Лекция 12. Рекурсия. Шаблоны функций. Введение в концепты

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

Содержание лекции

1. Рекурсия: определение, структура, стек вызовов, примеры.
2. Преимущества и недостатки рекурсии, сравнение с итерацией.
3. Шаблоны функций: синтаксис, инстанцирование, специализация.
4. `constexpr`, `static_assert`, константные аргументы шаблонов.
5. Введение в концепты C++20: мотивация, синтаксис, примеры.

Что такое рекурсия?

Определение

Рекурсия — это способ определения функции, при котором она вызывает саму себя.

- Рекурсивная функция состоит из:
  1. **Базовый случай** — условие остановки, при котором функция возвращает результат без рекурсивного вызова.
  2. **Рекурсивный случай** — вызов функции с изменёнными параметрами, приближающими к базовому случаю.

Пример

Простейший пример — вычисление факториала:

```
int factorial(int n) {
    if (n <= 1) return 1;       // базовый случай
    return n * factorial(n - 1); // рекурсивный случай
}
```

Рекурсивные определения структур

Формы Бэкуса-Наура (BNF)

Способ описания синтаксиса языков с помощью рекурсивных правил.

Пример

Список целых чисел: Список ::= Элемент | Элемент Список

- Рекурсивное определение позволяет описать бесконечное множество объектов конечным образом.
- Пример: односвязный список в C++ — это рекурсивная структура.

```
struct Node {
    int data;
    Node* next;  // ссылка на объект того же типа
};
```

Важно

Хотя бы одна ветка определения должна завершаться без рекурсии (базовый случай).

Структура рекурсивной функции

Общий шаблон

```
тип_возврата имя(параметры) {
    if (условие_останова) {
        return значение_для_базового_случая;
    }
    // рекурсивный случай
    return выражение_с_вызовом_имя(изменённые_параметры);
}
```

- Базовый случай должен быть достижим за конечное число шагов.
- Каждый рекурсивный вызов должен приближать к базовому случаю.

Прямая и косвенная рекурсия

Прямая рекурсия

Функция вызывает саму себя.

```
void direct() {
    // ...
    direct();  // вызов самой себя
}
```

Косвенная рекурсия

Функция A вызывает B, а B вызывает A (или более длинная цепочка).

```
void A() { B(); }
void B() { A(); }
```

Глубина рекурсии

Максимальное количество вложенных рекурсивных вызовов.

Стек вызовов и запись активации

Стек вызовов (call stack)

Область памяти для хранения:

- локальных переменных функций,
- параметров,
- адресов возврата,
- промежуточных значений.

Запись активации (фрейм)

Структура, добавляемая в стек при каждом вызове функции.

- Каждый рекурсивный вызов создаёт новый фрейм.
- При возврате фрейм удаляется.
- При слишком глубокой рекурсии стек переполняется (Stack Overflow).

Фаза спуска — последовательные вызовы, стек растёт. **Фаза подъёма** — возвраты из вызовов, стек уменьшается.

Иллюстрация стека вызовов для функции `factorial`



Для просмотра стека вызовов в VS можно поставить точку останова (слева), например, на строке с условием останова. Затем запустить отладку клавишей F5.

Пример 1: счётчик

```cpp
#include <iostream>
void countDown(int n) {
    if (n < 0) return;           // базовый случай
    std::cout << n << " ";
    countDown(n - 1);            // рекурсивный вызов
    std::cout << n << " ";       // после возврата
}
int main() {
    countDown(3);
    return 0;
}
```

- **Действия на спуске** выполняются в прямом порядке: 3 2 1 0.
- **Действия на возврате** выполняются в обратном порядке: 0 1 2 3.

Графическое представление рекурсии



- Синий — рекурсивный спуск (прямой порядок).
- Красный — рекурсивный возврат (обратный порядок).

Пример 2: факториал

Рекурсивная версия

```
int factRec(int n) {
    if (n <= 1) return 1;
    return n * factRec(n - 1);
}
```

Итеративная версия

```
int factIter(int n) {
    int res = 1;
    for (int i = 2; i <= n; ++i) res *= i;
    return res;
}
```

- Рекурсия даёт компактный код, но может быть менее эффективной.
- Итерация использует цикл и не наращивает стек рекурсивными вызовами.

