i = 0; i!=vr.size(); i+=grain)
        res.push_back(async(find_rec<Pred>,ref(vr),i,i+grain,pr));

    for (int count = res.size(); count; --count) {
        int i = wait_for_any(res,microseconds{10});   // ждем завершенную задачу
        if (auto p = res[i].get())                    // задача нашла совпадение?
            return p;
    }

    return nullptr; // совпадение не найдено
}
```

Вызов **get()** делает **future** недействительным, так что мы не сможем заглянуть в частичный результат дважды.

Я использовал **count()** для того, чтобы удостовериться в том, что работа не продолжается после того, как получены результаты от всех задач. Превосходит ли **pfind_any()** в производительности **pfind()**, зависит от многих факторов, но более всего от используемого алгоритма. Часто наилучшим параллельным вариантом алгоритма будет вариация последовательного алгоритма, а не тупое повторение обычного последовательного алгоритма.

Остается вопрос: «вам действительно достаточно одного совпадения?». При параллелизме больший смысл имеет нахождение всех совпадений. Это сделать легко. Все, что нужно, это позволить каждой задаче возвращать **vector** совпадений, а не просто совпадение:

```
template<typename Pred>
vector<Record*> find_all_rec(vector<Record>& vr, int first, int last, Pred pr)
{
    vector<Record*> res;
    for (int i=first; i!=last; ++i)
        if (pr(vr[i]))
            res.push_back(&vr[i]);
    return res;
}
```

Теперь нам нужно запустить **find_all_rec()** подходящее число раз и ждать результат:

```
template<typename Pred>
vector<Record*> pfind_all(vector<Record>& vr, Pred pr)
{
    vector<future<vector<Record*>>> res;

    for (int i = 0; i!=vr.size(); i+=grain)
        res.push_back(async(find_all_rec<Pred>,ref(vr),i,i+grain,pr));

    vector<vector<Record*>> r2 = wait_for_all(res);

    vector<Record*> r;
    for (auto& x : r2)   // объединяем результаты
        for (auto p : x)
            r.push_back(p);
    return r;
}
```

Если бы я вернул **vector<vector<Record*>>**, то этот **pfind_all()** была бы простейшей параллельной функцией. Однако, соединением векторов, возвращаемых как один, **pfind_all()** превращается в пример популярной группы параллельных алгоритмов:

1. Создаем некоторое число задач, подлежащих запуску.
2. Запускаем задачи параллельно.
3. Объединяем результаты.

Наш пример запускается следующим образом:

```
void find_all_cheap_red()
{
    assert(goods.size()%grain==0);

    auto vp = pfind_all(goods,
        [](Record& r) { return r.price<200 && r.color==Color::red; });
    for (auto p : vp)
        cout << "record "<< *p << '\n';
}
```

Наконец, нужно решить, стоят ли того усилия по параллелизации. Для оценки я добавил простую последовательную версию для тестирования:

```
void just_find_cheap_red()
{
    auto p = find_if(goods.begin(),goods.end(),
        [](Record& r) { return r.price<200 && r.color==Color::red; });
    if (p!=goods.end())
            cout << "record "<< *p << '\n';
    else
            cout << "not found\n";
}

void just_find_all_cheap_red()
{
    auto vp = find_all_rec(goods,0,goods.size(),
        [](Record& r) { return r.price<200 && r.color==Color::red; });
    for (auto p : vp)
        cout << "record "<< *p << '\n';
}
```

Для моих достаточно простых данных и моего простого ноутбука с четырьмя ядрами я не заметил существенной разницы. Это означает, что стоимость создания **thread** в моей несовершенной версии **async()** доминирует над эффектами от параллелизма. Для серьезной прибавки в эффективности я бы реализовал свой вариант **async()**, основанный на созданном заранее наборе **thread** и рабочей очереди вроде **Sync_queue** (§42.3.4) для **packaged_task** (§42.4.3). Заметьте, что такая важная оптимизация может быть выполнена без изменения моей параллельной функции **find()**, базирующейся на модели задач.

42.5. Советы

[1] **thread** это безопасный по типам интерфейс к системным потокам; §42.2.

[2] Не следует уничтожать работающий **thread**; §42.2.2..

[3] Используйте **join()**, ожидая завершения **thread**; §42.2.4.

[4] Для использования **thread** в парадигме «захват ресурса есть инициализация» применяйте **guarded_thread**; §42.2.4.

[5] Не следует вызывать **detach()** для **thread**, если только вы не вынуждены сделать это; §42.2.4.

[6] Для управления мьютексами используйте **lock_guard** или **unique_lock**; §42.3.1.4.

[7] Для запроса множественных блокировок используйте **lock()**; §42.3.2.

[8] Для взаимодейтсивя между разными **thread** используйте **conditional_variable**; §42.3.4.

[9] Мыслите в терминах задач, выполняемых параллельно, а не в понятиях **thread**; §42.4.

[10] Цените простоту; §42.4.

[11] Возвращайте результат посредством **promise** и получайте его из **future**; §42.4.1.

[12] Нельзя дважды вызывать **set_value()** или **set_exception()** для **promise**; §42.4.2.

[13] Для работы с исключениями, генерируемыми задачами, используйте **packaged_task**; §42.4.3.

[14] Используйте **packaged_task** и **future** для формирования запроса к внешнему сервису и ожидания результата от него; §42.4.3.

[15] Не пытайтесь дважды выполнять **get()** из **future**; §42.4.4.

[16] Для запуска простых задач используйтие **async()**; §42.4.6.

[17] Выбор хорошей грануляции для параллельных задач зависит от эксперимента и измерений; §42.4.7.

[18] Где только возможно, скрывайте параллелизм за интерфейсом к параллельному алгоритму; §42.4.7.

[19] Параллельный алгоритм может семантически отличаться от последовательной версии (например, **pfind_all()** вместо **find()**); §42.4.7.

[20] Иногда последовательное решение проще и быстрее параллельного решения; §42.4.7.

43

Стандартная библиотека C

C – это строго типизированный,
но слабо проверяемый язык
— Деннис Ритчи

- Введение
- Файлы
- Семейство функций **printf()**
- Строки в С-стиле
- Память
- Даты и время
- Прочие функции
- Советы

43.1. Введение

Стандартная библиотека языка С с минимальными модификациями инкорпорирована в стандартную библиотеку языка С++. Стандартная библиотека С предоставляет небольшой набор функций, доказавших свою полезность в течение десятилетий во многих контекстах — особенно в низкоуровневом программировании.

Стандартная библиотека языка С содержит больше функций, чем тот небольшой набор, которые мы здесь рассматриваем; можете ознакомиться с функциями стандартной библиотеки С по учебникам, например «Керниган и Ритчи» [Kernighan,1988], или по стандарту ISO C [C,2011].

43.2. Файлы

Система ввода/вывода представлена в файле **<stdio>** и базируется на *файлах*. Файл (**FILE***) может ссылаться на физический файл на диске компьютера или на один из стандартных потоков ввода и вывода: **stdin**, **stdout** и **stderr**. Стандартные потоки доступны по умолчанию; остальные файлы подлежат открытию:

Открытие и закрытие файлов	
f=fopen(s,m)	Открыть файловый поток для файла **s** в режиме **m f** есть **FILE*** для открытого файла, в случае успеха, или **nullptr**
x=fclose(f)	Закрыть файловый поток **f**; возвращает **0** в случае успеха

Файл, открытый функцией **fopen()**, должен быть закрыт функцией **fclose()**, или он будет оставаться открытым до тех пор, пока его не закроет операционная система. Если это проблема (вообще-то, это вариант утечки ресурса), используйте **fstream** (§38.2.1).

Режим (*mode*) это строка в C-стиле, содержащая один или более символов, специфицирующих детали открытия файла (и его использования после открытия):

Режимы файла	
"r"	Чтение
"w"	Запись (перетирает предыдущее содержание)
"a"	Добавление (в конец файла)
"r+"	Чтение и запись
"w+"	Чтение и запись (перетирает предыдущее содержание)
"b"	Бинарный; применяйте совместно с другими режимами

Для конкретных систем обычно имеются дополнительные режимы открытия. К примеру, **x** иногда означает, что «файл не должен существовать до этой операции». Некоторые опции можно комбинировать, например **fopen("foo","rb")** пытается открыть файл с именем **foo** для бинарного чтения. Режимы ввода/вывода должны быть одинаковыми для C-вариантов и **iostream** (§38.2.1).

43.3. Семейство функций printf()

Наиболее популярными функциями стандартной библиотеки C являются функции вывода. Я, правда, предпочитаю **iostream** за безопасность по типам и расширяемость. Форматируемая функция вывода **printf()** широко используется на практике (даже в программах на C++):

printf()	
n=printf(fmt,args)	Вывести форматированную строку **fmt** в **stdout**, подставляя аргументы **args** соответствующим образом
n=fprintf(f,fmt,args)	Вывести форматированную строку **fmt** в файл **f**, подставляя аргументы **args** соответствующим образом
n=sprintf(s,fmt,args)	Вывести форматированную строку **fmt** в строку **s** в C-стиле, подставляя аргументы **args** соответствующим образом

Для каждой версии **n** есть число выведенных символов или отрицательное число в случае краха попытки вывода. Часто, возвращаемое функцией **printf()** значение, просто игнорируется.

Вот объявление функции **printf()**:

