HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 04 – Типы. Шаблоны

Глава 1

Анатомия типов и оптимизация памяти (Layout & EBO)

Семантика и механика типов

Тип в C++ — это контракт, определяющий множество допустимых операций над объектом и интерпретацию битов в памяти. В отличие от языка ассемблера, где байт — это просто байт, система типов C++ накладывает семантические ограничения (invariants), которые компилятор использует для генерации корректного машинного кода и оптимизаций.

Struct vs Class

Существует распространенное заблуждение о фундаментальных различиях между `struct` и `class`. Технически, в C++ это идентичные конструкции за исключением одного свойства: модификатора доступа по умолчанию.

> **Struct vs Class**
>
> - **class**: Все поля и наследование по умолчанию `private`.
> - **struct**: Все поля и наследование по умолчанию `public`.

Следующие два объявления генерируют идентичный layout в памяти и идентичный машинный код:

```
class A {
public:
    int x;
};

struct B {
    int x;
};
```

Выбор между ними — вопрос конвенции. Обычно `struct` используется для POD-типов (Plain Old Data), представляющих собой набор открытых полей без сложной инвариантной логики, а `class` — для сущностей, требующих инкапсуляции и поддержания инвариантов.

Инвариант ненулевого размера

Одним из фундаментальных правил объектной модели C++ является гарантия уникальности адресов (identity) для различных объектов одного типа.

> **Важно!**
>
> В C++ размер любого законченного объекта типа не может быть равен нулю. `sizeof(T)` $\geqslant$ 1.

Даже если структура не содержит полей данных, компилятор обязан выделить ей минимум 1 байт памяти.

```
struct Empty {};
static_assert(sizeof(Empty) == 1);
```

Причина ограничения

Это требование продиктовано необходимостью адресной арифметики, в частности для массивов. Рассмотрим массив из двух пустых структур:

```
Empty arr[2];
```

Если бы `sizeof(Empty)` был равен 0, то адреса `&arr[0]` и `&arr[1]` совпадали бы. Это нарушило бы логику указателей: указатель на первый элемент массива был бы неотличим от указателя на второй. Чтобы гарантировать `&arr[i]` $\neq$ `&arr[j]` при `i` $\neq$ `j`, каждый элемент должен занимать физическое место в памяти.

> **На заметку**
>
> В языке Rust существуют *Zero Sized Types (ZST)*, которые действительно занимают 0 байт. Это возможно, так как в Rust иная модель памяти и правил адресной арифметики для таких типов. В C++ это невозможно из-за гарантий обратной совместимости и модели указателей.

Проблема накладных расходов (Memory Overhead)

Инвариант ненулевого размера создает проблему при композиции объектов. Рассмотрим классический пример контейнера, такого как `std::vector`. Вектор должен хранить аллокатор для управления памятью. Аллокатор по умолчанию `std::allocator` не имеет состояния (stateless), то есть не содержит полей данных.

Наивная реализация вектора могла бы выглядеть так:

```
template <typename T, typename Alloc = std::allocator<T>>
class Vector {
    T* begin_;      // 8 байт (на 64-бит)
    T* end_;        // 8 байт
```

```
    T* cap_;        // 8 байт
    Alloc alloc_;   // 1 байт (минимум)
    // + 7 байт padding для выравнивания
};
```

В 64-битной архитектуре указатели требуют выравнивания по границе 8 байт. Если структура содержит три указателя (24 байта) и поле типа `Alloc` (1 байт), компилятор вынужден добавить 7 байт padding'а, чтобы размер структуры был кратен выравниванию самого строгого поля (8 байт).

Итоговый размер: $24 + 1 + 7 = 32$ байта. Мы платим 8 байт памяти за хранение объекта, который логически пуст. Это недопустимый overhead для системных библиотек.

Empty Base Optimization (EBO)

Стандарт C++ предоставляет исключение из правила ненулевого размера. Оно применимо только к базовым классам.

> **Empty Base Optimization (EBO)**
>
> Если пустой класс используется в качестве базового класса, компилятору разрешено не выделять под него отдельную память, при условии, что это не нарушает требование уникальности адресов.

Если мы перепишем `Vector`, используя наследование от аллокатора, размер уменьшится:

```
template <typename T, typename Alloc>
class Vector : private Alloc { // Наследование вместо композиции
    T* begin_;
    T* end_;
    T* cap_;
};
```

Здесь размер `Vector` составит ровно 24 байта. Базовый класс `Alloc` "схлопывается" и имеет нулевой размер внутри layout'а наследника.

Для реализации этого паттерна в стандартной библиотеке долгое время использовался вспомогательный шаблон `std::compressed_pair` (или внутренние реализации типа `__compressed_`, который наследовался от одного или обоих типов, если они пусты.

Ограничения EBO и коллизии адресов

EBO не применяется, если нарушается инвариант уникальности адресов подобъектов. Если первый член структуры имеет тот же тип, что и пустая база, EBO отключается.

```
struct Empty {};

struct BadEBO : Empty {
    Empty e; // Первое поле совпадает по типу с базой
};
```

В данном случае:

1. Адрес объекта `BadEBO` совпадает с адресом его базы `Empty`.
2. Адрес первого поля `e` также совпадает с адресом начала структуры.

