HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 01 – Введение. Память

Глава 1

Экосистема разработки и Инструментарий

Разработка на C++ существенно отличается от работы с интерпретируемыми языками (Python, JS) или языками с мощной стандартной экосистемой "из коробки" (Go, Rust). Здесь нет единого стандарта на сборку, управление пакетами или линтинг. Однако индустрия выработала набор инструментов де-факто, владение которыми является обязательным для написания качественного, переносимого и безопасного кода. В этой главе мы рассмотрим пайплайн сборки CMake, настройку IDE через LSP (ClangD) и инструменты динамического анализа памяти (Sanitizers).

Система сборки CMake

Нажатие кнопки "Play" в IDE скрывает за собой сложный процесс препроцессинга, компиляции каждого файла (Translation Unit), линковки объектных файлов и подключения библиотек. При переходе к проектам, состоящим из более чем одного файла, ручной вызов компилятора (например, `g++ main.cpp utils.cpp -o app`) становится неуправляемым.

> **CMake**
>
> Кроссплатформенная система мета-сборки. Она не собирает проект сама, а генерирует инструкции для нативной системы сборки (Makefiles, Ninja, Visual Studio Project и т.д.) на основе конфигурационного файла `CMakeLists.txt`.

Процесс работы с CMake всегда состоит из двух этапов:

1. **Конфигурация (Configuration):** CMake читает `CMakeLists.txt`, проверяет наличие компилятора, библиотек и генерирует дерево сборки.
2. **Сборка (Build):** Вызов нативного инструмента (например, `make`) для непосредственной компиляции исходного кода.

Пример минимального `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Добавляем цель для сборки исполняемого файла
```

```
add_executable(my_app main.cpp utils.cpp)

# Пример подключения санитайзеров (опционально)
target_compile_options(my_app PRIVATE -fsanitize=address -g)
target_link_options(my_app PRIVATE -fsanitize=address)
```

Для запуска сборки в терминале используется следующий паттерн out-of-source build (сборка в отдельной директории, чтобы не засорять исходники):

```
# 1. Создаем директорию build и запускаем конфигурацию
# -S . указывает, что исходники в текущей папке
# -B build указывает, что артефакты сборки кладем в папку build
cmake -S . -B build

# 2. Запускаем непосредственно сборку
# --build build — абстрактная команда, вызывающая make/ninja внутри папки build
cmake --build build
```

> **Важно!**
>
> Никогда не редактируйте сгенерированные файлы (Makefiles) внутри папки `build`. Все изменения конфигурации должны производиться только в `CMakeLists.txt`. Если вам нужно пересобрать проект с нуля, безопасно просто удалить папку `build` и запустить шаг конфигурации заново.

Интегрированная среда разработки (IDE) и LSP

Для комфортной работы с C++ (автодополнение, навигация по коду, подсветка ошибок) обычного текстового редактора недостаточно. Современный стандарт — использование протокола Language Server Protocol (LSP).

ClangD vs Стандартное расширение C++

В редакторе VS Code по умолчанию предлагается расширение от Microsoft (C/C++). Однако для Unix-систем и сложного C++ кода предпочтительнее использовать **clangd**.

- **Стабильность:** Clangd использует тот же фронтенд, что и компилятор Clang. Это гарантирует, что если код подсвечивается как ошибочный, он действительно не скомпилируется.
- **Скорость:** Индексация проекта происходит быстрее и точнее.

Для того чтобы Clangd (и любой другой языковой сервер) понимал структуру вашего проекта, ему нужен файл `compile_commands.json`. Этот файл содержит точные команды, которые используются для компиляции каждого файла (включая флаги, пути к заголовочным файлам и макросы).

Чтобы CMake сгенерировал этот файл, необходимо добавить флаг при конфигурации:

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

После этого файл `compile_commands.json` появится в папке `build`. Для корректной работы IDE часто требуется создать символическую ссылку на этот файл в корень проекта или настроить IDE на поиск файла в подпапках.

На заметку

Если IDE подчеркивает валидный код красным (например, не видит `<iostream>`), в 99% случаев проблема в отсутствии или некорректности `compile_commands.json`.

Динамический анализ: AddressSanitizer (ASan)

Ошибки работы с памятью — самый коварный класс багов в C++. Они могут приводить к неопределенному поведению (UB), которое не проявляется при локальном запуске, но роняет продакшн. Статический анализ не может поймать все ошибки. Здесь на помощь приходят санитайзеры.

> **AddressSanitizer (ASan)**
>
> Инструмент динамического анализа, который встраивает проверки доступа к памяти непосредственно в исполняемый файл на этапе компиляции.

Механизм работы

При компиляции с флагом `-fsanitize=address` компилятор инструментирует код, окружая каждый выделенный объект в памяти (как на стеке, так и в куче) специальными "красными зонами" (redzones). При любом обращении к памяти программа проверяет, не попадает ли адрес в красную зону. Если попадает — выполнение немедленно прерывается с подробным отчетом об ошибке.

ASan позволяет обнаруживать:

- Выход за границы массива (Out-of-bounds access) на стеке и в куче.
- Использование памяти после освобождения (Use-after-free).
- Двойное освобождение (Double-free).
- Утечки памяти (Memory leaks) — работает как LeakSanitizer.

Стоимость использования

Использование ASan замедляет выполнение программы примерно в 2 раза и увеличивает потребление памяти. Это приемлемо для этапа тестирования и отладки (Debug builds), но обычно отключается в финальной сборке (Release builds), где важна максимальная производительность.

Пример включения в `CMakeLists.txt` (рекомендуется выносить в отдельный тип сборки):

```
if(ENABLE_ASAN)
    add_compile_options(-fsanitize=address -g)
    add_link_options(-fsanitize=address)
endif()
```

Отладка: GDB

Хотя современные IDE предоставляют графические интерфейсы для отладки, умение работать с консольным отладчиком GDB (GNU Debugger) является критическим навыком.

- **Работа на сервере:** Если ваше приложение упало на удаленном сервере (headless environment), у вас не будет доступа к GUI. GDB — единственный способ проанализировать `core dump` (слепок памяти упавшего процесса).
- **Автоматизация:** GDB поддерживает скриптинг. Вы можете написать сценарий, который автоматически выполняет 100 итераций цикла и останавливается только при выполнении сложного условия, что утомительно делать "прокликиванием" в GUI.

Базовый сеанс отладки в GDB:

```
$ gdb ./my_app
(gdb) break main          # Поставить точку останова на main
(gdb) run                 # Запустить программу
(gdb) next                # Следующая строка (шаг без захода внутрь)
(gdb) step                # Шаг с заходом внутрь функции
(gdb) print variable_name # Вывести значение переменной
(gdb) continue            # Продолжить выполнение до следующей точки
(gdb) bt                  # Backtrace: показать стек вызовов
```

Резюме раздела

Грамотно настроенное окружение — половина успеха.

1. Используйте **CMake** для управления сборкой, разделяя исходный код и артефакты.
2. Настройте **ClangD** и генерацию `compile_commands.json` для умного автодополнения.
3. Всегда собирайте отладочные версии с **AddressSanitizer** (`-fsanitize=address`), чтобы ловить ошибки памяти на ранних этапах.
4. Изучите основы **GDB** для отладки в сложных окружениях.

Глава 2

Модель памяти и Жизненный цикл объектов

Понимание работы с памятью — фундаментальный навык разработчика на C++. В отличие от языков с управляемой памятью (Java, Python, Go), где о расположении объектов заботится виртуальная машина и сборщик мусора, C++ предоставляет прямой доступ к управлению ресурсами. Это дает огромную власть, но налагает не меньшую ответственность: по статистике, около 70-80% всех уязвимостей в безопасности ПО связаны именно с ошибками работы с памятью (memory safety issues).

В этой главе мы рассмотрим, что такое память с точки зрения стандарта C++, что такое "объект" и какие существуют классы хранения данных.

Фундаментальная модель памяти

Миф о "Большом массиве байт"

Интуитивно память часто представляют как единый, непрерывный массив байт, пронумерованный от 0 до N. На физическом уровне (RAM) это отчасти верно, однако программа на C++ работает не с физическими чипами, и даже не напрямую с виртуальной памятью операционной системы. Она работает с **Абстрактной машиной C++**.

> **Важно!**
>
> С точки зрения стандарта C++, память — это не "просто массив", а набор доступных **объектов**. Доступ к памяти, где не создан объект (или его время жизни истекло), является неопределенным поведением (Undefined Behavior).

Операционная система предоставляет процессу *виртуальное адресное пространство*. Это пространство изолировано: программа не может случайно (или специально) прочитать данные соседнего процесса (например, браузера), просто обратившись по адресу $0x1234$. Адресное пространство процесса фрагментировано: оно состоит из сегментов кода, данных, стека и кучи, между которыми могут находиться огромные "дыры" невыделенной памяти.

Байт и слово

Минимальной адресуемой единицей памяти в C++ является байт.

- `sizeof(char)` всегда равен 1. Это определение единицы измерения в языке.
- Количество бит в байте определяется макросом `CHAR_BIT` из заголовка `<climits>`.

На заметку

Формально стандарт требует, чтобы `CHAR_BIT` $\geqslant$ `8`. Исторически существовали архитектуры с 9-битными или 16-битными байтами, но в современной индустрии (x86, ARM) байт всегда равен 8 битам (октету). Тем не менее, полагаться на хардкод константы 8 — плохой тон; лучше использовать `CHAR_BIT`.

Анатомия Объекта

В C++ всё есть объект (или ссылка на него). Понимание свойств объекта критично для избежания UB.

> **Объект в C++**
>
> Объект — это область памяти (region of storage), которая имеет:
>
> - **Size (Размер):** определяется `sizeof(T)`.
> - **Alignment (Выравнивание):** требование процессора к адресу начала объекта (например, `int` часто должен лежать по адресу, кратному 4).
> - **Storage Duration (Тип хранения):** где и сколько живет объект.
> - **Lifetime (Время жизни):** интервал времени выполнения, когда объект существует.
> - **Type (Тип):** интерпретация битов в этой памяти.
> - **Value (Значение):** конкретное содержимое.

Ключевой концепт: **Lifetime**.

1. Память выделяется (allocation).
2. Отрабатывает конструктор — начинается lifetime.
3. ... Использование объекта ...
4. Отрабатывает деструктор — заканчивается lifetime.
5. Память освобождается (deallocation).

Любое использование указателя или ссылки на объект до шага 2 или после шага 4 — это ошибка **Use-after-free** или использование неинициализированной памяти.

Категории хранения (Storage Durations)

Стандарт выделяет четыре типа продолжительности хранения. Понимание различий между ними позволяет отвечать на вопрос: "Когда умрет эта переменная?".

1. **Automatic (Автоматическая):** Локальные переменные функций, аргументы. Живут на стеке.

2. **Static (Статическая):** Глобальные переменные, `static` поля классов, `static` переменные внутри функций. Живут всё время работы программы.
3. **Dynamic (Динамическая):** Объекты, созданные через `new`/`malloc`. Живут пока их явно не удалят. (Подробнее в следующей главе).
4. **Thread-local (Локальная для потока):** Живут пока жив поток, создавший их (спецификатор `thread_local`).

Автоматическая память (Stack)

Самый быстрый и безопасный тип памяти. Обычно называется "стеком", так как реализуется через аппаратный стек процессора (регистры SP/ESP/RSP).

Принцип работы

При входе в функцию (scope) указатель стека сдвигается, выделяя место под все локальные переменные разом. При выходе — сдвигается обратно.

- **Аллокация:** мгновенная (просто сложение регистра с константой).
- **Деаллокация:** автоматическая при выходе из скоупа (закрывающая фигурная скобка `}`).

> **Важно!**
>
> Порядок разрушения объектов на стеке **строго обратен** порядку их создания. Это принцип LIFO (Last In, First Out).

```cpp
void logic() {
    std::string a = "First";
    std::vector<int> b = {1, 2};
    // ... работа ...
} // Сначала вызывается ~vector (b), затем ~string (a)
```

Это свойство является основой идиомы RAII (Resource Acquisition Is Initialization), о которой мы поговорим позже.

Проблема Stack Overflow (Задача с семинара)

Размер стека ограничен. В Linux это обычно 8 MiB, в Windows — 1-2 MiB (настраивается при компиляции). Исчерпание стека приводит к аварийному завершению программы (Segmentation Fault).

Рассмотрим задачу с семинара, демонстрирующую опасность рекурсии:

```cpp
void recursive_boom(int depth) {
    // Массив выделяется на стеке в КАЖДОМ кадре рекурсии
    char buffer[1024];

    // Предотвращение оптимизации (чтобы компилятор не выкинул массив)
    buffer[0] = depth % 256;
```

```
7
8       if (depth > 0) {
9           recursive_boom(depth - 1);
10      }
11  }
12
13  int main() {
14      recursive_boom(100000); // Глубина 100 тысяч
15  }
```

Анализ: Каждый вызов функции `recursive_boom` создает новый стековый кадр (stack frame). В этом кадре хранится:

1. Адрес возврата (8 байт на x64).
2. Аргумент `depth` (4-8 байт с учетом выравнивания).
3. Массив `buffer` (1024 байта).

Итого один вызов занимает чуть больше 1 Кб. При глубине 100 000 рекурсивных вызовов мы пытаемся занять $100\,000 \times 1$ KiB ≈ 100 MiB. Это значительно превышает лимит стека (8 MiB), что гарантированно приводит к краху программы.

> **На заметку**
>
> Крупные объекты (массивы на миллион элементов, тяжелые структуры) никогда не должны создаваться на стеке. Используйте `std::vector` или умные указатели для размещения их данных в куче.

Статическая память (Static)

Переменные со статической продолжительностью хранения инициализируются один раз (обычно до вызова `main` или при первом проходе управления) и разрушаются при завершении программы.

Сюда относятся:

- Глобальные переменные.
- Переменные, объявленные как `static` внутри функций или классов.

Static Initialization Order Fiasco

Самая большая проблема глобальных переменных — неопределенный порядок инициализации между разными единицами трансляции (cpp-файлами).

Представьте два файла:

```
1  // File A.cpp
2  int x = func(); // func() возвращает 42
3
4  // File B.cpp
5  extern int x;
```

```
int y = x + 1;
```

Стандарт C++ **не гарантирует**, что `A.cpp` будет инициализирован раньше `B.cpp`. Если инициализация `B` произойдет первой, переменная `x` будет иметь значение 0 (zero-initialization), и `y` станет равно 1. Если порядок будет обратным, `y` станет 43. Это классический "плавающий" баг, зависящий от фазы луны и линковщика.

Решение: Meyers Singleton

Для безопасной работы с глобальными состояниями (если они необходимы) Скотт Мейерс предложил паттерн, использующий `static` переменную *внутри функции*.

Стандарт C++11 гарантирует, что статические переменные внутри функции инициализируются:

1. Ровно один раз.
2. В момент, когда поток управления **впервые** проходит через их объявление.
3. Потокобезопасно (компилятор вставляет неявные блокировки/guard variables).

```
class Database {
    // ... подключение к БД ...
};

Database& GetDB() {
    // Инициализация произойдет только при первом вызове GetDB()
    static Database db_instance;
    return db_instance;
}

void ClientCode() {
    // Гарантированно инициализированный объект
    GetDB().query("SELECT * FROM users");
}
```

Этот подход решает проблему порядка инициализации (ленивая инициализация по требованию) и является стандартным способом реализации синглтонов в современном C++.

Резюме раздела

- Память в C++ — это набор объектов с временем жизни, а не просто байты.
- **Стек (Automatic)**: быстро, автоматически чистится, но размер ограничен. LIFO.
- **Глобальные (Static)**: живут вечно, опасны из-за порядка инициализации. Используйте локальные статики (Meyers Singleton).
- Избегайте глубокой рекурсии и больших массивов на стеке во избежание **Stack Overflow**.

Глава 3

Динамическая память и RAII

Мы разобрали автоматическую (стековую) и статическую память. Однако стека недостаточно для решения большинства реальных задач: его размер мал (мегабайты), а время жизни объектов жестко привязано к области видимости. Для работы с большими объемами данных (изображения, базы данных, сетевые буферы) или объектами, время жизни которых определяется бизнес-логикой, используется динамическая память (Heap или "Куча").

В "старом" C++ (до C++11) и в языке C работа с кучей требовала ручного микроменеджмента. В современном C++ мы используем идиому RAII, чтобы автоматизировать этот процесс.

Низкоуровневые механизмы: malloc vs new

В C++ существуют два способа выделить сырую память. Важно понимать разницу между ними, так как смешивать их — грубая ошибка.

1. **C-style (`malloc/free`):** Функции из стандартной библиотеки C (`<cstdlib>`).
    - **Типизация:** Возвращают `void*`. Требуют приведения типов.
    - **Объекты:** Только выделяют байты. **Не вызывают конструкторы** и деструкторы.
    - **Ошибки:** При нехватке памяти возвращают `nullptr`.
2. **C++ Operators (`new/delete`):** Операторы языка.
    - **Типизация:** Типизированы, возвращают `T*`.
    - **Объекты:** Сначала выделяют память (через `operator new`), затем **вызывают конструктор**. Оператор `delete` вызывает деструктор, затем освобождает память.
    - **Ошибки:** При нехватке памяти бросают исключение `std::bad_alloc`.

> **Важно!**
>
> В C++ использование `malloc` практически всегда является ошибкой, кроме случаев взаимодействия с legacy C-библиотеками или написания собственных аллокаторов. Для создания объектов всегда используется `new` (или его обертки).

Проблема Exception Safety

Почему ручные `new` и `delete` считаются плохим тоном ("моветоном")? Главная причина — сложность написания кода, устойчивого к исключениям (Exception Safety).

Рассмотрим классический пример утечки ресурсов, который сложно заметить невооруженным глазом:

```
void process(Widget* w1, Widget* w2);

void problematic_usage() {
    // ОПАСНО: Порядок вычисления аргументов не определен (до C++17)
    // Даже в C++17, если конструктор второго Widget кинет исключение,
    // первый Widget уже создан, но указатель на него потерян.
    process(new Widget(1), new Widget(2));
}
```

Разбор сценария утечки: Компилятор может сгенерировать код в таком порядке:

1. Выделение памяти под первый Widget.
2. Вызов конструктора первого Widget (успех). Указатель лежит во временном регистре процессора.
3. Выделение памяти под второй Widget.
4. **Вызов конструктора второго Widget — бросает исключение!**

В этот момент стек начинает разматываться (stack unwinding). Мы вылетаем из функции `problematic_usage`. Но указатель на первый созданный Widget нигде не был сохранен в переменную, у которой есть деструктор. Адрес потерян. Память утекла.

Чтобы исправить это на "сырых" указателях, пришлось бы писать громоздкий код:

```
void safe_but_ugly() {
    Widget* w1 = new Widget(1);
    try {
        Widget* w2 = new Widget(2);
        try {
            process(w1, w2);
        } catch (...) {
            delete w2;
            throw;
        }
    } catch (...) {
        delete w1; // Очистка w1 в случае ошибки создания w2
        throw;
    }
    delete w2;
    delete w1;
}
```

Очевидно, что писать так — невозможно. Здесь на помощь приходит RAII.

Идиома RAII (Resource Acquisition Is Initialization)

Это, пожалуй, самая важная идиома C++. Идея гениальна в своей простоте: **связать время жизни ресурса (память, файл, мьютекс) с временем жизни объекта на стеке**.

> **RAII**
>
> Захват ресурса происходит в конструкторе объекта, а освобождение — в его деструкторе. Поскольку объекты на стеке гарантированно разрушаются при выходе из области видимости (в том числе при вылете исключения), утечки ресурсов становятся невозможными.

Вместо сырого указателя мы используем "умный" объект-обертку. Стандартная библиотека предоставляет готовое решение.

Умный указатель std::unique_ptr

`std::unique_ptr<T>` (из заголовка `<memory>`) — это легковесная обертка над сырым указателем.

- **Семантика:** Я владею объектом единолично.
- **Конструктор:** Принимает сырой указатель.
- **Деструктор:** Вызывает `delete` для хранимого указателя.
- **Копирование:** Запрещено (удален copy constructor).
- **Перемещение:** Разрешено (move constructor передает владение).

Исправим пример с `process` используя современный подход:

```cpp
#include <memory>

void process(std::unique_ptr<Widget> w1, std::unique_ptr<Widget> w2);

void safe_usage() {
    // std::make_unique создает объект и сразу возвращает unique_ptr.
    // Если make_unique для w2 кинет исключение,
    // unique_ptr для w1 уже будет создан и корректно уничтожится.
    process(
        std::make_unique<Widget>(1),
        std::make_unique<Widget>(2)
    );
}
```

> **На заметку**
>
> Функция `std::make_unique<T>(args...)` появилась в C++14. Всегда предпочитайте её использованию `new`. Она не только безопаснее (исключает утечку, описанную выше), но и избавляет от необходимости писать имя типа `T` дважды.

Массивы: new[] и delete[]

В C++ одиночные объекты и массивы объектов — это разные сущности с точки зрения управления памятью.

- `new T` требует `delete ptr`.
- `new T[N]` требует `delete[] ptr`.

> **Важно!**
>
> Перепутать операторы удаления — это Неопределенное поведение (UB). Если вы выделили массив через `new int[10]`, а удалили через `delete`, программа может упасть или повредить кучу (Heap Corruption).

Почему? Обычно при выделении массива `new[]` компилятор сохраняет размер массива (например, 4 байта перед началом возвращаемого указателя), чтобы знать, сколько раз вызвать деструктор. Оператор `delete[]` читает этот размер и идет в цикле. Обычный `delete` просто удаляет один объект.

В современном C++ сырые массивы в динамической памяти (`new[]`) практически не нужны. Используйте:

- `std::vector<T>` — для динамических массивов с изменяемым размером.
- `std::string` — для строк.

Резюме раздела

1. Никогда не используйте `malloc` в C++ коде.
2. Избегайте явных вызовов `new` и `delete`.
3. Используйте `std::unique_ptr` и `std::make_unique` для управления одиночными объектами.
4. Используйте `std::vector` вместо динамических массивов `new T[]`.
5. **RAII** — ваш главный инструмент борьбы с утечками. Ресурс должен принадлежать объекту на стеке.

Глава 4

Низкоуровневая работа с памятью: Placement New и Выравнивание

До этого момента мы рассматривали стандартные способы создания объектов: на стеке (автоматически) или в куче (через `new` или `std::make_unique`). В обоих случаях компилятор и рантайм берут на себя две задачи одновременно:

1. Выделение сырой памяти (allocation).
2. Конструирование объекта в этой памяти (construction).

Однако в системном программировании, при написании собственных контейнеров (например, `std::vector`) или аллокаторов, эти этапы необходимо разделять. Нам нужно уметь создать объект в *уже выделенной* памяти по конкретному адресу. Этот механизм называется **Placement New**.

Это "черная магия" C++, открывающая ящик Пандоры с неопределенным поведением. Ошибиться здесь легко, а цена ошибки — трудноуловимые баги, связанные с выравниванием (alignment) и временем жизни.

Разделяющий властвуй: Placement New

Обычный оператор `new T(...)` делает две вещи: вызывает функцию выделения памяти (`operator new`), а затем вызывает конструктор. Placement New — это перегрузка оператора `new`, которая принимает дополнительный аргумент (адрес памяти) и **не выделяет память**, а просто вызывает конструктор по этому адресу.

Синтаксис и Жизненный цикл

Синтаксис выглядит так:

```
new (адрес) Тип(аргументы);
```

Для использования необходимо подключить заголовок `<new>`.

Рассмотрим полный цикл жизни объекта, созданного вручную. Поскольку память не выделялась через обычный `new`, мы не имеем права вызывать `delete`. Мы обязаны вызвать деструктор вручную.

