Программирование на C++

Лекция 6

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2026

Проблема C-строк

C-строка = char* с \0
Программист сам следит за длиной, памятью, переполнением

```
char s[10];
strcpy(s, "Hello");
strcat(s, " world");  // переполнение!
if (strcmp(s, "Hello world") == 0) ...
```

Тезис: Источник багов
Решение: std::string

Строки в стиле C++. Базовые операции

Класс string:

```
string s1 = "Hello ";
string s2 = "world !!!";

s1.size();    // s1 — экземпляр класса, он помнит свою длину
s1[0];        // Индексация символов с нуля
s1 = s2;      // Строки можно присваивать друг другу
s1 += s2;   // Строки можно прибавлять друг другу
s1 == s2;
s1 < s2;     // Строки можно сравнивать
```

Преимущества

- Скрывает способ хранения строки
- Длина строки контролируется и автоматически увеличивается при необходимости
- Снижается вероятность ошибок
- Перегружены операции + и +=
- Можно присваивать одной строке значение другой

Несмотря на то, что скорость работы С-строк немного выше, чем у класса string, на прикладном уровне лучше использовать string.

Однако системным программистам чаще приходится пользоваться char*.

С-строка	С++-строка
char s[100]	string s
strcpy, strcat	=, +=
strcmp	==, <
ручная длина	s.size()
риск переполнения	автоматически

Создание и инициализация

```
string s1;                  // пустая
string s2 = "Hello";        // копирующая инициализация
string s3("Hello");         // прямая инициализация
string s4(s3);              // копия
string s5(5, 'a');          // заполнение "aaaaa"
string s6 = s3 + " " + s4;  // конкатенация
```

Ввод-вывод

```
cout << s
cin >> s;
```

Опять пробелы((
Если мы вводим "Hello world" в s будет храниться только "Hello", поэтому надо использовать

```
getline(cin, s);
```
или
```
getline(cin, s,'!’);
```

Считывает неформатированные данные из потока в строку. Останавливается, как только найден символ, равный разделителю, или исчерпан поток.
Первая версия использует в качестве разделителя '\n', вторая — ‘!’.
Символ-разделитель удаляется из потока и не помещается в строку.

Как передавать в функции C++-строки

C++-строки
void p1(string &s) // s можно менять
void p1(const string &s) // s нельзя менять

void p1(string s); // копия — дорого

//Если строка **не меняется** — всегда **const string&**

Шпаргалка по методам string

Категория	Методы
Размер	size(), length(), empty(), capacity()
Доступ	[], at(), front(), back(), data(), c_str()
Поиск	find(), rfind(), find_first_of()
Модификация	append(), insert(), erase(), replace(), clear()
Подстроки	substr()
Сравнение	compare(), ==, <

Пример с find и npos

```
string s;
 s = "string srt srt";
 cout <<  s.length() << endl;
 int pos = s.find("St");
 if (pos != string::npos)
   cout<<pos<<endl;
 else
   cout <<"Not found\n";
```

Переменная **string::npos** является статическим элементом класса string

Ее значение **зависит от платформы**, может равняться максимально возможному количеству символов в строковом объекте 4294967295

Это значение используется как **признак неудачного** завершения **поиска**

Никогда не сравнивайте pos с числом — только с string::npos

Посимвольный обход

```
string s = "Hello!";
for (int i=0; i <s.length(); ++i)
      cout<<s[i];
s[5]='N';
```

```
for (char c : s) cout << c;            // range-based for
for (char& c : s) c = toupper(c); // изменение
```

Запомните
s[i] не проверяет границы, s.at(i) — проверяет

Рекомендуется не смешивать использование строк в стиле С и строк в стиле С++.

Для преобразования строк из типа string в тип char* используется метод **c_str()** класса string, который возвращает указатель типа const char * на строку, содержащую те же символы, что и строка типа string.