Если бы EBO сработало, то база и поле `e` имели бы один и тот же адрес. В C++ два *разных* объекта (базовый подобъект и поле-член) одного типа не могут иметь одинаковый адрес. Компилятор вынужден добавить отступ (padding) для поля `e`.

Результат: `sizeof(BadEBO)` равен 2 (1 байт под базу + 1 байт под поле, без учета выравнивания).

Современная оптимизация: атрибут [[no_unique_address]]

С выходом стандарта C++20 необходимость в трюках с наследованием (вроде `compressed_pair`) отпала. Появился атрибут `[[no_unique_address]]`.

Этот атрибут сообщает компилятору, что данное поле не обязательно должно иметь уникальный адрес среди других полей, если оно пустое. Фактически это EBO для композиции.

```
struct Vector20 {
    [[no_unique_address]] std::allocator<int> alloc; // Занимает 0 байт
    int* begin;
    int* end;
    int* cap;
};
```

`sizeof(Vector20)` будет равен 24 байтам (на 64-битной системе). Компилятор переиспользует padding, имеющийся в структуре, или просто не выделяет место под поле.

Это позволяет писать более понятный код, используя композицию вместо приватного наследования, сохраняя при этом эффективность памяти.

Визуализация Layout

Сравним расположение в памяти для разных подходов (схематично для 64-bit):

1. Наивная композиция (32 байта):

`[ ptr (8) ][ ptr (8) ][ ptr (8) ][ alloc (1) ][ padding (7) ]`

2. EBO через наследование (24 байта):

`[ Alloc (0) | ptr (8) ][ ptr (8) ][ ptr (8) ]`

Базовый класс Alloc виртуально накладывается на начало структуры, не занимая места.

3. Атрибут [[no_unique_address]] (24 байта):

`[ ptr (8) ][ ptr (8) ][ ptr (8) ] (alloc field is optimized out)`

Влияние виртуальных функций

Важно помнить, что наличие хотя бы одной виртуальной функции делает класс непустым, даже если у него нет полей данных.

```
struct VEmpty {
    virtual ~VEmpty() = default;
};
static_assert(sizeof(VEmpty) == 8); // На 64-bit
```

Причина: Объект должен хранить указатель на таблицу виртуальных функций (`vptr`), который добавляется компилятором неявно. EBO к таким классам неприменимо в том смысле, что размер не станет нулевым — он останется равным размеру указателя.

Резюме раздела

- Пустые типы имеют размер 1 байт для обеспечения уникальности адресов.
- Композиция с пустыми типами вызывает overhead из-за выравнивания.
- **EBO** (через наследование) позволяет обнулить размер пустой базы.
- **C++20 [[no_unique_address]]** позволяет достичь того же эффекта при композиции.
- При совпадении типов базы и первого поля оптимизация отключается.

Глава 2

Шаблонная магия: NTTP и дедукция типов

Система шаблонов C++ выходит далеко за рамки простой подстановки типов. Это полноценный тьюрин-полный язык, исполняемый на этапе компиляции. В этой главе мы рассмотрим механизмы параметризации значений (NTTP), революционные изменения C++20 в работе со строковыми литералами в шаблонах, а также тонкости вывода типов (deduction), которые часто становятся источником неочевидных ошибок.

Терминология: class vs typename

В объявлении параметров шаблона ключевые слова `class` и `typename` являются полностью взаимозаменяемыми.

```
1 template <class T> void foo(T t);      // Вариант 1
2 template <typename T> void bar(T t);   // Вариант 2
```

С точки зрения компилятора разницы нет никакой. Исторически `class` появился раньше, но `typename` считается более семантически верным, так как параметром может быть не только класс, но и примитивный тип (например, `int`). В современном C++ предпочтение отдается `typename`, однако в существующем коде вы встретите оба варианта.

> **На заметку**
>
> Ключевое слово `struct` в параметрах шаблона использовать нельзя.

Non-Type Template Parameters (NTTP)

Шаблоны могут принимать не только типы, но и значения. Это называется *Non-Type Template Parameters* (NTTP). До стандарта C++20 список допустимых типов для NTTP был строго ограничен:

- Целочисленные типы (int, long, char, bool и т.д.).
- Перечисления (enum).
- Указатели и ссылки на объекты со статической продолжительностью жизни (редко используется).
- `std::nullptr_t`.

Классический пример использования NTTP — контейнер `std::array`, размер которого должен быть известен на этапе компиляции:

```
template <typename T, size_t N>
struct Array {
    T data[N];
};

Array<int, 5> arr; // N = 5 подставляется при компиляции
```

Значение NTTP является константой времени компиляции. Попытка передать runtime-значение приведет к ошибке компиляции.

Революция C++20: Structural Types

До C++20 передать строковый литерал или объект пользовательского класса в качестве параметра шаблона было невозможно.

```
// C++17: Ошибка компиляции
template <auto S> struct Wrapper {};
Wrapper<"Hello"> w; // Строковые литералы запрещены
```

C++20 ослабил это ограничение, введя понятие *Structural Types*. Теперь в качестве NTTP можно использовать классы, если они удовлетворяют ряду требований (в основном: все поля `public` и сами являются структурными типами/примитивами).