```cpp
#include <new>
#include <string>
#include <iostream>

int main() {
    // 1. Выделяем сырую память (достаточную для хранения string)
    // Внимание: здесь есть проблема с выравниванием (см. далее)!
    char buffer[sizeof(std::string)];

    // 2. Конструируем объект в буфере (Placement New)
    std::string* s_ptr = new (buffer) std::string("Hello, Placement New!");

    // 3. Используем объект
    std::cout << *s_ptr << ", length: " << s_ptr->size() << std::endl;

    // 4. ЯВНО вызываем деструктор
    s_ptr->~basic_string();

    // 5. Память буфера освободится автоматически (так как это стек)
    // Если бы buffer был выделен malloc-ом, нужно было бы сделать free(buffer).
}
```

Важно!

Никогда не вызывайте `delete s_ptr` для объекта, созданного через placement new на стековом буфере или внутри другого объекта. Оператор `delete` попытается передать адрес буфера системному аллокатору памяти, который понятия не имеет об этом адресе (если это стек) или уже считает его занятым. Это гарантированный крэш или коррупция кучи.

Выравнивание (Alignment)

В примере выше (`char buffer[...]`) мы допустили критическую ошибку, которая на архитектуре x86 может остаться незамеченной, но приведет к падению программы (SIGBUS) на ARM или SPARC, а также к существенному падению производительности.

Почему sizeof недостаточно?

Процессор работает с памятью не побайтово, а машинными словами (4, 8, 16 байт). Для эффективного (а иногда и корректного) доступа к данным адреса должны быть кратны размеру типа.

- `int` (4 байта) обычно должен лежать по адресу, кратному 4.
- Указатели (8 байт на 64-bit) — по адресу, кратному 8.
- Векторные инструкции (SIMD, SSE/AVX) могут требовать выравнивания по 16, 32 или 64 байтам.

Массив `char buffer[N]` имеет выравнивание 1 (так как `alignof(char) == 1`). Он может начаться по любому адресу (например, 0x1001). Тип `std::string` внутри себя содержит указатели и счетчики размеров, требующие выравнивания 8 (на 64-битных системах).

Если мы разместим `std::string` по адресу 0x1001, мы нарушим требования выравнивания (**Misaligned Access**). Это **Undefined Behavior**.

Решение: alignas и alignof

Чтобы исправить это, буфер должен быть выровнен так же, как и тип, который мы собираемся в него положить.

- `alignof(T)` — возвращает требование к выравниванию для типа `T`.
- `alignas(N)` — спецификатор, заставляющий компилятор выровнять переменную по границе `N`.

Правильная реализация буфера:

```
template <typename T>
struct AlignedStorage {
    // Выравниваем массив байт так же, как тип T
    alignas(T) std::byte data[sizeof(T)];

    // Вспомогательные методы для удобства
    T* get() { return reinterpret_cast<T*>(data); }
    const T* get() const { return reinterpret_cast<const T*>(data); }
};

int main() {
    AlignedStorage<std::string> storage;

    // Теперь адрес storage.data гарантированно кратен alignof(string)
    std::string* s = new (storage.get()) std::string("Safe!");

    s->~basic_string();
}
```

Также в стандарте C++ (заголовок `<type_traits>`) есть `std::aligned_storage`, но начиная с C++23 он объявлен устаревшим (deprecated) в пользу использования `alignas` вручную, как показано выше, из-за сложности правильного использования API.

Тривиальная разрушаемость (Trivially Destructible)

Обязательно ли всегда вызывать деструктор при ручном управлении памятью?

```
{
    char buf[sizeof(int)];
    int* p = new (buf) int(42);
    // Нужно ли звать p->~int()?
}
```

Для фундаментальных типов (`int`, `float`, указатели) и простых C-структур (POD) деструктор ничего не делает (no-op). Вызов `p→~int()` корректен синтаксически, но компилятор его просто вырежет.

Существует трейт (свойство типа) `std::is_trivially_destructible_v<T>`.

- Если **true**: Вызов деструктора можно безопасно пропустить. Память можно просто "забыть" или перезаписать.
- Если **false** (например, `std::string`, `std::vector`): Вызов деструктора обязателен. Если его пропустить, утекут ресурсы, которыми владел объект (внутренние буферы в куче).

> **Важно!**
>
> Оптимизация с пропуском деструкторов широко используется в реализации `std::vector::clea` Если элементы тривиально разрушаемы, вектор просто обнуляет свой размер (`size = 0`), не пробегая циклом по всем элементам. Это дает колоссальный прирост производительности.

Практический пример: простейший Any

Чтобы закрепить материал, напишем упрощенный класс, который может хранить либо `int`, либо `double` (похоже на `union`, но с активным переключением).

```
#include <iostream>
#include <new>

struct NumberHolder {
    // Достаточно места для самого большого типа
    // Выравнивание по максимуму из двух
    alignas(double) std::byte storage[sizeof(double)];

    enum Type { INT, DOUBLE, NONE } currentType = NONE;

    void setInt(int val) {
        destroyCurrent(); // Сначала очищаем старое
        new (storage) int(val);
        currentType = INT;
    }

    void setDouble(double val) {
        destroyCurrent();
        new (storage) double(val);
        currentType = DOUBLE;
    }

    void destroyCurrent() {
        if (currentType == INT) {
            reinterpret_cast<int*>(storage)->~int(); // Тривиально, но для
            ↪ порядка
        } else if (currentType == DOUBLE) {
            reinterpret_cast<double*>(storage)->~double();
        }
        currentType = NONE;
    }

    ~NumberHolder() {
        destroyCurrent();
```

```
34         }
35 };
```

Резюме раздела

1. **Placement New** позволяет конструировать объекты в заранее выделенной памяти. Он не выделяет память сам.
2. Объекты, созданные через placement new, требуют явного вызова деструктора: `ptr→T()`. Использование `delete` запрещено.
3. **Выравнивание** критически важно. Простого массива `char` недостаточно. Используйте `alignas(T)`.
4. Доступ к невыровненным данным — это Undefined Behavior.
5. Для типов с `is_trivially_destructible` вызов деструктора можно опустить (оптимизация).

Глава 5

Оптимизация строк: SSO и String View

Строки — один из самых часто используемых типов данных в прикладном программировании. Поэтому эффективность реализации `std::string` критически важна для общей производительности C++ приложений. В этой главе мы разберем две важнейшие концепции: оптимизацию коротких строк (SSO), которая "прячет" аллокации, и `std::string_view` — инструмент для эффективной работы с подстроками без копирования.

Анатомия std::string и SSO

Наивная реализация строки могла бы выглядеть как простая обертка над динамическим массивом:

```
class NaiveString {
    char* data_;
    size_t size_;
    size_t capacity_;
};
```

На 64-битной архитектуре размер такого объекта составляет $8+8+8 = 24$ байта. Сама строка хранится в куче (heap). Это означает, что даже для строки "hello" (5 байт) мы вынуждены делать дорогостоящий системный вызов `malloc/new`, ловить cache miss при обращении к данным и фрагментировать память.

Учитывая, что в реальных программах огромное количество строк очень короткие (имена, ключи в JSON, идентификаторы), разработчики стандартной библиотеки применили оптимизацию **SSO (Small String Optimization)**.

Механика SSO

Идея SSO заключается в использовании памяти самого объекта (этих 24 байт на стеке) для хранения содержимого строки, если оно туда помещается.

Внутреннее устройство `std::string` можно представить как объединение (`union`) двух состояний:

1. **Long String (Куча):** Классическая схема (Pointer, Size, Capacity).
2. **Short String (SSO):** Буфер символов прямо внутри структуры + 1 байт для размера.

Псевдокод реализации (упрощенно):

```
class String {
    static const size_t SSO_CAPACITY = 23; // 24 байта - 1 байт на размер/null

    union {
        // Режим "Длинная строка"
        struct {
            char* ptr;
            size_t size;
            size_t capacity;
        } heap_buf;

        // Режим "Короткая строка"
        struct {
            char buffer[SSO_CAPACITY];
            unsigned char size_byte; // Хранит размер и флаг режима
        } stack_buf;
    };

public:
    // ... методы ...
};
```

Как это работает?

Поскольку объект занимает 24 байта, мы можем хранить строку длиной до 23 символов **без аллокации памяти**.

- **23 символа:** полезная нагрузка.
- **24-й байт:** используется двояко. В режиме SSO он хранит размер оставшегося места (или длину). В режиме длинной строки это часть поля `capacity` или специальный флаг.

Компиляторы используют битовые хаки. Например, младший бит последнего байта может служить флагом:

- 0: Режим SSO.
- 1: Режим длинной строки (указатель).

> **На заметку**
>
> Благодаря SSO, создание `std::string s = "test";` не приводит к обращению к куче. Это делает работу с короткими строками такой же быстрой, как использование автоматических массивов `char[N]`.

std::string_view (C++17)

До C++17 существовала проблема передачи строк в функции.

- **Передача по значению (`void f(std::string s)`):** Всегда вызывает копирование (глубокую аллокацию), если строка длинная. Дорого.

- **Передача по константной ссылке (`void f(const std::string& s)`):** Избавляет от копирования. Но что, если мы хотим передать подстроку?

```cpp
void process(const std::string& s);

std::string data = "Header: Value";
// Чтобы передать "Value", нужно создать временную строку (аллокация!)
process(data.substr(8));
```

Решением стал `std::string_view` — легковесная "вьюшка" (view), которая смотрит на кусок памяти, но не владеет им.

Структура и Производительность

`std::string_view` состоит всего из двух полей:

```cpp
class string_view {
    const char* ptr_; // Указатель на начало
    size_t size_;     // Длина
};
```

Размер объекта — 16 байт (на 64-bit).

> **Важно!**
>
> **Передача параметров:** `std::string_view` принято передавать **по значению**, а не по ссылке.
>
> - `void f(std::string_view sv)`
>
> Почему? Объект размером 16 байт идеально ложится в два 64-битных регистра процессора (например, RDI и RSI в System V ABI). Передача через регистры быстрее, чем передача ссылки (которая суть указатель), так как избегается лишнее разыменование (double indirection) при доступе к полям `ptr_` и `size_`.

Операция взятия подстроки (`substr`) для `string_view` имеет сложность $O(1)$ — это просто сдвиг указателя и уменьшение размера. Никакой памяти не выделяется.

Подводные камни (Pitfalls)

С большой силой приходит большая ответственность. Поскольку `string_view` **не владеет** памятью, программист обязан следить за тем, чтобы исходная строка жила дольше, чем `string_view`.

Проблема 1: Dangling View (Висячая ссылка)

Классическая ошибка — создание `string_view` от временного объекта.

```cpp
std::string_view get_greeting() {
    std::string s = "Hello, World!"; // Локальная переменная
```

```
    return s; // Неявное преобразование string -> string_view
} // s уничтожается, память освобождается

int main() {
    std::string_view sv = get_greeting();
    std::cout << sv; // UB: чтение освобожденной памяти (Use-after-free)
}
```

Также опасно комбинировать создание и использование в одной строке, если есть временные объекты:

```
std::string_view sv = std::string("Temporary") + " string";
// Временная строка умирает в конце выражения. sv смотрит в мусор.
```

Проблема 2: Отсутствие Null-Terminator

`std::string` гарантирует, что после данных следует байт `\0`, что позволяет использовать метод `.c_str()` для совместимости с C-функциями (`printf`, `fopen`).

`std::string_view` **не гарантирует** наличие нуль-терминатора, так как может указывать на середину другой строки.

```
std::string s = "Hello World";
std::string_view sv = s.substr(0, 5); // "Hello"

// ОШИБКА:
printf("%s", sv.data());
// sv.data() указывает на 'H'. printf будет печатать байты, пока не встретит \0.
// Он напечатает "Hello World" (в лучшем случае) или мусор за пределами s.
```

Если вам нужно передать `string_view` в функцию, ожидающую C-строку, вам придется сначала скопировать его в `std::string`:

```
std::string temp(sv);
legacy_function(temp.c_str());
```

Резюме раздела

1. **SSO** позволяет хранить короткие строки (обычно до 23 символов) без динамической аллокации. Это делает `std::string` очень эффективным для мелких данных.
2. Используйте `std::string_view` для передачи строк в функции и парсинга. Это избегает лишних копирований.
3. Передавайте `std::string_view` **по значению**.
4. Будьте предельно осторожны с временем жизни. **Никогда** не сохраняйте `string_view` на временные строки.
5. Помните, что `string_view` не является нуль-терминированной строкой.

Глава 6

Кейс-стади: Внутреннее устройство std::deque

Контейнер `std::deque` (Double-Ended Queue) — один из самых недооцененных и сложных контейнеров стандартной библиотеки. Часто его воспринимают просто как "вектор, в который можно быстро писать в начало". Однако его внутреннее устройство представляет собой уникальный компромисс между производительностью вектора и стабильностью ссылок связанного списка.

В этой главе мы разберем архитектуру `deque`, как она реализует "магию" расширения в обе стороны без копирования элементов и почему это важно для системного программирования.

Архитектура: Карта и Чанки

Наивная реализация двусторонней очереди могла бы быть кольцевым буфером на одном динамическом массиве. Но такая реализация имела бы проблему `std::vector`: при исчерпании емкости пришлось бы выделять новый огромный массив и переносить туда все элементы.

`std::deque` решает эту задачу иначе. Это **фрагментированный** контейнер.

> **Структура Deque**
>
> Логически `deque` — это бесконечный непрерывный массив. Физически — это набор фиксированных блоков памяти (**Chunks** или Buffers), управляемых центральным массивом указателей (**Map**).

Map (Карта)

Центральный элемент управления — это динамический массив указателей (`T** map`).

- Каждая ячейка карты указывает на отдельный блок памяти (чанк), хранящий элементы.
- Карта работает как кольцевой буфер: у нас есть индексы начала и конца последовательности заполненных блоков внутри карты.

Chunks (Блоки)

Сами данные хранятся в блоках фиксированного размера. Размер блока зависит от реализации (GCC, Clang, MSVC), но типичная эвристика — статический буфер размером около 512 байт (или минимум 16 элементов для мелких типов).

```
template <typename T>
class Deque {
private:
    // Параметры чанка
    static const size_t CHUNK_SIZE = 512 / sizeof(T);

    // Центральная "карта" - массив указателей на блоки
    T** map_;
    size_t map_size_;    // Размер самой карты (емкость указателей)

    // Итераторы начала и конца
    // Хранят индекс в карте + индекс внутри конкретного чанка
    iterator start_;
    iterator finish_;
};
```

Двухуровневая индексация

Доступ к произвольному элементу по индексу (`operator[]`) в `deque` сложнее, чем в векторе. Если в векторе это одна инструкция ассемблера (base + index * scale), то в деке это двухуровневая арифметика.

Чтобы получить элемент `deque[n]`, нужно:

1. Определить, в каком блоке находится элемент.
2. Определить смещение внутри этого блока.

Формула (упрощенно):

```
T& operator[](size_t n) {
    // 1. Учитываем смещение начала данных
    size_t pos = start_.cur_index + n;

    // 2. Вычисляем индекс блока в карте
    size_t node_index = pos / CHUNK_SIZE;

    // 3. Вычисляем смещение внутри блока
    size_t offset = pos % CHUNK_SIZE;

    // 4. Двойное разыменование: сначала карту, потом блок
    return map_[node_index][offset];
}
```

На заметку

Именно из-за двойного разыменования (double indirection) `std::deque` немного медленнее `std::vector` при последовательном проходе и случайном доступе. Однако он значительно быстрее `std::list`, так как данные в чанках лежат плотно, что обеспечивает хорошую локальность кэша (Cache Locality).

Главная особенность: Reference Stability

Киллер-фича `std::deque`, отличающая его от вектора — **стабильность ссылок**.

Рассмотрим сценарий:

```
std::vector<int> v = {1, 2, 3};
int& ref = v[0];
v.push_back(4); // Если capacity превышен, вектор реаллоцируется
// ref теперь указывает в освобожденную память (Dangling Reference)!
```

У вектора при расширении происходит выделение нового буфера и перемещение всех элементов. Все итераторы и ссылки инвалидируются.

У `deque` поведение принципиально иное:

1. При добавлении элементов (`push_back`/`push_front`) заполняются существующие чанки.
2. Если текущий чанк полон, выделяется **новый чанк** и добавляется в `map`.
3. Существующие элементы остаются на своих местах в старых чанках.

А что, если переполнилась сама карта (map)? В этот момент происходит реаллокация карты:

1. Выделяется новый массив указателей (обычно в 2 раза больше).
2. Указатели из старой карты копируются в центр новой.
3. Сами чанки данных **не трогаются**. Они остаются по тем же адресам в куче.

> **Важно!**
>
> Вставка в начало или конец `std::deque` **инвалидирует итераторы** (так как меняется внутренняя структура карты), но **НЕ инвалидирует ссылки и указатели** на элементы. Это критически важно, если вы храните указатели на элементы контейнера во внешних структурах.

Сценарии использования

Когда стоит выбрать `deque` вместо `vector`?

1. **Очередь сообщений:** Если вам нужно активно добавлять и удалять элементы с *обоих* концов. Вектор умеет быстро работать только с концом (`push_back`). Вставка в начало вектора — $O(N)$.

2. **Неинвалидация ссылок:** Если архитектура приложения требует, чтобы указатели на объекты сохранялись при росте контейнера.
3. **Фрагментация памяти:** Если вы храните миллионы объектов, вектор потребует найти непрерывный кусок памяти в сотни мегабайт, что может быть невозможно. Дек спокойно "съест" память кусками по 512 байт, разбросанными по всей куче.

Резюме раздела

- `std::deque` — это массив указателей на блоки фиксированного размера.
- Обеспечивает $O(1)$ вставку в начало и конец.
- Доступ по индексу чуть медленнее вектора (две инструкции чтения памяти).
- При реаллокации (расширении) элементы не перемещаются в памяти, ссылки остаются валидными.

Часть II

Лекция 02 – Динамическая память. Move семантика

Глава 7

Фундаментальное управление памятью и RAII

Управление памятью является краеугольным камнем системного программирования на C++. Понимание того, где и как хранятся объекты, определяет не только производительность приложения, но и его корректность. Данная глава посвящена анатомии памяти процесса, проблемам ручного управления ресурсами, идиоме RAII и современным механизмам эксклюзивного владения.

1.1 Анатомия памяти процесса

В контексте C++ модель памяти процесса операционной системы традиционно разделяется на четыре основных сегмента. Каждый из них имеет свои характеристики времени жизни объектов и накладных расходов на доступ.

- **Стек (Stack, Automatic Storage Duration):** Область памяти, работающая по принципу LIFO (Last In, First Out). Выделение памяти сводится к изменению значения регистра указателя стека (Stack Pointer), что делает операции аллокации и деаллокации чрезвычайно быстрыми (наносекунды). Размер стека фиксирован и невелик (обычно 2–8 МБ), что делает его непригодным для хранения больших структур данных.
- **Куча (Heap, Dynamic Storage Duration):** Область памяти произвольного доступа. Управление осуществляется через системные вызовы (например, `malloc`/`new`). Аллокация требует поиска свободного блока памяти подходящего размера, что значительно медленнее стековых операций и может приводить к фрагментации памяти. Время жизни объектов здесь полностью контролируется программистом.
- **Статическая память (Static Storage Duration):** Предназначена для глобальных переменных и переменных, объявленных с модификатором `static`. Память выделяется при запуске программы и освобождается при ее завершении. Объекты инициализируются один раз.
- **Thread-Local Storage (TLS):** Специфическая область памяти, уникальная для каждого потока выполнения. Переменные с модификатором `thread_local` существуют в течение времени жизни потока.

1.2 Проблемы ручного управления памятью (C-style)

В языке C и в "legacy" C++ коде управление динамической памятью осуществлялось вручную через пары функций `malloc/free` или операторов `new/delete`. Этот подход порождает класс критических ошибок, связанных с человеческим фактором.

Утечки памяти и обработка ошибок

При возникновении исключительной ситуации или преждевременном возврате из функции программист обязан гарантировать вызов функции освобождения ресурсов. В коде с множественными точками выхода это приводит к дублированию логики очистки или использованию запутанных конструкций `goto`.

Рассмотрим пример обработки ресурсов в стиле C:

```
void process_data() {
    int* buffer = (int*)malloc(1024 * sizeof(int));
    if (!buffer) return;

    if (!init_network()) {
        free(buffer); // Ручная очистка при ошибке
        return;
    }

    if (!compute_hash(buffer)) {
        close_network(); // Необходимо закрыть предыдущий ресурс
        free(buffer);    // И освободить память
        return;
    }

    // Основная логика
    free(buffer);
    close_network();
}
```

Любое изменение логики (добавление нового ресурса или условия) требует модификации всех веток обработки ошибок, что экспоненциально увеличивает риск утечки ресурса (resource leak) или двойного освобождения (double free).

1.3 Идиома RAII (Resource Acquisition Is Initialization)

RAII — фундаментальная идиома C++, связывающая время жизни ресурса (памяти, файлового дескриптора, мьютекса) с временем жизни объекта на стеке.

> **RAII**
>
> Захват ресурса происходит в конструкторе объекта, а освобождение — в его деструкторе. Поскольку деструкторы автоматических переменных гарантированно вызываются при выходе из области видимости (в том числе при раскрутке стека из-за исключения), утечки ресурсов становятся невозможными при корректном использовании.

Пример RAII-обертки для файлового дескриптора C-style:

```cpp
#include <cstdio>
#include <stdexcept>

class FileHandle {
    FILE* file_;

public:
    explicit FileHandle(const char* filename) {
        file_ = std::fopen(filename, "r");
        if (!file_) {
            throw std::runtime_error("Failed to open file");
        }
    }

    ~FileHandle() {
        if (file_) {
            std::fclose(file_); // Гарантированное закрытие
        }
    }

    // Запрет копирования для избежания double-close
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    // Метод доступа
    FILE* get() const { return file_; }
};

void safe_processing() {
    FileHandle fh("data.txt");
    // ... работа с файлом ...
    throw std::runtime_error("Error");
    // Деструктор fh вызывается автоматически, файл закрывается.
}
```

Механизм размотки стека (stack unwinding) гарантирует вызов деструкторов для всех полностью сконструированных объектов в блоке до передачи управления обработчику исключения.

1.4 Механика new и delete

В C++ операторы `new` и `delete` выполняют две задачи: управление памятью и управление временем жизни объекта.

- **new expression:** 1. Вызывает функцию `operator new` для выделения "сырой" памяти (аналог `malloc`). 2. Конструирует объект в этой памяти (вызывает конструктор). 3. Возвращает типизированный указатель.
- **delete expression:** 1. Вызывает деструктор объекта. 2. Вызывает функцию `operator delete` для освобождения памяти (аналог `free`).

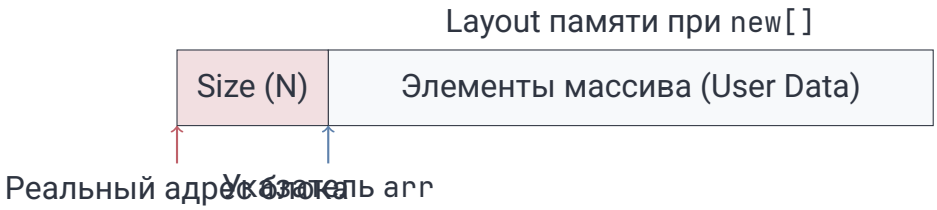
> **Важно!**
>
> C++ требует строгого соответствия форм аллокации и деаллокации:
>
> - `new T` → `delete ptr`
> - `new T[]` → `delete[] ptr`
>
> Смешивание этих форм (например, `delete` для массива) является Undefined Behavior.

Undefined Behavior при смешивании new[] и delete

Рассмотрим детально, почему следующий код вызывает неопределенное поведение, часто приводящее к повреждению кучи (heap corruption).

```
int* arr = new int[10];
delete arr; // ОШИБКА! UB
```

При использовании `new[]`, компилятор и аллокатор должны сохранить информацию о количестве элементов массива, чтобы `delete[]` мог вызвать деструкторы для **каждого** элемента. Часто это реализуется путем записи размера массива в памяти непосредственно *перед* адресом, возвращаемым пользователю (так называемый "cookie" или "overhead").



Сценарий сбоя:

1. `new int[10]` выделяет блок памяти размером $10 \times$ sizeof(int) $+$ sizeof(size_t). Размер (10) записывается в начало, указатель сдвигается и возвращается программе.
2. `delete arr` (без скобок) предполагает, что удаляется *один* объект.
3. Если тип тривиальный (как `int`), деструкторы не вызываются, но `operator delete` получает адрес `arr`. Аллокатор ожидает, что ему передадут *реальный адрес начала блока* (который может быть сдвинут на размер cookie).
4. Передача неверного адреса в функцию освобождения памяти ломает внутренние структуры аллокатора (free list, метаданные страниц), что приводит к падению программы не сразу, а при следующей аллокации (delayed crash).

Для нетривиальных типов последствия еще хуже: вызывается деструктор только первого элемента, а остальные 9 объектов остаются несконструированными "зомби" или утекают, если они владели ресурсами.

1.5 Умные указатели: std::unique_ptr

Стандарт C++11 ввел `std::unique_ptr` как замену устаревшему `std::auto_ptr` и сырым указателям. Это шаблонный класс, реализующий семантику эксклюзивного владения (exclusive ownership).

> **std::unique_ptr<T>**
>
> Обертка над сырым указателем, которая:
>
> - Гарантирует вызов `delete` (или `delete[]`) в деструкторе.
> - Не имеет оверхеда по памяти (размер равен `sizeof(void*)`, если используется дефолтный удалитель).
> - Запрещает копирование (конструктор копирования и оператор присваивания удалены).
> - Разрешает перемещение (Move Semantics), передавая владение другому объекту.

Специализация `std::unique_ptr<T[]>` корректно обрабатывает массивы, автоматически вызывая `delete[]`.