Пример 3: числа Фибоначчи

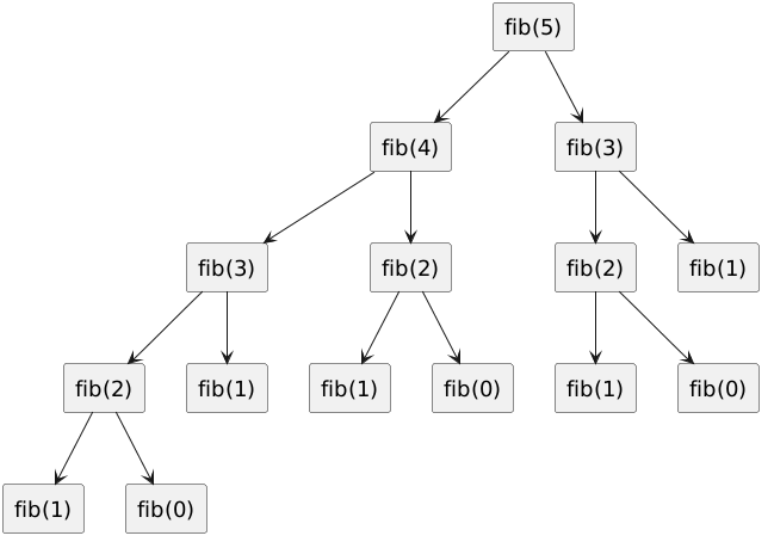
Рекурсивная реализация

```
int fib(int n) {
    if (n <= 1) return n;   // два базовых случая
    return fib(n - 1) + fib(n - 2);
}
```

Важно

Рекурсия: время $O(2^n)$; более точная оценка — $O(\varphi^n)$, где $\varphi \approx 1.618$. Память стека: $O(n)$.
Итерация: время $O(n)$, дополнительная память $O(1)$.

Пример 3: дерево вызовов



Пример 3: чем он плох?

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Вопрос: Как называется такая рекурсия? Почему этот код плох?

Пример 3: чем он плох?

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Вопрос: Как называется такая рекурсия? Почему этот код плох?

Ответ

Это **каскадная рекурсия**: каждый небазовый вызов порождает два новых.

Проблемы:

1. Временная сложность: $\mathrm{O}(2^n)$.
2. Многократные повторные вычисления одних и тех же значений.

Вывод

Эта реализация неэффективна из-за многократного вычисления одних и тех же значений.

Пример 3: улучшенная реализация

Идея

Передавать два последних значения как аккумуляторы, избегая каскада.

```
int fib(int n, int a = 0, int b = 1) {
    if (n == 0) return a;
    if (n == 1) return b;
    return fib(n - 1, b, a + b);
}
```

Вопрос: Как изменилась сложность?

Пример 3: улучшенная реализация

Идея

Передавать два последних значения как аккумуляторы, избегая каскада.

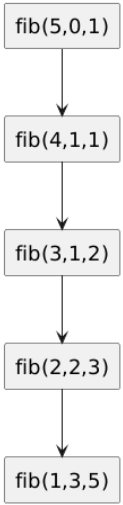
```
int fib(int n, int a = 0, int b = 1) {
    if (n == 0) return a;
    if (n == 1) return b;
    return fib(n - 1, b, a + b);
}
```

Вопрос: Как изменилась сложность?

Ответ

- Теперь каждый вызов порождает только один новый — **линейная рекурсия**. Нет повторных вычислений.
- Временная сложность: $\mathrm{O}(n)$. Пространственная сложность: $\mathrm{O}(n)$. Глубина: n.
- Здесь итеративное решение лучше.

Пример 3: дерево вызовов для улучшенной реализации



Вложенная рекурсия (функция Аккермана)

Определение

Вложенная рекурсия – вид рекурсии, при котором рекурсивный вызов функции осуществляется с параметрами, содержащими другие рекурсивные вызовы той же функции.

```
int ackermann(int m, int n) {
    if (m == 0) return n + 1;
    if (n == 0) return ackermann(m - 1, 1);
    return ackermann(m - 1, ackermann(m, n - 1));
}
```

- **Функция Аккермана** — наиболее известный пример вложенной рекурсии.
- Функция Аккермана очень быстро растёт.
- Хороший пример того, что рекурсия не всегда практична.