Необходимость такого преобразования возникает крайне редко только в тех случаях, когда требуется использовать для обработки строк функции библиотеки cstring.

```
string s1="C++";
const char *s2 = s1.c_str();
printf("%s\n", s2);  // для С-функций
```

ReplaceAll.h + идея: найти все вхождения и заменить

Пример. Заменить в строке s1 каждое вхождение подстроки s2 подстрокой s3.

```
//ReplaceAll.h

#pragma once

#include <string>

void replaceAll (string &context, const string &from,  const string & to);
```

ReplaceAll.h: реализация

```
//ReplaceAll.cpp

#include "ReplaceAll.h"
using namespace std;

void replaceAll (string & context, const string & from, const string & to) {
  if (from.empty()) return;  // защита от бесконечного цикла
  size_t lookHere=0;
  size_t foundHere;
  while ((foundHere = context.find(from, lookHere)) != string::npos) {
    context.replace(foundHere, from.size(), to);
    lookHere = foundHere + to.size();
  }
}
```

Типичные ошибки со строками

cin >> s вместо getline	— теряются пробелы
s.length() возвращает size_t	— сравнение с int даёт warning
s.c_str()	— указатель живёт, пока строка не изменилась
s[i]	— не проверяет границы
replaceAll с пустой from	— бесконечный цикл

Современный C++

std::string_view (C++17) — объект для строки без копирования
R"(C:\path\file)" — сырые литералы без экранирования
std::to_string, std::stoi — преобразования
std::format (C++20) — замена printf

std::string_view

std::string_view в C++ — это облегчённый объект, который предоставляет представление строки (или части строки) без фактического копирования данных. Он был введён в стандарте C++17.

Конструирование string_view

```
const char arr[] = { 'H', 'e', 'l', 'l', 'o' };
string_view v(arr, 5);
assert(v.length() == 5);
assert(v.data() == &arr[0]); // view uses arr's
data

string s("Hello");
string_view sv = s;
assert(sv.data() == s.data());
assert(sv.length() == s.length());

sv = string_view(s.data(), 4);
assert(sv.length() == 4);
```

Как это работает

std::string_view не владеет памятью и не копирует символы. Он просто хранит указатель на начало последовательности символов и длину строки. Это позволяет работать с подстроками без создания новых объектов и выделения памяти.

Ключевые особенности std::string_view

- **Не владеет данными**. std::string_view не отвечает за выделение или освобождение памяти. Он лишь ссылается на уже существующий буфер (например, на объект std::string или строковый литерал)
- **Только для чтения**. Нельзя изменить символы в строке через std::string_view
- **Может стать некорректным**. Если буфер, на который ссылается std::string_view, будет изменён или уничтожен, представление станет некорректным (неопределённое поведение).
- **Поддерживает большинство методов** std::**string**. Например, size(), empty(), доступ по индексу, data(), substr(), compare(), find().
- **Не требует завершающего нуля**. В отличие от C-строк, std::string_view хранит длину, поэтому ему не нужен нулевой ограничитель в конце

Когда использовать: std::string_view особенно полезен в следующих случаях:

- **В параметрах функций**. Если функция только читает строку, предпочтительнее принимать std::string_view, чтобы избежать лишнего копирования
- **Для подстрок**. Вместо того чтобы вызывать substr(), который создаёт новый std::string, можно сразу передать std::string_view на поддиапазон исходной строки
- Для взаимодействия с C-API. std::string_view легко взаимодействует с функциями, ожидающими const char*

Важные ограничения std::string_view

- Не продлевайте жизнь данных. Не храните std::string_view дольше, чем живёт объект, на который она ссылается.
- Не используйте на изменяемых строках. Если вы планируете изменять данные через std::string методы push_back, append, resize), не передавайте std::string_view на них
- Для Unicode. Для работы с широкими символами (wchar_t и т.п.) существуют специализированные типы: std::wstring_view, std::u16string_view, std::u32string_view

Вывод
std::string_view — мощный инструмент для оптимизации, но его нужно применять осознанно, учитывая особенности владения памятью и жизненный цикл данных