```
// Безопасное владение массивом
std::unique_ptr<int[]> safe_arr(new int[10]);
// При выходе из scope вызовется delete[]
```

1.6 std::make_unique vs new

Начиная с C++14, рекомендуется использовать фабричную функцию `std::make_unique` вместо прямого использования оператора `new`.

Проблема безопасности исключений:

Рассмотрим вызов функции, принимающей два уникальных указателя:

```
void process(std::unique_ptr<A> a, std::unique_ptr<B> b);

// ОПАСНЫЙ ВЫЗОВ
process(std::unique_ptr<A>(new A()), std::unique_ptr<B>(new B()));
```

C++ не гарантирует порядок вычисления аргументов функции. Возможна следующая последовательность операций компилятором:

1. `new A()` → выделена память для A.
2. `new B()` → **Выброшено исключение** (например, `std::bad_alloc`).
3. Конструктор `std::unique_ptr<A>` еще не был вызван.
4. Указатель на A теряется, происходит утечка памяти.

`std::make_unique` решает эту проблему, так как внутри функции создание объекта и оборачивание в умный указатель — это одна неразрывная операция.

```
// БЕЗОПАСНЫЙ ВЫЗОВ
process(std::make_unique<A>(), std::make_unique<B>());
```

Кроме безопасности, `make_unique` избавляет от дублирования типа (`unique_ptr<T>(new T)`) и делает код чище (принцип DRY).

Резюме раздела

- Используйте RAII для всех ресурсов.
- Избегайте сырых `new/delete`. Используйте `std::unique_ptr` и `std::make_unique`.
- Никогда не смешивайте скалярные и векторные формы `new/delete`.
- `std::unique_ptr` — это "бесплатная" абстракция с точки зрения производительности, обеспечивающая строгую семантику владения.

Глава 8

Система типов: Категории значений (Value Categories)

Понимание категорий значений — это необходимый фундамент для освоения семантики перемещения (Move Semantics) и идеальной передачи (Perfect Forwarding). В C++ каждое выражение (expression) характеризуется двумя независимыми свойствами: **типом** (например, `int`, `std::string`) и **категорией значения**.

Если тип определяет операции и представление в памяти, то категория значения определяет *время жизни* результата выражения и возможность его использования в качестве операнда для перемещения.

2.1 Таксономия значений (C++17)

До стандарта C++11 мир был прост: существовали только `lvalue` (left-hand side) и `rvalue` (right-hand side). С появлением семантики перемещения этой классификации стало недостаточно. Стандарт C++11 (и уточненный C++17) ввел более гранулярную систему.

В основе классификации лежат два ортогональных свойства выражения:

1. **Identity (m):** Обладает ли выражение идентичностью? Иными словами, есть ли у результата выражения стабильный адрес в памяти, по которому можно к нему обратиться?
2. **Movability (m):** Можно ли безопасно переместить ресурсы из объекта, который является результатом выражения? (Допускается ли "кража" состояния?)

На пересечении этих свойств образуются три фундаментальные (листовые) категории:

- **lvalue (identity + !movable):** Именованные объекты, функции, поля данных. У них есть адрес, но перемещать из них неявно нельзя (чтобы не сломать логику программы).
- **prvalue (pure rvalue, !identity + movable):** "Чистые" значения. Временные объекты, литералы (кроме строковых), результаты функций, возвращающих значение. Не имеют имени, существуют эфемерно.
- **xvalue (eXpiring value, identity + movable):** Объекты, у которых есть адрес ("плоть"), но которые помечены как "умирающие". Программа явно разрешила перемещение из них (например, результат `std::move`).

Для удобства стандарт вводит две группирующие категории:

- **glvalue (generalized lvalue):** Всё, что имеет идентичность (lvalue ∪ xvalue).

- **rvalue:** Всё, из чего можно перемещать (xvalue ∪ prvalue).

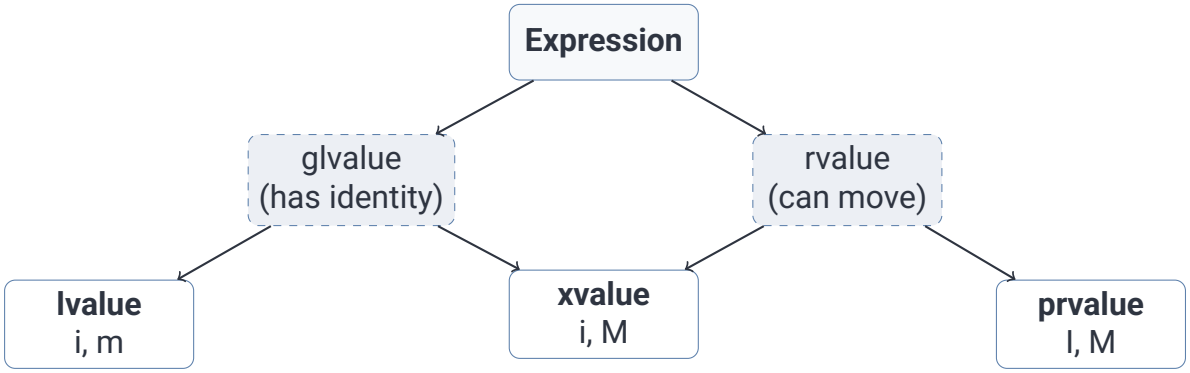


Рис. 8.1: Иерархия категорий значений в C++17

2.2 Практический анализ категорий

Рассмотрим код и определим категории значений для различных выражений. Ключевой маркер `lvalue` — возможность взять адрес через унарный оператор `&`.

```
int i = 1;
int* y = &i;                    // OK: i - это lvalue
// y = &42;                     // ОШИБКА: 42 - это prvalue, адреса нет
// 1 = i;                       // ОШИБКА: 1 - prvalue, нельзя присвоить В него

const char(*ptr)[5] = &"abcd";  // ВНИМАНИЕ: "abcd" - это lvalue!
char arr[20];
arr[10] = 'z';                  // Результат arr[10] - lvalue (ссылка на char)

int a, b;
(a, b) = 5;                   // Оператор запятая возвращает lvalue (ссылку на b)
```

Почему строковый литерал — это lvalue?

Это классический вопрос на понимание. Литерал `42` или `3.14` является `prvalue` — это просто битовое представление значения, которое может быть "зашито" прямо в инструкции процессора (immediate operand). У него нет адреса в оперативной памяти с точки зрения абстрактной машины C++.

Строковый литерал `"abcd"`, напротив, имеет тип `const char[5]`. Массивы в C++ не могут быть значениями-без-адреса. Компилятор размещает байты строки в сегменте статических данных (обычно `.rodata`), и выражение `"abcd"` обозначает именно этот массив в памяти. Следовательно, у него есть идентичность (адрес), и это **lvalue**.

2.3 Rvalue-ссылки (T&&)

Для захвата и работы с rvalue-категориями (временными объектами и xvalues) в C++11 был введен новый ссылочный тип: **Rvalue reference**, обозначаемый синтаксисом `T&&`.

- `T&` (lvalue reference): вяжется только к lvalue. (Исключение: `const T&` может вязаться к rvalue, продлевая ему жизнь, но объект будет неизменяемым).
- `T&&` (rvalue reference): вяжется только к rvalue. Позволяет изменять временный объект (например, "обворовывать" его).

```
int GetValue() { return 42; } // Возвращает prvalue

void demo() {
    int i = 10;

    int&  ref1 = i;           // OK: i is lvalue
    // int&& ref2 = i;        // ERROR: cannot bind rvalue ref to lvalue

    // int&  ref3 = GetValue(); // ERROR: cannot bind lvalue ref to prvalue
    const int& ref4 = GetValue(); // OK: Lifetime Extension (но read-only)

    int&& ref5 = GetValue();  // OK: ref5 ссылается на временный int(42)
    ref5 = 100;               // Мы можем менять временный объект!
}
```

Важно!

Имя переменной — это всегда lvalue. Если у вас есть переменная типа rvalue-ссылка:

```
void foo(int&& x) {
    // x имеет тип "rvalue reference to int"
    // Но само выражение 'x' является lvalue!
}
```

У `x` есть имя, к нему можно обратиться, взять его адрес. Тот факт, что он ссылается на что-то, что *было* временным, не делает саму переменную `x` временной. Это критически важно для понимания того, почему внутри move-конструкторов нужно писать `std::move`.

2.4 Materialization (C++17)

До C++17 `prvalue` часто трактовалось как временный объект. C++17 изменил определение: `prvalue` — это "инициализатор", рецепт создания объекта. Временный объект (в памяти) создается только тогда, когда это необходимо. Этот процесс называется **Temporary Materialization**.

Пример:

```
struct Big { int data[100]; };

Big make() { return Big(); } // Возвращает prvalue (инициализатор)

void use() {
    Big b = make();
    // В C++17 здесь нет ни копирования, ни перемещения.
```

```
    // prvalue из make() материализуется прямо в память переменной 'b'.
    // Это гарантированная RVO (Return Value Optimization).
}
```

Материализация происходит, когда prvalue нужно превратить в glvalue (например, чтобы привязать к ссылке или обратиться к полю).

Резюме раздела

- **lvalue:** Есть имя, есть адрес. Нельзя мувить неявно.
- **prvalue:** Чистое значение, инициализатор. Нет адреса.
- **xvalue:** Экс-lvalue, которое разрешили "грабить" (результат `std::move`).
- **glvalue:** lvalue + xvalue.
- **rvalue:** prvalue + xvalue.
- Переменная типа `T&&` сама по себе является **lvalue** в выражениях.

Глава 9

Семантика перемещения (Move Semantics): Механика и История

Семантика перемещения — это, пожалуй, самое значительное изменение в C++11, фундаментально изменившее подход к управлению ресурсами. Она позволяет не копировать данные, а передавать владение ими от одного объекта к другому. Чтобы понять, зачем это нужно, полезно взглянуть на историю языка.

3.1 Исторический контекст: Проблема лишних копий

До стандарта C++11 возврат тяжелых объектов из функций был сопряжен с накладными расходами. Рассмотрим классический пример фабричной функции, возвращающей вектор:

```
std::vector<int> LoadBigData() {
    std::vector<int> data(1000000);
    // ... заполнение данными ...
    return data;
}

void Process() {
    std::vector<int> my_data = LoadBigData();
}
```

В наивной реализации компилятора C++98 здесь происходило следующее:

1. Внутри `LoadBigData` создается локальный вектор (аллокация 4 МБ памяти).
2. При возврате создается временный объект-результат: вызывается конструктор копирования. Выделяются новые 4 МБ, данные копируются (`memcpy`). Локальный вектор уничтожается (освобождение 4 МБ).
3. В `Process` объект `my_data` инициализируется копированием из временного объекта. Еще одна аллокация, еще одно копирование. Временный объект уничтожается.

Итого: 3 аллокации, 2 глубоких копирования массивов. Хотя RVO (Return Value Optimization) существовала и раньше, она не была гарантирована во всех случаях (например, при сложной логике возврата или присваивании существующему объекту).

Трюк со swap

Чтобы избежать копирования, программисты использовали идиому `swap`. Вместо конструктора копирования "перемещение" имитировалось обменом внутреннего состояния с пустым объектом.

```cpp
// Pre-C++11 стиль
std::vector<int> temp;
LoadBigData(temp); // Передача по ссылке (out-parameter)
std::vector<int> my_data;
my_data.swap(temp); // Обмен указателями за O(1)
```

Это работало быстро, но делало код громоздким и лишало функции возможности возвращать значения естественным образом.

3.2 Катастрофа std::auto_ptr

Попытка реализовать семантику владения до C++11 привела к созданию `std::auto_ptr`. Это был класс умного указателя, который пытался "перемещать" ресурс при копировании.

```cpp
// Упрощенная логика auto_ptr
template <typename T>
struct auto_ptr {
    T* ptr;

    // "Копирующий" конструктор, который на самом деле перемещает
    auto_ptr(auto_ptr& other) {
        ptr = other.ptr;
        other.ptr = nullptr; // МОДИФИКАЦИЯ ИСТОЧНИКА!
    }
};

void bad_idea() {
    std::auto_ptr<int> a(new int(10));
    std::auto_ptr<int> b = a;
    // Теперь 'b' владеет ресурсом, а 'a' стал nullptr.
    // Это произошло при синтаксисе КОПИРОВАНИЯ.
}
```

Почему это плохо? Семантика копирования подразумевает создание независимого дубликата: оригинал не должен меняться. `auto_ptr` нарушал этот контракт. Это приводило к багам при использовании в стандартных контейнерах. Например, `std::sort` может брать "опорный" элемент копированием. В случае `auto_ptr` оригинал в массиве внезапно обнулялся, что приводило к потере данных и крашам. В C++11 `auto_ptr` был объявлен deprecated, а в C++17 удален из стандарта.

3.3 Философия Move Semantics

C++11 решил проблему, введя Rvalue-ссылки. Теперь мы можем перегружать функции для двух случаев:

1. `const T& source` — мы хотим **копировать** (источник важен, он останется неизменным).
2. `T&& source` — мы хотим **перемещать** (источник временный или нам явно разрешили его "разграбить").

Анатомия Move Constructor

Перемещающий конструктор "крадет" ресурсы у rvalue-объекта. Критически важно оставить источник в валидном состоянии, чтобы его деструктор отработал корректно.

Рассмотрим класс `Holder`, управляющий буфером памяти:

```
class Holder {
    int* data_;
    size_t size_;

public:
    Holder(int size) : size_(size), data_(new int[size]) {}

    ~Holder() {
        delete[] data_; // Деструктор должен быть безопасным для nullptr
    }

    // Move Constructor
    // Принимает неконстантную rvalue-ссылку
    Holder(Holder&& other) noexcept
        : data_(other.data_)  // 1. Крадем указатель
        , size_(other.size_)  // 2. Крадем метаданные
    {
        // 3. Зануляем источник!
        // Если этого не сделать, деструктор 'other' удалит память,
        // которой теперь владеем мы. Будет double free.
        other.data_ = nullptr;
        other.size_ = 0;
    }
};
```

Затраты: копирование двух скаляров (указатель + size). 0 аллокаций.

Анатомия Move Assignment Operator

Оператор присваивания сложнее, так как объект уже может владеть ресурсом, который нужно предварительно освободить. Также обязательна защита от самоприсваивания (`x = std::move(x)`).

```
Holder& operator=(Holder&& other) noexcept {
    if (this == &other) {
        return *this; // Защита от перемещения в себя
    }

```

```
    // 1. Освобождаем свой текущий ресурс
    delete[] data_;

    // 2. Крадем ресурсы
    data_ = other.data_;
    size_ = other.size_;

    // 3. Зануляем источник
    other.data_ = nullptr;
    other.size_ = 0;

    return *this;
}
```

3.4 std::move — это ложь

Одна из самых распространенных ошибок новичков — думать, что функция `std::move` что-то перемещает.

> **std::move**
>
> Это функция приведения типа. Она принимает объект любого типа (lvalue или rvalue) и безусловно приводит его к **rvalue-ссылке** (`T&&`).

Сам по себе вызов `std::move(x)` не генерирует никакого машинного кода для переноса данных. Он просто говорит компилятору: "Смотри на `x` как на временный объект". Перемещение происходит только тогда, когда результат `std::move` передается в конструктор или оператор присваивания, принимающий `T&&`.

```
std::vector<int> v1 = {1, 2, 3};
std::move(v1); // Ничего не происходит. v1 остался цел.

auto v2 = std::move(v1); // Вызывается vector(vector&&), вот теперь v1 пуст.
```

3.5 Ловушка реализации: Именованные Rvalue-ссылки

Это самый тонкий момент, на котором ошибаются даже опытные разработчики.

> **Важно!**
>
> **Если у rvalue-ссылки есть имя, то она — lvalue.**

Рассмотрим некорректную реализацию конструктора обертки:

```
class Wrapper {
    std::string str_;
public:
    // Мы принимаем строку как rvalue-ссылку (text)
    explicit Wrapper(std::string&& text)
```

```
        : str_(text) // ОШИБКА! Вызывается COPY constructor!
    {}
};
```

Разбор ошибки:

1. Параметр `text` имеет тип `std::string&&`.
2. Внутри тела (или списка инициализации) функции, `text` — это переменная с именем. У нее есть адрес. Следовательно, выражение `text` имеет категорию значения **lvalue**.
3. Конструктор `std::string` видит lvalue и вызывает копирование (`const string&`).

Мы потеряли эффективность. Чтобы исправить это, нужно снова превратить lvalue `text` в xvalue с помощью `std::move`:

```
explicit Wrapper(std::string&& text)
    : str_(std::move(text)) // Правильно: вызывается MOVE constructor
{}
```

3.6 Правило пяти (Rule of 5)

С появлением move-семантики "Правило трех" расширилось до "Правила пяти". Если классу требуется пользовательский деструктор (для освобождения ресурса), то, скорее всего, ему требуются:

1. Деструктор
2. Конструктор копирования
3. Оператор присваивания копированием
4. **Конструктор перемещения**
5. **Оператор присваивания перемещением**

Если вы не реализуете move-методы, компилятор не сгенерирует их автоматически, если у вас уже есть пользовательский деструктор или copy-методы. В этом случае класс будет только копируемым, что "убьет" производительность при работе с контейнерами.

Правило нуля (Rule of 0): Лучший способ написать класс — не управлять ресурсами вручную. Используйте `std::unique_ptr`, `std::vector`, `std::string`. У этих классов уже реализованы все 5 методов. Компилятор автоматически сгенерирует корректные дефолтные версии для вашего класса, которые просто вызовут соответствующие методы для полей.

```
class ModernHolder {
    // std::vector сам управляет памятью.
    // Нам не нужно писать ни деструктор, ни конструкторы.
    // Все сгенерируется автоматически и будет поддерживать move.
    std::vector<int> data_;
};
```

Резюме раздела

- Move-семантика заменяет глубокое копирование передачей указателей.
- `std::move` — это каст к rvalue, а не действие.
- Аргумент функции типа `T&&` внутри функции является lvalue. Используйте `std::move` для передачи его дальше.
- После перемещения объект находится в валидном, но неопределенном состоянии (valid but unspecified). Его можно уничтожить или присвоить ему новое значение, но читать из него нельзя.

Глава 10

Глава 4. Безопасность исключений и Контейнеры

Взаимодействие семантики перемещения с контейнерами стандартной библиотеки (в частности, `std::vector`) имеет нюанс, который часто упускают из виду. Этот нюанс касается гарантий безопасности исключений (Exception Safety Guarantees). Оказывается, наличие перемещающего конструктора не гарантирует, что вектор будет его использовать.

4.1 Проблема реалокации вектора

Вспомним, как работает `std::vector::push_back` (или `emplace_back`), когда его емкость (`capacity`) исчерпана:

1. Выделяется новый блок памяти большего размера (обычно $\times 2$).
2. Элементы из старого блока переносятся в новый.
3. Старый блок удаляется.

В C++98 элементы всегда **копировались**. Если конструктор копирования одного из элементов бросал исключение, вектор просто уничтожал уже созданные копии в новом буфере и освобождал его. Старый буфер оставался нетронутым. Это обеспечивало **Строгую гарантию исключений (Strong Exception Guarantee)**: операция либо выполняется успешно, либо не меняет состояние программы (транзакционность).

С появлением C++11 возникло желание **перемещать** элементы вместо копирования. Это намного быстрее. Но что, если перемещающий конструктор бросит исключение на середине процесса?

Сценарий сбоя:

1. Вектор начал перемещать элементы. Элементы 1 и 2 успешно перемещены в новый буфер. В старом буфере они теперь "пустые" (moved-from).
2. При перемещении Элемента 3 возникает исключение.
3. Мы должны откатить операцию ("unwind"), чтобы обеспечить Strong Guarantee.
4. Мы уничтожаем копии в новом буфере.
5. **Проблема:** Мы не можем "вернуть" Элементы 1 и 2 обратно в старый буфер, потому что операция обратного перемещения *тоже может бросить исключение*!

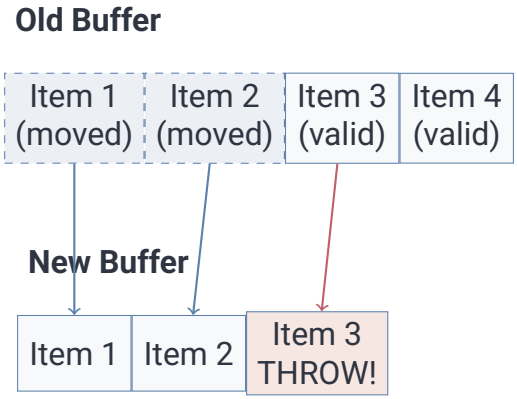


Рис. 10.1: Катастрофа при исключении во время Move

В итоге, у нас остались испорченные данные в старом буфере. Транзакционность нарушена.

4.2 Ключевое слово noexcept

Чтобы решить эту дилемму, `std::vector` использует стратегию: *"Перемещать только если это безопасно, иначе копировать"*.

Как компилятор узнает, безопасно ли перемещать объект? Для этого используется спецификатор `noexcept` в объявлении перемещающего конструктора.

```
class SafeMover {
public:
    // Мы гарантируем, что этот конструктор не бросит исключение
    SafeMover(SafeMover&&) noexcept {
        // ...
    }

    // Для копирования гарантий нет (может не хватить памяти)
    SafeMover(const SafeMover&) {
        // ...
    }
};

class UnsafeMover {
public:
    // noexcept отсутствует!
    UnsafeMover(UnsafeMover&&) {
        // ...
    }
    UnsafeMover(const UnsafeMover&) { /*...*/ }
};
```

При реалокации `std::vector` проверяет свойство типа через type traits (в частности, `std::is_nothr`

Algorithm 1: Псевдокод логики реалокации вектора

```
if std::is_nothrow_move_constructible<T>::value then
    Move: Используем std::move для переноса элементов.;
    Это быстро и безопасно (исключений не будет).;
else
    if std::is_copy_constructible<T>::value then
        Copy: Используем копирование.;
        Медленно, но если будет исключение, старый буфер останется цел.;
    else
        Move (Unsafe): Если тип нельзя копировать (например, unique_ptr), у нас нет
          выбора. Рискуем и перемещаем.;
```

4.3 std::move_if_noexcept

Стандартная библиотека предоставляет утилиту `std::move_if_noexcept`, которая инкапсулирует эту логику выбора. Она возвращает rvalue-ссылку (разрешая мув) только если мув-конструктор помечен как `noexcept`, иначе она возвращает const lvalue-ссылку (форсируя копирование).

```cpp
template <typename T>
void vector_realloc_stub(T* old_buf, T* new_buf, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        // Магия выбора:
        new (new_buf + i) T(std::move_if_noexcept(old_buf[i]));
    }
}
```

4.4 Идиома "Move or Copy"

Из вышесказанного следует важная рекомендация для разработчиков классов:

> **Важно!**
>
> **Всегда помечайте перемещающий конструктор и оператор присваивания как `noexcept`, если они не бросают исключений.**

Обычно перемещение сводится к копированию указателей и скалярных типов, что никогда не бросает исключений. Если вы забудете `noexcept`, ваш класс будет работать корректно, но `std::vector<YourClass>` будет молча копировать элементы при расширении, что может катастрофически снизить производительность (с $O(1)$ до $O(N)$ глубоких копий).

Пример правильного объявления (Rule of 5 с noexcept):

```cpp
class Efficient {
    std::string data;
public:
    Efficient(Efficient&& other) noexcept
        : data(std::move(other.data)) {} // std::string move is noexcept
```

```
6
7       Efficient& operator=(Efficient&& other) noexcept {
8           data = std::move(other.data);
9           return *this;
10      }
11
12      // Copy operations (can throw std::bad_alloc)
13      Efficient(const Efficient&) = default;
14      Efficient& operator=(const Efficient&) = default;
15  };
```

Резюме раздела

- `std::vector` гарантирует Strong Exception Safety.
- При реалокации вектор выбирает между Move и Copy.
- Выбор Move происходит **только** если Move Constructor помечен как `noexcept`.
- Если `noexcept` нет, происходит "молчаливое" копирование (Silent Pessimization).

Глава 11

Время жизни объектов (Object Lifetime) и Оптимизации

Управление временем жизни объектов — одна из самых сложных тем в C++. В отличие от языков с Garbage Collection, в C++ объект живет ровно столько, сколько определено его областью видимости (scope) или временем жизни контейнера. Однако существуют специальные правила, позволяющие продлевать жизнь временным объектам, а также оптимизации компилятора, которые могут эту жизнь "элиминировать" ради производительности.

5.1 Temporary Lifetime Extension

В C++ временные объекты (prvalue), созданные в выражении, обычно уничтожаются в конце выполнения этого "полного выражения" (full-expression), то есть на ближайшей точке с запятой.

Однако существует исключение: **Продление времени жизни (Lifetime Extension)**. Если временный объект привязывается к ссылке (константной lvalue-ссылке или rvalue-ссылке), то время жизни этого объекта продлевается до времени жизни самой ссылки.