Это открыло возможность передавать строки в шаблоны, предварительно обернув их в структурный тип `fixed_string`.

Реализация fixed_string

Для передачи строки `"text"` в шаблон, она должна быть скопирована в буфер внутри структурного типа во время компиляции.

```
template<size_t N>
struct fixed_string {
    char buf[N + 1]{}; // Публичный массив (обязательно)

    constexpr fixed_string(char const* s) {
        for (unsigned i = 0; i != N; ++i) buf[i] = s[i];
    }

    // Оператор для удобного приведения к строке
    constexpr operator char const*() const { return buf; }
};

// Deduction guide для вывода N из длины литерала
template<size_t N> fixed_string(char const (&)[N]) -> fixed_string<N - 1>;
```

Теперь мы можем использовать этот тип как NTTP:

```cpp
template <fixed_string Str>
struct Logger {
    void Log() {
        std::cout << "Prefix: " << Str.buf << "\n";
    }
};

int main() {
    // Работает в C++20!
    // Компилятор создает уникальный тип Logger для строки "Debug"
    Logger<"Debug"> logger;
    logger.Log();
}
```

Case Study: Compile Time Regular Expressions (CTRE)

Главное применение расширенных NTTP — библиотека CTRE (Compile Time Regular Expressions). Традиционный `std::regex` парсит строку регулярного выражения в рантайме, строит конечный автомат (DFA/NFA) в динамической памяти, что крайне медленно.

Благодаря возможности передать строку паттерна как шаблонный параметр, мы можем:

1. Распарсить регулярное выражение на этапе компиляции (`constexpr`).
2. Построить конечный автомат как набор типов или switch-case конструкций.
3. Сгенерировать оптимизированный машинный код под конкретный паттерн.

Примерный синтаксис (концептуально):

```cpp
// Паттерн передается как NTTP
auto match = ctre::match<"([0-9]+)-([a-z]+)">("123-abc");
```

> **Важно!**
>
> Здесь строка `"([0-9]+)-([a-z]+)"` обрабатывается компилятором. Если в регулярном выражении есть синтаксическая ошибка, программа **не скомпилируется**. Это дает гарантию корректности регулярных выражений до запуска программы.

Производительность такого решения на порядки выше `std::regex` и сравнима с лучшими JIT-движками (PCRE, RE2), но без оверхеда на инициализацию.

Вывод типов (Template Argument Deduction)

Компилятор умеет выводить шаблонные аргументы из аргументов функции. Однако этот механизм не производит неявных преобразований типов (implicit conversions), что часто сбивает с толку новичков.

Конфликт типов в std::max

Рассмотрим классическую ошибку:

```
void test() {
    int a = 42;
    unsigned int b = 100;

    // ОШИБКА КОМПИЛЯЦИИ:
    // deduced conflicting types for parameter 'T' ('int' vs 'unsigned int')
    auto m = std::max(a, b);
}
```

Шаблон `std::max` объявлен как:

```
template <typename T>
const T& max(const T& a, const T& b);
```

Компилятор видит первый аргумент `int` и выводит `T = int`. Затем видит второй аргумент `unsigned int` и выводит `T = unsigned int`. Возникает конфликт. Компилятор не имеет права самовольно решать, какой тип "главнее" или шире, так как это может привести к потере данных (знаковости или точности).

Решения:

1. Явное указание типа (отключает вывод):

```
std::max<int>(a, b); // b приводится к int (опасно переполнением)
// или
std::max<unsigned>(a, b); // a приводится к unsigned
```

2. Приведение аргументов (предпочтительно):

```
std::max(static_cast<unsigned>(a), b);
```

3. Использование `std::common_type` (C++20 подход): Если вы пишете свой шаблон, можно использовать трейт для вычисления общего типа.

CTAD: Class Template Argument Deduction

До C++17 при создании экземпляра шаблонного класса мы обязаны были указывать типы:

```
std::pair<int, double> p(1, 2.5); // C++14
```

Начиная с C++17, компилятор умеет выводить параметры шаблона класса из аргументов конструктора. Это называется CTAD.

```
std::pair p(1, 2.5); // p имеет тип std::pair<int, double>
std::vector v = {1, 2, 3}; // v имеет тип std::vector<int>
```

На заметку

CTAD работает только если вы создаете объект класса напрямую. Он **не работает** для алиасов типов (`using MyVec = std::vector; MyVec v = ...; // Ошибка`).

Deduction Guides

Иногда автоматический вывод типов работает не так, как задумано, особенно при работе со ссылками. Стандартные правила вывода (decay) склонны отбрасывать ссылочность и константность, превращая типы в значения.

Рассмотрим кастомный класс пары:

```
template <typename T, typename U>
struct MyPair {
    T first;
    U second;

    MyPair(const T& t, const U& u) : first(t), second(u) {}
};
```

Использование CTAD:

```
int x = 10;
const int& rx = x;

MyPair p(rx, rx);
// T выведется как int, U как int.
// Поля first и second будут копиями x.
```

Если мы хотим, чтобы `MyPair` мог хранить ссылки, нам нужно подсказать компилятору, как выводить типы. Для этого используются *Deduction Guides* (руководства по выводу).