Сырые литералы без экранирования

R"(C:\path\file)" — сырые литералы без экранирования. Это удобная фишка из стандарта C++11, которая позволяет писать пути к файлам в Windows без постоянного экранирования обратных слешей (\)

```
#include <iostream>
#include <string>

int main() {
    std::string path = R"(C:\Users\Name\Documents\file.txt)";
    std::cout << path << std::endl;
    return 0;
}
```

Этот код выведет полный путь без лишних слешей — C:\Users\Name\Documents\file.txt

Как это работает

Ключевое слово R (от «raw» — сырой) сообщает компилятору, что дальше идёт исходный текст, а не шаблон с экранированными символами. Всё, что находится между R" и ", компилятор передаёт в строковый литерал как есть, без обработки escape-последовательностей.

В обычном строковом литерале каждый обратный слеш нужно экранировать, написав \\, иначе компилятор воспримет его как часть последовательности для экранирования.

std::format (C++20)

```
#include <format>
#include <iostream>
#include <string>

int main() {
    std::string name = "Алиса";
    auto s = std::format("Привет, {}!", name); // Вызов возвращает строку
    std::cout << s << '\n'; // Вывод: Привет, Алиса!

    // Примеры со спецификаторами
    std::cout << std::format("{:#x}\n", 255); // Шестнадцатеричная с префиксом
    std::cout << std::format("{:#b}\n", 10); // Двоичная

    // Для вещественных чисел
    std::cout << std::format("{:.2f}\n", 3.14159); // 3.14

    // Форматирование времени
    auto now = std::chrono::system_clock::now();
    std::string time_str = std::format("{:%Y-%m-%d %H:%M:%S}", now);
    std::cout << time_str << '\n';
    return 0;
}
```

Проблема конфликта имён

Две библиотеки определяют print(). Как различить?

```
namespace LibA { void print(); }
namespace LibB { void print(); }
LibA::print();  // однозначно
```

Пространство имён — это «фамилия» для идентификаторов ☺

Пространства имён

- Пространство имён — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.).

- Пространства имён используются для организации кода в виде логических групп и **с целью избежания конфликтов имён**, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек.

Доступ

Все идентификаторы в пределах пространства имён доступны друг другу без уточнения.

Идентификаторы за пределами пространства имен могут получить доступ к членам с помощью:

1. полного имени для каждого идентификатора, например std::vector<std::string> vec;
2. объявления для одного идентификатора using std::string;
3. директивы using для всех идентификаторов в пространстве имен using namespace std;

Способ	Синтаксис	Область	Риск конфликта
Полное имя	std::string	везде	нет
using-объявление	using std::string;	только string	низкий
using-директива	using namespace std;	всё пространство	высокий

Код в **файлах заголовков** всегда должен содержать **полное имя** в пространстве имен.

Создание пространства имён

```
namespace MyNamespace {
	class ObjectManager {
		public: void DoSomething() {…}
	};
	void Func(ObjectManager) {…}
}
```

Использование полного имени

```
namespace MyNamespace {
        class ObjectManager {
                public: void DoSomething() {…}
        };
        void Func(ObjectManager) {…}
}
```

```
MyNamespace ::ObjectManager mgr;
mgr.DoSomething();
MyNamespace ::Func(mgr);
```

Добавить в область видимости один идентификатор

```
namespace MyNamespace {
	class ObjectManager {
		public: void DoSomething() {…}
	};
	void Func(ObjectManager) {…}
}
```

```
using MyNamespace ::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
MyNamespace ::Func(mgr);
```