```
struct Data {
    ~Data() { std::cout << "Dead\n"; }
};

Data create() { return Data(); }

void demo() {
    // 1. Обычное поведение
    create();
    std::cout << "After statement\n";
    // Output: Dead -> After statement

    // 2. Продление жизни
    const Data& ref = create(); // Временный объект НЕ умирает здесь!
    std::cout << "Using ref\n";
    // Output: Using ref -> Dead (при выходе из scope)
}
```

Это позволяет безопасно работать с временными результатами, не копируя их.

5.2 Опасные паттерны и висячие ссылки (Dangling Refs)

Правило продления жизни имеет критическое ограничение: оно **не транзитивно**. Оно работает только при *прямой* привязке результата выражения к ссылке. Если временный объект "спрятан" внутри другого объекта или возвращается через цепочку функций, продление может не сработать или сработать не так, как вы ожидаете.

Ловушка 1: Возврат ссылки на временный объект

Это классическое UB, но с нюансом продления.

```
const int& get_ref(const int& x) { return x; }

void trap() {
    // 1. Создается временный int(10).
    // 2. Он передается в get_ref.
    // 3. get_ref возвращает ссылку на него.
    // 4. Ссылка привязывается к val.
    // ПРОДЛЕНИЯ ЖИЗНИ НЕ ПРОИСХОДИТ!
    // Правило работает только если ссылка инициализируется САМИМ prvalue,
    // а здесь она инициализируется результатом функции (lvalue/xvalue).
    const int& val = get_ref(10);

    std::cout << val; // UB: обращение к памяти мертвого объекта
}
```

Ловушка 2: Доступ к полю временного объекта

Очень частая ошибка при использовании геттеров.

```
struct Wrapper {
    std::string name;
    const std::string& get_name() const { return name; }
};

Wrapper make_wrapper() { return Wrapper{"test"}; }

void disaster() {
    // make_wrapper() создает временный Wrapper.
    // Мы берем ссылку на его поле.
    // В конце строки временный Wrapper уничтожается.
    // Поле name уничтожается вместе с ним.
    const std::string& s = make_wrapper().get_name();

    std::cout << s; // UB! Ссылка 's' висит.
}
```

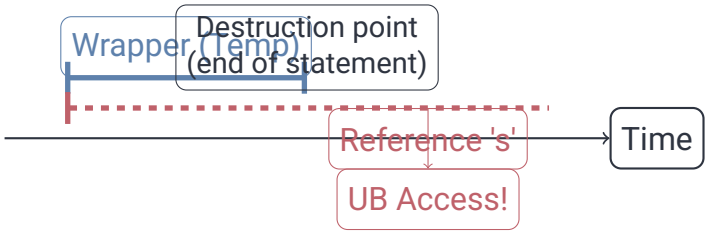


Рис. 11.1: Диаграмма времени жизни при доступе к полю временного объекта

5.3 RVO и Copy Elision

Компиляторы C++ обладают правом (а начиная с C++17 — обязанностью в некоторых случаях) полностью исключать создание временных объектов. Это называется **Return Value Optimization (RVO)**.

```
std::vector<int> generate() {
    return std::vector<int>(1000); // prvalue
}

void use() {
    std::vector<int> v = generate();
}
```

Без RVO: 1. Создается временный вектор внутри `generate`. 2. Он перемещается/копируется во временный результат возврата. 3. Результат перемещается/копируется в `v`.

С RVO (Copy Elision): Компилятор передает адрес переменной `v` внутрь функции `generate`. Вектор конструируется **прямо на месте** `v`. Никаких копий, никаких перемещений.

5.4 Pessimizing Move

С появлением `std::move` программисты начали писать его везде, думая, что помогают компилятору. В случае возврата значения это приводит к обратному эффекту — **Pessimizing Move**.

```
std::vector<int> bad_func() {
    std::vector<int> local_vec;
    // ...
    // ОШИБКА! Мы явно приводим к rvalue-ссылке.
    // Это ЗАПРЕЩАЕТ RVO, так как типы не совпадают (Obj vs Obj&&).
    // Компилятор обязан вызвать Move Constructor.
    return std::move(local_vec);
}

std::vector<int> good_func() {
    std::vector<int> local_vec;
    // ...
    // ПРАВИЛЬНО! Работает NRVO (Named RVO).
    // Объект будет построен сразу в целевой памяти.
    return local_vec;
}
```

Важно!

Никогда не делайте `std::move` для локальной переменной в операторе `return`. Компилятор и так попробует применить NRVO. Если NRVO невозможно (сложная логика условий), компилятор *автоматически* применит move (как будто вы написали `std::move`). Ручной `std::move` здесь только мешает оптимизации.

Резюме раздела

- Ссылки могут продлевать жизнь временным объектам, но только "на один шаг".
- Остерегайтесь ссылок на части временных объектов.
- RVO/NRVO — самая мощная оптимизация в C++.
- `return std::move(x)` — это вредная привычка (Pessimizing Move).

Глава 12

Perfect Forwarding и Универсальные ссылки

Одной из самых мощных, но в то же время запутанных возможностей C++11 стала "Идеальная передача" (Perfect Forwarding). Эта механика позволяет писать обобщенные функции-обертки (wrappers), которые передают свои аргументы в другие функции **в точности** так, как они были получены: с сохранением категории значения (lvalue/rvalue) и cv-квалификаторов (const/volatile).

6.1 Проблема передачи аргументов

Представьте, что мы пишем фабричную функцию `make_shared`, которая должна создать объект типа `T`, передав аргументы в его конструктор.

Наивный подход с использованием константных ссылок:

```
template <typename T, typename Arg>
T* factory(const Arg& arg) {
    return new T(arg);
}
```

Проблема: Если конструктор `T` принимает неконстантную ссылку (хочет изменить аргумент), этот код не скомпилируется. Если конструктор принимает rvalue-ссылку (move-семантика), мы передадим ему lvalue (так как `arg` имеет имя), и перемещения не произойдет.

Попытка перегрузки:

```
template <typename T, typename Arg>
void wrapper(Arg& arg) { func(arg); }

template <typename T, typename Arg>
void wrapper(const Arg& arg) { func(arg); }
```

Проблема: Количество перегрузок растет экспоненциально (2^N) от количества аргументов. Для 5 аргументов нужно написать 32 версии функции. Это неприемлемо.

6.2 Универсальные ссылки (Forwarding References)

Скотт Мейерс ввел термин "Universal Reference" (в стандарте C++17 закреплен термин **Forwarding Reference**), чтобы описать особый вид ссылок в шаблонах.

> **Forwarding Reference (T&&)**
>
> Если переменная или параметр объявлены как `T&&`, где `T` — это **выводимый** тип шаблона (deduced type), то это не rvalue-ссылка, а универсальная ссылка. Она может "превращаться" как в lvalue-ссылку, так и в rvalue-ссылку в зависимости от того, что передали на вход.

Синтаксис `T&&` работает по-разному в двух контекстах:

```
// 1. Rvalue Reference
void foo(int&& x); // Нет вывода типов. Это точно rvalue ref.

template <typename T>
void bar(std::vector<T>&& x); // T выводится, но && относится к vector, а не к T.

// 2. Forwarding Reference
template <typename T>
void func(T&& x); // T выводится. Это универсальная ссылка.

auto&& var = some_expression; // auto выводится. Это универсальная ссылка.
```

6.3 Правила схлопывания ссылок (Reference Collapsing)

Как компилятор понимает, во что превратить `T&&`? Это определяется механизмом дедукции типа и правилами схлопывания ссылок.

Когда мы передаем аргумент в шаблонную функцию `func(T&& x)`:

1. Если передан **lvalue** (например, `int i`), то `T` выводится как `int&`.
2. Если передан **rvalue** (например, `42`), то `T` выводится как `int`.

Теперь подставим выведенные типы в сигнатуру `T&&`. В C++ запрещены ссылки на ссылки (нельзя написать `int& &`), но компилятор может их генерировать в процессе инстанцирования. В этот момент вступают в силу правила "схлопывания":

Тип T	Сигнатура (T + &&)	Результат
`int&` (lvalue)	`int& &&`	`int&` (lvalue ref)
`int` (rvalue)	`int &&`	`int&&` (rvalue ref)

Таблица 12.1: Упрощенное правило: "Lvalue выигрывает всегда"

Полная таблица Collapsing Rules:

- `T&` + `&` → `T&`
- `T&` + `&&` → `T&`
- `T&&` + `&` → `T&`
- `T&&` + `&&` → `T&&`

Именно это позволяет одной сигнатуре T&& принимать всё.

6.4 Механика std::forward

Теперь, когда у нас есть универсальная ссылка x, нам нужно передать её дальше. Но помните: **именованная rvalue-ссылка — это lvalue**. Если мы просто напишем `call(x)`, мы всегда будем передавать lvalue, теряя move-семантику. Нам нужен механизм, который делает `std::move`, но *только если изначально пришло rvalue*.

Эту роль выполняет `std::forward<T>`.

```
template <typename T>
void wrapper(T&& arg) {
    // std::forward кастит arg обратно к его "изначальной" категории
    target_func(std::forward<T>(arg));
}
```

Как это работает "под капотом"? Реализация `std::forward` выглядит примерно так (упрощенно):

```
// Если T = int& (передали lvalue)
// Возвращает: int& && -> int& (lvalue)
template <typename T>
T&& forward(typename remove_reference<T>::type& t) {
    return static_cast<T&&>(t);
}

// Если T = int (передали rvalue)
// Возвращает: int&& (rvalue)
template <typename T>
T&& forward(typename remove_reference<T>::type& t) {
    return static_cast<T&&>(t);
}
```

Таким образом, `std::forward` — это **условный cast**:

- Если `T` — ссылочный тип, он кастит к ссылке (lvalue).
- Если `T` — не ссылочный тип, он кастит к rvalue-ссылке (действует как `std::move`).

6.5 Практическое применение

1. Emplace-методы контейнеров

Методы `emplace_back` вектора используют perfect forwarding для конструирования элемента прямо в памяти контейнера, минуя создание временных объектов.

```
std::vector<std::string> v;

// Плохо: создание временного string, затем move
```

```
v.push_back(std::string("hello"));

// Идеально: аргумент "hello" пробрасывается в конструктор string,
// который вызывается прямо в памяти вектора (placement new).
v.emplace_back("hello");
```

2. make_unique / make_shared

Эти функции просто пересылают все свои аргументы в конструктор целевого типа.

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args) {
    return unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Резюме раздела

- T&& в шаблоне — это Forwarding Reference, а не rvalue reference.
- Правила схлопывания ссылок гарантируют, что lvalue остается lvalue.
- `std::move` — безусловное приведение к rvalue.
- `std::forward<T>` — условное приведение, восстанавливающее исходную категорию значения.

Глава 13

Продвинутые идиомы C++ и Архитектурные паттерны

Завершая изучение семантики перемещения и управления памятью, мы рассмотрим продвинутые техники, которые позволяют выжать максимум из системы типов C++. Эти идиомы часто используются в библиотечном коде (например, STL или Boost) для оптимизации использования памяти и улучшения эргономики API.

7.1 Ref-qualifiers (Квалификаторы ссылок для методов)

Традиционно методы класса в C++ перегружались только по константности объекта (`this`):

```
struct Data {
    std::string value;
    const std::string& get() const { return value; } // Для const объектов
    std::string& get() { return value; }             // Для non-const объектов
};
```

Однако, что если мы вызываем `get()` у временного объекта (rvalue)?

```
auto str = Data{"heavy_string"}.get();
```

В данном случае вызовется неконстантная версия `get()`, которая вернет lvalue-ссылку на поле. Затем произойдет **копирование** строки в переменную `str`, так как lvalue-ссылка не позволяет применить move-семантику. И это несмотря на то, что сам объект `Data` является временным и будет уничтожен сразу после этой строки! Мы упускаем возможность "украсть" данные.

C++11 позволяет перегружать методы по категории значения объекта `*this`, используя синтаксис & и &&.

```
struct ModernData {
    std::vector<int> heavy;

    // 1. Вызывается для lvalue (именованных объектов)
    // Мы вынуждены вернуть const reference (копирование на стороне клиента)
```

```
    const std::vector<int>& items() const& {
        return heavy;
    }

    // 2. Вызывается для rvalue (временных объектов)
    // Мы можем безопасно отдать ресурсы, так как объект скоро умрет
    std::vector<int> items() && {
        return std::move(heavy);
    }
};

void demo() {
    ModernData d;
    auto v1 = d.items();              // Вызов (1): Копирование (безопасно)

    auto v2 = ModernData{}.items();  // Вызов (2): Перемещение!
    // v2 забирает буфер у временного вектора. 0 копий.
}
```

Этот механизм позволяет писать API, которые автоматически оптимизируются для временных объектов, предотвращая ненужные глубокие копии.

7.2 Empty Base Optimization (EBO)

В C++ любой объект должен иметь уникальный адрес. Из этого следует, что размер даже пустой структуры не может быть равен нулю.

```
struct Empty {};
static_assert(sizeof(Empty) >= 1); // Обычно 1 байт
```

Это создает проблему при композиции. Если мы включим пустой класс как поле в другой класс, он "съест" минимум 1 байт + возможный padding (выравнивание).

```
struct A {
    Empty e; // 1 байт
    int i;   // 4 байта
    // Итоговый размер может быть 8 байт из-за выравнивания!
};
```

Однако стандарт разрешает оптимизацию: если пустой класс является **базовым**, он может иметь нулевой размер (его часть в layout объекта "схлопывается"). Это называется **Empty Base Optimization (EBO)**.

```
struct B : Empty { // EBO работает!
    int i;
};
static_assert(sizeof(B) == sizeof(int)); // 4 байта
```

Где это используется? В аллокаторах, делетерах и компараторах STL. Например, `std::unique_ptr` хранит указатель и делетер. Если делетер — это пустая структура (без состояния, только

`operator()`), то благодаря EBO размер `unique_ptr` остается равным размеру сырого указателя. Если бы делетер хранился как поле, размер указателя удвоился бы.

7.3 Опасность const T&&

Тип `const T&&` (константная rvalue-ссылка) является синтаксически валидным, но семантически бесполезным и даже вредным.

1. **Нельзя изменить:** Мы не можем "обворовать" объект, так как он `const`.
2. **Нельзя смувить:** Конструктор перемещения требует `T&&`, а не `const T&&`.

При попытке применить `std::move` к константному объекту происходит тихое "предательство":

```
const std::string s = "data";
// std::move(s) кастит к 'const string&&'
// Нет конструктора string(const string&&)
// Компилятор ищет ближайшее совпадение... string(const string&)
// ВЫЗЫВАЕТСЯ КОПИРОВАНИЕ!
std::string s2 = std::move(s);
```

Вывод: Никогда не используйте `const` для rvalue-ссылок и возвращаемых значений, если планируете перемещение. `const` блокирует move-семантику.

7.4 Destructive Move vs Non-destructive Move

C++ реализует **неразрушающее перемещение (Non-destructive Move)**. Это означает, что после перемещения исходный объект (*source*) остается жить. Он переходит в состояние "валидное, но неопределенное". Для него обязательно будет вызван деструктор.

- **Плюсы:** Совместимость с RAII. Деструктор всегда очищает ресурсы, даже у moved-from объекта (который просто ничего не делает, так как ресурсы обнулены).
- **Минусы:** Оверхед. Нам нужно вручную занулять указатели в источнике. Деструктор вызывается "вхолостую".

В языке **Rust**, напротив, реализовано **разрушающее перемещение (Destructive Move)**.

- Перемещение — это просто `memcpy` битов.
- Исходный объект считается "мертвым" сразу после перемещения.
- Деструктор для исходного объекта **не вызывается**.
- Компилятор статически запрещает обращение к перемещенной переменной.

Подход Rust более эффективен (нет лишних записей нулей, нет лишних вызовов деструкторов), но требует более строгой системы типов (borrow checker), чтобы гарантировать безопасность на этапе компиляции. В C++ "зомби-объекты" (moved-from) — это плата за гибкость и совместимость с легаси-кодом.

Резюме раздела

- **Ref-qualifiers** позволяют перегружать методы для rvalue-объектов, оптимизируя цепочки вызовов.
- **EBO** позволяет "бесплатно" хранить stateless-объекты (политики, аллокаторы) внутри классов.
- `const` и `move` несовместимы.
- В C++ перемещение оставляет "пустую оболочку", которую нужно корректно уничтожить. В Rust перемещение уничтожает оболочку логически.

Часть III

Лекция 03 – Продолжаем про move семантику

Глава 14

Механика перемещения и специальные функции класса

Современный C++ (начиная с C++11) ввел концепцию Move Semantics (семантика перемещения), которая фундаментально изменила подход к управлению ресурсами. Если раньше передача тяжелого объекта (например, `std::vector` на гигабайт данных) означала глубокое копирование, то теперь мы можем передать владение ресурсом за константное время. В этой главе мы детально разберем, как реализовать перемещающие операции, как избежать распространенных ошибок и как компилятор генерирует (или удаляет) специальные функции класса.

Анатомия Move Assignment Operator

Оператор перемещающего присваивания (`move assignment operator`) — это специальная функция-член класса, которая вызывается, когда объекту присваивается *rvalue* того же типа. В отличие от копирующего присваивания, цель этой операции — не дублировать ресурс, а *украсть* его у временного объекта, оставив последний в состоянии, пригодном для безопасного уничтожения.

Паттерн реализации идиомы copy-and-swap

Один из классических способов реализации операторов присваивания — идиома **copy-and-swap** (в контексте move — скорее *move-and-swap*). Она выглядит следующим образом:

```
class Holder {
public:
    // ... конструкторы ...

    // Move assignment
    Holder& operator=(Holder&& other) noexcept {
        // Меняем местами текущее состояние с состоянием other
        swap(*this, other);
        return *this;
    }

private:
    void swap(Holder& a, Holder& b) {
```

```
        // ... реализация обмена полей ...
    }
};
```

На заметку

Механика: При вызове `swap`, текущий ресурс объекта (который нужно уничтожить) перемещается в `other`, а ресурс из `other` перемещается в `*this`. Поскольку `other` — это *rvalue*, который скоро будет уничтожен, он заберет старый ресурс нашего объекта с собой в могилу (вызовет деструктор).

Однако, у этого подхода есть недостатки:

1. **Лишняя работа:** `swap` выполняет три перемещения (A -> tmp, B -> A, tmp -> B). Для простого присваивания это может быть избыточно.
2. **Отложенное уничтожение:** Старый ресурс уничтожается не сразу, а только когда будет вызван деструктор `other`. Это может быть критично, если ресурс удерживает, например, файловый дескриптор или мьютекс.

Оптимизированная реализация через std::exchange

Более современный и эффективный подход — использование `std::exchange` (C++14). Эта функция заменяет значение объекта новым и возвращает старое. Это позволяет реализовать перемещение "в одну строчку" с четкой семантикой передачи владения.

```
#include <utility> // для std::exchange

class Buffer {
    char* data_ = nullptr;
    size_t size_ = 0;

public:
    // ...

    Buffer& operator=(Buffer&& other) noexcept {
        if (this == &other) {
            return *this; // Защита от self-assignment
        }

        // 1. Освобождаем текущий ресурс
        delete[] data_;

        // 2. Забираем ресурс у other и "зануляем" его
        data_ = std::exchange(other.data_, nullptr);
        size_ = std::exchange(other.size_, 0);

        return *this;
    }
};
```

В этом коде `std::exchange(other.data_, nullptr)` делает следующее:

1. Читает текущее значение `other.data_`.
2. Присваивает `other.data_` значение `nullptr`.
3. Возвращает прочитанное (старое) значение, которое мы записываем в `this→data_`.

Это устраняет промежуточные свопы и гарантирует, что объект-источник (`other`) останется в пустом состоянии.

Проблема Self-Assignment (Самоприсваивание)

В копирующем присваивании проверка `if (this == &other)` обязательна, чтобы не удалить свой собственный ресурс перед копированием. В перемещающем присваивании ситуация тоньше.

> **Важно!**
>
> Если вы используете реализацию через `swap`, самоприсваивание безопасно (своп объекта с самим собой ничего не ломает), но бесполезно тратит такты процессора. Если вы используете прямую реализацию с `delete[] data_`, то без проверки `if (this == &other)` вы удалите ресурс, который собираетесь переместить в себя же -> **Undefined Behavior** (use-after-free).

Хотя самоприсваивание rvalue (написание `x = std::move(x)`) — это редкая и странная операция, корректная реализация должна её обрабатывать. Стандартная библиотека требует, чтобы объекты были устойчивы к самоприсваиванию.

Концепция "Moved-from state"

Когда мы перемещаем данные из объекта (источника), он остается в состоянии, которое стандарт называет "**valid but unspecified state**" (валидное, но неопределенное).

> **Moved-from state**
>
> Объект, из которого переместили данные, должен находиться в таком состоянии, что:
>
> - Его деструктор отработает корректно (не упадет, не сделает double free).
> - Ему можно присвоить новое значение (оператор присваивания сработает корректно).
> - Значение его полей не гарантировано (если явно не оговорено классом).

Рассмотрим пример "плохого" перемещения:

```
class BadString {
    char* str;
public:
    BadString(BadString&& other) {
        // Мы скопировали указатель...
        this->str = other.str;
        // ...НО ЗАБЫЛИ занулить other.str!
    }

    ~BadString() {
```

```
        delete[] str;
    }
};
```

В этом случае, когда `other` выйдет из области видимости, его деструктор удалит память по адресу `str`. Позже деструктор нашего нового объекта снова попытается удалить `str`. Это классический **double free**.

Правило: **Всегда оставляйте источник в состоянии, безопасном для разрушения (обычно это null pointers, zero size).**

Rule of 5 vs Rule of 0

C++11 расширил "Правило трех" (Rule of 3) до "Правила пяти" (Rule of 5) из-за добавления семантики перемещения.

1. **Деструктор**
2. **Копирующий конструктор**
3. **Копирующий оператор присваивания**
4. **Перемещающий конструктор**
5. **Перемещающий оператор присваивания**

Rule of 5

Если классу требуется ручное управление ресурсом (например, сырой указатель, дескриптор файла), вы, скорее всего, должны реализовать (или явно объявить) все 5 специальных функций.