Синтаксис deduction guide (пишется вне класса, как свободная функция):

```
// deduction-guide:
// Если конструктор вызывается с аргументами T и U,
// выводим тип класса как MyPair<T, U>.
template <typename T, typename U>
MyPair(T, U) -> MyPair<T, U>;
```

Этот гайд, по сути, стандартный. Но предположим, мы хотим изменить поведение и форсировать создание ссылок для определенных ситуаций (что в общем случае опасно, но показательно).

Создадим гайд, который сохраняет ссылочность аргументов:

```
// Опасный гайд: захватывает ссылки
template <typename T, typename U>
MyPair(const T&, const U&) -> MyPair<const T&, const U&>;
```

Теперь:

```
int x = 5;
MyPair p(x, x);
// Сработает гайд. T=int, U=int.
// Результат вывода -> MyPair<const int&, const int&>.
// p.first теперь ссылка на x.
```

Важно!

Deduction Guides часто необходимы, когда тип конструктора отличается от типа хранения. Например, конструктор принимает итераторы `It, It`, а класс должен быть `vector<typename iterator_traits<It>::value_type>`. Без явного гайда компилятор не сможет совершить этот скачок логики.

Резюме раздела

- `class` и `typename` в параметрах шаблона эквивалентны.
- NTTP позволяют передавать значения. С C++20 можно передавать строки через structural types (например, `fixed_string`).
- CTRE использует NTTP строки для compile-time парсинга регулярок.
- Вывод типов функций (Deduction) не делает неявных приведений (`max(int, unsigned)` — ошибка).
- CTAD позволяет опускать угловые скобки при создании классов: `pair(1, 2)`.
- Deduction Guides позволяют корректировать логику CTAD, особенно для сложных преобразований (итераторы -> контейнер).

Глава 3

Метапрограммирование: Traits и Control Flow

Метапрограммирование в C++ — это написание программ, которые выполняются компилятором и манипулируют другими программами (типами и константами) как данными. Фундаментом этой парадигмы являются *Traits* (свойства типов) и механизмы управления потоком компиляции.

Трейты (Traits): API для типов

Трейт (trait) — это класс-шаблон, который ничего не делает в рантайме, но предоставляет информацию о типе на этапе компиляции. Это "мета-функция", принимающая тип и возвращающая значение (обычно `bool`) или другой тип.

Стандартная библиотека предоставляет богатый набор трейтов в заголовке `<type_traits>`: `std::is_integral`, `std::remove_reference`, `std::enable_if` и т.д.

Механика: Частичная специализация

В основе работы трейтов лежит механизм частичной специализации шаблонов. Рассмотрим реализацию простейшего трейта `is_pointer`, который определяет, является ли тип указателем.

```
// 1. Базовый шаблон (General Case)
// По умолчанию считаем, что T — не указатель.
template <typename T>
struct is_pointer {
    static constexpr bool value = false;
};

// 2. Частичная специализация для указателей
// Если тип совпадает с паттерном T*, выбирается эта версия.
template <typename T>
struct is_pointer<T*> {
    static constexpr bool value = true;
};
```

При использовании `is_pointer<int*>::value`:

1. Компилятор видит, что `int*` подходит под специализацию `T*` (где `T = int`).
2. Специализация более специфична, чем общий шаблон.
3. Выбирается версия с `value = true`.

Проблема обобщенного доступа: Iterator Traits

Рассмотрим классическую проблему написания обобщенного алгоритма. Мы пишем функцию, которая должна работать как с итераторами контейнеров (`std::vector<int>::iterator`), так и с обычными указателями (`int*`).

```
template <typename Iter>
void algorithm(Iter it) {
    // Нам нужно создать временную переменную того типа,
    // на который указывает итератор.
    typename Iter::value_type temp = *it; // ОШИБКА для int*
}
```

У класса итератора есть вложенный тип `value_type`. Но у встроенного типа `int*` нет вложенных типов. Это делает невозможным единообразное обращение к свойствам типа напрямую.

Решение: Слой косвенности

Для решения этой проблемы стандарт вводит сущность `std::iterator_traits`. Это посредник, который унифицирует интерфейс доступа к свойствам итераторов.

Мы можем реализовать его самостоятельно:

```
// 1. Общая версия: делегирует запрос самому итератору
template <typename Iter>
struct iterator_traits {
    using value_type = typename Iter::value_type;
    using pointer    = typename Iter::pointer;
    // ... другие свойства
};

// 2. Специализация для сырых указателей (T*)
template <typename T>
struct iterator_traits<T*> {
    using value_type = T;   // Для int* значением является int
    using pointer    = T*;
    // ...
};
```

Теперь наш алгоритм работает корректно для любых видов итераторов:

```
template <typename Iter>
void algorithm(Iter it) {
    // Работает и для std::vector::iterator, и для int*
```

```
    using T = typename std::iterator_traits<Iter>::value_type;
    T temp = *it;
}
```

Синтаксический ад: typename и template

При написании шаблонов вы неизбежно столкнетесь с ошибками парсинга, требующими ключевых слов `typename` и `template` в неочевидных местах.