Опасность: бесконечная рекурсия

Неправильная рекурсия

```
void infinite(int n) {
    std::cout << n << " ";
    infinite(n + 1);   // нет базового случая
}
```

- Отсутствие базового случая или неправильное изменение параметра приводит к бесконечной рекурсии.
- Стек переполняется, программа аварийно завершается.

Ошибка

```
Unhandled exception: Stack overflow
```

Преимущества и недостатки рекурсии

Аспект	**Рекурсия**	**Итерация**
Читаемость	Часто более естественна для рекурсивных структур (деревья)	Более громоздкая для вложенных структур
Использование памяти	Стек O(глубина) при постоянном размере фрейма	Зависит от алгоритма; для факториала O(1)
Скорость	Возможны накладные расходы на вызовы	Может быть быстрее при том же алгоритме
Риск ошибок	Недостижимый базовый случай, переполнение стека	Неверное условие или шаг цикла

Когда использовать:

- Задачи с древовидной структурой (обход дерева или графа, генерация перестановок, Ханойские башни).
- Когда рекурсивное решение значительно короче и понятнее.
- Когда глубина рекурсии невелика (гарантированно ограничена).

Классификация видов рекурсии

Вид рекурсии	**Описание**
Действия на спуске	Вычисления выполняются при входе в функцию (перед рекурсивным вызовом).
Действия на возврате	Вычисления выполняются после рекурсивного вызова (при возврате).
Действия на спуске и возврате	Вычисления выполняются в обеих фазах.
Каскадная (древовидная)	Каждый вызов порождает два и более рекурсивных вызова.
Концевая (tail recursion)	Рекурсивный вызов является последней операцией в функции.

Сравнение рекурсии и итерации

Критерий	**Рекурсия**	**Итерация**
Определение	Функция вызывает саму себя	Цикл (for, while)
Базовый случай	Обязателен для остановки	Условие продолжения цикла
Память	Стек зависит от глубины	Зависит от используемых структур данных
Скорость	Возможны расходы на вызовы	Может быть быстрее при том же алгоритме
Применение	Рекурсивные структуры, задачи с ветвлением	Простые повторяющиеся действия
Читаемость	Компактна для сложных алгоритмов	Линейна, понятна

Концевая рекурсия

Определение

Концевая рекурсия — когда рекурсивный вызов является **последней** операцией в функции.

```
// Концевая рекурсия
void traverse(Node* p) {
    if (p == nullptr) return;
    std::cout << p->data << " ";
    traverse(p->next);  // последняя операция
}
```

- Компилятор может оптимизировать концевую рекурсию в цикл (Tail Call Optimization). Это уменьшает глубину стека до O(1).
- Не все компиляторы выполняют оптимизацию (в C++ это не гарантируется).
- В C++11 и новее можно использовать `constexpr` для вычислений на этапе компиляции.

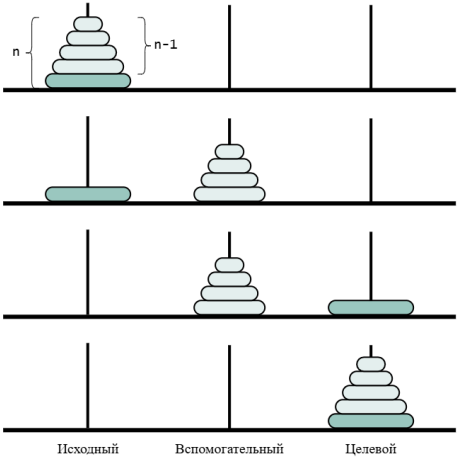
Пример 4: Ханойские башни (постановка задачи)

- Даны три стержня: A (исходный), B (вспомогательный), C (целевой).
- На стержне A находится n дисков разного размера, уложенных в порядке убывания (снизу большой, сверху малый).
- Задача: перенести все диски на стержень C, используя B как вспомогательный.
- Правила:
  1. За один раз можно перемещать только один верхний диск со стержня.
  2. Нельзя класть больший диск на меньший.