```
class ResourceManager {
    Resource* res;
public:
    ~ResourceManager() { delete res; } // 1

    ResourceManager(const ResourceManager& other) { ... } // 2
    ResourceManager& operator=(const ResourceManager& other) { ... } // 3

    ResourceManager(ResourceManager&& other) noexcept { ... } // 4
    ResourceManager& operator=(ResourceManager&& other) noexcept { ... } // 5
};
```

Rule of 0

Если ваш класс не управляет ресурсами напрямую, а использует RAII-обертки (`std::string`, `std::vector`, `std::unique_ptr`), вам **не нужно** писать ни одну из специальных функций. Компилятор сгенерирует их автоматически и корректно.

> **Важно!**
>
> **Rule of 0** — предпочтительный подход. Это принцип "Пиши меньше кода". Если вы можете выразить семантику класса через комбинацию стандартных типов, делайте это.

Implicit Deletion (Неявное удаление)

Правила генерации специальных функций в C++ довольно запутаны. Одно из важнейших правил:

> **На заметку**
>
> Если вы объявляете (или определяете) любую перемещающую операцию (конструктор или оператор присваивания), то **копирующие операции неявно удаляются** (становятся `delete`).

Это сделано для безопасности: если класс поддерживает перемещение, скорее всего, он владеет уникальным ресурсом (как `unique_ptr`), который нельзя просто скопировать.

Пример ловушки:

```
class Widget {
public:
    Widget(Widget&&) noexcept; // Пользовательский move ctor
    // Copy ctor неявно удален!
    // Copy assignment неявно удален!
    // Move assignment не объявлен (но и не удален, просто не сгенерирован)
};

void func() {
    Widget w1;
    Widget w2 = w1; // ОШИБКА КОМПИЛЯЦИИ: call to deleted constructor
}
```

Если вы хотите вернуть копирование, вы должны объявить его явно: `Widget(const Widget&) = default;`

noexcept в перемещающих операциях

Ключевое слово `noexcept` критически важно для производительности перемещения, особенно при работе со стандартными контейнерами, такими как `std::vector`.

Проблема транзакционности вектора

Рассмотрим, что происходит, когда `std::vector` исчерпывает свою емкость (`capacity`) и должен расшириться: 1. Выделяется новый, больший буфер памяти. 2. Элементы переносятся из старого буфера в новый. 3. Старый буфер удаляется.

В C++98 элементы всегда копировались. Если при копировании N-го элемента возникало исключение, вектор просто уничтожал уже созданные копии в новом буфере и освобождал

его. Старый буфер оставался нетронутым. Это обеспечивало **Strong Exception Guarantee** (строгую гарантию исключений): операция либо выполняется полностью, либо не меняет состояние программы.

В C++11 мы хотим перемещать элементы (это дешевле). Но если перемещающий конструктор кинет исключение на N-м элементе, мы оказываемся в беде:

- Часть элементов уже перемещена в новый буфер.
- Их "оригиналы" в старом буфере теперь находятся в *moved-from state* (по сути, разрушены или пусты).
- Мы не можем "откатить" операцию, потому что обратное перемещение тоже может кинуть исключение!
- Данные безвозвратно потеряны.

std::move_if_noexcept

Чтобы сохранить Строгую гарантию исключений, `std::vector` использует утилиту `std::move_if_n`
Логика при реаллокации следующая:

- Если перемещающий конструктор типа элемента помечен как `noexcept`, вектор использует перемещение.
- Если `noexcept` нет, вектор **откатывается к копированию** (использует copy constructor), даже если доступен move constructor.
- Если тип *move-only* (как `unique_ptr`) и не `noexcept`, вектор не дает гарантий безопасности исключений (или не компилируется, в зависимости от реализации).

> **Важно!**
>
> Если вы забыли написать `noexcept` у перемещающего конструктора, ваш код будет компилироваться и работать, но `std::vector<MyType>` будет работать **медленно**, выполняя глубокое копирование при каждом расширении. Это "тихий убийца производительности".

```
class FastWidget {
public:
    // Обязательно noexcept!
    FastWidget(FastWidget&&) noexcept { /* ... */ }

    // Аналогично для присваивания
    FastWidget& operator=(FastWidget&&) noexcept { /* ... */ }
};
```

Техническое отступление: noexcept оператор

В C++ `noexcept` — это не только спецификатор, но и оператор, который возвращает `bool` на этапе компиляции. Он проверяет, может ли выражение теоретически кинуть исключение.

```
template <typename T>
class Wrapper {
```

```
    T value;
public:
    // Move ctor является noexcept ТОЛЬКО если T имеет noexcept move ctor
    Wrapper(Wrapper&& other) noexcept(std::is_nothrow_move_constructible_v<T>)
        : value(std::move(other.value)) {}
};
```

Это позволяет создавать гибкие шаблоны, которые транслируют гарантии исключений вложенных типов.

Глава 15

Универсальные ссылки и идеальная передача (Perfect Forwarding)

В предыдущей главе мы обсуждали rvalue-ссылки (`T&&`) как инструмент для реализации семантики перемещения. Однако в шаблонах C++ синтаксис `T&&` приобретает совершенно иное значение. Это одна из самых запутанных тем для новичков: одни и те же символы могут означать "rvalue-ссылка" или "универсальная ссылка" (Universal Reference), в зависимости от контекста.

В этой главе мы разберем механику вывода типов, правила схлопывания ссылок (Reference Collapsing) и современные способы работы с квалификаторами значений, включая нововведения C++23.

Универсальные ссылки (Forwarding References)

Термин "Universal Reference" был введен Скоттом Мейерсом, но в стандарте C++ он называется **Forwarding Reference**. Это ссылка, которая может вести себя и как lvalue-ссылка, и как rvalue-ссылка, в зависимости от того, чем она инициализирована.

> **Универсальная ссылка**
>
> Ссылка является универсальной только при выполнении двух условий одновременно:
>
> 1. Вывод типа (*type deduction*) происходит именно для этой переменной.
> 2. Переменная объявлена строго как `T&&` (где `T` — имя шаблонного параметра).

Рассмотрим разницу на примерах:

```
// 1. Обычная Rvalue-ссылка
void func(Widget&& w); // Тип Widget конкретен, вывода типов нет.
                       // Принимает ТОЛЬКО rvalue.

// 2. Универсальная ссылка
template <typename T>
void wrapper(T&& arg); // T выводится компилятором.
                       // Принимает И lvalue, И rvalue.

```

```cpp
// 3. Подвох: это НЕ универсальная ссылка
template <typename T>
void vector_push(std::vector<T>&& v); // T выводится, но вид ссылки
                                      // искажен (vector<T>&&).
                                      // Это rvalue-ссылка на вектор.

// 4. Еще подвох: const убивает универсальность
template <typename T>
void const_wrapper(const T&& arg); // const T&& — это всегда rvalue-ссылка,
                                   // но только на const объекты.
```

Главное свойство универсальной ссылки: она "впитывает" категорию значения (value category) переданного аргумента.

Математика ссылок: Reference Collapsing

Как компилятор понимает, во что превратить T&&? Здесь вступает в силу правило **схлопывания ссылок** (Reference Collapsing).

В C++ запрещено создавать "ссылку на ссылку" явно (нельзя написать `int& & x`), но компилятор может сгенерировать такую конструкцию в процессе инстанцирования шаблона. Когда это происходит, две ссылки объединяются в одну по строгим правилам.

Пусть мы передаем аргумент в функцию `template <typename T> void f(T&& param)`.

1. **Сценарий 1: Передаем lvalue (например, int x).** Компилятор выводит T как `int&` (lvalue-ссылка). Подставляем в сигнатуру T&&:

   $$\texttt{int\&} + \&\& \rightarrow \texttt{int\&}$$

   Результат: Функция принимает lvalue-ссылку.

2. **Сценарий 2: Передаем rvalue (например, 42).** Компилятор выводит T как `int` (не ссылка). Подставляем в сигнатуру T&&:

   $$\texttt{int} + \&\& \rightarrow \texttt{int\&\&}$$

   Результат: Функция принимает rvalue-ссылку.

Полная таблица правил схлопывания (где левая часть — тип T, правая — спецификатор параметра):

Тип T	Спецификатор	Результат
type&	&	type&
type&	&&	type&
type&&	&	type&
type&&	&&	type&&

Таблица 15.1: Правила Reference Collapsing

> **Важно!**
>
> Интуитивное правило: **Lvalue-ссылка заразна**. Если в уравнении появляется хотя бы один одиночный амперсанд (&), результат всегда будет lvalue-ссылкой. Rvalue-ссылка получается только из комбинации двух двойных амперсандов (или типа без ссылок + &&).

Идеальная передача: std::forward vs std::move

Универсальные ссылки чаще всего используются для *Perfect Forwarding* — передачи аргументов в другую функцию с сохранением их категории (lvalue остается lvalue, rvalue остается rvalue).

Почему нельзя использовать std::move?

Рассмотрим, что произойдет, если мы применим `std::move` к универсальной ссылке.

```
void process(const Widget& lval) { std::cout << "Lvalue processed\n"; }
void process(Widget&& rval)      { std::cout << "Rvalue processed\n"; }

template <typename T>
void log_and_call(T&& arg) {
    std::cout << "Logging...\n";
    // ОШИБКА! std::move безусловно кастит к rvalue!
    process(std::move(arg));
}

int main() {
    Widget w;
    log_and_call(w); // Мы передаем lvalue...
}
```

В примере выше: 1. Вызывается `log_and_call(w)`. `T` выводится как `Widget&`. 2. `arg` имеет тип `Widget&`. 3. `std::move(arg)` кастит его к `Widget&&` (rvalue). 4. Вызывается перегрузка `process(Widget&&)`. 5. **Результат:** Мы передали в функцию живой lvalue объект `w`, но функция `process` считает его временным и может "украсть" его данные. После возврата из `log_and_call` объект `w` может оказаться пустым. Это катастрофа.

Решение: std::forward

Для решения этой проблемы используется `std::forward<T>`. В отличие от `std::move`, который делает *безусловный* каст, `std::forward` делает *условный* каст.

- Если `T` — lvalue-ссылка, `forward` возвращает lvalue.
- Если `T` — не ссылка (или rvalue-ссылка), `forward` возвращает rvalue.

```
template <typename T>
void log_and_call(T&& arg) {
    std::cout << "Logging...\n";
    // T передается явно как параметр шаблона forward
    process(std::forward<T>(arg));
}
```

Теперь при вызове `log_and_call(w)` (lvalue), `std::forward` вернет `Widget&`, и вызовется безопасная перегрузка. При вызове `log_and_call(Widget())` (rvalue), вернется `Widget&&`, и ресурсы будут эффективно перемещены.

Ref-qualifiers: Перегрузка методов для *this

Иногда нам нужно знать категорию значения не аргумента, а самого объекта, у которого вызывается метод (`*this`). Это особенно актуально для паттерна **Builder** или для оптимизации геттеров.

До C++11 мы могли перегружать методы только по `const`. C++11 добавил **Ref-qualifiers**: возможность указывать & или && после списка аргументов метода.

Пример: Паттерн Builder

Представим построитель сложного объекта (например, поискового индекса).

```
class IndexBuilder {
    std::vector<int> data;
public:
    void add(int x) { data.push_back(x); }

    // Возвращаем результат
    std::vector<int> finish() {
        return data; // Копирование! (или NRVO)
    }
};
```

Мы хотим, чтобы метод `finish()` перемещал данные, если `Builder` является временным объектом, и копировал, если долгоживущим.

```
class EfficientBuilder {
    std::vector<int> data;
public:
    void add(int x) { data.push_back(x); }

    // Версия для lvalue (долгоживущий объект): копируем
    std::vector<int> finish() & {
        return data;
    }

    // Версия для rvalue (временный объект): перемещаем
    std::vector<int> finish() && {
        return std::move(data);
    }
};

void usage() {
    EfficientBuilder b;
    b.add(1);

    // Вызывается finish() & -> копирование
    auto v1 = b.finish();

    // Вызывается finish() && -> перемещение
    // b становится пустым, но это безопасно, так как мы сделали move(b)
    auto v2 = std::move(b).finish();
```

```
27
28      // Строим и сразу забираем -> перемещение
29      auto v3 = EfficientBuilder().finish();
30  }
```

Это мощный механизм, но он имеет недостаток: дублирование кода. Если логика метода сложная, нам приходится писать две (или четыре, с учетом `const`) почти одинаковые функции.

C++23: Deducing This

Стандарт C++23 представил революционное изменение в синтаксисе методов классов, названное **Deducing This** (или *Explicit Object Parameter*).

Теперь мы можем явно объявить параметр, который будет представлять `this`, и сделать его шаблонным. Это позволяет "выводить" категорию значения самого объекта и использовать **Perfect Forwarding** для `*this`.

Синтаксис выглядит так: первым параметром метода идет `this Self&& self`.

Упрощение Ref-qualifiers

Перепишем пример с `EfficientBuilder` на C++23. Вместо двух функций нам понадобится одна.

```
1  #include <vector>
2  #include <utility> // для std::forward
3
4  class ModernBuilder {
5      std::vector<int> data;
6  public:
7      void add(int x) { data.push_back(x); }
8
9      // Единый шаблонный метод
10     // Self выведется как ModernBuilder& (для lvalue)
11     // или ModernBuilder (для rvalue)
12     template <typename Self>
13     auto finish(this Self&& self) {
14         // self — это наш *this, но теперь это обычный аргумент.
15         // Мы можем применить std::forward к самому объекту!
16
17         // forward_like - новая утилита C++23, аналогичная forward,
18         // но применяющая категорию self к полю data.
19         return std::forward<Self>(self).data;
20     }
21 };
```

На заметку

Как это работает:

1. Если мы зовем `b.finish()` (где `b` — lvalue), `Self` выводится как `ModernBuilder&`. `std::forward` возвращает ссылку, происходит копирование вектора.
2. Если мы зовем `std::move(b).finish()`, `Self` выводится как `ModernBuilder`. `std::forward` кастит `self` к rvalue. При доступе к `.data` мы получаем rvalue-доступ к полю, и срабатывает move-конструктор возвращаемого значения.

Преимущества Deducing This:

- **Дедупликация:** Одна реализация вместо 4-х (const/non-const, &/&&).
- **Рекурсивные лямбды:** Теперь лямбда может легко ссылаться на саму себя через явный параметр `this auto&& self`.
- **CRTP:** Паттерн Curiously Recurring Template Pattern становится проще, так как в базовом классе можно принимать `this Derived&& self` без приведения типов.

Резюме раздела

- `T&&` в шаблонах — это **Forwarding Reference**, а не rvalue-ссылка.
- `std::forward` обязателен при передаче универсальных ссылок дальше по стеку вызовов.
- Правила **Reference Collapsing** гарантируют, что lvalue превращает всю цепочку ссылок в lvalue.
- C++23 **Deducing This** позволяет писать обобщенный код, который автоматически адаптируется к `const` и `rvalue/lvalue` состоянию объекта.

Глава 16

Архитектура эксклюзивного владения: std::unique_ptr

Умные указатели — краеугольный камень современного C++. В этой главе мы не просто рассмотрим, "как пользоваться" `std::unique_ptr`, а разберем его устройство до байта. Мы реализуем собственный `unique_ptr`, столкнемся с проблемой пустого базового класса (EBO) и узнаем, как специализировать шаблоны для массивов.

Концепция эксклюзивного владения

`std::unique_ptr<T>` моделирует семантику единоличного владения ресурсом. Это означает:

1. В любой момент времени ресурсом владеет ровно один указатель.
2. Копирование указателя запрещено (компилятор выдаст ошибку).
3. Перемещение разрешено (владение переходит от A к B, A становится пустым).
4. При уничтожении указателя ресурс освобождается автоматически.

Это идеальная абстракция для ресурсов, которые не должны быть разделены: дескрипторы файлов, соединения с базой данных, динамические массивы внутри классов.

Базовый интерфейс

Минимальный интерфейс `unique_ptr` выглядит так:

```
template <typename T>
class UniquePtr {
    T* ptr_ = nullptr;

public:
    // Конструктор по умолчанию и от сырого указателя
    explicit UniquePtr(T* ptr = nullptr) : ptr_(ptr) {}

    // ЗАПРЕТ копирования (Rule of 5)
    UniquePtr(const UniquePtr&) = delete;
    UniquePtr& operator=(const UniquePtr&) = delete;
```

```
12
13      // Move-конструктор
14      UniquePtr(UniquePtr&& other) noexcept
15          : ptr_(std::exchange(other.ptr_, nullptr)) {}
16
17      // Move-оператор присваивания
18      UniquePtr& operator=(UniquePtr&& other) noexcept {
19          if (this != &other) {
20              delete ptr_; // Очищаем свой ресурс
21              ptr_ = std::exchange(other.ptr_, nullptr); // Забираем чужой
22          }
23          return *this;
24      }
25
26      // Деструктор
27      ~UniquePtr() {
28          delete ptr_;
29      }
30
31      // Доступ к данным
32      T& operator*() const { return *ptr_; }
33      T* operator->() const { return ptr_; }
34      T* get() const { return ptr_; }
35  };
```

Release vs Reset

Два важнейших метода управления владением:

- `reset(ptr)`: Уничтожает старый объект (вызывает `delete`) и захватывает новый `ptr`. Если вызван без аргументов, просто уничтожает объект.
- `release()`: Отказывается от владения, **не уничтожая** объект. Возвращает сырой указатель и зануляет внутреннее поле. Используется для передачи владения в C-API или легаси код.

```
1  void reset(T* ptr = nullptr) {
2      T* old_ptr = std::exchange(ptr_, ptr);
3      delete old_ptr; // Удаляем старый ресурс
4  }
5
6  T* release() {
7      return std::exchange(ptr_, nullptr); // Просто отдаем и забываем
8  }
```

Проблема размера (The Sizeof Problem)

В реальном `std::unique_ptr` есть второй шаблонный параметр — **Deleter**. Это функтор, который определяет, *как* удалять ресурс.

```cpp
template <typename T, typename Deleter = std::default_delete<T>>
class unique_ptr {
    T* ptr_;
    Deleter deleter_; // Храним экземпляр дилитера
public:
    ~unique_ptr() {
        if (ptr_) deleter_(ptr_); // Вызов функтора
    }
};
```

По умолчанию `Deleter` — это пустая структура:

```cpp
template <typename T>
struct default_delete {
    void operator()(T* ptr) const { delete ptr; }
};
```

Проблема:

1. `sizeof(T*)` = 8 байт (на 64-бит).
2. `sizeof(default_delete)` = 1 байт (пустая структура не может иметь размер 0, иначе у неё не будет адреса).
3. Из-за выравнивания (alignment) компилятор добавит padding.

Итог: `sizeof(unique_ptr)` становится $8 + 1 + \text{padding}(7) = 16$ байт. Это недопустимо! Умный указатель должен весить столько же, сколько сырой (Zero Overhead Principle).

Empty Base Optimization (EBO)

В C++ есть специальное правило: **базовый класс** может иметь размер 0 байт, если он пустой. Это называется *Empty Base Optimization*.

Чтобы воспользоваться этим, мы не можем хранить `Deleter` как поле. Мы должны от него **унаследоваться**. Но мы не можем просто унаследовать `unique_ptr` от `Deleter`, так как `Deleter` может быть:

- Финальным классом (final).
- Указателем на функцию (наследоваться нельзя).
- Ссылкой.

Решение: использование вспомогательной структуры `CompressedPair`.

Реализация CompressedPair

Идея: храним два значения. Если второе пустое — наследуемся от него.

```cpp
// Основной шаблон (если T2 не пустой)
template <typename T1, typename T2, bool = std::is_empty_v<T2>>
class CompressedPair {
```

```
    T1 first_;
    T2 second_;
public:
    T1& GetFirst() { return first_; }
    T2& GetSecond() { return second_; }
};

// Специализация для пустого T2 (включаем EBO)
template <typename T1, typename T2>
class CompressedPair<T1, T2, true> : private T2 { // Наследуемся!
    T1 first_;
public:
    T1& GetFirst() { return first_; }
    T2& GetSecond() { return *this; } // Кастим себя к базе
};
```

Теперь, используя `CompressedPair<T*, Deleter>` внутри `unique_ptr`, мы получаем размер 8 байт для stateless дилитеров.

Специализация для массивов

`std::unique_ptr` поддерживает массивы: `std::unique_ptr<int[]>`. Это критически важно, потому что для массивов нужно вызывать `delete[]` вместо `delete`.

Если бы специализации не было:

```
std::unique_ptr<int> p(new int[10]);
// Деструктор вызовет delete p (без скобок) -> UB!
```

Как реализована специализация `unique_ptr<T[]>`: 1. Дефолтный дилитер вызывает `delete[]`. 2. Оператор `*` и `->` удалены (логически к массиву нельзя применить `->`). 3. Добавлен оператор индексации `operator[]`.

```
template <typename T, typename Deleter>
class unique_ptr<T[], Deleter> {
public:
    // ...
    T& operator[](size_t i) const {
        return ptr_[i];
    }

    // operator* и operator-> отсутствуют
};
```

Угловой кейс: unique_ptr<void>

Можно ли создать `std::unique_ptr<void>`?

С дефолтным дилитером — **нет**.

```
std::unique_ptr<void> p(malloc(100)); // Ошибка компиляции или UB
```

Причина: `delete ptr`, где `ptr` имеет тип `void*`, является Undefined Behavior (если мы не знаем реальный тип объекта, мы не можем вызвать его деструктор). Стандартная библиотека часто ставит `static_assert(!is_void_v<T>)` в `default_delete`.

Однако, `unique_ptr<void>` полезен для управления памятью, выделенной через `malloc`, или для C-API хендлов. Для этого **обязательно** нужно предоставить кастомный дилитер.

```
struct FreeDeleter {
    void operator()(void* p) const {
        std::free(p); // free корректно работает с void*
    }
};

// Теперь это легально и безопасно
std::unique_ptr<void, FreeDeleter> memory(std::malloc(1024));
```

Это мощный паттерн для RAII-оберток над любыми ресурсами C-библиотек (FILE*, SDL_Surface*, SSL*).

Резюме раздела

- `unique_ptr` — это "золотой стандарт" владения. Нулевой оверхед.
- **EBO (Empty Base Optimization)** позволяет дилитеру не занимать память, если он не имеет состояния.
- Специализация для массивов `<T[]>` меняет `delete` на `delete[]` и добавляет `operator[]`.
- Для `void*` или системных ресурсов всегда используйте кастомный дилитер.

Глава 17

Внутреннее устройство разделяемого владения: std::shared_ptr

В отличие от `std::unique_ptr`, который обеспечивает эксклюзивное владение, `std::shared_ptr` реализует модель совместного (разделяемого) владения ресурсом. Несколько указателей могут ссылаться на один и тот же объект, и этот объект будет уничтожен только тогда, когда исчезнет *последний* указатель.

В этой главе мы погрузимся в архитектуру `shared_ptr`, разберем структуру контрольного блока, поймем, почему `sizeof(shared_ptr)` равен двум указателям, и научимся использовать "Aliasing Constructor" — одну из самых мощных, но малоизвестных возможностей C++.

Анатомия Shared Ptr: Два указателя

На первый взгляд может показаться, что `shared_ptr` просто хранит указатель на объект и счетчик ссылок. Но где хранится этот счетчик? Он не может быть внутри объекта (это требовало бы изменения класса объекта, что называется интрузивным подходом). Он не может быть статическим полем (тогда счетчик был бы общим для всех объектов типа T).

Решение: `shared_ptr` создает вспомогательный объект в куче, который называется **Control Block** (контрольный блок).

Сам объект `shared_ptr` на стеке состоит из двух полей:

1. `T* stored_ptr` — "сырой" указатель на управляемый объект. Используется для операторов разыменования (*, →).
2. `ControlBlock* cb_ptr` — указатель на контрольный блок.

Таким образом, на 64-битной архитектуре `sizeof(std::shared_ptr<T>) == 16 байт`.

Структура Control Block

Контрольный блок — это сердце механизма shared ownership. Он содержит:

- **Strong Ref Count** (счетчик сильных ссылок): количество живых `shared_ptr`.
- **Weak Ref Count** (счетчик слабых ссылок): количество живых `weak_ptr`.
- **Deleter**: функтор для уничтожения объекта (стирание типа через виртуальный вызов).

- **Allocator**: аллокатор, использованный для выделения памяти под контрольный блок.

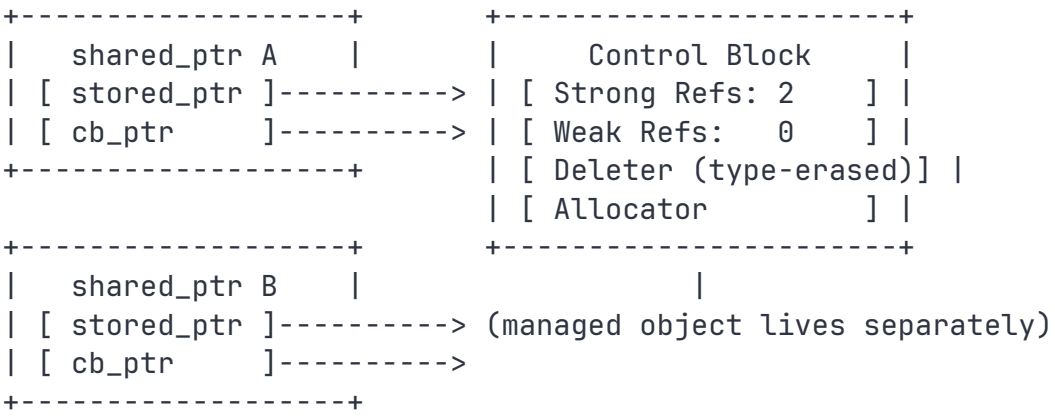


Рис. 17.1: Схема памяти обычного shared_ptr

std::make_shared vs std::shared_ptr(new T)

Существует два способа создать `shared_ptr`:

```
// 1. Через конструктор
std::shared_ptr<int> p1(new int(42));

// 2. Через make_shared
auto p2 = std::make_shared<int>(42);
```

Разница между ними фундаментальна с точки зрения производительности и размещения памяти.

Проблема двойной аллокации

При вызове `shared_ptr(new T)` происходят две независимые аллокации памяти: 1. `new int(42)` выделяет память под объект. 2. Конструктор `shared_ptr` выделяет память под **Control Block**.

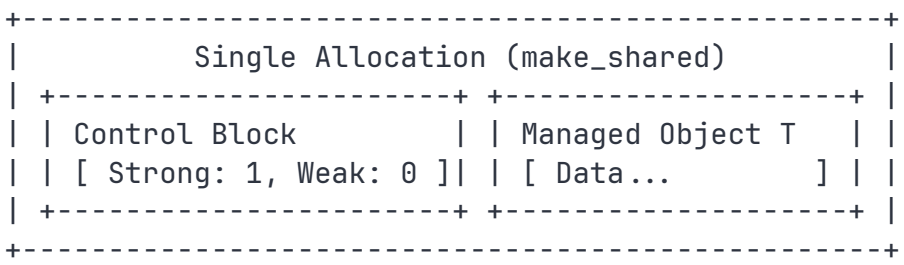
Это плохо по двум причинам:

- Лишняя нагрузка на аллокатор (медленно).
- Плохая локальность данных (объект и счетчик могут лежать далеко друг от друга, вызывая cache miss при доступе).
- **Небезопасность исключений:** Если мы пишем `f(shared_ptr<int>(new int(42)), g())`, и `g()` кидает исключение *после* `new int`, но *до* конструктора `shared_ptr`, мы получаем утечку памяти.

Оптимизация make_shared

`std::make_shared` совершает **одну большую аллокацию**, в которой размещает и Control Block, и сам объект T "паровозиком" (рядом друг с другом).

Это экономит память (меньше оверхеда на заголовки аллокатора) и процессорное время.



> **Важно!**
>
> У `make_shared` есть один неочевидный недостаток. Память под объект T не может быть освобождена, пока жив хотя бы один `weak_ptr`. Поскольку блок и объект — это единый кусок памяти, они живут и умирают вместе. Если у вас огромный объект и долгоживущие слабые ссылки, лучше использовать раздельные аллокации.

Aliasing Constructor (Конструктор псевдонимов)

Это "секретное оружие" `shared_ptr`. Сигнатура конструктора выглядит так:

```
template <typename Y>
shared_ptr(const shared_ptr<Y>& r, T* ptr) noexcept;
```

Что он делает:

- Он берет **Control Block** (владение) от указателя `r`. То есть он увеличивает счетчик ссылок того объекта, которым владеет `r`.
- Но в качестве `stored_ptr` (то, что возвращает `operator*`) он запоминает `ptr`.

Суть: Мы владеем одним объектом (и продлеваем ему жизнь), но указываем на другой (обычно — на его часть).

Пример: Указатель на поле структуры

Представьте, что у нас есть структура `Message`, и мы хотим передать в функцию только одну её строку `payload`, но так, чтобы вся структура `Message` не была удалена, пока мы работаем со строкой.