Добавить в область видимости все идентификаторы пространства имён

```
namespace MyNamespace {
	class ObjectManager {
		public: void DoSomething() {…}
	};
	void Func(ObjectManager) {…}
}
```

```
using namespace MyNamespace;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

Рекомендации

- Директиву using можно поместить в верхнюю часть cpp-файла (в области видимости файла) или внутрь определения класса или функции.

- Без особой необходимости не размещайте директивы using в файлах заголовков (*.h), так как любой файл, содержащий этот заголовок, добавит все идентификаторы пространства имён в область видимости, что может вызвать скрытые конфликты имён или конфликты, которые очень трудно отлаживать.

- В файлах заголовков всегда используйте полные имена. Если эти имена получаются слишком длинными, используйте псевдоним пространства имен для их сокращения.

Реализации функций в .cpp-файле

```
//contosoData.h
#pragma once
namespace ContosoDataServer {
    void Foo();
    int Bar();
}
```

```
//contosoData.cpp
#include "contosodata.h"
using namespace ContosoDataServer;

// use fully-qualified name here
void ContosoDataServer::Foo() {
      Bar(); // no qualification needed for Bar()
}

int ContosoDataServer::Bar(){return 0;}
```

Реализации функций в .cpp-файле должны использовать полное имя, даже если вы размещаете using директиву в верхней части файла

Объявления в нескольких файлах

- Пространство имён может быть объявлено в нескольких блоках в одном файле или в нескольких файлах.
- Компилятор соединит вместе все части во время предварительной обработки и полученное в результате пространство имён будет содержать все члены, объявленные во всех частях.
- Примером этого является пространство имён std, которое объявляется в каждом из файлов заголовка в стандартной библиотеке.

Ошибка порядка объявлений

```
namespace V {
   void f();
}

void V::f() { }       // ok
void V::g() { }       // C2039, g() is not yet a member of V

namespace V {
   void g();
}
```

Эта ошибка может возникнуть, когда члены пространства имён объявляются в нескольких файлах заголовка и эти заголовки не включены в правильном порядке.

Глобальное пространство имён

Если идентификатор не объявлен явно в пространстве имён, он неявно считается входящим в глобальное пространство имён

Старайтесь избегать объявления в глобальной области, если это возможно, за исключением основной функции точки входа, которая должна находиться в глобальном пространстве имён

Чтобы явно указать глобальный идентификатор, используйте оператор разрешения области видимости без имени
::SomeFunction(x);

Пространство имён std

Все типы и функции стандартной библиотеки C++ объявляются в пространстве имён std или пространствах имён, вложенных внутри std.

Примеры: std::string, std::vector, std::cout, std::sort

Вложенные пространства имён

```
namespace MyNamespace {
        void Foo();

        namespace Details {
                int CountImpl;
                void Ban() { return Foo(); }
        }

           int Bar(){...};
           int Baz(int i) { return Details::CountImpl;}
}
```

Псевдонимы пространств имён

Имена пространств имен должны быть уникальными, из-за чего зачастую они получаются не слишком короткими.

Можно создать псевдоним пространства имен, который служит сокращенным для фактического имени.

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }
```

Анонимные или безымянные пространства имён

```
namespace {
   int MyFunc(){}
}
```

Это называется неименованным или анонимным пространством имён, и полезно, если нужно сделать объявления переменных невидимыми для кода в других файлах, не создавая именованное пространство имён.

Весь код, находящийся в том же файле, может видеть идентификаторы в безымянном пространстве имён, но эти идентификаторы, а также само пространство имён, будет невидимым за пределами этого.

Типичные ошибки с пространствами имён

using namespace std; в .h — конфликты

Определение функции без полного имени в .cpp — ошибка линковки

Объявление в неправильном порядке — ошибка компиляции

Анонимное пространство без закрывающей } — ошибка

Best practices

В .h — только полные имена

В .cpp — using в узкой области

Для длинных имён — псевдонимы

Анонимные пространства — для внутренних функций файла

using namespace std; — только в учебных примерах

Шпаргалка

Строки: string, getline, find, substr, c_str, npos

Пространства: namespace, using, ::, анонимные

Правила: в .h — полные имена; в .cpp — можно using;

Не смешивать C и C++ строки без нужды