Пример 4: Ханойские башни (рекурсивный алгоритм)

- Рекурсивно перенести $n-1$ диск с исходного стержня на вспомогательный.
- Перенести оставшийся диск с исходного стержня на целевой.
- Рекурсивно перенести $n-1$ диск с вспомогательного стержня на целевой.



Пример 4: Ханойские башни (решение)

Рекурсивный алгоритм

```
void hanoi(int n, char from, char aux, char to) {
    if (n == 0)
        return;
    hanoi(n - 1, from, to, aux); // исходный -> вспомогательный
    std::cout << "Move disk " << n << " from " << from << " to " << to << std::endl;
    hanoi(n - 1, aux, from, to); // вспомогательный -> целевой
}
```

- Количество перекладываний: $2^n - 1$. Время: $\mathrm{O}(2^n)$, память стека: $\mathrm{O}(n)$.
- Классический пример задачи, которая естественно решается рекурсией.

Пример 5. Бинарный поиск: алгоритм

Постановка задачи

Дан отсортированный массив. Найти индекс элемента `x`. Если элемента нет, вернуть -1.

Алгоритм

- Если `l > r`, диапазон пуст: возвращаем `-1`.
- Иначе вычисляем индекс среднего элемента массива `m = (l+r) / 2`, где `l` и `r` — индексы первого и последнего элементов массива.
- Берем средний элемент массива `a[m]` и сравниваем его с `x`.
- Если они равны, то индекс найден и равен `m`, выход.
- Если `x < a[m]`, то рекурсивно ищем среди элементов с индексами от `l` и `m-1`.
- Иначе (`x > a[m]`) рекурсивно ищем среди элементов с индексами от `m+1` и `r`.

Бинарный поиск: решение

Рекурсивный алгоритм

```
int binarySearch(const int* a, int x, int l, int r) {
    if (l > r) return -1;
    int m = (l + r) / 2;
    if (x == a[m]) return m;
    if (x < a[m]) return binarySearch(a, x, l, m - 1);
    return binarySearch(a, x, m + 1, r);
}
```

- Сложность: $\mathrm{O}(\log n)$ — каждый раз массив делится пополам.
- Глубина рекурсии и память стека: $\mathrm{O}(\log n)$.

Вопрос: что выведет этот код?

```
#include <iostream>
void foo(int n) {
    if (n == 0) return;
    foo(n - 1);
    std::cout << n << " ";
}
int main() {
    foo(3);
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
void foo(int n) {
    if (n == 0) return;
    foo(n - 1);
    std::cout << n << " ";
}
int main() {
    foo(3);
    return 0;
}
```

Ответ: 1 2 3 (сначала спуск до 0, затем вывод при возврате)

Проблема: перегрузка для разных типов

```
int maxInt(int a, int b) { return (a > b) ? a : b; }
double maxDouble(double a, double b) { return (a > b) ? a : b; }
char maxChar(char a, char b) { return (a > b) ? a : b; }
// ... и так для каждого типа
```

- Для каждого типа нужно писать отдельную функцию (или использовать перегрузку).
- Код дублируется, трудно поддерживать.
- Что если нужен новый тип? Придётся писать ещё одну функцию.

Макросы как (плохая) альтернатива

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int main() {
    int x = 5, y = 3;
    int z = MAX(x, y);          // 5, OK
    double d = MAX(2.5, 3.7);   // 3.7, OK
    int a = 5, b = 3;
    int s = MAX(a, b) + 2;      // 7, раскрывается как ((a)>(b)?(a):(b))+2
    int c = MAX(++a, b);        // 7, побочные эффекты
    return 0;
}
```

Недостатки макросов

- Макрос не задаёт типы параметров; проверяется код после текстовой подстановки.
- Могут возникать побочные эффекты из-за многократного вычисления аргументов.
- Сложно отлаживать.

Шаблоны функций: синтаксис

Определение шаблона

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

- `template <typename T>` — объявление шаблона с параметром типа `T`.
- Можно использовать `class T` вместо `typename T` (эквивалентно).
- `T` заменяется на конкретный тип при вызове.

Шаблон функции — это «трафарет» для создания семейства функций.

Пример: шаблон max