```
int printf(const char* format ...);
```

Эта функция принимает C-строку, за которой следует произвольное число аргументов произвольных типов. Смысл этих дополнительных аргументов контролируется посредством форматирующих спецификаций, таких как **%c** (вывести как символ) и **%d** (вывести как десятичное целое), находящихся в форматирующей строке. Например:

```
int x = 5;
const char* p = "Pedersen";
printf("the value of x is '%d' and the value of s is '%s'\n",x,s);
```

Символ, расположенный сразу за %, управляет обработкой аргумента. Первый % относится к первому аргументу (здесь **%d** относится к **x**), второй % относится ко второму аргументу (здесь **%s** относится к **s**) и так далее. В частности, вот такой вывод соответствует представленному выше примеру вызова функции **printf()**:

```
the value of x is '5' and the value of s is 'Pedersen'
```

за которым выполняется перевод на новую строку.

В общем случае нет возможности проконтролировать соответствие между форматирующими символами и типами аргументов. Пример:

```
printf("the value of x is '%s' and the value of s is '%x'\n",x,s);  // oops
```

Набор спецификаций форматирующих символов весьма обширен (и рос с течением времени) и дает большую степень гибкости. Отдельные системы добавляют свои собственные опции, выходящие за рамки С-стандарта. См. также набор опций для форматирования **strftime()** (§43.6).

За символом % могут следовать:

- **–** опциональный знак минус, специфицирующий выравнивание влево форматируемого значения в рамках поля форматирования;
- **+** опциональный знак плюс, означающий, что значение знакового типа всегда начинается с + или –;
- **0** опциональный нуль, означающий, что лидирующие нули используются для выравнивания числового значения. Если имеются знак – или спецификации точности, то **0** игнорируется;
- **#** опциональный знак **#**, специфицирующий тот факт, что числа с плавающей запятой будут печататься с десятичной точкой, даже если за ней не следуют никакие ненулевые цифры, что будут печататься завершающие нули, что восьмеричные значения будут печататься с начальным **0**, и что шестнадцатеричные значения будут начинаться на **0x** или **0X**;
- *d* опциональная строка цифр, специфицирующая ширину поля вывода; если у выводимого значения меньше символов, чем ширина поля, то они будут дополняться пробелами слева (или справа при наличии индикатора для выравнивания влево) до ширины поля; если ширина поля начинается с нуля, то заполнение будет нулями, а не пробелами;
- **.** служит для отделения ширины поля от последующей строки цифр;
- *d* опциональная строка цифр, специфицирующая точность, что означает число цифр, появляющееся после десятичной точки, для **e-** и **f-** форматирования, или максимальное число символов печати;
- ***** ширина поля или точность вместо строки цифр. В этом случае целочисленный аргумент предоставляет ширину поля или точность;
- **h** опциональный символ, указывающий на то, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют (знаковому или беззнаковому) короткому целому аргументу;
- **hh** опциональная пара символов, указывающая на то, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют (знаковому или беззнаковому) аргументу типа **char**;
- **l** опциональный символ (эл), указывающий на то, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют (знаковому или беззнаковому) длинному целому аргументу;
- **ll** опциональная пара символов (эл эл), указывающая на то, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют (знаковому или беззнаковому) длинному длинному целому аргументу;

L опциональный символ, указывающий на то, что последующие **a**, **A**, **e**, **E**, **f**, **F**, **g** или **G** соответствуют аргументу типа **long double**;

j специфицирует, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют аргументам типа **intmax_t** или **uintmax_t**;

z специфицирует, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют аргументу типа **size_t**;

t специфицирует, что последующие **d**, **i**, **o**, **u**, **x** или **X** соответствуют аргументу типа **ptrdiff_t**;

% указывает на то, что символ **%** должен печататься; аргументы не используются;

c символ, индицирующий тип выполняемой конверсии. Символы конверсии (приведения) и их смысл таковы:

- **d** целый аргумент конвертируется в десятичную нотацию;
- **i** целый аргумент конвертируется в десятичную нотацию;
- **o** целый аргумент конвертируется в восьмеричную нотацию;
- **x** целый аргумент конвертируется в шестнадцатеричную нотацию;
- **X** целый аргумент конвертируется в шестнадцатеричную нотацию;
- **f** аргумент типа **float** или **double** конвертируется в десятичную нотацию в стиле [–]*ddd.ddd*. Число букв *d* после десятичной точки равно точности аргумента. При необходимости число округляется. Если точность отсутствует, то даются шесть цифр; если явно указана точность **0** и **#** не специфицирован, то десятичная точка не печатается;
- **F** То же, что и **%f**, но используются заглавные буквы для **INF**, **INFINITY** и **NAN**;
- **e** аргумент типа **float** или **double** конвертируется в десятичную нотацию в научном стиле [–]*d.ddd***e+***dd* или [–]*d.ddd***e–***dd*, где имеется одна цифра до десятичной точки и число букв d после десятичной точки равно спецификации точности аргумента. При необходимости число округляется. Если точность отсутствует, то даются шесть цифр; если явно указана точность **0** и **#** не специфицирован, то никаких цифр и десятичной точки не печатается;
- **E** то же, что и **e**, но на печати используется **E** для идентификации экспоненты;
- **g** аргументы типа **float** или **double** печатаются в стиле d, в стиле f, или в стиле e, а выбор осуществляется в пользу варианта, при котором достигается максимальная точность при минимальном размере вывода;
- **G** как и **g**, но на печати используется **E** для идентификации экспоненты;
- **a** аргумент типа **double** печатается в шестнадцатеричном формате [–]**0x***h.hhhh***p+**d или [–]**0x***h.hhhh***p–**d;
- **A** как **%a**, но используются **X** и **P** вместо **x** или **p**;
- **c** выводится символьный аргумент;
- **s** аргумент трактуется как строка (указатель на символы) и символы строки печатаются до тех пор, пока не встретится нулевой символ или же не будет исчерпано число символов, указуемое спецификацией точности; если точность есть **0** или отсутствует, то выводятся все символы до нулевого;
- **p** аргумент трактуется как указатель. Выводимое представление зависит от реализации;
- **u** беззнаковый целый аргумент конвертируется к десятичной нотации;
- **n** число символов, напечатанное до этого момента вызовами **printf()**, **fprintf()** или **sprintf()**, выводится в позицию **int**, указуемую указателем на **int** аргумент.

Ни в каких случаях несуществующая или маленькая ширина поля не приводят к усечению поля; заполнение имеет место только, если указанная ширина поля превышает действительную ширину.

Вот более сложный пример:

```
char* line_format = "#line %d \"%s\"\n";
int line = 13;
char* file_name = "C++/main.c";

printf("int a;\n");
printf(line_format,line,file_name);
```

А вот вывод этого примера:

```
int a;
#line 13 "C++/main.c"
```

Использование **printf()** небезопасно в том смысле, что проверка типа не выполняется. К примеру, покажем известный способ получить неопределенный вывод или ошибку сегментирования (или того хуже):

```
char x = 'q';
printf("bad input char: %s",x);      // %s должен был быть %c
```

В то же время, функция **printf()** предоставляет огромную гибкость в форме, которая хорошо известна С-программистам.

Поскольку в языке С нет пользовательских типов, то не существует способа определения форматов для таких типов, как **complex**, **vector** или **string**. Формат для **strftime()** (§43.6) показывает, в какие дебри вы попадаете, пытаясь изобрести другой набор спецификаторов формата.

Стандартный вывод **stdout** в языке С соответствует **cout**. Стандартный ввод **stdin** в языке С соответствует **cin**. Стандартный вывод ошибок **stderr** в языке С соответствует **cerr**. Эта аналогия столь велика, что ввод/вывод в стиле С и ввод/вывод потоков в С++ могут разделять одни и те же буферы. Например, смесь операций с **cout** и **stdout** может использоваться для достижения единого потока вывода (что нередко встречается в смешанном С и С++ коде). Такая гибкость не бесплатна. Ради большей производительности не следует смешивать эти операции в едином потоке вывода. Для гарантии, вызовите **ios_base::sync_with_stdio(false)** перед первыми операциями ввода/вывода (§38.4.4).

В стандартной библиотеке есть функция ввода **scanf()**, которая имитирует стиль **printf()**. Например:

```
int x;
char s[buf_size];
int i = scanf("the value of x is '%d' and the value of s is '%s'\n",&x,s);
```

Здесь **scanf()** пытается читать целое значение в переменную **x** и последовательность непробельных символов в **s**. Неформатирующий символ указывает на то, что ввод должен содержать этот символ. Например:

```
the value of x is '123' and the value of s is 'string '\n"
```

прочтет **123** в **x** и строку, за которой идет **0**, в **s**. Если вызов **scanf()** завершается успешно, то возврат (**i** в нашем примере) означает число аргументов-указателей, которым были присвоены значения (в нашем примере мы надеемся на число **2**); в противном случае это **EOF**. Указанный способ ввода чреват ошибками. Все аргументы функции **scanf()** должны быть указателями. Я настоятельно не рекомендую использовать **scanf()**.

Итак, что мы можем сделать, если обязаны использовать стандартный ввод/вывод языка C? Иногда популярным ответом на этот вопрос служит совет «используйте функцию **gets()**»:

```
// очень опасный код:
char s[buf_size];
char* p = gets(s);  // читаем строку в s
```

Вызов **p=gets(s)** читает символы в **s** до тех пор, пока не встретится символ перехода на новую строку или символ конца файла, а затем **'\0'** помещается сразу за последним прочитанным в **s** символом. В случае «конца файла» или возникшей ошибки в **p** устанавливается значение **nullptr**; в противном случае этот указатель устанавливается на **s**. Не стоит использовать **gets(s)** или его приблизительный эквивалент **scanf("%s",s)**. Эти средства много лет как используются разработчиками вирусов: предоставляя ввод, переполняющий входной буфер (**s** в нашем примере), мы допускаем порчу программы и возможное проникновение атакующего кода на ваш компьютер. Функция **sprintf()** страдает теми же самыми проблемами переполнения буфера. В стандарте C11 предлагается набор альтернативных функций ввода, принимающих дополнительный аргумент для защиты от переполнения, например, функция **gets_s(p,n)**. Как и в случае с потоковым вводом/выводом языка C++ пользователь сталкивается с проблемой: узнать в точности, что стало терминирующим условием (§38.4.1.2; слишком много символов, терминирующий символ или конец файла).

Библиотека ввода/вывода предоставляет простой и полезный способ читать и выводить (писать) символы:

Символьные функции Stdio	
x=getc(st)	Прочитать символ из потока ввода **st**; **x** есть целочисленное значение символа или **EOF**, если произошла ошибка или достигнут конец файла
x=putc(c,st)	Вывести символ **c** в поток вывода **st**; **x** есть целочисленное значение символа или **EOF**, если произошла ошибка
x=getchar()	**x=getc(stdin)**
x=putchar(c)	**x=putc(c,stdout)**
x=ungetc(c,st)	Поместить **c** обратно в поток ввода **st**; **x** есть целочисленное значение символа или **EOF**, если произошла ошибка

Результатом этих операций является **int** (не **char**, и не **EOF**). Например, вот типичный цикл ввода для языка C:

```
int ch;   // внимание: не "char ch;"
while ((ch=getchar())!=EOF) { /* делаем что-то */ }
```

Не выполняйте двух последовательных вызова **ungetc()** над потоком; результат не определен и не переносим.

Существует много других функций ввода/вывода; см. хорошую книгу по языку C (например, «K&R»).

43.4. Строки в C-стиле

Строки в C-стиле являются массивами **char** с терминирующим нулем. Такое понятие строк поддерживается набором функций из файлов **<cstring>** (или **<string.h>**, но не **<string>**) и **<cstdlib>**. Эти функциии оперируют над C-строками посредством указателей на **char** (**const char*** для памяти «только для чтения», но не указателями **unsigned char***):

Строковые операции в стиле C	
x=strlen(s)	Сосчитать символы (исключая терминирующий 0)
p=strcpy(s,s2)	Копировать s2 в s; [s:s+n) и [s2:s2+n) не могут перекрываться; p=s; терминирующий 0 копируется
p=strcat(s,s2)	Копировать s2 в конец s; p=s; терминирующий 0 копируется
x=strcmp(s, s2)	Сравнить лексикографически: если s<s2, то x отрицателен; если s==s2, то x==0; если s>s2, то x положителен
p=strncpy(s,s2,n)	strcpy максимум n символов; может не скопировать терминирущий 0
p=strncat(s,s2,n)	strcat максимум n символов; может не скопировать терминирущий 0
x=strncmp(s,s2,n)	strcmp максимум n символов
p=strchr(s,c)	p указывает на первый c в s
p=strrchr(s,c)	p указывает на последний c в s
p=strstr(s,s2)	p указывает на первый символ s, к-рый начинает подстроку, равную s2
p=strpbrk(s,s2)	p указывает на первый символ s, также находящийся в s2

Отметим, что в C++ функции **strchr()** и **strstr()** продублированы, чтобы сделать их безопасными по типам (например, они не могут конвертировать **const char*** в **char***, как могут их аналоги из языка C). См. также §36.3.2, §36.3.3 и §36.3.7.

Числовые преобразования в стиле C p указывает на первый символ в s, который не использ-ся в преобразовании; b есть основание числа [2:36] или 0, когда использ-ся числа в стиле C	
x=atof(s)	x есть **double**, представленный в s
x=atoi(s)	x есть **int**, представленный в s
x=atol(s)	x есть **long**, представленный в s
x=atoll(s)	x есть **long long**, представленный в s
x=strtod(s,p)	x есть **double**, представленный в s
x=strtof(s,p)	x есть **float**, представленный в s
x=strtold(s,p)	x есть **long double**, представленный в s;
x=strtol(s,p,b)	x есть **long**, представленный в s
x=strtoll(s,p,b)	x есть **long long**, представленный в s
x=strtoul(s,p,b)	x есть **unsigned long**, представленный в s
x=strtoull(s,p,b)	x есть **unsigned long long**, представленный в s

При конвертировании в тип с плавающей запятой **errno** устанавливается в **ERANGE** (§40.3), если результат не соответствует целевому типу. См. также §36.3.5.

43.5. Память

Функции манипуляции памятью работают над «сырой памятью» (безтиповой) через указатели типа **void*** (указатели **const void*** в случае памяти «только для чтения»):

Операции с памятью в стиле C	
q=memcpy(p,p2,n)	Копировать n байт из p2 в p (подобно **strcpy**); [p:p+n) и [p2:p2+n) не могут перекрываться; q=p
q=memmove(p,p2,n)	Копировать n байт из p2 в p; q=p
x=memcmp(p,p2,n)	Сравнить n байт из p2 с соответствующими n байт из p; x<0 значит <, x==0 значит ==, 0<x значит >

Продолжение

Операции с памятью в стиле C	
q=memchr(p,c,n)	Найти **c** (преобразованный в **unsigned char**) в [**p**:**p+n**); **q** указывает на тот эл-т; **q=0**, если **c** не найден
q=memset(p,c,n)	Копировать **c** (преобразованный в **unsigned char**) в каждый из [**p**:**p+n**); **q=p**
p=calloc(n,s)	**p** указывает на **n*s** байт из свободной памяти, инициализированных **0**; **p=nullptr**, если байты не были выделены
p=malloc(n)	**p** указывает на **n** неинициализированных байт из свободной памяти; **p=nullptr**, если **s** байт не могут быть выделены
q=realloc(p,n)	**q** указывает на **n** байт из свободной памяти; **p** должен быть указателем, возвращенным **malloc()** или **calloc()** или **nullptr**; если возможно, повторно применить память, на к-рую указывает **p**; если нет, копировать все байты из области, на к-рую указывает **p** в новую область; **q=nullptr**, если **s** байт не могут быть выделены
free(p)	Осовободить память, на которую указывает **p**; **p** должен быть **nullptr** или указателем, возвращенным **malloc()**, **calloc()** или **realloc()**

Заметьте, что **malloc()** и тому подобные функции не активируют конструкторы, а **free()** не активирует деструктор. Не используйте эти функции для типов с конструкторами или деструкторами.

Отметим, что **realloc(p,n)** будет релоцировать (копировать) хранящиеся данные, от указателя **p** и далее тогда, когда возникает нужда в дополнительной памяти, большей чем та, что указуется через **p**. Пример:

```
int max = 1024;
char* p = static_cast<char*>(malloc(max));
char* current_word = nullptr;
bool in_word = false;
int i=0;
while (cin.get(&p[i]) {
    if (isletter(p[i])) {
        if (!in_word)
            current_word = p;
        in_word = true;
    }
    else
        in_word = false;
    if (++i==max)
        p = static_cast<char*>(realloc(p,max*=2)); // двойное выделение
    // ...
}
```

Надеюсь, что вы зафиксировали грубую ошибку: если был вызван **realloc()**, то **current_word** может указывать на местоположение в памяти за пределами текущей области, указуемой через **p**.

В большинстве случаев вместо **realloc()** лучше использовать **vector** (§31.4.1).

Функции **mem*** находятся в файле **<cstring>**, а функции, аллоцирующие память — в файле **<cstdlib>**.

43.6. Даты и время

В файле **<ctime>** находятся некоторые типы и функции для работы с датами и временем:

Типы для времени и дат	
clock_t	Арифметический тип для хранения коротких интервалов времени (напр., несколько минут)
time_t	Арифметический тип для хранения длинных интервалов времени (напр., веков)
tm	**struct**, содержащий время даты (с 1900 года)

Структура **tm** определена следующим образом:

```
struct tm {
    int tm_sec;    // секунда [0:61]; 60 и 61 представляют високостные секунды
    int tm_min;    // минута [0:59]
    int tm_hour;   // час [0:23]
    int tm_mday;   // день месяца [1:31]
    int tm_mon;    // месяц [0:11]; 0 обозначает январь (внимание: не [1:12])
    int tm_year;   // год после 1900; 0 обозначает 1900, а 115 обозначает 2015
    int tm_wday;   // дней после воскресенья [0:6]; 0 обозначает воскресенье
    int tm_yday;   // дней после 1 января [0:365]; 0 обозначает 1 января
    int tm_isdst;  // часов сезонного времени
};
```

Системные часы поддерживаются функцией **clock()**, которая поддерживается несколькими функциями, придающими смысл вовзращаемому типу **clock_t**:

Функции для времени и даты	
t=clock()	**t** – количество «тиков» часов с начала программы; **t** есть **clock_t**
t=time(pt)	**t** – текущее календарное время; **pt** есть **time_t** или **nullptr**; **t** есть **clock_t**; если **pt!=nullptr**, ***pt=t**
d=difftime(t2,t1)	**d** есть **double**, представляющий **t2–t1** в секундах
ptm=localtime(pt)	Если **pt==nullptr**, **ptm=nullptr**; иначе **ptm** указывает на **time_t** с локальным временем для ***pt**
ptm=gmtime(pt)	Если **pt==nullptr**, **ptm=nullptr**; иначе **ptm** указывает на **time_t** с **Greenwich Mean Time (GMT)** **tm** для ***pt**
t=mktime(ptm)	**time_t** для ***ptm** или **time_t(–1)**
p=asctime(ptm)	**p** есть предвставление в виде строки в С-стиле ***ptm**
p=ctime(t)	**p=asctime(localtime(t))**
n=strftime(p,max,fmt,ptm)	Копировать ***ptm** в **[p:p+n+1)** согласно форматирующей строке **fmt**; символы после **[p:p+m)** игнорируются; **n==0** в случае ошибок; **p[n]=0**

Вот пример результата вызова **asctime()**:

```
"Sun Sep 16 01:03:52 1973\n"
```

А вот пример того, как может использоваться функция **clock()** для таймирования функции:

```
int main(int argc, char* argv[])
{
    int n = atoi(argv[1]);

    clock_t t1 = clock();
    if (t1 == clock_t(–1)) {   // значит "clock() не сработала"
        cerr << "sorry, no clock\n";
        exit(1);
    }
```

```
        for (int i = 0; i<n; i++)
            do_something();      // таймируемый цикл
        clock_t t2 = clock();
        if (t2 == clock_t(-1)) {
            cerr << "sorry, clock overflow\n";
            exit(2);
        }
        cout << "do_something() " << n << " times took "
            << double(t2-t1)/CLOCKS_PER_SEC << " seconds"
            << " (measurement granularity: " << CLOCKS_PER_SEC
            << " of a second)\n";
}
```

Явная конверсия **double(t2-t1)** перед делением необходима, поскольку **clock_t** может быть целым. Для величин **t1** и **t2**, возвращаемым функцией **clock()**, **double(t2-t1)/CLOCKS_PER_SEC** есть наилучшая системная аппроксимация времени в секундах между двумя вызовами.

Сравните **<ctime>** со средствами, предоставляемыми в **<chrono>**; см. §35.2.

Если **clock()** не предоставляется для вашего процессора или интервал времени слишком велик для измерения, то **clock()** возвращает **clock_t(-1)**.

Функция **strftime()** использует форматирующую строку **printf()** для управления выводом **tm**. Например:

```
void almost_C()
{
    const int max = 80;
    char str[max];
    time_t t = time(nullptr);
    tm* pt = localtime(&t);
    strftime(str,max,"%D, %H:%M (%I:%M%p)\n",pt);
    printf(str);
}
```

Вывод будет таков:

```
06/28/12, 15:38 (03:38PM)
```

Форматирующие символы функции **strftime()** образуют практически небольшой язык программирования:

Форматирование даты и времени	
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Представление даты и времени
%C	Год, разделенный на 100 и срезанный до десятичного целого [00:99]
%d	День месяца десятичным числом [01:31]
%D	Эквивалентно **%m/%d/%y**
%e	День месяца десятичным числом [1:31]; одиночная цифра предваряется пробелом
%F	Эквивалентно **%Y-%m-%d**; формат даты ISO 8601
%g	Последние два разряда года, содержащего данную неделю, десятичным числом [00:99]
%G	Год, содержащий данную неделю, десятичным числом (напр., 2012)
%h	Эквивалентно **%b**

Продолжение

Форматирование даты и времени	
%H	Час (24-х часовой формат) десятичным числом [00:23]
%I	Час (12-и часовой формат) десятичным числом [01:12]
%j	День года десятичным числом [001:366]
%m	Месяц десятичным числом [01:12]
%M	Минута десятичным числом [00:59]
%n	Символ перевода строки
%p	Локализованный вариант AM/PM для 12-и часового формата
%r	Время в 12-и часовом формате
%R	Эквивалентно **%H:%M**
%S	Секунда десятичным числом [00:60]
%t	Символ горизонтальной табуляции
%T	Эквивалентно **%H:%M:%S**; формат времени ISO 8601
%u	День недели ISO 8601 десятичным числом [1:7]; 1 – понедельник
%U	Номер недели в году (первое воскресенье – первый день недели 1) десятичным числом [00:53]
%V	Номер недели в году ISO 8601 десятичным числом [01:53]
%w	День недели десятичным числом [0:6]; 0 – воскресенье
%W	Номер недели в году (первый понедельник – первый день недели 1) десятичным числом [00:53]
%x	Локализованное представление даты
%X	Локализованное представление времени
%y	Последние 2 цифры года десятичным числом [00:99]
%Y	Год десятичным числом (напр., 2012)
%z	Смещение по UTC в формате ISO 8601 **–0430** (на 4.5 часа позже UTC, Гринвича); пустая строка, если часовой пояс не определен
%Z	Локализованное название или сокращение часового пояса; ничего, если часовой пояс неизвестен
%%	Символ %

Здесь схема локализации — это глобальная локализация программы.

Некоторые спецификаторы конвертирования могут модифицироваться посредством **E** или **O** модификаторов, указывающих на детали, зависящие от реализации и специфики локализации. Например:

Примеры модификаторов формата даты и времени	
%Ec	Дата и время в альтернативном локализованном представлении
%EC	Название эры, применяемое в альтернативном локализованном представлении
%OH	Час (24-х часовой формат) альтернативными символами цифр
%Oy	Последние два разряда года альтернативными символами цифр

Функция **strftime()** используется фасетом **put_time** (§39.4.4.1).

По поводу средств работы со временем в стиле языка C++ см. §35.2.

43.7. Прочие функции

В файле **<cstdlib>** находим:

Прочие <stdlib.h> функции	
abort()	Завершить программу "аномально"
exit(n)	Завершить программу со значением **n**; **n==0** обозначает успешное завершение
system(s)	Исполнить строку, как команду (системно-зависимо)
qsort(b,n,s,cmp)	Отсортировать массив, начиная с **b** с **n** элементов, размера **s**, используя функцию сравнения **cmp**
bsearch(k,b,n,s,cmp)	Искать **k** в сортированном масиве, начиная с **b**, с **n** эл-тов размера **s**, используя ф-ию сравнения **cmp**
d=rand()	**d** – псевдослучайное число в диапазоне **[0:RAND_MAX]**
srand(d)	Создать последовательность псевдослучайных чисел с **d** в роли зерна

Функция сравнения (**cmp**), используемая **qsort()** и **bsort()**, должна иметь тип