Зависимые имена типов (Dependent Types)

Когда компилятор парсит шаблон, он еще не знает конкретного типа `T`. Рассмотрим выражение:

```
T::x * y;
```

Как это интерпретировать?

1. Это умножение статической переменной `T::x` на переменную `y`?
2. Или это объявление указателя `y` на тип `T::x`?

В C++ принято правило: **по умолчанию любое зависимое имя считается значением (переменной или функцией)**. Если вы хотите сказать компилятору, что это тип, вы *обязаны* использовать ключевое слово `typename`.

```
template <typename T>
void foo() {
    // Ошибка: компилятор думает, что const_iterator — это статическое поле
    // T::const_iterator * it;

    // Правильно:
    typename T::const_iterator * it;
}
```

Зависимые шаблоны (Dependent Templates)

Аналогичная проблема возникает, когда у зависимого типа есть шаблонный метод.

```
template <typename T>
void call_wrapper(T& obj) {
    // Ошибка: < интерпретируется как оператор "меньше"
    // obj.foo<int>();

    // Правильно:
    obj.template foo<int>();
}
```

Без слова `template` компилятор распарсит строку как `(obj.foo) < (int) > ...`.

Compile-Time Control Flow

В рантайме мы используем `if`, чтобы выбрать ветку исполнения. В шаблонах нам часто нужно выбрать ветку *компиляции* в зависимости от свойств типа.

Tag Dispatching (Диспетчеризация по тегам)

До C++17 стандартным способом выбора алгоритма была перегрузка функций. Рассмотрим `std::advance`, которая сдвигает итератор на N шагов. Для векторов это просто `it += n` ($O(1)$), для списков — цикл ($O(N)$).

Мы используем пустые структуры-теги для выбора перегрузки:

```
// Реализация для Random Access (быстрая)
template <typename Iter>
void advance_impl(Iter& it, int n, std::random_access_iterator_tag) {
    it += n;
}

// Реализация для остальных (медленная)
template <typename Iter>
void advance_impl(Iter& it, int n, std::input_iterator_tag) {
    while (n--) ++it;
}

// Фасад
template <typename Iter>
void advance(Iter& it, int n) {
    // Извлекаем категорию итератора и создаем объект-тег
    using category = typename std::iterator_traits<Iter>::iterator_category;
    advance_impl(it, n, category{});
}
```

If Constexpr (C++17)

C++17 ввел конструкцию `if constexpr`, которая позволяет писать условную логику в одной функции. Компилятор вычисляет условие и инстанцирует **только одну** ветку. Вторая ветка отбрасывается (discarded statement).

```
template <typename Iter>
void advance(Iter& it, int n) {
    using cat = typename std::iterator_traits<Iter>::iterator_category;

    if constexpr (std::is_base_of_v<std::random_access_iterator_tag, cat>) {
        it += n; // Компилируется только для RA итераторов
    } else {
        while (n--) ++it;
    }
}
```

Преимущество `if constexpr` в том, что код внутри отброшенной ветки не обязан быть полностью валидным для текущего типа `T`, достаточно лишь синтаксической корректно-

сти. Например, выражение `it += n` вызвало бы ошибку компиляции для `std::list`, если бы мы использовали обычный `if`, но с `constexpr` эта проверка отключается.

Ловушка безусловного static_assert

При использовании `if constexpr` часто возникает соблазн написать проверку на "неподдерживаемый тип" в ветке `else`.

> **Важно!**
>
> Следующий код **не скомпилируется никогда**, даже если мы заходим в ветку `true`.

```
template <typename T>
void process(T t) {
    if constexpr (std::is_integral_v<T>) {
        // ...
    } else {
        // ОШИБКА: static_assert срабатывает всегда!
        static_assert(false, "Type not supported");
    }
}
```

Причина: Согласно стандарту, если `static_assert` не зависит от шаблонного параметра (в данном случае `false` — это константа), он срабатывает на этапе разбора шаблона, до инстанциации. Компилятор видит "всегда ложь" и останавливает сборку.

Решение: Dependent False

Чтобы `static_assert` срабатывал только при инстанциации ветки `else`, условие должно зависеть от `T`.

```
// Хелпер, который всегда равен false, но зависит от типа
template <typename T>
struct always_false : std::false_type {};

template <typename T>
inline constexpr bool always_false_v = always_false<T>::value;

template <typename T>
void process(T t) {
    if constexpr (std::is_integral_v<T>) {
        // ...
    } else {
        // Теперь проверка отложена до инстанциации этой ветки
        static_assert(always_false_v<T>, "Type not supported");
    }
}
```

В C++23 планируется разрешить `static_assert(false)` в таких контекстах, но для текущих стандартов (C++17/20) использование идиомы `always_false` обязательно.

Резюме раздела

- **Traits** позволяют получать свойства типов через специализацию шаблонов.
- **iterator_traits** — необходимый слой абстракции для поддержки встроенных типов (указателей) в обобщенном коде.
- Используйте **typename** перед типами, зависящими от параметра шаблона, и **template** перед шаблонными методами зависимых объектов.
- **Tag Dispatching** — старый, но рабочий способ выбора реализации через перегрузки.
- **if constexpr** — современный способ compile-time ветвления.
- Остерегайтесь `static_assert(false)` в ветках `if constexpr`, делайте условие зависимым от `T`.