```cpp
#include <iostream>
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
int main() {
    std::cout << max(3, 5) << std::endl;          // int
    std::cout << max(3.14, 2.71) << std::endl;    // double
    std::cout << max('a', 'z') << std::endl;      // char
    return 0;
}
```

Вывод: 5, 3.14, z

Шаблонные функции vs шаблоны функций

Шаблон функции (function template)

Это описание общего алгоритма, не скомпилированный код.

Шаблонная функция (template function)

Это конкретная функция, полученная из шаблона путём подстановки типов (инстанцирования).

Пример

`template <typename T> T max(T a, T b)` — это **шаблон функции**. `max<int>(3, 5)` — это **шаблонная функция** (инстанцирование с `int`).

Инстанцирование шаблона

- Инстанцирование — процесс создания конкретной функции из шаблона.
- Компилятор автоматически выводит типы параметров шаблона по аргументам вызова.
- Можно указать типы явно: `max<double>(3, 5)`.

```
// Вызов max(3, 5) приводит к инстанцированию:
int max(int a, int b) {
    return (a > b) ? a : b;
}
// Вызов max(3.14, 2.71) приводит к инстанцированию:
double max(double a, double b) {
    return (a > b) ? a : b;
}
```

Два этапа компиляции шаблонов

1. **Первый этап** (компиляция шаблона):
   - Проверяется синтаксис, ошибки, не зависящие от параметров.
   - Код не генерируется.
2. **Второй этап** (инстанцирование):
   - При вызове функции компилятор подставляет конкретные типы.
   - Генерируется код для каждого использованного типа.
   - Проверяется семантика для данных типов (например, наличие оператора >).

Важно

Ошибки, связанные с отсутствием операций для типа, возникают на втором этапе.

Где размещать шаблоны?

- Обычно определение шаблона размещают в заголовочном файле.
- Для неявного инстанцирования определение должно быть доступно компилятору.
- **Правило:**
  1. При многофайловой программе — обычно в заголовочном файле (`.h`).
  2. При однофайловой — в том же `.cpp`, где используется.

Почему?

Компилятору нужно знать полное определение шаблона, чтобы создать конкретную функцию для данного типа.

Специализация шаблона

- Иногда общее определение шаблона не подходит для некоторых типов.
- Например, для `const char*` сравнение указателей, а не строк.
- Можно предоставить специализацию для конкретного типа.

```
template <>
const char* max(const char* a, const char* b) {
    return (strcmp(a, b) > 0) ? a : b;
}
```

- Синтаксис: `template <>` перед определением.
- Специализация переопределяет поведение для данного типа.

Специализация: синтаксис

```
// Общий шаблон
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

// Явная специализация для const char*
template <>
const char* max(const char* a, const char* b) {
    return (strcmp(a, b) > 0) ? a : b;
}
```

- После `template <>` указываем конкретный тип вместо `T`.
- Специализация должна быть объявлена до первого использования для этого типа.

Перегрузка vs специализация

- **Перегрузка** — создание функции с тем же именем, но другим набором параметров (может быть нешаблонной).
- **Специализация** — создание конкретной версии шаблона для определённого типа.

Как выбирается функция

1. Среди подходящих перегрузок выбирается лучшее соответствие аргументам.
2. При одинаково хорошем соответствии обычная функция предпочтительнее шаблонной.
3. Если выбран шаблон, используется его явная специализация для данных типов, если она есть.

Пример

При наличии `int max(int, int)` вызов `max(3, 5)` выберет обычную функцию. Вызов `max<int>(3, 5)` обращается к шаблону.

Пример: шаблон printArray

```cpp
#include <iostream>
template <typename T>
void printArray(const T* arr, int size) {
    for (int i = 0; i < size; ++i)
        std::cout << arr[i] << " ";
    std::cout << std::endl;
}
int main() {
    int a[] = {1, 2, 3};
    double b[] = {1.1, 2.2, 3.3};
    char c[] = "abc";
    printArray(a, 3);
    printArray(b, 3);
    printArray(c, 3);
    return 0;
}
```

Пример: шаблон sumArray

```
#include <iostream>
template <typename T>
T sumArray(const T* arr, int size) {
    T sum = 0;
    for (int i = 0; i < size; ++i)
        sum += arr[i];
    return sum;
}
int main() {
    int a[] = {1, 2, 3};
    double b[] = {1.1, 2.2, 3.3};
    std::cout << sumArray(a, 3) << std::endl;
    std::cout << sumArray(b, 3) << std::endl;
    return 0;
}
```

Variadic Templates (шаблоны с переменным числом аргументов)

- Позволяют создавать шаблоны, принимающие произвольное количество аргументов.
- Пакет параметров обозначается `typename... Args`.