```
struct Message {
    Header header;
    std::string payload; // Хотим указывать сюда
};

void ProcessPayload(std::shared_ptr<std::string> str_ptr) {
    std::cout << *str_ptr << "\n";
}

int main() {
    // Создаем сообщение (владеем всем объектом)
    auto msg = std::make_shared<Message>();
    msg->payload = "Hello World";
```

```
14
15      // Создаем алиас-указатель
16      // 1. Владеем msg (увеличиваем refcount для Message)
17      // 2. Указываем на msg->payload
18      std::shared_ptr<std::string> payload_ptr(msg, &msg->payload);
19
20      // Теперь msg можно "забыть" в этой scope
21      msg.reset();
22
23      // Но объект Message НЕ будет удален!
24      // Потому что payload_ptr держит Control Block от Message.
25      ProcessPayload(payload_ptr);
26
27      // Message удалится только здесь, когда умрет payload_ptr
28  }
```

Если бы мы просто создали `shared_ptr<string>(&msg→payload)`, это привело бы к катастрофе: новый `shared_ptr` попытался бы сделать `delete` для поля внутри структуры, что является некорректным (freeing stack/middle memory). Aliasing constructor решает эту проблему, разделяя понятия "чем владеем" и "на что указываем".

Жизненный цикл Control Block

Механизм удаления в `shared_ptr` работает в два этапа, и это важно понимать при реализации.

1. **Уничтожение Объекта (Object Destruction):** Когда `Strong Ref Count` достигает 0:
   - Вызывается `Deleter(object_ptr)`.
   - Вызывается деструктор объекта T.
   - Память под объект освобождается (если это не `make_shared`).

   На этом этапе `weak_ptr` переходят в состояние *expired*.
2. **Уничтожение Контрольного блока (Block Deallocation):** Когда `Strong Ref Count` == 0 **И** `Weak Ref Count` == 0:
   - Удаляется сам Control Block.
   - Если использовался `make_shared`, только в этот момент освобождается вся память (и блока, и объекта).

Именно поэтому `weak_ptr` удерживает память контрольного блока (а в случае `make_shared` — и память мертвого объекта) от физического освобождения.

Реализация логики деструктора

В упрощенном виде логика декремента счетчика в деструкторе `shared_ptr` выглядит так:

```
1  ~shared_ptr() {
2      if (control_block) {
3          // Атомарный декремент
```

```
        if (control_block->strong_refs.fetch_sub(1) == 1) {
            // Мы были последним владельцем
            control_block->dispose_object(); // Уничтожить T

            // Проверяем слабые ссылки
            if (control_block->weak_refs.load() == 0) {
                control_block->destroy_self(); // Уничтожить блок
            }
        }
        // Важно: декремент слабых ссылок происходит в ~weak_ptr
        // и там тоже есть проверка на удаление блока.
    }
}
```

На самом деле, логика немного сложнее: сильный счетчик тоже неявно считается "одной слабой ссылкой", чтобы блок жил, пока жив хоть один владелец. Обычно используется схема: "Block живет, пока `Weak + (Strong > 0 ? 1 : 0) > 0`".

Резюме раздела

- `shared_ptr` состоит из двух указателей: на объект и на Control Block.
- `make_shared` эффективнее (1 аллокация), но может удерживать память дольше из-за слабых ссылок.
- **Aliasing Constructor** позволяет создавать `shared_ptr`, которые владеют объектом A, но указывают на объект B.
- Контрольный блок живет дольше управляемого объекта, если есть `weak_ptr`.

Глава 18

Безопасное управление жизненным циклом: WeakPtr и ESFT

В этой главе мы разберем две взаимосвязанные концепции, без которых экосистема `shared_ptr` была бы неполной и опасной: `std::weak_ptr` и идиому `enable_shared_from_this`. Мы увидим, как разрывать циклические ссылки, почему нельзя создавать умные указатели из `this` напрямую, и как C++ решает эту проблему через сложную машинерию SFINAE и наследования.

Проклятие циклических ссылок

Модель подсчета ссылок (*Reference Counting*) имеет фундаментальный недостаток: она не умеет обрабатывать циклы.

Представьте два объекта, которые ссылаются друг на друга через `shared_ptr`:

```
struct Node {
    std::shared_ptr<Node> neighbor;
    ~Node() { std::cout << "Deleted\n"; }
};

void create_cycle() {
    auto a = std::make_shared<Node>(); // A refs=1
    auto b = std::make_shared<Node>(); // B refs=1

    a->neighbor = b; // B refs=2
    b->neighbor = a; // A refs=2
}
// При выходе из функции:
// a уничтожается -> A refs=1
// b уничтожается -> B refs=1
// Итог: оба объекта живы, память утекла навсегда. Деструкторы не вызваны.
```

Чтобы разорвать цикл, одна из ссылок должна быть "слабой" — то есть не влиять на время жизни объекта, но позволять проверить, жив ли он.

std::weak_ptr: Наблюдатель

`std::weak_ptr` — это умный указатель, который ссылается на объект, управляемый `shared_ptr`, но не увеличивает *Strong Ref Count*. Вместо этого он увеличивает *Weak Ref Count* в контрольном блоке.

Свойства `weak_ptr`:

- Его нельзя разыменовать напрямую (операторов `*` и `->` нет). Это сделано специально: объект может исчезнуть в любой момент (в другом потоке), поэтому доступ к нему должен быть транзакционным.
- Метод `expired()` проверяет, удален ли объект.
- Метод `lock()` пытается создать `shared_ptr` из слабой ссылки. Если объект жив, мы получаем валидный `shared_ptr` (и счетчик сильных ссылок временно растет). Если мертв — получаем `nullptr` (пустой `shared_ptr`).

Исправление примера с циклом:

```
struct Node {
    // Используем weak_ptr для обратной ссылки
    std::weak_ptr<Node> neighbor;
};

void safe_cycle() {
    auto a = std::make_shared<Node>();
    auto b = std::make_shared<Node>();

    a->neighbor = b; // std::shared_ptr конвертируется в weak_ptr
    b->neighbor = a;
}
// При выходе:
// a уничтожается -> A refs=0 (удаляется!)
// Деструктор A уничтожает поле neighbor (слабую ссылку на B).
// b уничтожается -> B refs=0 (удаляется!).
// Память чиста.
```

Проблема "this" и enable_shared_from_this

Часто возникает ситуация, когда объект внутри своего метода хочет передать указатель на самого себя в функцию, принимающую `shared_ptr`.

```
struct Widget {
    void register_in_system(std::vector<std::shared_ptr<Widget>>& registry) {
        // ОШИБКА! Создается НОВЫЙ контрольный блок!
        registry.emplace_back(this);
    }
};

int main() {
    auto w = std::make_shared<Widget>(); // Control Block 1 (refs=1)

```

```
    std::vector<std::shared_ptr<Widget>> reg;
    w->register_in_system(reg); // Создает Control Block 2 (refs=1)

    // При выходе:
    // w уничтожается -> удаляет Widget (Block 1)
    // reg уничтожается -> удаляет ТОТ ЖЕ Widget (Block 2)
    // Double Free! Крэш программы.
}
```

Проблема в том, что сырой указатель `this` ничего не знает о том, что он уже управляется каким-то `shared_ptr` где-то снаружи. Создавая `shared_ptr(this)`, мы создаем *вторую независимую иерархию владения* для того же адреса памяти.

Решение: std::enable_shared_from_this

Стандартная библиотека предлагает паттерн (Mix-in класс) `std::enable_shared_from_this<T>`.

Механика работы "под капотом": 1. Класс `Widget` наследуется от `std::enable_shared_from_this<W` 2. В базовом классе хранится скрытое поле `std::weak_ptr<Widget> weak_this`. 3. Когда мы создаем `shared_ptr<Widget>`, конструктор `shared_ptr` проверяет (через SFINAE или концепты), является ли `Widget` наследником `ESFT`. 4. Если да, то `shared_ptr` инициализирует поле `weak_this` внутри объекта, записывая туда слабую ссылку на *тот самый* контрольный блок, который он только что создал.

Теперь объект может восстановить `shared_ptr` на себя, используя сохраненный `weak_this`.

```
// 1. Наследуемся (CRTP паттерн)
struct Widget : public std::enable_shared_from_this<Widget> {

    void register_in_system(std::vector<std::shared_ptr<Widget>>& registry) {
        // 2. Используем shared_from_this()
        registry.push_back(shared_from_this());
    }
};

int main() {
    auto w = std::make_shared<Widget>();
    // Внутри make_shared происходит магия инициализации weak_this

    std::vector<std::shared_ptr<Widget>> reg;
    w->register_in_system(reg); // Всё безопасно, счетчик увеличился до 2
}
```

Ловушки ESFT

Несмотря на удобство, `enable_shared_from_this` имеет строгие ограничения.

Ловушка 1: Вызов в конструкторе

> **Важно!**
>
> Нельзя вызывать `shared_from_this()` в конструкторе объекта!

Почему? В момент работы конструктора `shared_ptr`, который будет владеть объектом, **еще не завершил инициализацию**. Контрольный блок, возможно, уже есть, но инициализация поля `weak_this` происходит *после* завершения конструктора объекта (обычно в конструкторе `shared_ptr`).

Если вызвать `shared_from_this()` в конструкторе, поле `weak_this` еще пустое (expired), и будет выброшено исключение `std::bad_weak_ptr`.

Ловушка 2: Объект на стеке

Если вы создадите объект на стеке:

```
Widget w;
w.shared_from_this(); // Крэш (bad_weak_ptr)
```

У стекового объекта нет контрольного блока, `weak_this` не инициализирован. `ESFT` работает **только** если объект управляется через `shared_ptr`.

Реализация инициализации weak_this (Deep Dive)

Как именно `shared_ptr` узнает, что нужно инициализировать поле внутри T? Это делается через шаблонную магию. Упрощенно:

```
template<typename T>
class shared_ptr {
public:
    template<typename Y>
    shared_ptr(Y* ptr) {
        // ... создание control block ...

        // Магия: если Y наследуется от enable_shared_from_this,
        // вызвать приватный метод инициализации.
        if constexpr (std::is_base_of_v<std::enable_shared_from_this<Y>, Y>) {
            ptr->weak_this = *this; // присваиваем shared_ptr в weak_ptr
        }
    }
};
```

В реальности поле `weak_this` приватное, и `shared_ptr` объявлен другом для `enable_shared_from_t` чтобы иметь к нему доступ.

Резюме раздела

- `weak_ptr` — единственный способ разорвать циклические ссылки при использовании `shared_ptr`.
- Никогда не делайте `shared_ptr(this)`. Это приведет к Double Free.
- Используйте `std::enable_shared_from_this` для безопасного получения `shared_ptr` на себя.
- Помните, что `shared_from_this()` не работает в конструкторах и для объектов на стеке.

Глава 19

Стирание типов и Интрузивные указатели

В завершающей главе этого раздела мы рассмотрим продвинутые паттерны управления памятью и типами, которые выходят за рамки стандартных `unique_ptr` и `shared_ptr`. Мы реализуем собственный аналог `std::any` с использованием техники Type Erasure (стирание типов) и разберем концепцию интрузивных указателей — высокопроизводительной альтернативы `shared_ptr`, используемой в ядрах ОС, игровых движках и LLVM.

Паттерн Type Erasure: Реализация std::any

C++ — язык со строгой статической типизацией. Однако иногда нам нужен полиморфизм без наследования: возможность положить в контейнер объект *любого* типа (int, string, UserClass) и корректно управлять его временем жизни. Стандартный класс `std::any` делает именно это.

Как он работает внутри? Как он может хранить `T`, не будучи шаблонным классом `any<T>`? Ответ кроется в идиоме Type Erasure.

Архитектура Any

Суть идиомы:

1. Класс-обертка (`Any`) хранит указатель на *нешаблонный* абстрактный базовый класс (например, `Base`).
2. Внутри методов (обычно в конструкторе) мы создаем *шаблонного* наследника (`Derived<T>`), который знает конкретный тип `T`.
3. Указатель на наследника сохраняется в поле типа `Base*`.
4. Виртуальные функции в `Base` (в первую очередь деструктор) позволяют манипулировать объектом, не зная его реального типа в месте хранения.

Реализация

```
class Any {
private:
    // 1. Абстрактный интерфейс (Type Erased)
    struct Base {
```

```
        virtual ~Base() = default;
        virtual std::unique_ptr<Base> clone() const = 0;
    };

    // 2. Шаблонная реализация (Type Aware)
    template <typename T>
    struct Derived : Base {
        T value;

        Derived(const T& v) : value(v) {}
        Derived(T&& v) : value(std::move(v)) {}

        std::unique_ptr<Base> clone() const override {
            return std::make_unique<Derived<T>>(value);
        }
    };

    std::unique_ptr<Base> storage;

public:
    // Конструктор: принимает любой тип
    template <typename T>
    Any(T&& value) {
        // Выводим чистый тип (без const/volatile/reference)
        using CleanT = std::decay_t<T>;

        // Создаем конкретного наследника, но храним как Base
        storage = std::make_unique<Derived<CleanT>>(std::forward<T>(value));
    }

    // Copy Constructor (использует clone)
    Any(const Any& other) {
        if (other.storage) {
            storage = other.storage->clone();
        }
    }

    // Вспомогательный метод для any_cast (небезопасный для простоты)
    template <typename T>
    T* cast() {
        if (auto d = dynamic_cast<Derived<T>*>(storage.get())) {
            return &d->value;
        }
        return nullptr;
    }
};
```

На заметку

Обратите внимание: класс `Any` **не является** шаблонным. Вы можете объявить `std::vector<Any>`, и в нем будут лежать объекты разных типов. Магия происходит в конструкторе: именно там "стирается" тип `T`, превращаясь в полиморфный указатель `Base*`.

Деструктор `Any` автоматически вызовет деструктор `unique_ptr<Base>`, который вызовет

`virtual Base()`, который диспетчеризируется в `Derived<T>()`, корректно уничтожая поле `value` конкретного типа.

Интрузивные указатели (IntrusivePtr)

`std::shared_ptr` удобен, но имеет недостатки:

- **Размер:** 16 байт (2 указателя).
- **Аллокация:** Требует внешней памяти под Control Block (даже с `make_shared` это оверхед на структуру блока).
- **Оторванность:** Нельзя просто так взять сырой указатель на объект и превратить его в `shared_ptr` (нужен `enable_shared_from_this` или новый блок).

Интрузивные указатели решают эти проблемы, требуя от объекта *самому* хранить счетчик ссылок.

Концепция

Объект класса `T` должен содержать поле `int ref_count`. Обычно это делается через наследование от базового класса `RefCounted`.

Умный указатель `IntrusivePtr<T>` хранит **только** сырой указатель `T*` (размер 8 байт). При копировании он вызывает метод `obj→add_ref()`, при уничтожении — `obj→release()`.

Реализация RefCounted

```
class RefCounted {
    mutable std::atomic<int> ref_count_ = 0;

protected:
    // Деструктор виртуальный, чтобы корректно удалять наследников
    virtual ~RefCounted() = default;

public:
    void add_ref() const {
        ref_count_.fetch_add(1, std::memory_order_relaxed);
    }

    void release() const {
        if (ref_count_.fetch_sub(1, std::memory_order_acq_rel) == 1) {
            delete this; // Объект удаляет сам себя!
        }
    }
};
```

> **Важно!**
>
> Конструкция `delete this` абсолютно легальна в C++, если после неё не обращаться к полям объекта. Именно так работают COM-объекты в Windows и многие структуры ядра Linux.

Сравнение с shared_ptr

Характеристика	std::shared_ptr	IntrusivePtr
Размер	16 байт (2 ptr)	8 байт (1 ptr)
Аллокации	1 (make_shared) или 2	0 (дополнительных)
Локальность	Хорошая (make_shared)	Идеальная (счетчик внутри)
Восстановление	Опасно (без ESFT)	Безопасно (счетчик всегда с собой)
Weak Ptr	Есть	Нет (обычно)
Требования к T	Любой тип	Должен наследовать RefCounted

Пример использования

```
class Texture : public RefCounted {
    // ... данные текстуры ...
};

void use_texture(IntrusivePtr<Texture> tex) {
    // Копирование tex увеличивает счетчик внутри самого объекта Texture
}

int main() {
    // Обычный new, никаких make_shared
    IntrusivePtr<Texture> t(new Texture());

    // Можно восстановить умный указатель из сырого
    Texture* raw = t.get();
    IntrusivePtr<Texture> t2(raw); // OK! Счетчик инкрементируется.
    // Оба указателя смотрят на один объект, счетчик = 2.
}
```

Итоги раздела Smart Pointers

Мы прошли путь от базового `unique_ptr` до написания собственной системы типов и интрузивного управления памятью.

Резюме раздела

- **Type Erasure** позволяет создавать контейнеры для разнородных типов (`std::any`, `std::function`), пряча шаблонную логику за полиморфным фасадом.
- **IntrusivePtr** — выбор для высоконагруженных систем, где критичен каждый байт и такт процессора, и где вы контролируете иерархию классов.
- Понимание устройства Control Block и EBO делает вас не просто пользователем C++, а инженером, понимающим цену каждой абстракции.

Часть IV

Лекция 04 – Типы. Шаблоны

Глава 20

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 21

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.

```
template <class T> void foo(T t);      // Вариант 1
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 22

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`.

```cpp
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`.

```cpp
// 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 23

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.

```cpp
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`.

```cpp
#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 24

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.

Часть V

Лекция 05 – Ошибки. Исключения. Noexcept

Глава 25

Эволюция обработки ошибок: От C до C++17

Обработка ошибок — это механизм управления потоком выполнения при возникновении нештатных ситуаций. Любая операция, связанная с ресурсами (память, ввод-вывод), является потенциально сбойной.

Пример аллокации памяти:

- **Явный сбой:** Запрос объема памяти, превышающего физические возможности (32 ГБ на машине с 16 ГБ).
- **Скрытый сбой:** Фрагментация адресного пространства, когда суммарно свободной памяти достаточно, но нет непрерывного блока нужного размера.
- **Внешний сбой:** OOM-killer (Out of Memory killer) в Linux, который может завершить процесс при оверкоммите (overcommit).

До появления стандартизированных исключений в C++ (и в языке C) доминировали подходы, основанные на кодах возврата. Рассмотрим их эволюцию и архитектурные недостатки, которые привели к появлению механизма исключений.

Подход языка C: Коды возврата и errno

В языке C функции сообщают об ошибке через возвращаемое значение. Это создает семантическую неоднозначность: одно и то же значение (например, `int`) используется и как результат вычисления, и как индикатор статуса.

Глобальная переменная errno

Классический механизм UNIX — использование глобальной переменной `errno`. Функция возвращает специальное маркерное значение (например, `-1` или `NULL`), а код ошибки записывается в глобальную целочисленную переменную.

```
FILE* f = fopen("config.txt", "r");
if (f == NULL) {
    // Код ошибки в errno
    if (errno == ENOENT) {
        // Файл не найден
```

```
    } else if (errno == EACCES) {
        // Нет прав доступа
    }
}
```

Недостатки подхода: 1. **Потеря контекста.** `errno` — это просто число. Оно не сообщает, какой именно файл не удалось открыть или почему операция ввода-вывода была прервана. 2. **Игнорирование ошибок.** Программист обязан проверять результат *каждого* вызова. На практике это часто игнорируется (например, проверка результата `printf` или `close`). 3. **Проблемы многопоточности.** Изначально `errno` была глобальной, что делало невозможным её использование в multithreading. В современных стандартах это *thread-local* переменная, но архитектурная проблема разделения состояния остается.

Паттерн очистки ресурсов в ядре Linux

В отсутствие деструкторов (RAII) язык C требует ручной очистки ресурсов. Если функция захватывает несколько ресурсов (память, мьютексы, дескрипторы), обработка ошибок превращается в сложную задачу.

В ядре Linux стандартным паттерном является использование `goto` для обратной раскрутки инициализации. Это эмуляция деструкторов: метки располагаются в порядке, обратном захвату ресурсов.

```
int process_file(const char* path) {
    int err = 0;

    void* page = alloc_page();
    if (!page) return -ENOMEM;

    struct inode* node = get_inode(path);
    if (!node) {
        err = -ENOENT;
        goto out_free_page; // Прыжок к освобождению памяти
    }

    if (lock_inode(node) != 0) {
        err = -EIO;
        goto out_put_inode; // Прыжок к освобождению иноды
    }

    // Основная работа...

    unlock_inode(node); // Успешное завершение: освобождаем в прямом порядке
out_put_inode:
    put_inode(node);
out_free_page:
    free_page(page);

    return err;
}
```

Данный подход имитирует `switch-case` без `break`. Чем глубже произошла ошибка, тем «выше» по стеку очистки нужно прыгнуть. Это эффективно, но требует высокой дисциплины

и подвержено ошибкам при рефакторинге (copy-paste ошибок меток).

Подход Go и кортежи возврата (Tuple Returns)

Современные языки (например, Go) пытаются решить проблему неявности `errno`, возвращая пару значений: `(результат, ошибка)`.

```
// Пример на псевдокоде Go
file, err := os.Open("config.txt")
if err ≠ nil {
    return err
}
// Работа с file...
```

В C++ это можно эмулировать через возврат структуры или `std::pair`.

> **Product Type vs Sum Type**
>
> С точки зрения теории типов, возврат пары `(Value, Error)` — это **Тип-Произведение** (Product Type). Множество возможных состояний равно декартову произведению $S = V \times E$.
>
> Это создает семантически некорректные состояния:
>
> 1. `Value` валиден, `Error` валиден (Противоречие: успех и ошибка одновременно).
> 2. `Value` невалиден, `Error` невалиден (Отсутствие результата и отсутствие ошибки).
>
> Корректным подходом является **Тип-Сумма** (Sum Type), реализуемый через `std::variant<Valu Error>` или `std::expected` (C++23), где состояние может быть *либо* результатом, *либо* ошибкой, но не обоими сразу.

Недостатки подхода Go/Tuple в C++: 1. **Раздувание кода.** Проверка `if (err)` занимает 3-4 строки на каждый вызов. 2. **Накладные расходы.** Постоянное конструирование и копирование объектов ошибок, даже если они не нужны (happy path).