Глава 4

Concepts: Новая эра ограничений (C++20)

До стандарта C++20 шаблоны страдали от фундаментальной проблемы: отсутствие явного интерфейса для типов. Шаблонная функция принимала "что угодно", и проверка совместимости типа с алгоритмом происходила лишь в момент инстанциации тела функции. Это приводило к двум последствиям: чудовищным сообщениям об ошибках и сложным техникам метапрограммирования (SFINAE) для перегрузки функций.

C++20 представил механизм **Concepts** (Концепты) — способ явно декларировать требования к шаблонным аргументам.

Проблема SFINAE и читаемость ошибок

Рассмотрим классический пример использования `std::map`. Ключ карты должен быть упорядочиваемым (иметь оператор <).

```
struct Key {
    int v;
    // Нет оператора <
};

void test() {
    std::map<Key, int> m;
    m[Key{1}] = 10;
}
```

Без концептов компилятор начнет инстанцировать шаблон `std::map`, затем внутреннее дерево, затем узлы дерева, и где-то на глубине 15-го вызова внутри STL обнаружит, что выражение `a < b` невозможно. В результате программист получает сотни строк диагностики, указывающих на внутренности библиотеки, а не на строку `m[Key{1}] = 10`.

Для решения этой проблемы (а также для выбора перегрузок) ранее использовалась техника **SFINAE** (Substitution Failure Is Not An Error) и метафункция `std::enable_if`.

Код "старой школы" выглядел так:

```
template <typename T,
          typename = std::enable_if_t<std::is_integral_v<T>>>
void foo(T t) { /*...*/ }
```

Это синтаксически тяжело, неочевидно для чтения и увеличивает время компиляции.

Определение Концепта

Концепт — это именованный набор требований к типу. Технически это шаблонная переменная типа `bool`, вычисляемая на этапе компиляции, но с особым синтаксисом и семантикой.

```
template <typename T>
concept Integral = std::is_integral_v<T>;
```

Requires-выражение

Самым мощным инструментом создания концептов является `requires-expression`. Оно позволяет проверить валидность произвольного кода без его выполнения. Если код внутри блока `requires` некорректен (нет метода, несовместимые типы), выражение возвращает `false`, не вызывая ошибки компиляции.

Синтаксис:

```
requires (параметры) {
    требования;
}
```

Рассмотрим создание концепта `Hashable`. Тип `T` является хешируемым, если:

1. Существует специализация `std::hash<T>`.
2. У объекта хешера есть оператор вызова, принимающий `T`.
3. Результат вызова конвертируем в `std::size_t`.

```
#include <concepts>
#include <functional>

template <typename T>
concept Hashable = requires(T a) {
    // 1. Простое требование: выражение должно быть валидным
    std::hash<T>{}(a);

    // 2. Составное требование (Compound Requirement)
    // Проверяет валидность + тип возвращаемого значения
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};
```

Здесь `std::convertible_to` — это стандартный концепт из библиотеки `<concepts>`.

Использование Концептов

C++20 предоставляет три способа наложить ограничение на шаблон. Все они эквивалентны по смыслу, выбор зависит от стиля и сложности требований.

1. Requires clause (Предложение requires)

Наиболее гибкий способ. Ключевое слово `requires` ставится после списка шаблонов или после сигнатуры функции.

```
template <typename T>
requires Hashable<T>
void process(T key) {
    // ...
}

// Или trailing requires clause (полезно для методов класса)
template <typename T>
void process(T key) requires Hashable<T> {
    // ...
}
```

2. Ad-hoc type constraint

Имя концепта используется вместо `typename` или `class` в списке параметров. Это самый распространенный стиль.

```
template <Hashable T>
class HashSet {
    // ...
};
```

Если `T` не удовлетворяет `Hashable`, компилятор выдаст короткую и ясную ошибку **в месте инстанциации**: "constraints not satisfied for class template HashSet".

3. Terse syntax (Сокращенный синтаксис)

Позволяет вообще избавиться от слова `template`. Используется `auto` в сочетании с именем концепта в списке аргументов функции.

```
// Эквивалентно template <Hashable T> void foo(T key);
void foo(Hashable auto key) {
    // ...
}
```

Это делает шаблонные функции визуально почти неотличимыми от обычных функций, снижая порог входа.

Концепты и перегрузка функций

Концепты участвуют в разрешении перегрузок. Компилятор выбирает функцию с **наиболее строгим** (more constrained) ограничением. Это позволяет элегантно заменять `std::enable_if` и Tag Dispatching.

```cpp
template <typename T>
concept RandomAccess = requires(T t, int n) { t + n; t[n]; };

template <typename T>
concept Bidirectional = requires(T t) { --t; };

// Версия 1: Для любых итераторов
template <typename Iter>
void advance(Iter& it, int n) {
    while(n--) ++it;
}

// Версия 2: Только для Random Access
// Эта версия "более ограничена" (subsumes), поэтому компилятор выберет её,
// если Iter удовлетворяет RandomAccess.
template <RandomAccess Iter>
void advance(Iter& it, int n) {
    it += n;
}
```

В отличие от специализации шаблонов, здесь не нужно наследование или сложная логика частичного упорядочивания. Если концепт A включает в себя требования концепта B, то A считается "более строгим".