```
template <typename... Args>
void print(Args... args) {
    // ...
}
// Пример использования:
print(1, 2.5, "hello", 'a');
```

- Подробно изучим в следующем семестре.
- Позволяет создавать гибкие функции, например, для форматированного вывода.

Fold Expressions (C++17)

Fold expression — способ развёртывания пакета параметров шаблона с помощью операторов.

```cpp
#include <iostream>

template <typename... Args>
void example(Args... args) {
    ((std::cout << args << ' '), ...);
}

template <typename... Args>
auto sum(Args... args) {
    return (0 + ... + args);
}

int main() {
    example(1, 2.5, "hello"); // 1 2.5 hello
    std::cout << '\n';
    std::cout << sum(1.2, 2, 'a') << std::endl; // 100.2
    return 0;
}
```

Подробно изучим в следующем семестре.

Вопрос: что выведет этот код?

```
#include <iostream>
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
int main() {
    std::cout << max(5, 3.14);
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
int main() {
    std::cout << max(5, 3.14);
    return 0;
}
```

Ответ: Ошибка компиляции: неоднозначность вывода типа (int и double). Нужно явно указать тип: `max<double>(5, 3.14)` или привести оба аргумента к одному типу.

constexpr

Определение

`constexpr` позволяет выполнять вычисления при компиляции, ещё до запуска программы.

- Для переменной: начальное значение должно быть вычислено при компиляции.
- Для функции: её можно использовать в таких вычислениях. Она также может вызываться как обычная функция во время работы программы.
- `const` запрещает менять значение переменной, но не требует вычислять его при компиляции.

```
constexpr double square(double x) { return x * x; }
int main() {
    constexpr double s = square(5.0); // вычисляется на этапе компиляции

    double x = 3.0;
    constexpr double s2 = square(x); // ошибка, т.к. x не константа

    const double s3 = square(x); // может быть вычислено во время выполнения
    return 0;
}
```

constexpr

```cpp
#include <limits>
constexpr int Sum(int a, int b) {
    return a + b;
}
int main() {
    constexpr int a = Sum(2, 3);                // 5
    constexpr int n = std::numeric_limits<int>::max(); // OK в C++20
    constexpr int m = std::numeric_limits<int>::lowest();
    return 0;
}
```

Внутри constexpr-функции можно использовать условия и циклы, создавать локальные переменные и изменять их значения.
При вычислении во время компиляции нельзя, например, читать данные с клавиатуры или изменять глобальные переменные.

constexpr и рекурсия

```cpp
#include <iostream>
constexpr int factorial(const int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
int main() {
    constexpr int f = factorial(5);    // вычисляется на этапе компиляции
    std::cout << f << std::endl;

    int x;
    std::cin >> x;
    int g = factorial(x);              // вычисляется во время выполнения
    std::cout << g << std::endl;

    static_assert(factorial(5) == 120, "Wrong factorial");
    return 0;
}
```

- Рекурсивные constexpr функции допустимы.
- В constexpr-инициализаторе и static_assert допустимый вызов вычисляется при компиляции.

static_assert

Определение

static_assert — проверка условия на этапе компиляции. Если условие ложно, компиляция останавливается с сообщением об ошибке.

```cpp
#include <limits>

template <typename T>
void check() {
    static_assert(std::numeric_limits<T>::is_integer, "Только для целочисленных типов");
    // остальной код
}

int main() {
    check<int>();    // OK
    // check<double>(); // Ошибка компиляции: "Только для целочисленных типов"
    return 0;
}
```

template с константами и constexpr

Шаблоны могут иметь нетиповые параметры (константы времени компиляции).