```
int (*cmp)(const void* p, const void* q);
```

Функции сортировки «видят» свои аргументы просто как последовательности байт. Возвращаемое целое таково:

- Отрицательное значение, если ***p** меньше, чем ***q**
- Нуль, если ***p** является равным ***q**
- Положительное значение, если ***p** больше, чем ***q**

В этом отличе от **sort()**, использующей традиционную операцию сравнения **<**.

Заметьте, что **exit()** и **abort()** не возбуждают исключений. Если вам требуется вызов деструкторов для ранее сконструированных объектов, генерируйте исключение (§13.5.1).

Аналогично, функция **longjmp()** из **<csetjmp>** — это нелокальный **goto**, который раскручивает стек до тех пор, пока не найдет результат, соответствующий **setjmp()**. При этом деструкторы не вызываются. Поведение этой функции не определено в случае, если деструктор будет вызван посредством **throw** из той же точки программы. Никогда не используйте **setjmp()** в C++ программе.

По поводу остальных функций стандартной библиотеки языка C смотри [Kernighan, 1988].

В файле **<cstdint>** находятся **int_fast16_t** и другие стандартные синонимы целых:

Псевдонимы целых типов **N** может быть **8**, **16**, **32** или **64**	
int_N_t	Целый тип из N байт, напр., **int_8_t**
uint_N_t	Беззнаковый целый тип из N байт, напр., **uint_16_t**
int_leastN_t	Целый тип из как минимум N байт, напр., **int_least16_t**
uint_leastN_t	Беззнаковый целый тип из как минимум N байт, напр., **uint_least32_t**
int_fastN_t	Целый тип из как минимум N байт, напр., **int_fast32_t**
uint_leastN_t	Беззнаковый целый тип из как минимум N байт, напр., **uint_fast64_t**

Также в файле **<cstdint>** находятся синонимы для наибольшего знакового и беззнакового типа целых. Пример:

```
typedef long long intmax_t;  // largest signed integer type
typedef unsigned long long uintmax_t; // largest unsigned integer type
```

43.8. Советы

[1] Если вы беспокоитесь насчет утечек ресурсов, используйте **fstream** вместо **fopen()**/**fclose()**; §43.2.

[2] По причинам безопасности типов и расширяемости отдавайте предпочтение **<iostream>** перед **<stdlib>**; §43.3.

[3] Никогда не используйте **gets()** или **scanf("%s",s)**; §43.3.

[4] Используйте **<string>**, а не **<cstring>**, из-за простоты в управлении ресурсом; §43.4.

[5] Применяйте средства управления памятью языка C, вроде **memcpy()**, только для работы с «сырой» памятью; §43.5.

[6] Вместо **malloc()** и **realloc()** используйте **vector**; §43.5.

[7] Не забывайте, что стандартной библиотеке языка C ничего не известно о конструкторах и деструкторах; §43.5.

[8] При таймировании отдавайте предпочтение **<chrono>**, а не **<ctime>**; §43.6.

[9] По причинам простоты и производительности отдавайте предпочтение **sort()** перед **qsort()**; §43.7.

[10] Не используйте **exit()**; вместо этого генерируйте исключение; §43.7.

[11] Не используйте **longjmp()**; вместо этого генерируйте исключение; §43.7.

44

Совместимость

Идите вперед, следуя своим привычкам,
а я буду следовать своим.
— С. Напьер

- Введение
- Расширения C++11

  Средства языка; Компоненты стандартной библиотеки; Устаревшие средства; Сравнение с ранними реализациями C++
- Совместимость C/C++

  C и C++ братья; «Тихие» отличия; C код, который не есть C++ код; C++ код, который не есть C код
- Советы

44.1. Введение

Настоящая глава посвящена рассмотрению отличий между стандартом C++ (ISO/ISEC 14882-2011) и более ранними версиями (ISO/ISEC 14882-1998), стандартом C (ISO/IEC 9899-2011) и более ранними версиями (Classic C). При этом мы рассмотрим:

- Точный список новаций в C++11
- Различия, которые могут вызывать проблемы у программистов
- Способы преодоления проблем

Большинство проблем совместимости возникают тогда, когда люди пытаются апгрейдировать C программы в их C++ аналоги, стараются портировать C++ программы более ранних версий C++ на более новые (например, с C++98 на C++11) или при попытке скомпилировать C++ программу с новыми средствами посредством старого компилятора. Мы не будем здесь стремиться перечислить абсолютно все проблемы совместимости, а остановимся лишь на наиболее типичных, которые чаще всего встречаются на практике, а также представим стандартные решения для таких случаев.

Начнем с того, что остановимся на вопросе о наборе реализаций C++, для которых программы должны работать. Для изучения C++ разумно ограничиться лишь самыми полными и самыми полезными реализациями. Если же речь идет о разработке продукта и последующем его внедрении на рынок, то разумно следовать консервативной стратегии по максимизации систем, на которых продукт должен работать без проблем. Раньше это было причиной для того, чтобы избегать кажущихся новаторскими средств языка C++. Теперь же, по мере совершенствования и сближения между собой разных реализаций C++, нужда в переносимости менее влияет на крайнюю осторожность в использовании новых черт и средств C++.

44.2. Расширения C++11

Сначала я перечислю средства языка и компоненты стандартной библиотеки, которые были добавлены в C++ в рамках нового стандарта C++11. Затем я обсужу вопрос о том, что делать со старыми версиями (в основном, с C++98).

44.2.1. Средства языка

Может показаться, что список средств языка просто огромный. Но не надо забывать, что средства языка не используются изолированно. В частности, большинство этих средств, новых в C++11, не имеют смысла вне рамок, сформированных более старыми версиями. Порядок перечисления приблизительно соответствует порядку появления информации о средстве в данной книге:

1. Однородная и общая инициализация, использующая {}-списки (§2.2.2, §6.3.5)
2. Вывод типа из инициализатора: **auto** (§2.2.2, §6.3.6.1)
3. Предотвращение сужения (§2.2.2, §6.3.5)
4. Обобщенные и гарантированные константные выражения: **constexpr** (§2.2.3, §10.4, §12.1.6)
5. Оператор range-**for** (§2.2.5, §9.5.1)
6. Ключевое слово для нулевого указателя: **nullptr** (§2.2.5, §7.2.2)
7. Сильно типизированные в рамках области видимости перечисления: **enum class** (§2.2.3, §8.4.1)
8. Утверждения времени компиляции: **static_assert** (§2.4.3.3, §24.4)
9. Соответствие между {}-списками и **std::initializer_list** (§3.2.1.3, §17.3.4)
10. rvalue-ссылки (допускающие перемещающую семантику; §3.3.2, §7.7.2)
11. Вложенные аргументы шаблона с окончанием на **>>** (нет пробелами между знаками **>**; §3.4.1)
12. Лямбды (§3.4.3, §11.4)
13. Шаблоны с переменным числом параметров (§3.4.4, §28.6)
14. Синонимы типов и шаблонов (§3.4.5, §6.5, §23.6)
15. Символы Unicode (§6.2.3.2, §7.3.2.2)
16. Целый тип **long long** (§6.2.4)
17. Управление выравниванием: **alignas** и **alignof** (§6.2.9)
18. Возможность использовать тип выражения в качестве типа в объявлении: **decltype** (§6.3.6.1)
19. «Сырые» строковые литералы (§7.3.2.1)
20. Обобщенные POD (§8.2.6)
21. Обобщенный **union** (§8.3.1)
22. Локальные классы в качестве аргументов шаблонов (§11.4.2, §25.2.1)
23. Синтаксис суффиксов типа (§12.1.4)
24. Синтаксис атрибутов и два стандартных атрибута: **[[carries_dependency]]** (§41.3) и **[[noreturn]]** (§12.1.7)
25. Прекращение распространения исключений: спецификатор **noexcept** (§13.5.1.1)
26. Тестирование возможности исключений в выражениях: операция **noexcept** (§13.5.1.1)
27. Средства C99: расширенные интегральные типы, конкатенация узких/широких строк; **_func_** и **_STDC_HOSTED_** (§12.6.2); **_Pragma(X)** (§12.6.3); макросы vararg и пустые аргументы макроса (§12.6)

28. **inline** пространства имен (§14.4.6)
29. Делегирующие конструкторы (§17.4.3)
30. Внутриклассовые инициализаторы членов (§17.4.4)
31. Управление умолчаниями: **default** (§17.6) и **delete** (§17.6.4)
32. Явные операции конвертирования (§18.4.2)
33. Пользовательские литералы (§19.2.6)
34. Более явный контроль за конкретизацией шаблонов: **extern template** (§26.2.2)
35. Умолчательные аргументы шаблонов для шаблонов функций (§25.2.5.1)
36. Наследуемые конструкторы (§20.3.5.1)
37. Управление замещением: **override** и **final** (§20.3.4)
38. Правила перегрузки шаблонных функций (§23.5.3.2)
39. Модель памяти (§41.2)
40. Локальная память потоков: **thread_local** (§42.2.8)

Я не старался перечислить буквально каждое мелкое отличие от C++98. Историческая перспектива на этот вопрос рассмотрена в §1.4.

44.2.2. Компоненты стандартной библиотеки

В рамках стандарта C++11 новые компоненты стандартной библиотеки имеют две разновидности: действительно новые компоненты (как библиотека регулярных выражений), и улучшения компонентов библиотеки из C++98 (например, перемещающие конструкторы для контейнеров).

1. **initializer_list** конструкторы для контейнеров (§3.2.1.3, §17.3.4, §31.3.2)
2. Перемещающая семантика для контейнеров (§3.3.1, §17.5.2, §31.3.2)
3. Односвязанный список: **forward_list** (§4.4.55, §31.4.2)
4. Хэшированные контейнеры: **unordered_map**, **unordered_multimap**, **unordered_set** и **unordered_multiset** (§4.4.5, §31.4.3)
5. Указатели для управления ресурсами: **unique_ptr**, **shared_ptr** и **weak_ptr** (§5.2.1, §34.3)
6. Поддержка параллелизма: **thread** (§5.3.1, §42.2), мьютексы (§5.3.4, §42.3.1), блокировки (§5.3.4, §42.3.2) и условные переменные (§5.3.4.1, §42.3.4)
7. Высокоуровневая поддержка параллелизма: **packaged_task**, **future**, **promise** и **async()** (§5.3.5, §42.4)
8. **tuple** (§5.4.3, §28.5, §34.2.4.2)
9. Регулярные выражения: **regex** (§5.5, Глава 37)
10. Случайные числа: **uniform_int_distribution**, **normal_distribution**, **random_engine** и т. д. (§5.6.3, §40.7)
11. Имена целых типов, вроде **int16_t**, **uint32_t** и **int_fast64_t** (§6.2.8, §43.7)
12. Контейнер смежных последовательностей фиксированного размера: **array** (§8.2.4, §34.2.1)
13. Копирование исключений и повторная генерация (§30.4.1.2)
14. Сообщение об ошибках посредством кодов ошибок: **system_error** (§30.4.3)
15. Операции **emplace()** для контейнеров (§31.3.6)
16. Широкое использование **constexpr**-функций
17. Систематическое использование **noexcept**-функций
18. Улучшенные адаптеры функций: **function** и **bind()** (§33.5)

19. Конверсия **string** в числовые значения (§36.3.5)
20. Аллокаторы в области видимости (§34.4.4)
21. Характеристики типов, такие как **is_integral** и **is_base_of** (§35.4)
22. Средства работы со временем: **duration** и **time_point** (§35.2)
23. Рациональная арифметика времени компиляции: **ratio** (§35.3)
24. Завершение программы: **quick_exit** (§15.4.3)
25. Дополнительные алгоритмы, такие как **move()**, **copy_if()** и **is_sorted()** (Глава 32)
26. Сборка мусора (§34.5)
27. Поддержка низкоуровневого параллелизма: **atomic** (§41.3).

Более подробные сведения о стандартной библиотеке располагаются в

- Главе 4, Главе 5 и Части IV.
- Примеры техники реализации: **vector** (§13.6), **string** (§19.3) и **tuple** (§28.5).
- Специализированная литература по C++11 вроде [Williams,2012].
- Краткий исторический обзор в §1.4.

44.2.3. Устаревшие средства

Помечая средства языка как устаревшие, комитет по стандартам выражает желание, чтобы это средство когда-либо было исключено (§iso.D). Но у комитета нет мандата немедленно исключить средство, которое широко до этого использовалось на практике, независимо от его избыточности и потенциальной опасности. Так что все сводится к намеку на то, что лучше это средство больше не применять. От компиляторов ожидается, что они будут выдавать предупреждения в таких случаях.

- Генерация копирующих конструкторов и копирующих присваиваний для классов с деструктором.
- Присваивание строкового литерала в **char*** (§7.3.2)
- Спецификации исключений из C++98:

```
void f() throw(X,Y); // C++98; now deprecated
```

  Средства поддержки спецификации исключений, **unexcepted_handler**, **set_unexcepted()**, **get_unexcepted()** и **unexcepted()**. Вместо них используется **noexcept**(§13.5.1.1).
- Некоторые функциональные объекты стандартной библиотеки и ассоциированные с ними функции: **unary_function**, **binary_function**, **pointer_to_unary_function**, **pointer_to_binary_function**, **ptr_fun()**, **mem_fun()**, **mem_fun_t**, **mem_fun1_t**, **mem_fun_ref_t**, **mem_fun_ref1_t**, **mem_fun()**, **const_mem_fun_t**, **const_mem_fun1_t**, **const_mem_fun_ref_t**, **const_mem_fun_ref1_t**, **binder1st**, **bind1st()**, **binder2nd**, **bind2nd()**. Вместо них используются **function** и **bind()** (§33.5).
- **auto_ptr**. Вместо него используется **unique_ptr** (§5.2.1, §434.3.1).

Кроме того, комитет исключил по существу неиспользуемое средство **export**, потому что оно было довольно сложным и разноречивым в разных реализациях основных поставщиков.

Приведения в С-стиле трактуются как устаревшие после того, как были введены именованные преобразования (§11.5.2). Программистам настоятельно рекомендуется не применять в своих программах приведения в С-стиле. Там, где нужны явные приведения типов, используйте **static_cast**, **reinterpret_cast**, **const_cast**; они точнее отражают семантику и более заметны в тексте программы.

44.2.4. Сравнение с ранними реализациями C++

Язык C++ постоянно используется, начиная с 1983 года (§1.4). С тех пор вышло несколько версий, для которых были выпущены разные независимые реализации. Фундаментальной целью усилий по стандартизации было желание сделать так, чтобы и пользователи, и реализаторы имели единое определение C++. С 1998 года программисты опирались на стандарт ISO C++98, а теперь мы имеем стандарт ISO C++11.

К сожалению, нередки случаи, когда программисты в своих первоначальных контактах с языком C++ используют старые реализации. Например потому, что такие реализации широко распространены. Конечно, при осознанном выборе никакой уважающий себя профессионал не будет связываться с таким антикварианом, тем более, что многие современные реализации доступны свободно и бесплатно. А для новичка старые реализации в любом случае будут иметь свою скрытую цену (и немалую): отсутствие многих черт современного C++ и многих компонентов стандартной библиотеки потребуют от программиста дополнительных усилий, которые в современных реализациях просто не нужны, так как многие проблемы уже решены. Применение старых реализаций может сформировать у новичка искаженное представление о языке C++ и неправильный стиль программирования. Если нужно, чтобы новичку были созданы тепличные условия, то я бы ему порекомендовал плотнее налечь на стандартную библиотеку, что заставит его в том числе использовать классы, шаблоны и исключения.

Все еще существуют отдельные области, для которых применение языка C попрежнему актуально. Если вам требуется использовать C, то в таком случае будет лучше всего опереться на общее подмножество языков C и C++; при этом вы получите некоторую степень поддержки в безопасности типов, переносимости и в получении опыта, который вам пригодится в будущем, когда будет возможность перейти на C++. См. §1.3.3.

Всюду, где только возможно, старайтесь соответствовать стандарту, минимизируя опору на средства, зависящие от реализации или просто нестандартные. Проектируйте в рамках существующего стандарта и только в тех случаях, когда отдельные стандартные средства недоступны, вырабатывайте обходные пути. В любом случае, такой подход приведет к программам, которые будет легче поддерживать со временем, чем варианты, опирающиеся на минимальное подмножество C++. Расширения языка, предоставляемые отдельными реализациями, также должны быть минимизированы в ваших программах. См. §1.3.2.

44.3. Совместимость C/C++

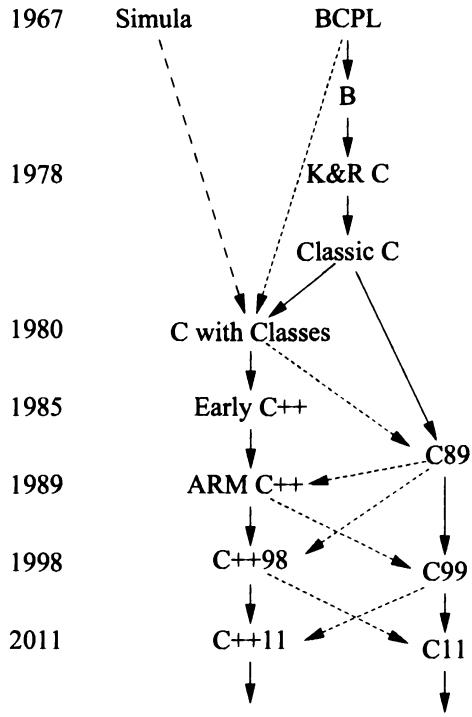
С минимальными исключениями C++ является надмножеством языка C (в смысле C11, определенным стандартом ISO/IEC 9899:2011(E)). Большая часть различий проистекает из-за большей нацеленности C++ на проверку типов. Грамотно написанные C программы имеют тенденцию быть одновременно и C++ программами. Компилятор в состоянии диагностировать каждое различие в C и C++. Несовместимости в паре C99/C++11 перечислены в §iso.C. До последнего времени C11 был внове для программистов, и большая част кода на C соответствовала классическому C или C99.

44.3.1. C и C++ братья

Классический C имеет двух основных наследников: ISO C и ISO C++. С годами эти языки эволюционировали в разных направлениях и с разными скоростями. Это привело к тому, что каждый из этих языков предоставляет поддержку классическому стилю C программирования несколько различающимися способами. Результирующие отличия могут

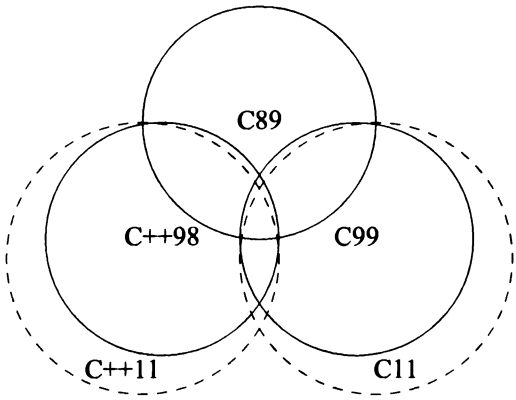
сильно навредить программистам, которые используют как C, так и C++; проблемы возникают у тех, кто пишет на одном языке, а использует библиотеки от другого языка.

Как же я могу называть C и C++ братьями? Ясно, что C++ является наследником C. Однако, давайте взглянем на следующее фамильное дерево:



Сплошные линии означают массивное наследование средств языка, пунктирные линии — наследование основных средств, а точечные линии означают минимальное заимствование. При таком взгляде на ISO C и ISO C++, эти языки можно считать основными наследниками K&R C, то есть их можно считать братьями. Каждый из них содержит ключевые аспекты классического C и никто из них на 100% не совместим с классическим C. Здесь я использую термин «классический C» в смысле «K&R C» плюс перечисления и присваивания структур.

Несовместимости сильно мешают программистам, поскольку создают комбинаторный взрыв альтернатив. Рассмотрим простую диаграмму:



Как C++11, так и C11, имеют большую часть «K&R C» в качестве подмножества. C++11 имеет большую часть C11 в качестве подмножества. Это средства, имеющие отношение к четко разграниченным областям.

C89 только	Вызов необъявленной функции
C99 только	Массивы переменной длины (VLA)
C++ только	Шаблоны
C89 и C99	Объявления функций в стиле Algol
C89 и C++	Использование ключевого слова C99 **restrict** в кач-ве идентификатора
C++ и C99	// комментарии
C89, C++, и C99	**struct**'ы
C++11 только	Перемещающая семантика (используя rvalue-ссылки; **&&**)
C11 только	Обобщенные выражения, использующие ключевое слово **_Generic**
C++11 и C11	Атомарность

Часто различия между C и C++ не являются следствием изменений в C, сделанных при разработке C++. Многие из различий возникали уже после возникновения C++. Примерами служат возможность присваивать **T*** в **void***, а также компоновка с глобальными константами [Stroustrup,2002]. Иногда такие различия возникали в процессе изменений в C после того, как они уже были инкорпорированы в стандарт ISO C++, вроде разного смысла **inline**.

44.3.2. «Тихие» отличия

С небольшими исключениями, программы, которые одновременно являются C++ и C программами, имеют одинаковый смысл в обоих языках. По счастью, эти исключения (часто называемые «тихими отличиями») довольно экзотические:

- В языке C размер символьной константы и энумераторов равен **sizeof(int)**. В C++ **sizeof('a')** равен **sizeof(char)**.
- В C энумератор есть **int**, а реализациям C++ разрешается выбирать наиболее подходящий размер (§8.4.2).
- В C++ имя для **struct** вводится в область видимости, в котором оно объявляется; а в C нет. В результате, в C++ имя **struct**, определенное в более вложенной области видимости, скрывает такое же имя в объемлющей области видимости. Пример:

```
int x[99];
void f()
{
    struct x { int a; };
    sizeof(x); /* размер массива в C, размер структуры в C++ */
    sizeof(struct x);    /* размер структуры */
}
```

44.3.3. C код, который не есть C++ код

Несовместимости в C/C++, которые приносят наибольшее число проблем в реальной работе, не такие уж и тонкие. Большинство из них обнаруживаются компиляторами. В настоящем разделе мы приводим примеры, когда C код не есть C++ код. Большая их часть рассматривается как плохой стиль или даже помечены устаревшими в современном C. Исчерпывающий список несовместимостей представлен в §iso.C.

- В C большую часть функций можно вызывать без предварительного объявления. Например:

```
int main()      // не C++; плохой стиль в C
{
    double sq2 = sqrt(2);                          /* вызов не определенной функции */
    printf("the square root of 2 is %g\n",sq2); /* вызов не определенной функции */
}
```

Полное и согласованное применение объявлений функций (прототипов функций) рекомендуется в C. Когда следуют этому разумному совету, и особенно в случае, когда компиляторы этому способствуют, C код соответствует правилам C++. В случае же вызова необъявленных функций вы берете на себя ответственность за то, что функция вызвана корректно, или же получен источник несовместимостей.

- В C функция, объявленная без указания типов аргументов, может принять любое число аргументов любых типов.

```
void f();  /* типы аргументов не указаны */

void g()
{
    f(2); /* плохой стиль в C; не C++ */
}
```

Это считается устревшим в ISO C.

- В C возможно определение функций, в котором спецификация типов аргументов располагается после списка аргументов:

```
void f(a,p,c) char *p; char c; { /* ... */ }      /* C; не C++ */
```

Такие определения следует переписать:

```
void f(int a, char* p, char c) { /* ... */ }
```

- В C структуры можно определять в типе возврата и в объявлениях типа аргумента. Например:

```
struct S { int x,y; } f();  /* C; не C++ */
void g(struct S { int x,y; } y);      /* C; не C++ */
```

Правила определения типов в C++ делают такие объявления бесполезными, и они не разрешены.

- В C целые можно присваивать переменным, имеющими тип перечисления:

```
enum Direction { up, down };
enum Direction d = 1;   /* ошибка: int присваивается Direction; OK в C */
```

- C++ предоставляет много ключевых слов, которых нет в C. Если такое ключевое слово присутствует в C программе в качестве идентификатора, то такую программу надлежит модифицировать, чтобы она стала C++ программой:

Ключевые слова C++, не являющиеся ключевыми словами C

alignas	alignof	and	and_eq	asm	bitand
bitor	bool	catch	char16_t	char32_t	class
compl	const_cast	constexpr	decltype	delete	dynamic_cast
explicit	false	friend	inline	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or_eq	private	protected	public	reinterpret_cast	static_assert
static_cast	template	this	thread_local	throw	true
try	typeid	typename	using	virtual	wchar_t
xor	xor_eq				

Кроме того, ключевое слово **export** зарезервировано для будущего. C99 включает ключевое слово **inline**.

- В C некоторые ключевые слова языка C++ являются макросами, определеными в стандартных заголовочных файлах:

Ключевые слова C++, являющиеся макросами C								
and	**and_eq**	**bitand**	**bitor**	**bool**	**compl**	**false**	**not**	**not_eq**
or	**or_eq**	**true**	**wchar_t**	**xor**	**xor_eq**			

Из этого следует, что в C их можно проверять посредством **#ifdef**.

- В C объект глобальных данных может объявляться много раз в одной и той же единице трансляции без использования спецификатора **extern**. До тех пор, пока максимум одно объявление содержит инициализатор, объект считается определеным только один раз. Например:

```
int i;
int i; /* еще одно определение одной и той же целочисленной "i"; не C++ */
```

В C++ сущность должна определяться ровно один раз; §15.2.3.

- В C **void*** может использоваться в качестве правого операнда присваивания или в инициализации переменной любого указательного типа; в C++ нельзя этого делать (§7.2.1). Пример:

```
void f(int n)
{
    int* p = malloc(n*sizeof(int)); /* не C++; в C++, выделяйте, используя "new" */
}
```

Возможно, это единственное отличие, с которым труднее всего бороться. Заметьте, что неявное приведение **void*** к другим указательным типам не является источником вреда в общем случае:

```
char ch;
void* pv = &ch;
int* pi = pv;    // не C++
*pi = 666;       // перепишет ch и другие байты возле ch
```

Если вы применяете оба языка, то выполните приведение типа для результата вызова функции **malloc()** к правильному варианту. Если же вы используете только C++, то лучше вообще избегать использования **malloc()**.

- В C тип строкового литерала есть массив **char**, а в C++ это массив **const char**, так что:

```
char* p = "a string literal is not mutable";  // ошибка в C++; OK в C
p[7] = 'd';
```

- В C допускается передача управления на оператор с меткой (**switch** или **goto**; §9.6), обходя инициализацию; C++ этого не допускает. Например:

```
goto foo;       // OK в C; не C++
// ...
{
    int x = 1;
foo:
    if (x!=1) abort();
    /* ... */
}
```

- В C глобальный **const** по умолчанию имеет внешнее связывание; в C++ это не так и должна иметь место инициализация, если только нет объявления **extern** (§7.5). Например:

```
const int ci;   // OK в C; ошибка const not initialized в C++
```

- В C имена вложенных структур помещаются в ту же самую область видимости, что и структура, в которую они вложены. Пример:

```
struct S {
    struct T { /* ... */ } t;
    // ...
};

struct T x;          // OK в C, означает "S::T x;"; не C++
```

- В C++ имя класса вносится в область видимости, в которой оно объявляется; оно не может иметь то же самое имя, что и другой тип, объявленный в той же самой области видимости. Пример:

```
struct X { /* ... */ };
typedef int X;       // OK в C; не C++
```

- В C массив может инициализироваться инициализатором, который имеет больше элементов, чем того требует массив. Пример:

```
char v[5] = "Oscar";     // OK в C, завершающий 0 не используется; не C++
printf("%s",v);          // вероятная катастрофа
```

44.3.3.1. Проблемы «классического C»

Если вам требуется апгрейдировать программы на «классическом C» («K&R C») или C89 программы, то возникает еще некоторое количество проблем:

- В C89 нет // комментариев (хотя большинство C89-компиляторов их добавляют самостоятельно):

```
int x;     // не C89
```

- В C89 спецификатор типа имеет умолчательный вариант **int**. Например:

```
const a = 7;   /* в C89, подразумевается тип int; не C++ или C99 */

f()   /* тип возвращаемого значения f() по умолчанию int; не C++ или C99 */
{
    /* .. */
}
```

44.3.3.2. Средства C, которые не были приняты в C++

Несколько добавлений в C99 (по сравнению с C89) были намеренно не приняты в C++:

1. Массивы переменной длины; используйте **vector** или другие варианты динамических массивов
2. Помеченные инициализаторы; используйте конструкторы.

Средства C11 весьма новы для того, чтобы их можно было рассматривать в C++, за исключением таких черт, как модель памяти и атомарность (§41.3), которые и пришли туда из C++.

44.3.4. C++ код, который не есть C код

В этом разделе перечисляются средства, которые предлагаются языком C++, но не входят в C. Возможна разная классификация этих средств, так что представленная ниже классификация не претендует на универсальность.

- Средства, предназначенные в основном для удобства нотации:
  1. // комментарии (§2.2.1, §9.7); добавлены в C99
  2. Поддержка ограниченных наборов символов (§iso.2.4); частично добавлена в C99
  3. Поддержка расширенных наборов символов (§6.2.3); добавлена в C99
  4. Неконстантные инициализаторы для объектов в статическом хранилище (§15.4.1)
  5. **const** в константных выражениях (§2.2.3, §10.4.2)
  6. Объявления как операторы (§9.3); добавлены в C99
  7. Объявления в инициализаторах циклов **for** (§9.5); добавлены в C99
  8. Объявления в условиях (§9.4.3)
  9. Имена структур не нуждаются в префиксе **struct** (§8.2.2)
  10. Анонимные **union** (§8.3.2); добавлены в C11
- Средства усиления системы типов:
  1. Проверка типов аргументов (§12.1); частично добавлены в C (§44.3.3)
  2. Компоновка, безопасная по типам (§15.2, §15.2.3)
  3. Работа со свободной памятью посредством **new** и **delete** (§11.2)
  4. **const** (§7.5); частично добавлено в C
  5. Логический тип **bool** (§6.2.2); частично добавлен в C99
  6. Именованные конверсии типов (§11.5.2)
- Средства для пользовательских типов:
  1. Классы (Глава 16)
  2. Члены-функции (§16.2.1) и члены-классы (§16.2.13)
  3. Конструкторы и деструкторы (§16.2.5, Глава 17)
  4. Производные классы (Глава 20, Глава 21)
  5. Виртуальные функции и абстрактные классы (§20.3.2, §20.4)
  6. Управление доступом: **public/protected/private** (§16.2.3, §20.5)
  7. **friend** (§19.4)
  8. Указатели на члены (§20.6)
  9. Статические члены (§16.2.12)
  10. Члены **mutable** (§16.2.9.3)
  11. Перегрузка операций (Глава 18)
  12. Ссылки (§7.7)
- Средства организации программ (в дополнение к классам):
  1. Шаблоны (Глава 23)
  2. Встраиваемые функции (§12.1.3); добавлены в C99
  3. Умолчательные аргументы (§12.2.5)
  4. Перегрузка функций (§12.3)
  5. Пространства имен (§14.3.1)
  6. Явная квалификация области видимости (операция ::; §6.3.4)
  7. Исключения (§2.4.3.1, Глава 13)

8. RTTI (Глава 22)

9. Обобщенные константные выражения (**constexpr**; §2.2.3, §10.4, §12.1.6)

Средства C++11, перечисленные в §44.2, отсутствуют в C.

Ключевые слова, добавленные в C++ (§44.3.3), могут использоваться как маркеры большинства специфических для C++ возможностей. Однако, некоторые средства, такие как перегрузка функций и константные выражения, не идентифицируются ключевыми словами.

Компоновка функций в C++ безопасна по типам, в то время как правила C такого не предоставляют. Отсюда вытекает, что в большинстве реализаций C++ функцию нужно маркировать **extern "C"**, чтобы она была C++ функцией, но в то же время соответствовала соглашениям о вызове языка C (§15.2.5). Например:

```
double sin(double);                 // не будет компоноваться с C кодом
extern "C" double cos(double);      // будет компоноваться с C кодом
```

Макрос **__cplusplus** может использоваться для определения того, какой компилятор будет применяться — компилятор языка C или компилятор языка C++ (§15.2.5).

Помимо перечисленных средств нужно еще отметить, что C++ библиотека (§30.1.1, §30.2) по большей части специфична именно для C++. Библиотека C предлагает общие макросы для типов в **<tgmath.h>** и поддержку чисел **_Complex** в **<complex.h>**, аппроксимирующем **<complex>**.

В C также есть **<stdbool.h>**, в котором предоставляется **_Bool** и синоним **bool** (для аппроксимации типа **bool** языка C++).

44.4. Советы

[1] Перед тем, как использовать новое средство в производственном коде, напишите небольшую программу, с помощью которой можно протестировать совместимость со стандартом и уровень производительности; §44.1.

[2] Для изучения C++ используйте самые полные и самые современные реализации C++, которые вы только можете достать; §44.2.4.

[3] Общее подмножество C и C++ не есть наилучший вариант подмножества C++ для изучения; §1.2.3; §44.2.4.

[4] Отдавайте предпочтение стандартным средствам перед нестандартными; §36.1, §44.2.4.

[5] Избегайте средств, помеченных как устаревшие, вроде **throw**-спецификаций; §44.2.3, §13.5.1.3.

[6] Избегайте приведений типа в C-стиле; §44.2.3, §11.5.

[7] Правило «неявный **int**» поставлено «вне закона», так что явным образом специфицируйте типы функций, переменных, констант и т. д.; §44.3.3.

[8] При конвертировании C программы в C++ программу сначала проверьте, что объявления функций и стандартные заголовочные файлы используются согласованным образом; §44.3.3.

[9] При конвертировании C программы в C++ программу переименуйте переменные, имена которых совпадают с ключевыми словами языка C++; §44.3.3.

[10] Если вы должны применить язык C, то ради переносимости и безопасности типов следует писать программу на общем подмножестве языков C и C++; §44.2.4.

[11] При конвертировании C программы в C++ программу, выполните конвертирование возврата функции **malloc()** в надлежащий тип, или же замените вызовы этой функции на **new**; §44.3.3.

[12] При замене **malloc()** и **free()** на **new** и **delete**, подумайте о том, чтобы использовать **vector**, **push_back()** и **reserve()** вместо **realloc()**; §3.4.2, §43.5.

[13] При конвертировании С программы в С++ программу не забывайте, что отсутствует неявное приведение от **int** к перечислениям; используйте явное приведение; §44.3.4, §8.4.

[14] Средства из пространства имен **std** определяются в заголовочном файле без суффикса **h** (например, **std::cout** определяется в **<iostream>**); §30.2.

[15] Для **std::string** используется **<string>** (**<string.h>** содержит строковые функции в С-стиле); §15.2.4.

[16] Для каждого стандартного заголовочного файла **<X.h>**, помещающего имена в глобальное пространство имен, заголовочный файл **<cX>** помещает имена в пространство имен **std**; §15.2.2.

[17] Используйте **extern "C"**, когда объявляете С функции; §15.2.5.

Научно-техническое издание

Бьерн Страуструп

Язык программирования C++. Четвертое издание

Редактор А. Боборыкин

Компьютерная верстка *А. Егоровой*

ISBN 978-5-6045724-6-7

Подписано в печать 20.08.2021. Формат 70x100/16.

Печатн. лист. 76. Бумага специальная

Тираж 1000 экз. Заказ 4157

Издательство БИНОМ-МЕД

127018, Москва, ул. 1-Я Ямская, 8

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт www.chpd.ru, E-mail: sales@chpd.ru