Использование в if constexpr

Поскольку концепт — это `constexpr bool`, его можно использовать внутри функции для условной компиляции. Это замена SFINAE, когда нам нужно изменить поведение внутри одного тела функции.

```cpp
template <typename T>
void serialize(const T& obj) {
    if constexpr (requires { obj.to_json(); }) {
        // Если есть метод to_json(), используем его
        std::cout << obj.to_json();
    } else if constexpr (std::is_integral_v<T>) {
        // Если это число
        std::cout << std::to_string(obj);
    } else {
        // Fallback
        static_assert(always_false_v<T>, "Cannot serialize type");
    }
}
```

Это значительно чище, чем написание трех разных перегрузок с `enable_if`.

Резюме раздела

- **Concepts** решают проблему читаемости ошибок шаблонов и документирования интерфейсов.
- `requires-expression` позволяет проверить компилируемость кода (наличие методов, операторов) без его выполнения.
- Синтаксис варьируется от verbose (`requires clause`) до terse (`Concept auto`).
- Концепты позволяют перегружать функции по свойствам типов без использования SFINAE хаков.
- Стандартная библиотека C++20 вводит заголовок `<concepts>` с набором готовых предикатов (`std::integral`, `std::copyable`, `std::predicate` и др.).

Глава 5

Лямбда-выражения: От сахара до мета-типов

Лямбда-выражения, появившиеся в C++11, часто воспринимаются как синтаксический сахар для создания анонимных функций. Однако с точки зрения языка это гораздо более мощный механизм, создающий уникальные типы (closure types) с состоянием. В современном C++ (C++20/23) лямбды эволюционировали в инструмент метапрограммирования, позволяющий манипулировать типами и контекстами на этапе компиляции.

Анатомия замыкания (Closure Type)

Когда компилятор встречает лямбда-выражение, он генерирует уникальный класс (функтор), который называется *closure type*.

```
int x = 10;
auto l = [x](int y) { return x + y; };
```

Этот код разворачивается компилятором примерно в следующую структуру:

```
class __lambda_unique_name {
    int x; // Захваченная переменная (по значению)

public:
    __lambda_unique_name(int x_val) : x(x_val) {}

    // Оператор вызова по умолчанию const!
    int operator()(int y) const {
        return x + y;
    }
};
```

> **Важно!**
>
> Каждое лямбда-выражение имеет свой **уникальный тип**, даже если их сигнатуры и тела полностью совпадают.

```cpp
auto l1 = []{};
auto l2 = []{};
static_assert(!std::is_same_v<decltype(l1), decltype(l2)>); // Типы разные
```

По этой причине нельзя объявить переменную типа "лямбда" без `auto` или шаблона, если не использовать type-erasure обертки вроде `std::function`.

Механика захвата (Captures)

Список захвата `[]` определяет, какие переменные из окружающего контекста будут доступны внутри лямбды и как именно они будут храниться в объекте замыкания.

Reference vs Value

- `[x]`: Создает копию x внутри объекта лямбды (поле класса).
- `[&x]`: Сохраняет ссылку (технически — указатель) на переменную в стеке.
- `[=]`: Копирует все используемые внешние переменные.
- `[&]`: Захватывает все используемые внешние переменные по ссылке.

> **Важно!**
>
> Захват по ссылке `[&]` опасен висячими ссылками (dangling references). Если время жизни лямбды превышает время жизни захваченных переменных (например, лямбда возвращается из функции или передается в другой поток), программа падает или работает некорректно.

Ловушка неявного захвата this

При использовании `[=]` внутри метода класса программисты часто ожидают, что члены класса будут скопированы. Это ошибка.

```cpp
struct Processor {
    std::vector<int> data;

    auto get_filter() {
        // ОШИБКА: [=] захватывает this, а не копию data!
        return [=](int val) {
            // Эквивалентно this->data.size()
            return val > data.size();
        };
    }
};
```

Если объект `Processor` будет уничтожен, а лямбда продолжит жить, обращение к `data` приведет к Use-After-Free, так как лямбда хранит сырой указатель `this`, а не копию вектора. Для решения этой проблемы в C++17 ввели явный захват `[*this]`, который копирует объект целиком.

Init-capture и Move-semantics (C++14)

До C++14 было невозможно захватить move-only типы (например, `std::unique_ptr`) в лямбду, так как захват по значению требовал копирования.

C++14 ввел *generalized lambda capture* (init-capture), позволяющий объявлять новые переменные прямо в списке захвата.

```
auto make_callback() {
    auto ptr = std::make_unique<int>(42);

    // Перемещаем ptr внутрь лямбды.
    // Внешний ptr становится null.
    return [u = std::move(ptr)]() {
        std::cout << *u << "\n";
    };
}
```

Здесь `u` — это новое поле внутри сгенерированного класса, инициализируемое результатом `std::move(ptr)`. Тип `u` выводится автоматически.

Ключевое слово mutable

По умолчанию `operator()` у сгенерированного класса помечен как `const`. Это означает, что лямбда не может изменять свои захваченные по значению переменные (так как они являются полями класса).