```
template <typename T, int N>
struct Buffer {
    T data[N];
    constexpr int size() const { return N; }
};
int main() {
    Buffer<char, 1024> buf;
    constexpr int sz = buf.size(); // OK
    return 0;
}
```

constexpr-функция может вычислить константный аргумент шаблона.

Константные выражения как аргументы шаблона

```
template <int N>
class List { /* ... */ };

constexpr int sqr1(int arg) { return arg * arg; }
int sqr2(int arg) { return arg * arg; }

int main() {
    const int X = 2;
    List<sqr1(X)> mylist1;   // OK: sqr1 - constexpr
    // List<sqr2(X)> mylist2; // Ошибка: sqr2 не constexpr
   return 0;
}
```

- Нетиповые параметры шаблонов должны быть константными выражениями.
- constexpr функции позволяют вычислять такие выражения на этапе компиляции.

Итог по constexpr и static_assert

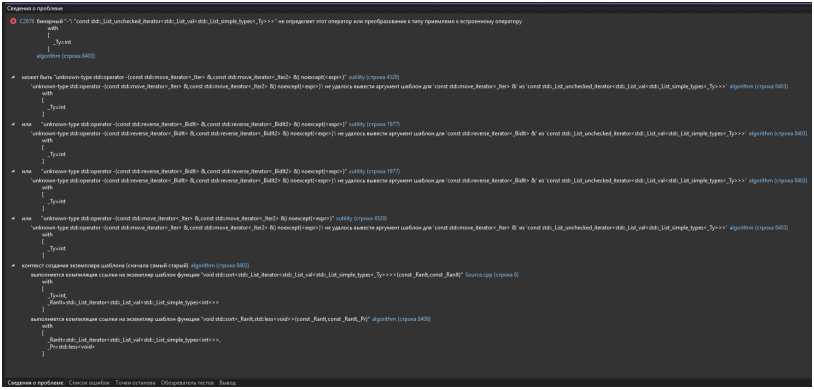
- constexpr — мощный инструмент для вычислений на этапе компиляции. Позволяет повысить производительность (константы вычисляются заранее).
- static_assert — проверка условий на этапе компиляции, улучшает диагностику.
- Используйте constexpr для функций, которые могут быть вычислены во время компиляции.
- static_assert полезен для проверки предусловий в шаблонах.

Проблема: нечитаемые ошибки компиляции

```cpp
#include <list>
#include <algorithm>
int main() {
    std::list<int> l = {1, 2, 3};
    std::sort(l.begin(), l.end()); // ошибка компиляции
    return 0;
}
```

- `std::sort` требует итераторов произвольного доступа, а `std::list` предоставляет двунаправленные.
- Ошибка возникает внутри шаблонного кода, сообщение может занимать сотни строк.
- Сложно понять, в чём причина.

Текст ошибки



Проблема

Сообщения об ошибках в шаблонах часто трудночитаемы.

requires

Определение

Языковая конструкция C++, позволяющая явно задать ограничения (constraints) на параметры шаблона. Для шаблонов функций ограничения учитываются при выборе подходящей перегрузки.

```
#include <iostream>
#include <type_traits>
template <typename TFirst, typename TSecond>
requires std::is_arithmetic_v<TFirst> && std::is_arithmetic_v<TSecond>
auto Average(TFirst x, TSecond y) {
    return (x + y) / 2;
}
int main() {
    // Average("Hello", "World"); // Ошибка компиляции
    std::cout << Average(10, 20.5) << std::endl; // OK
    return 0;
}
```

- Требования к типам параметров шаблона заданы в объявлении шаблона.
- Если ограничения не выполнены, этот шаблон не подходит для вызова.
- Здесь ограничения записаны напрямую, без отдельного именованного концепта.

Концепты (C++20)

Определение

Именованный набор требований к параметрам шаблона, проверяемых на этапе компиляции и пригодных для повторного использования.