Современные оптимизации: std::from_chars

В C++17 был введен заголовок `<charconv>` и функции `std::from_chars`, которые используют подход возврата структуры, напоминающий C-style, но по причинам производительности.

```
#include <charconv>

struct from_chars_result {
    const char* ptr; // Указатель на первый нераспарсенный символ
    std::errc ec;    // Код ошибки (0, если успех)
};

// ...
auto [ptr, ec] = std::from_chars(begin, end, value);
```

```
if (ec != std::errc()) {
    // Обработка ошибки
}
```

На заметку

Почему не исключения? Парсинг чисел — операция, которая часто может завершаться неудачей (например, валидация пользовательского ввода). Выброс исключения — это тяжелая операция (раскрутка стека, RTTI), которая может быть в 100-1000 раз медленнее простой проверки кода возврата. Для высоконагруженных низкоуровневых операций (парсинг JSON, логов) возврат структуры оправдан.

Фундаментальная проблема конструкторов

Ключевая причина введения исключений в C++ — это архитектура объектно-ориентированного программирования, в частности, конструкторов.

Конструктор не имеет возвращаемого значения. Он возвращает сам созданный объект. Если внутри конструктора происходит ошибка (например, `new` не смог выделить память под буфер вектора), у него нет штатного способа сообщить об этом вызывающему коду через `return`.

Антипаттерн: Двухфазная инициализация

До исключений использовался подход с разделением создания и инициализации:

```
class Vector {
    int* data = nullptr;
public:
    Vector() {} // 1. Дешевое создание "пустого" объекта

    // 2. Метод, который может вернуть ошибку
    bool init(size_t size) {
        data = (int*)malloc(size * sizeof(int));
        return data != nullptr;
    }
};

// Клиентский код
Vector v;
if (!v.init(100)) {
    // Обработка ошибки
}
```

> **Важно!**
>
> Этот подход порождает "**Zombie Objects**" — объекты, которые существуют, но находятся в невалидном состоянии.
>
> - Программист может забыть вызвать `init()`.
> - Объект может быть передан в функцию, которая ожидает валидное состояние.
> - Требуется постоянная проверка флагов `is_initialized` внутри каждого метода.

RAII и Исключения

C++ решает эту проблему через идиому **RAII** (Resource Acquisition Is Initialization). Инвариант класса должен быть установлен в конструкторе. Если это невозможно (ошибка), конструктор должен прервать выполнение через выброс исключения.

```
class Vector {
public:
    Vector(size_t size) {
        data = new int[size]; // Если new упадет, полетит std::bad_alloc
        // Если мы здесь, объект ГАРАНТИРОВАННО валиден
    }
};
```

Это гарантирует, что в программе не существует "полуживых" объектов. Либо объект создан корректно, либо его создание было прервано исключением и память очищена (при условии корректного Stack Unwinding, который мы рассмотрим далее).

Резюме раздела

1. Коды возврата (C-style) приводят к игнорированию ошибок и смешиванию бизнес-логики с обработкой сбоев.
2. Подход "Результат + Ошибка" (Go) теоретически некорректен (Product Type) и многословен.
3. Исключения в C++ необходимы для корректной работы конструкторов и предотвращения появления Zombie Objects.
4. Исключения имеют накладные расходы, поэтому в узких местах (парсинг) применяются специальные API без исключений (`std::from_chars`).

Глава 26

Механика исключений: Stack Unwinding и Гарантии

Механизм исключений в C++ — это не просто альтернатива кодам возврата, а сложная инфраструктура времени выполнения (Runtime), тесно связанная с управлением памятью и временем жизни объектов. Понимание процесса «раскрутки стека» (Stack Unwinding) критически важно для написания корректного C++ кода.

Анатомия раскрутки стека (Stack Unwinding)

Когда оператор `throw` выбрасывает исключение, программа перестает выполняться линейно. Среда выполнения (C++ Runtime) начинает процесс поиска подходящего обработчика (`catch`), поднимаясь вверх по стеку вызовов. Этот процесс называется **Stack Unwinding**.

Ключевая особенность этого процесса: по мере выхода из каждой функции (scope), Runtime обязан корректно уничтожить все локальные объекты, созданные на стеке в этом блоке. Для каждого такого объекта вызывается деструктор.

> **RAII (Resource Acquisition Is Initialization)**
>
> Именно гарантия вызова деструкторов при раскрутке стека делает возможной идиому RAII. Если ресурс (память, файл, мьютекс) обернут в объект с деструктором, исключение не приведет к утечке. Если же используется «сырой» `new` без `delete` (который мог бы стоять далее по коду, но выполнение перепрыгнуло его), произойдет утечка памяти.

Рассмотрим класс `Noisy`, который сообщает о своем рождении и смерти:

```
#include <iostream>

struct Noisy {
    int id;
    Noisy(int i) : id(i) {
        std::cout << "Ctor " << id << "\n";
    }
    ~Noisy() {
        std::cout << "Dtor " << id << "\n";
    }
};
```

```
12
13 void func() {
14     Noisy n1(1);
15     Noisy n2(2);
16     throw std::runtime_error("Error");
17     // Сюда управление никогда не дойдет
18     // Деструкторы n2 и n1 будут вызваны автоматически
19 }
```

При вызове `func()` вывод будет следующим:

```
Ctor 1
Ctor 2
Dtor 2  <-- Обратный порядок уничтожения
Dtor 1
```

Runtime гарантирует, что объекты уничтожаются строго в порядке, обратном их созданию (LIFO — Last In, First Out). Это важно, так как объект `n2` может зависеть от `n1`.

Деструкторы и спецификатор noexcept

До стандарта C++11 деструкторы могли выбрасывать исключения так же, как и любые другие функции. Однако практика показала, что это приводит к фатальным ошибкам в архитектуре приложений.

В современном C++ действует правило:

> **Важно!**
>
> Начиная с C++11, все деструкторы по умолчанию неявно помечены как `noexcept(true)`.

Это означает, что если вы попытаетесь выбросить исключение из деструктора и оно вылетит за его пределы, программа немедленно завершится вызовом `std::terminate()`.

Почему принято такое жесткое решение? Ответ кроется в механике взаимодействия двух исключений.

Катастрофа Double Fault: std::terminate

Представьте ситуацию: 1. В блоке `try` происходит ошибка, выбрасывается исключение `E1`. 2. Начинается раскрутка стека (Stack Unwinding). 3. Runtime находит на стеке локальный объект и вызывает его деструктор. 4. Внутри деструктора происходит сбой, и выбрасывается новое исключение `E2`, которое вылетает наружу.

В этот момент в одном потоке существуют два активных (unhandled) исключения: `E1` (которое еще не поймано) и `E2` (которое только что возникло). C++ Runtime не умеет обрабатывать две ошибки одновременно — непонятно, какую из них доставлять в `catch`, и как продолжать раскрутку.

Результат: Процесс немедленно убивается через `std::terminate()`. Никакие `catch` блоки не срабатывают, деструкторы остальных объектов не вызываются.

Пример кода «смертника»:

```cpp
struct Bomb {
    ~Bomb() {
        // Попытка выбросить исключение из деструктора
        throw std::runtime_error("Boom in dtor");
    }
};

int main() {
    try {
        Bomb b;
        // 1. Бросаем первичное исключение
        throw std::logic_error("Primary error");

        // 2. Начинается unwinding. Вызывается ~Bomb().
        // 3. Из ~Bomb вылетает "Boom".
        // 4. ДВА исключения -> std::terminate().
    } catch (...) {
        std::cout << "Never printed\n";
    }
}
```

Если убрать `throw "Primary error"`, программа все равно упадет (из-за `noexcept` по умолчанию на деструкторе), но уже по причине нарушения спецификации `noexcept`, а не из-за Double Fault. Если же явно пометить деструктор `noexcept(false)`, то одиночное исключение сработает, но двойное все равно убьет процесс.

Проблема очистки ресурсов (на примере std::ofstream)

Классический пример конфликта RAII и обработки ошибок — закрытие файла.

```cpp
class FileWriter {
    std::ofstream file;
public:
    ~FileWriter() {
        file.close(); // Может вернуть ошибку или кинуть исключение (если
        ↪ включено)
    }
};
```

Операция `close()` включает сброс буферов на диск (flush). Если диск переполнен, `close()` завершится с ошибкой.

- Если деструктор игнорирует ошибку (глотает исключение), пользователь теряет данные, не узнав об этом.
- Если деструктор выбрасывает исключение, программа рискует упасть с `std::terminate`, если `FileWriter` уничтожался в процессе обработки другой ошибки.

Решение: Предоставить явный метод `close()`, а в деструкторе оставить «аварийную» логику.

```
void manual_close() {
    file.close();
    if (file.fail()) throw std::ios_base::failure("Write failed");
}

~FileWriter() {
    try {
        if (file.is_open()) file.close();
    } catch (...) {
        // Логируем, но НЕ бросаем дальше
        std::cerr << "Error closing file in dtor\n";
    }
}
```

Это компромисс: ответственный пользователь вызывает `manual_close()` и обрабатывает ошибки. Забывчивый пользователь полагается на деструктор, который гарантирует отсутствие утечек дескрипторов, но может "проглотить" ошибку записи.

Проверка активных исключений: std::uncaught_exceptions

Иногда действительно необходимо выполнить опасную операцию в деструкторе, но только если это безопасно (т.е. если мы не находимся в процессе раскрутки стека).

Для этого существует функция `std::uncaught_exceptions()` (обратите внимание на множественное число, C++17). Она возвращает количество активных исключений в текущем потоке.

> **На заметку**
>
> В C++98 существовала функция `std::uncaught_exception()` (единственное число), возвращавшая `bool`. Она была признана архитектурно ошибочной и удалена в C++20, так как не позволяла корректно работать во вложенных сценариях (например, когда одно исключение уже обрабатывается, и внутри `catch` бросается новое).

Паттерн безопасного выброса из деструктора (используется крайне редко, например, в транзакционной памяти или сложных базах данных):

```
class Transaction {
public:
    ~Transaction() noexcept(false) { // Разрешаем исключения
        if (std::uncaught_exceptions() == 0) {
            // Стек спокоен, можно кидать исключения.
            // Например, коммит транзакции, который может упасть.
            commit_or_throw();
        } else {
            // Мы уже летим из-за ошибки.
            // Кидать новое нельзя -> terminate.
            // Поэтому делаем только безопасный откат.
            rollback_silent();
        }
    }
```

```
15 };
```

Резюме раздела

1. **Stack Unwinding** — механизм автоматического вызова деструкторов при возникновении исключения. Это основа RAII.
2. **Деструкторы по умолчанию noexcept**. Выброс исключения из деструктора при наличии другого активного исключения приводит к `std::terminate`.
3. Не стройте логику программы на исключениях в деструкторах. Деструктор должен заниматься освобождением ресурсов, которое (в идеале) не должно фейлиться.
4. Если необходимо сообщить об ошибке при закрытии ресурса, используйте отдельный метод (например, `close()`) и вызывайте его явно перед разрушением объекта.

Глава 27

Продвинутая работа с исключениями: Транспортировка и Slicing

Работа с исключениями в C++ не ограничивается простыми блоками `try-catch`. При построении сложных архитектур (например, асинхронных очередей задач или плагинных систем) возникает необходимость сохранять состояние ошибки, передавать его между потоками и корректно обрабатывать полиморфные иерархии исключений.

Физическое расположение исключений

Когда выполняется инструкция `throw`, среда выполнения (Runtime) должна аллоцировать память под объект исключения. Возникает вопрос: где именно живет этот объект?

1. **Не на стеке.** Стек раскручивается (unwinding) в процессе поиска обработчика, поэтому объект исключения не может быть локальной переменной.
2. **Не в статической памяти.** Исключения могут быть рекурсивными или возникать в нескольких потоках одновременно.

Фактически, компиляторы (в соответствии с ABI, например, Itanium C++ ABI) выделяют память в отдельной области кучи (heap). Это накладывает специфические требования:

- Операция `throw` может потребовать динамической аллокации.
- Если памяти нет (бросается `std::bad_alloc`), Runtime использует заранее зарезервированный "аварийный буфер", чтобы гарантировать возможность сообщить об ошибке нехватки памяти.

Время жизни этого объекта управляется Runtime. Он существует до тех пор, пока последний обработчик `catch` не завершит свою работу с ним.

Проблема срезки (Object Slicing)

Одной из самых коварных ошибок в C++ является перехват исключений по значению, а не по ссылке. Это приводит к потере полиморфизма и данных, известной как **Object Slicing** (срезка объекта).

Рассмотрим иерархию ошибок:

```cpp
struct ErrorBase {
    virtual const char* what() const { return "Base Error"; }
    virtual ~ErrorBase() = default;
};

struct DatabaseError : ErrorBase {
    int errorCode;
    DatabaseError(int code) : errorCode(code) {}

    const char* what() const override { return "DB Error"; }
};
```

Если мы выбросим `DatabaseError`, но поймаем его как `ErrorBase` (по значению), произойдет следующее:

```cpp
void unsafe_handler() {
    try {
        throw DatabaseError(505);
    } catch (ErrorBase e) { // ОШИБКА: Перехват по значению!
        // 1. Создается новый объект типа ErrorBase.
        // 2. Вызывается копирующий конструктор ErrorBase(const ErrorBase&).
        // 3. Поля DatabaseError (errorCode) отбрасываются.
        // 4. vptr теперь указывает на таблицу виртуальных функций Base.

        std::cout << e.what(); // Выведет "Base Error", а не "DB Error"
    }
}
```

Механика Slicing

При копировании объекта-наследника в переменную базового типа происходит физическое копирование только той части памяти, которая относится к базовому классу. Дополнительные поля наследника игнорируются. Указатель на таблицу виртуальных функций (`vptr`) инициализируется значением для базового класса, полностью стирая полиморфное поведение.

Важно!

Всегда перехватывайте исключения по константной ссылке: `catch (const ErrorBase& e)`. Это предотвращает копирование и сохраняет полиморфизм (вызов `e.what()` вернет "DB Error").

Механика повторного выброса (Rethrow)

Частый сценарий: перехватить ошибку, залогировать её и пробросить дальше для обработки на уровне выше. Здесь существует критическая разница между `throw e;` и `throw;`.

Ошибочный проброс (throw e)

```
try {
    // ... код ...
} catch (const ErrorBase& e) {
    log_error(e);
    throw e; // ОШИБКА: Повторная срезка!
}
```

Даже если `e` — это ссылка на `DatabaseError`, инструкция `throw e` создает *новый* объект исключения. Тип этого нового объекта определяется статическим типом переменной `e` (то есть `ErrorBase`). Мы снова теряем информацию о том, что это была ошибка базы данных.

Корректный проброс (throw)

```
try {
    // ... код ...
} catch (const ErrorBase& e) {
    log_error(e);
    throw; // КОРРЕКТНО: Проброс текущего активного исключения
}
```

Инструкция `throw;` (без аргументов) сообщает Runtime: «Возьми тот самый объект исключения, который сейчас обрабатывается (со всеми его полиморфными свойствами), и запусти процесс раскрутки стека дальше». Копирования не происходит, тип сохраняется.

Транспортировка исключений между потоками

Исключения привязаны к стеку текущего потока (Thread Local). Вы не можете выбросить исключение в одном потоке и автоматически поймать его в другом (например, в `main`).

Для решения этой задачи в C++11 был введен механизм захвата и транспортировки исключений: `std::exception_ptr`.

Это «умный указатель» (аналог `std::shared_ptr`) для объектов исключений. Он копирует (или продлевает жизнь) объекту исключения, позволяя сохранить его в переменную и передать в другой контекст.

Основные примитивы

1. `std::current_exception()` — вызывается внутри блока `catch`. Возвращает `std::exception_p` указывающий на текущую ошибку. Если исключений нет, возвращает `nullptr`.
2. `std::rethrow_exception(ptr)` — принимает указатель и выбрасывает исключение заново.

Реализация паттерна Worker-Result

Рассмотрим, как это работает на примере самодельного аналога `std::future`:

```cpp
#include <exception>
#include <thread>
#include <iostream>

std::exception_ptr globalException = nullptr;

void worker() {
    try {
        // Имитация работы
        throw std::runtime_error("Failure in worker thread");
    } catch (...) {
        // 1. Не знаем тип исключения, но можем его захватить
        globalException = std::current_exception();
        // Теперь объект исключения живет в куче и удерживается указателем
    }
}

int main() {
    std::thread t(worker);
    t.join();

    if (globalException) {
        try {
            // 2. Пробрасываем захваченное исключение в текущем потоке
            std::rethrow_exception(globalException);
        } catch (const std::exception& e) {
            std::cout << "Caught from worker: " << e.what() << "\n";
        }
    }
    return 0;
}
```

На заметку

`std::exception_ptr` корректно работает с любыми типами исключений, даже если они не наследники `std::exception` (например, `throw 42;`). Однако, чтобы обработать их после `rethrow_exception`, вам все равно понадобится соответствующий `catch`.

Резюме раздела

- Объекты исключений живут в специальной области памяти (не стек), их время жизни управляется Runtime.
- Перехват по значению (`catch (Base e)`) разрушает полиморфизм (Object Slicing). Всегда используйте ссылки.
- Для проброса исключения используйте `throw;` (без аргументов), чтобы избежать повторной срезки.
- Для передачи ошибок между потоками используйте пару `std::current_exception` и `std::rethrow_exception`.

Часть VI

Лекция 06 – Паттерны. ODR

Глава 28

Физическая структура программы: Компиляция, Линковка и ODR

Понимание того, как исходный код C++ превращается в исполняемый файл, является критическим навыком для системного программиста. В отличие от интерпретируемых языков или языков с JIT-компиляцией, C++ имеет строгую, многоступенчатую модель сборки, унаследованную от C. Ошибки на этапе линковки (Unresolved external symbol, Multiple definition) часто становятся следствием непонимания концепции единиц трансляции и правила одного определения (ODR).

Пайплайн сборки C++

Процесс преобразования кода можно разделить на три изолированных этапа: препроцессинг, компиляция и линковка.

1. Препроцессинг

Препроцессор — это инструмент текстовой подстановки. Он ничего не знает о синтаксисе C++, типах данных или областях видимости.

- `#include "file.h"`: Содержимое файла `file.h` копируется байт-в-байт в место вызова директивы. Рекурсивные включения раскрываются полностью.
- `#define`, `#ifdef`: Обрабатываются макросы и условная компиляция.
- Комментарии удаляются.

Результатом работы препроцессора является *Translation Unit* (TU) — полный текст программы, готовый к компиляции. Если в исходном файле было 10 строк, а он включил заголовок на 10 000 строк, компилятор получит на вход 10 010 строк кода.

2. Компиляция

Компилятор работает с каждой единицей трансляции **изолированно**. При компиляции `a.cpp` компилятор не знает о существовании `b.cpp`.

1. **Лексический и синтаксический анализ:** Построение абстрактного синтаксического дерева (AST).
2. **Семантический анализ:** Проверка типов, перегрузок, шаблонов.

3. **Оптимизация:** Преобразование AST в промежуточное представление (IR) и применение оптимизаций (inlining, loop unrolling).
4. **Кодогенерация:** Создание объектного файла (`.o` или `.obj`).

Объектный файл содержит машинный код функций и данных, а также **таблицу символов** (Symbol Table). Символы делятся на:

- **Экспортируемые (Defined):** Функции и глобальные переменные, определенные в этом TU.
- **Импортируемые (Undefined):** Функции, которые были объявлены (declared), но не определены (defined) в текущем TU.

3. Линковка (Компоновка)

Линкер (Linker) собирает объектные файлы в единый исполняемый файл или библиотеку. Его задача — разрешить все Undefined символы, сопоставив их с Defined символами из других объектных файлов. Именно здесь проверяется правило ODR.

One Definition Rule (ODR)

Фундаментальное правило C++ (стандарт ISO/IEC 14882) гласит:

> **One Definition Rule (ODR)**
>
> 1. В любой единице трансляции может быть не более одного определения любой переменной, функции, класса или шаблона.
> 2. Во всей программе должно быть **ровно одно** определение каждой используемой переменной или функции (non-inline).
> 3. Для классов и шаблонов разрешено иметь определения в нескольких TU, если они **побайтово идентичны** (token-for-token identical).

Нарушение пункта 2 приводит к ошибке линковки `multiple definition of symbol`. Нарушение пункта 3 (разные определения одного класса в разных файлах) приводит к неопределенному поведению (UB), которое крайне сложно диагностировать.

Declaration vs Definition

Важно различать объявление и определение:

- **Declaration (Объявление):** Сообщает компилятору о существовании сущности и её типе. Не выделяет память.

```
extern int x;        // Объявление переменной
void func();         // Объявление функции
class A;             // Forward declaration класса
```

- **Definition (Определение):** Создает сущность, выделяет память или генерирует код.

```cpp
int x = 0;          // Определение (даже без инициализатора int x;)
void func() { }     // Определение (есть тело)
class A { int f; }; // Определение класса
```

Проблема глобальных переменных в хедерах

Рассмотрим классическую ошибку нарушения ODR.

header.h

```cpp
#ifndef HEADER_H
#define HEADER_H
// ОШИБКА: Это определение глобальной переменной
int globalVar = 42;
#endif
```

a.cpp

```cpp
#include "header.h"
// После препроцессинга здесь появится: int globalVar = 42;
// Компилятор создаст a.o с символом globalVar (Defined)
```

b.cpp

```cpp
#include "header.h"
// После препроцессинга здесь тоже появится: int globalVar = 42;
// Компилятор создаст b.o с символом globalVar (Defined)
```

При попытке слинковать `a.o` и `b.o` линкер обнаружит два сильных символа с именем `globalVar` и выдаст ошибку `duplicate symbol`.

Стратегии решения конфликтов линковки

Существует три основных механизма для корректного разделения кода между единицами трансляции.

1. extern (External Linkage)

Традиционный подход C. В хедере мы только *объявляем* переменную, а *определяем* её ровно в одном .cpp файле.

```cpp
// header.h
extern int globalVar; // Только объявление

// a.cpp
#include "header.h"
int globalVar = 42;   // Определение (память выделена здесь)
```

```

// b.cpp
#include "header.h"
// Использует символ globalVar, который будет найден в a.o
```

2. static (Internal Linkage)

Ключевое слово `static`, примененное к глобальной переменной или свободной функции, меняет тип линковки на *внутренний*.

```
// header.h
static int globalVar = 42;
```

Если включить такой хедер в `a.cpp` и `b.cpp`:

- В `a.o` будет создана своя локальная копия `globalVar`.
- В `b.o` будет создана **другая**, независимая копия `globalVar`.

Линкер не увидит конфликта, так как символы не экспортируются наружу. Это решает ошибку сборки, но создает логическую ошибку, если вы ожидали, что переменная будет общей для всей программы (Singleton). Кроме того, это увеличивает размер бинарного файла.

3. inline (Weak Linkage / COMDAT Folding)

В C++17 появилась возможность объявлять переменные как `inline`. Это позволяет определять переменную в хедере.

```
// header.h
inline int globalVar = 42; // C++17
```

Семантика `inline`: "Разрешено множественное определение". Компилятор помечает символ как слабый (weak). Линкер, встречая несколько копий такого символа, просто выбирает одну (обычно первую попавшуюся) и отбрасывает остальные. Все единицы трансляции будут ссылаться на один и тот же адрес памяти. Это идеальное решение для констант и глобальных настроек в современных стандартах.

> **Важно!**
>
> Шаблоны функций и классов неявно являются `inline`. Методы, определенные внутри тела класса, также неявно являются `inline`. Именно поэтому мы можем писать реализацию шаблонных классов полностью в .h файлах.

Anonymous Namespaces

В C++ вместо `static` для ограничения видимости внутри файла рекомендуется использовать анонимные пространства имен.

```
namespace {
    int localHelper = 10;
    void doSomethingInternal() { ... }
}
```

С точки зрения линковки это эквивалентно генерации уникального имени для пространства имен (например, `namespace _unique_id_123`), доступного только в текущем TU. Это предпочтительнее `static`, так как работает и для типов (классов), а не только для переменных и функций, позволяя передавать локальные типы в шаблоны.

Библиотеки и LTO

Static (.a / .lib) vs Shared (.so / .dll)

- **Static Library:** Это просто архив объектных файлов (аналог `zip` или `tar` с индексом). При линковке нужные `.o` файлы извлекаются из архива и копируются в исполняемый файл. Размер бинарника растет, но он становится самодостаточным.
- **Shared Library:** Не копируется в бинарник. Линкер лишь проверяет наличие символов и оставляет "заглушки". При запуске программы операционная система (Dynamic Loader) загружает библиотеку в память.
  - **Плюс:** Экономия оперативной памяти. Код библиотеки загружается в физическую память один раз и отображается в виртуальную память всех процессов, использующих её (механизм memory mapping).
  - **Минус:** Проблема зависимостей (DLL Hell), необходимость распространять библиотеки вместе с приложением.