Чтобы разрешить модификацию внутреннего состояния, используется ключевое слово `mutable`.

```
int main() {
    int counter = 0;
    // mutable убирает const с operator()
    auto generator = [counter]() mutable {
        return ++counter; // Изменяем внутреннюю копию, а не внешнюю переменную
    };

    std::cout << generator(); // 1
    std::cout << generator(); // 2
    std::cout << counter;     // 0 (внешняя переменная не менялась)
}
```

Без `mutable` компилятор выдал бы ошибку на строке `++counter`, так как мы пытаемся изменить поле внутри константного метода.

Конвертация в указатель на функцию

Лямбды, не имеющие состояния (пустой список захвата `[]`), обладают особым свойством: они могут быть неявно преобразованы в сырой указатель на функцию.

```cpp
using FuncPtr = int(*)(int, int);

// Лямбда без захвата
auto l = [](int a, int b) { return a + b; };

// Неявная конвертация
FuncPtr p = l;
```

Это работает, потому что для stateless лямбд компилятор генерирует статический метод внутри класса-замыкания.

Хак с унарным плюсом

Иногда нужно форсировать выбор перегрузки функции, принимающей указатель на функцию, а не шаблон. Для этого используется унарный плюс перед лямбдой.

```cpp
// +l принудительно вызывает operator void(*)()
auto ptr = +[]() { std::cout << "Pure lambda"; };
static_assert(std::is_pointer_v<decltype(ptr)>);
```

Templated Lambdas (C++20)

В C++14 появились *generic lambdas* с параметрами `auto`. Однако у них есть недостаток: внутри тела лямбды нет легкого доступа к самому типу `T`, выведенному из `auto`.

```cpp
// C++14
auto l = [](auto x) {
    // Как получить тип элементов, если x - это std::vector<T>?
    using T = typename decltype(x)::value_type; // Громоздко
};
```

C++20 разрешил явные шаблонные параметры для лямбд:

```cpp
// C++20
auto l = []<typename T>(const std::vector<T>& vec) {
    T val = vec[0]; // T доступен напрямую
    std::cout << typeid(T).name();
};
```

Это также позволяет накладывать ограничения (Concepts) на аргументы лямбды.

Unevaluated Context: Лямбды в типах

До C++20 лямбды нельзя было использовать в *unevaluated operands* (таких как `decltype` или `sizeof`). C++20 снял это ограничение, что позволяет использовать лямбды для создания компараторов прямо в объявлении типа контейнера.

```cpp
// Custom Comparator для set без написания отдельной структуры
using MySet = std::set<int, decltype([](int a, int b) {
    return a > b; // Сортировка по убыванию
})>;

int main() {
    MySet s; // Работает в C++20
    s.insert(1);
    s.insert(2);
    // s содержит {2, 1}
}
```

Так как каждая лямбда имеет уникальный тип, `decltype` корректно пробрасывает этот тип в шаблон `std::set`. А поскольку лямбда без захвата является *default constructible* (начиная с C++20), контейнер может создать экземпляр компаратора.

Передача и хранение лямбд

Как принимать лямбду в функцию? Есть три основных стратегии, каждая со своей ценой.

1. Шаблон (Zero Overhead)

```cpp
template <typename F>
void run(F&& f) { f(); }
```

- **Плюсы:** Максимальная производительность. Тело лямбды инлайнится. Нет аллокаций.
- **Минусы:** Код функции `run` дублируется для каждой новой лямбды (code bloat). Требует нахождения реализации в заголовочном файле.

2. Type Erasure (std::function)

```cpp
void run(std::function<void()> f) { f(); }
```

`std::function` — это тяжелый объект, использующий технику стирания типа (похожую на vtable).

- **Плюсы:** Единый тип для любых вызываемых объектов. Можно хранить в векторе, скрывать реализацию в .cpp.
- **Минусы:**
  1. **Виртуальный вызов:** Вызов оператора `()` происходит через косвенную адресацию, что мешает инлайнингу.
  2. **Аллокация:** Если размер лямбды превышает размер SBO (Small Buffer Optimization, обычно 16-32 байта), происходит выделение памяти в куче.

3. Function Ref (C++26 / nonstd)

`std::function_ref` (или его аналоги в библиотеках) — это легковесная невладеющая ссылка на вызываемый объект.

```
1 void run(std::function_ref<void()> f) { f(); }
```

Внутренне это пара: `{void* obj, R(*callback)(void*)}`. При создании лямбда не копируется, и память не выделяется. Это идеальный вариант для передачи коллбеков в функции, которые не сохраняют их на будущее.

Резюме раздела

- Лямбда — это объект сгенерированного компилятором уникального класса.
- `[=]` захватывает `this`, а не члены класса, что опасно.
- Используйте init-capture `[u = std::move(p)]` для move-only типов.
- `mutable` позволяет менять состояние внутри лямбды.
- C++20 разрешил лямбды в `decltype` и шаблонные лямбды `[]<T>(T x)`.
- Избегайте `std::function`, если не нужно хранить лямбду долго; используйте шаблоны или `function_ref` для передачи параметров.