```
template <typename T>
concept MyConcept = /* выражение, проверяемое на этапе компиляции */;
```

- Ограничения на параметры шаблона имеют собственное имя и объявляются отдельно от шаблона.
- Один и тот же концепт может повторно использоваться в разных шаблонах.
- Если тип не удовлетворяет ограничениям функции, она исключается из подходящих кандидатов. Ошибка возникает, если подходящей перегрузки не осталось.
- Улучшают сообщения об ошибках (ошибка возникает на уровне объявления, а не внутри тела).
- Стандартные концепты `std::integral`, `std::floating_point` и другие объявлены в `<concepts>`.

Синтаксис: объявление концепта

```cpp
#include <iostream>
#include <concepts>
template <typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
template <Numeric TFirst, Numeric TSecond>
auto Average(TFirst x, TSecond y) {
    return (x + y) / 2;
}
int main() {
    // Average("Hello", "World"); // Ошибка компиляции
    std::cout << Average(10, 20.5) << std::endl; // OK
    return 0;
}
```

- Выражение должно быть константным и возвращать `bool`.
- Концепт `Numeric` проверяет, является ли тип целым или вещественным.
- Можно использовать стандартные трейты (`std::is_...`) или `requires`.

requires-выражения

Позволяют описывать набор требований к выражениям над типом, включая операции, вызовы методов и требования к типам их результатов, с использованием специального синтаксиса.

```
template <typename T>
concept HasSize = requires(const T& a) {
    a.size();           // проверка наличия метода size()
    { a.size() } -> std::convertible_to<int>; // и его возвращаемого типа
};
template <typename T>
concept Iterable = requires(T a) {
    begin(a); // проверка наличия итераторов
    end(a);
};
```

convertible_to — проверяет, можно ли результат выражения неявно привести к указанному типу.

Пример: концепт HasSize

```cpp
// подключить iostream, concepts, vector, string
template <typename T>
concept HasSize = requires(const T& a) {
    a.size();
   { a.size() } -> std::convertible_to<int>;
};
template <HasSize T>
void printSize(const T& container) {
    std::cout << "Size: " << container.size() << std::endl;
}
int main() {
    std::string s = "hello";
    std::cout << std::boolalpha << HasSize<std::string> << std::endl; // true
    printSize(s); // Size: 5
    std::vector<int> v{1, 2, 3};
    std::cout << std::boolalpha << HasSize<std::vector<int>> << std::endl; // true
    printSize(v); // Size: 3
    int x = 5;
    std::cout << std::boolalpha << HasSize<int> << std::endl; // false
    // printSize(x); // ошибка: int не имеет size()
    return 0;
}
```

Примеры

```
template <typename T>
concept Size = sizeof(T) <= sizeof(int);

template <typename T> requires Size<T> // полный синтаксис
T sum(T a, T b){ return a + b;}

template <Size T> // сокращенный синтаксис
T sum(T a, T b){ return a + b;}
```

Вопрос: что произойдёт, если тип не удовлетворяет концепту?

```cpp
#include <iostream>
#include <concepts>

template <typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

template <Numeric T>
T square(T x) { return x * x; }

int main() {
    std::cout << square(5) << std::endl;         // 25
    std::cout << square(3.14) << std::endl;      // 9.8596
    // std::cout << square("hello") << std::endl; // что произойдёт?
    return 0;
}
```

Вопрос: что произойдёт, если тип не удовлетворяет концепту?

```cpp
#include <iostream>
#include <concepts>

template <typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

template <Numeric T>
T square(T x) { return x * x; }

int main() {
    std::cout << square(5) << std::endl;         // 25
    std::cout << square(3.14) << std::endl;      // 9.8596
    // std::cout << square("hello") << std::endl; // что произойдёт?
    return 0;
}
```

Ответ: Ошибка компиляции: тип `const char*` не удовлетворяет концепту `Numeric`. Сообщение об ошибке будет ясным и укажет на невыполненное ограничение.

Итоги лекции

- **Рекурсия**: определение, структура, стек вызовов, примеры (факториал, Фибоначчи, Ханойские башни). Сравнение с итерацией.
- **Шаблоны функций**: синтаксис, инстанцирование, специализация. Решение проблемы дублирования кода.
- **constexpr и static_assert**: вычисления на этапе компиляции, проверка условий.
- **Концепты (C++20)**: мотивация (улучшение ошибок), синтаксис, использование в шаблонах.

Что дальше?

- Следующая лекция: линейные списки (односвязные и двусвязные).
- На практике: рекурсивные алгоритмы, шаблоны функций.
- Готовимся к работе с динамическими структурами данных.

Вопросы?