Link Time Optimization (LTO)

Традиционная модель компиляции имеет недостаток: компилятор видит только один TU. Он не может заинлайнить функцию, определенную в другом `.cpp` файле, так как на этапе компиляции её тело недоступно.

LTO меняет этот процесс. При включении LTO (флаг `-flto` в GCC/Clang) компилятор вместо машинного кода генерирует в объектных файлах промежуточное представление (Intermediate Representation, IR — например, GIMPLE или LLVM Bitcode).

Реальная генерация кода откладывается до этапа линковки. Линкер собирает IR со всех модулей, видит всю программу целиком и может выполнять агрессивные оптимизации:

- Cross-module inlining (инлайнинг функций между разными .cpp).
- Dead code elimination (удаление неиспользуемых функций, которые ранее считались экспортируемыми).

Цена LTO — значительное увеличение времени линковки и потребления памяти в процессе сборки.

Практические рекомендации

1. Используйте `#pragma once` для защиты хедеров. Это нестандартная директива, но она поддерживается всеми основными компиляторами и работает быстрее классических Include Guards (`#ifndef ...`), так как компилятору не нужно перечитывать файл.
2. Никогда не определяйте переменные в хедерах без `inline` или `constexpr`.
3. Шаблоны должны быть полностью определены в хедерах, так как компилятору нужно видеть их тело для инстанцирования конкретных типов.
4. Используйте анонимные пространства имен вместо `static` в `.cpp` файлах для сокрытия локальных помощников.

Глава 29

Управление зависимостями и идиома PImpl

Одной из фундаментальных проблем разработки крупных проектов на C++ является время компиляции. Из-за механизма включения заголовочных файлов (`#include`), изменение одной приватной переменной в низкоуровневом классе может вызвать каскадную перекомпиляцию сотен единиц трансляции, зависящих от этого заголовка.

Идиома PImpl (Pointer to Implementation), также известная как "Compilation Firewall" или "Opaque Pointer", — это архитектурный паттерн, призванный разорвать эти транзитивные зависимости и скрыть детали реализации от клиентского кода.

Проблема транзитивных зависимостей

В C++ определение класса в заголовочном файле должно быть полным. Компилятору необходимо знать точный размер объекта (значение `sizeof`), чтобы выделить память на стеке или внедрить объект в другой класс. Это означает, что все типы, используемые в качестве полей класса (даже приватных), должны быть полностью определены.

Рассмотрим пример класса `DatabaseConnection`, который использует тяжелую стороннюю библиотеку (например, драйвер PostgreSQL).

DatabaseConnection.h (Без PImpl)

```
#pragma once
// Мы вынуждены включить тяжелый хедер сторонней библиотеки,
// так как используем её типы в полях класса.
#include <libpq-fe.h>
#include <vector>
#include <string>

class DatabaseConnection {
public:
    void connect(const std::string& connectionString);
    void executeQuery(const std::string& query);

private:
    // Детали реализации "протекают" в интерфейс.
    // Любой файл, включающий DatabaseConnection.h,
```

```
    // неявно включает <libpq-fe.h> и все его зависимости.
    PGconn* pgConnection;
    std::vector<std::string> cache;
    int internalState;
};
```

Недостатки такого подхода:

1. **Header Bloat (Раздувание заголовков):** Клиентский код, которому нужен только метод `connect`, вынужден парсить тысячи строк кода из `<libpq-fe.h>`.
2. **Хрупкость сборки:** Если вы добавите новое приватное поле `int retryCount`, изменится `sizeof(DatabaseConnection)`. Все исходные файлы, использующие этот класс, обязаны быть перекомпилированы, иначе нарушится ODR (разные размеры объекта в разных TU).
3. **Загрязнение пространства имен:** Макросы и типы из сторонней библиотеки могут конфликтовать с кодом пользователя.

Реализация PImpl

Суть идиомы заключается в вынесении всех приватных полей в отдельную структуру (класс реализации), которая определяется только внутри `.cpp` файла. В публичном заголовке остается только указатель на эту структуру.

Поскольку размер указателя фиксирован и известен компилятору (обычно 8 байт на 64-битных системах), определение класса реализации в хедере не требуется. Достаточно *предварительного объявления* (forward declaration).

Современный PImpl с std::unique_ptr

В современном C++ для управления временем жизни реализации используется `std::unique_ptr`. Это обеспечивает RAII: реализация будет автоматически удалена вместе с публичным объектом.

DatabaseConnection.h (С PImpl)

```
#pragma once
#include <string>
#include <memory> // для std::unique_ptr

class DatabaseConnection {
public:
    DatabaseConnection();
    ~DatabaseConnection(); // Важно: объявляем, но не определяем здесь

    // Перемещающие операции (Move Semantics) необходимы,
    // так как unique_ptr нельзя копировать.
    DatabaseConnection(DatabaseConnection&&) noexcept;
    DatabaseConnection& operator=(DatabaseConnection&&) noexcept;

    // Копирование запрещаем (или реализуем через глубокое копирование Impl)
```

```
    DatabaseConnection(const DatabaseConnection&) = delete;
    DatabaseConnection& operator=(const DatabaseConnection&) = delete;

    void connect(const std::string& connectionString);
    void executeQuery(const std::string& query);

private:
    // Предварительное объявление.
    // Класс Impl определен где-то в другом месте.
    struct Impl;

    // Указатель на реализацию.
    // std::unique_ptr требует полного типа T только в момент вызова delete
    std::unique_ptr<Impl> pImpl;
};
```

Обратите внимание: в этом файле нет `#include <libpq-fe.h>`. Зависимость полностью устранена.

DatabaseConnection.cpp

```
#include "DatabaseConnection.h"
// Включаем тяжелые зависимости ТОЛЬКО в файле реализации
#include <libpq-fe.h>
#include <vector>

// Полное определение класса реализации
struct DatabaseConnection::Impl {
    PGconn* pgConnection = nullptr;
    std::vector<std::string> cache;
    int internalState = 0;

    // Можно выносить сюда и приватные методы
    void helperFunction() { /* ... */ }
};

// Конструктор: выделяем память под реализацию
DatabaseConnection::DatabaseConnection()
    : pImpl(std::make_unique<Impl>()) {}

// Деструктор: именно здесь компилятору нужен полный тип Impl
DatabaseConnection::~DatabaseConnection() = default;

// Реализация перемещения
DatabaseConnection::DatabaseConnection(DatabaseConnection&&) noexcept = default;
DatabaseConnection& DatabaseConnection::operator=(DatabaseConnection&&)
    noexcept = default;

// Проксирование вызовов
void DatabaseConnection::connect(const std::string& str) {
    // Обращение к полям через стрелку pImpl->
    pImpl->pgConnection = PQconnectdb(str.c_str());
}
```

Проблема неполного типа (Incomplete Type) в деструкторе

Распространенной ошибкой является попытка определить деструктор в заголовочном файле (или использовать деструктор по умолчанию, сгенерированный компилятором в хедере).

```
// DatabaseConnection.h
class DatabaseConnection {
    // ОШИБКА, если Impl объявлен только предварительно (class Impl;)
    ~DatabaseConnection() = default;
    std::unique_ptr<Impl> pImpl;
};
```

Причина ошибки кроется в устройстве `std::unique_ptr`. В его деструкторе происходит вызов `delete ptr`. Оператор `delete` перед освобождением памяти вызывает деструктор объекта. Чтобы вызвать деструктор объекта `Impl`, компилятор должен видеть определение класса `Impl`. В заголовочном файле `Impl` — это неполный тип (incomplete type).

Использование `delete` на неполном типе в C++ является либо ошибкой компиляции, либо (в случае старых стандартов и сырых указателей) неопределенным поведением (UB). `std::unique_ptr` защищает от UB с помощью `static_assert`, проверяющего `sizeof(T) > 0`.

> **Важно!**
>
> Вы обязаны объявить деструктор в хедере, а реализовать его (даже через `= default`) в `.cpp` файле, **после** того как определена структура `Impl`.

ABI Stability (Бинарная совместимость)

Application Binary Interface (ABI) определяет, как объекты располагаются в памяти. В примере без PImpl размер объекта `DatabaseConnection` зависит от реализации `std::vector` и внутренней структуры `PGconn`.

Если вы обновите версию компилятора (что может изменить `sizeof(std::vector)`) или добавите приватное поле, размер класса изменится. Если ваше приложение динамически линкуется с библиотекой, предоставляющей этот класс, изменение размера приведет к смещению адресов полей и краху программы, если библиотека обновлена, а приложение — нет.

С PImpl размер класса `DatabaseConnection` всегда равен размеру одного `std::unique_ptr` (обычно размер машинного слова). Вы можете менять содержимое `Impl` как угодно: добавлять поля, менять контейнеры, удалять зависимости. Бинарный интерфейс публичного класса остается замороженным. Это критически важно для разработки библиотек (DLL/Shared Objects), которые должны обновляться без перекомпиляции клиентов.

Цена абстракции (Performance Trade-offs)

PImpl не является бесплатным. За изоляцию зависимостей приходится платить производительностью.

1. **Динамическая аллокация:** При создании каждого объекта `DatabaseConnection` происходит обращение к куче (`new Impl`). Это медленнее, чем выделение памяти на стеке, и может вызвать фрагментацию памяти при создании миллионов мелких объектов.
2. **Indirection (Косвенная адресация):** Любой вызов метода требует разыменования указателя `pImpl→field`. Это дополнительная инструкция процессора и потенциальный промах кэша (Cache Miss), так как данные реализации лежат в куче, далеко от самого объекта.
3. **Размер кода:** Компилятор не может заинлайнить методы, реализация которых скрыта в `.cpp`.

Для решения проблемы с аллокацией используется техника **Fast PImpl**. Вместо указателя используется буфер на стеке (`std::aligned_storage`), размер которого подбирается вручную.

```
// Fast PImpl (упрощенно)
class FastWidget {
    // Резервируем память внутри объекта (на стеке)
    // Риск: если Impl станет больше 64 байт, придется менять хедер
    alignas(8) unsigned char storage[64];
    struct Impl; // Реализация будет placement new в этот буфер
};
```

Этот подход возвращает скорость доступа и убирает аллокацию, но возвращает зависимость от размера реализации (хотя и в меньшей степени — пока мы укладываемся в буфер, перекомпиляция не нужна).

Глава 30

Архитектурные паттерны: Singleton и Template Method

Паттерны проектирования (Design Patterns) — это не готовые библиотеки и не строгие алгоритмы, которые можно скопировать в код. Это формализованные описания способов решения часто встречающихся архитектурных задач. В контексте C++ паттерны приобретают специфику, связанную с управлением временем жизни объектов, полиморфизмом и возможностями метапрограммирования (шаблонов).

В этой главе мы рассмотрим два фундаментальных паттерна: *Template Method*, который позволяет выделять общий алгоритм в иерархиях классов, и *Singleton*, реализация которого в C++ прошла долгий путь эволюции от простых глобальных переменных до потокобезопасных конструкций на основе CRTP.

Шаблонный метод (Template Method)

Часто при проектировании иерархии классов возникает ситуация, когда общий алгоритм действий одинаков для всех наследников, но отдельные шаги этого алгоритма должны различаться.

В языках без множественного наследования реализации или без детерминированного разрушения объектов часто приходится дублировать логику. В C++ этот паттерн реализуется естественно через механизм виртуальных функций.

> **Template Method**
>
> Поведенческий паттерн, определяющий скелет алгоритма в базовом классе и делегирующий реализацию некоторых шагов подклассам. Подклассы могут переопределять шаги алгоритма, не меняя его общей структуры.

Рассмотрим пример симуляции жизни животных. У всех животных день проходит по одному сценарию: проснуться, издать звук, поесть. Процесс пробуждения и приема пищи (в нашей упрощенной модели) одинаков, а вот звуки — уникальны.

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

// Базовый класс определяет алгоритм
```

```cpp
class Animal {
public:
    virtual ~Animal() = default;

    // Шаблонный метод.
    // Он НЕ виртуальный, чтобы зафиксировать структуру алгоритма.
    void liveOneDay() {
        wakeUp();
        makeSound(); // Точка кастомизации (Virtual Hook)
        eat();
    }

protected:
    // Общие шаги реализации
    void wakeUp() const {
        std::cout << "Opening eyes..." << std::endl;
    }

    void eat() const {
        std::cout << "Eating food..." << std::endl;
    }

    // Чисто виртуальный метод — обязан быть реализован в наследнике
    virtual void makeSound() const = 0;
};

// Конкретная реализация
class Dog : public Animal {
protected:
    void makeSound() const override {
        std::cout << "Woof! Woof!" << std::endl;
    }
};

class Cat : public Animal {
protected:
    void makeSound() const override {
        std::cout << "Meow..." << std::endl;
    }
};
```

В идиоматике C++ этот подход часто пересекается с идиомой **NVI (Non-Virtual Interface)**. NVI рекомендует делать публичные методы невиртуальными (обеспечивая стабильный интерфейс и проверки инвариантов), а точки расширения (виртуальные методы) делать `protected` или `private`.

Это позволяет базовому классу `Animal` гарантировать, что звук будет издан именно между пробуждением и едой, и никакой наследник не сможет случайно сломать этот порядок, забыв вызвать базовый метод.

Паттерн Singleton (Одиночка)

Singleton — один из самых спорных, но часто используемых паттернов. Его задача:

1. Гарантировать, что у класса есть только один экземпляр.
2. Предоставить к нему глобальную точку доступа.

Примеры использования:

- **Аллокатор памяти:** Система управления памятью обычно глобальна для процесса.
- **Файловая система / Логгер:** Централизованная запись логов в один файл требует синхронизации, которую проще обеспечить через единый объект.
- **Драйвер видеокарты:** Физическое устройство одно, и доступ к нему должен быть монопольным.

Критика Singleton

Singleton часто называют "антипаттерном" из-за проблем, которые он привносит:

- **Скрытые зависимости:** Если функция принимает аргументы, её зависимости явны. Если функция внутри себя вызывает `Logger::getInstance()`, зависимость скрыта.
- **Проблемы тестирования:** Невозможно изолировать тесты. Если один тест изменил состояние синглтона, второй тест может упасть. Нельзя запустить два экземпляра "приложения" параллельно в одном тестовом процессе.
- **Глобальное состояние:** Усложняет понимание потока данных и многопоточности.

Тем не менее, в системном программировании Singleton остается необходимым инструментом.

Static Initialization Order Fiasco

Почему нельзя просто создать глобальную переменную?

```
// Logger.h
class Logger { ... };
extern Logger globalLogger;

// Logger.cpp
Logger globalLogger;
```

Проблема возникает, когда у нас есть **несколько** глобальных статических объектов в **разных** единицах трансляции (файлах .cpp), и один из них использует другой в своем конструкторе.

Представим ситуацию:

1. В `Logger.cpp` определен `globalLogger`.
2. В `Application.cpp` определен объект `Application app`, который в конструкторе пишет лог: `globalLogger.log("App started")`.

Стандарт C++ **не определяет** порядок инициализации глобальных переменных, находящихся в разных единицах трансляции. Линкер может собрать их в любом порядке.

> **Важно!**
>
> Если `app` будет инициализировано раньше, чем `globalLogger`, конструктор `Application` обратится к неинициализированной памяти (в лучшем случае заполненной нулями). Это приведет к краху программы (Segmentation Fault) или неопределенному поведению до начала функции `main`.

Это явление называется *Static Initialization Order Fiasco*.

Meyers Singleton

Решением проблемы инициализации является идиома, популяризированная Скоттом Мейерсом. Идея заключается в переносе статической переменной из глобальной области видимости внутрь функции.

```
class Logger {
private:
    Logger() { /* ... */ }
    // Запрещаем копирование
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

public:
    static Logger& getInstance() {
        // Статическая локальная переменная.
        // Инициализируется ПРИ ПЕРВОМ вызове функции.
        static Logger instance;
        return instance;
    }

    void log(const std::string& msg) { /* ... */ }
};
```

Теперь порядок инициализации детерминирован. Если конструктор `Application` вызовет `getInstance()`, логгер будет создан именно в этот момент.

Thread Safety (Потокобезопасность)

До стандарта C++11 инициализация статических локальных переменных не была потокобезопасной. Если два потока одновременно заходили в `getInstance()` в первый раз, мог возникнуть race condition (двойное создание). Приходилось использовать паттерн *Double-Checked Locking*.

В C++11 и новее стандарт гарантирует: **инициализация статических локальных переменных потокобезопасна**. Компилятор автоматически генерирует блокировки (обычно через атомарные флаги или мьютексы), чтобы только один поток выполнил инициализацию. Это называют "Magic Statics".

Проблема разрушения (Dead Reference)

Meyers Singleton решает проблему инициализации, но оставляет проблему разрушения. Статические объекты разрушаются в порядке, обратном созданию, после выхода из `main`.

Сценарий катастрофы:

1. `main` завершается.
2. Разрушается `Logger` (так как он был создан последним, лениво).
3. Разрушается глобальный объект `Application`.
4. В деструкторе `Ãpplication` происходит попытка записать прощальное сообщение: `Logger::getInstance().log("App stopped")`.
5. Метод `getInstance` возвращает ссылку на уже уничтоженный объект `Logger`.
6. Use-after-free, UB, крах.

Leaky Singleton

Чтобы избежать этой проблемы, в системном программировании часто используют "утекающий" синглтон. Мы создаем объект через `new`, но никогда не вызываем `delete`.

```
static Logger& getInstance() {
    // Память выделяется в куче и никогда не освобождается.
    static Logger* instance = new Logger();
    return *instance;
}
```

С точки зрения C++, это утечка памяти (Memory Leak). Однако, это *безопасная* утечка. Когда процесс завершается, операционная система всё равно реиспользует все страницы памяти, принадлежавшие процессу. Зато объект `Logger` гарантированно живёт до самого последнего момента существования процесса, и любой деструктор может безопасно им воспользоваться.

Обобщение через CRTP

В крупном проекте писать метод `getInstance` и закрывать конструкторы для каждого синглтона утомительно. Мы можем захотеть создать базовый класс `Singleton<T>`, который инкапсулирует эту логику.

Однако, `Singleton` должен знать тип создаваемого класса, а создаваемый класс должен наследоваться от `Singleton`. Возникает циклическая зависимость, которая разрешается с помощью идиомы **CRTP (Curiously Recurring Template Pattern)** — Странно Рекурсивный Шаблон.

```
#include <utility> // для std::forward

template <typename T>
class Singleton {
public:
```

```
    // Variadic templates позволяют передать аргументы в конструктор T
    template <typename... Args>
    static T& getInstance(Args&&... args) {
        // Создаем через new (Leaky Singleton), чтобы избежать проблем с
        ↪ деструкцией
        static T* instance = new T(std::forward<Args>(args)...);
        return *instance;
    }

protected:
    Singleton() = default;
    ~Singleton() = default;

public:
    // Запрещаем копирование и перемещение для самого Singleton
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};

// Пример использования
class FileSystem : public Singleton<FileSystem> {
    // Важно: Singleton<FileSystem> должен иметь доступ к приватному конструктору
    friend class Singleton<FileSystem>;

private:
    FileSystem(const std::string& root) {
        std::cout << "FS mounted at " << root << "\n";
    }

public:
    void readFile(const std::string& path) { /* ... */ }
};

int main() {
    // Первый вызов инициализирует объект
    FileSystem::getInstance("/var/data").readFile("config.txt");

    // Последующие вызовы возвращают тот же объект (аргументы игнорируются)
    FileSystem::getInstance().readFile("log.txt");
}
```

Разбор механизма CRTP:

1. `class FileSystem : public Singleton<FileSystem>`: В момент объявления класса `FileSystem`, шаблон `Singleton` инстанцируется типом `FileSystem`. Хотя `FileSystem` еще является неполным типом (incomplete type), его можно использовать как аргумент шаблона, если шаблон не обращается к внутренностям типа в момент объявления.
2. Метод `getInstance` является статическим членом базового класса, но возвращает ссылку на `T` (то есть на наследника).
3. `friend class Singleton<FileSystem>`: Это ключевой момент. Так как конструктор `FileSystem` приватен (чтобы никто не создал второй экземпляр), базовый класс не мог бы вызвать `new T(...)`. Дружба дает базовому классу права на доступ к приватным членам наследника.

Этот паттерн позволяет элегантно вынести инфраструктурную логику (управление единственностью) из бизнес-логики класса.

Глава 31

Глубокое погружение в исключения (Exception Safety)

Механизм исключений в C++ — это мощный инструмент обработки ошибок, который часто понимается поверхностно. В отличие от кодов возврата (как в C или Go) или монад (как `Result<T, E>` в Rust), исключения в C++ предлагают автоматическую раскрутку стека и нелокальную передачу управления. Однако неправильное использование исключений ведет к утечкам ресурсов, неопределенному поведению и деградации производительности.

Эта глава посвящена не синтаксису `try-catch`, а внутренней механике процесса (*Under the hood*), гарантиям безопасности (Exception Safety Guarantees) и правильному написанию устойчивого кода.

Механика исключений: Zero-cost vs Stack Unwinding

Часто можно услышать, что "исключения медленные". Это утверждение верно лишь отчасти. Современные компиляторы (GCC, Clang, MSVC) реализуют модель, называемую **Zero-cost exceptions** (на Itanium C++ ABI).

Happy Path (Путь без ошибок)

Пока исключение **не** брошено, накладные расходы на поддержку механизма исключений практически равны нулю. В коде нет проверок `if (error)` после каждого вызова функции. Процессор выполняет линейный поток инструкций, что благоприятно для предсказателя переходов (Branch Predictor) и кэша инструкций.

Error Path (Путь ошибки)

Когда выполняется оператор `throw`, происходит дорогая операция, называемая **Stack Unwinding** (Раскрутка стека):

1. **Поиск обработчика:** Runtime-система (например, `libunwind`) просматривает таблицу исключений (`.eh_frame` или аналогичную секцию), сгенерированную компилятором. Эта таблица сопоставляет диапазоны адресов инструкций (Program Counter) с информацией о том, какие деструкторы нужно вызвать в текущем кадре стека и есть ли здесь блок `catch`.

2. **Вызов деструкторов:** Если в текущей функции нет подходящего `catch`, система вызывает деструкторы всех локальных объектов, созданных с начала функции до точки выброса.
3. **Раскрутка:** Стек урезается (stack pointer смещается), управление передается вызывающей функции, и процесс повторяется (шаг 1), пока не будет найден `catch` или пока не будет достигнут `main`.
4. **Terminate:** Если стек раскрутился до `main`, а обработчик не найден, вызывается `std::terminat` (обычно `abort()`).

Этот процесс включает парсинг таблиц, множество косвенных переходов и вызовов, что действительно очень медленно (тысячи тактов процессора). Поэтому исключения не должны использоваться для управления потоком выполнения в штатном режиме (например, для выхода из цикла).

Правила перехвата исключений

Slicing (Срезка) объектов

Ловить исключения нужно **строго по ссылке** (желательно константной). Ловля по значению приводит к срезке полиморфной части объекта.

```
#include <iostream>
#include <exception>

struct MyError : std::exception {
    const char* what() const noexcept override {
        return "My Custom Error";
    }
    int errorCode = 500;
};

void badHandler() {
    try {
        throw MyError();
    }
    // ПЛОХО: catch по значению.
    // Создается копия std::exception, отбрасывая часть MyError.
    catch (std::exception e) {
        // e.what() вызовет метод базового класса std::exception,
        // а не переопределенный MyError::what()!
        std::cout << e.what() << "\n";
        // e.errorCode недоступен
    }
}

void goodHandler() {
    try {
        throw MyError();
    }
    // ХОРОШО: catch по ссылке. Полиморфизм работает.
    catch (const std::exception& e) {
```

```
        std::cout << e.what() << "\n"; // Выведет "My Custom Error"
    }
}
```

Порядок блоков catch

Блоки `catch` проверяются сверху вниз. Компилятор не всегда предупредит вас, если вы попытаетесь поймать базовый тип раньше производного.

```
try {
    // ...
}
catch (const std::exception& e) {
    // Сюда попадут ВСЕ наследники std::exception
}
catch (const std::runtime_error& e) {
    // Мертвый код (Dead Code)!
    // runtime_error наследуется от exception,
    // поэтому он будет перехвачен первым блоком.
}
```

Спецификатор noexcept

Ключевое слово `noexcept` — это часть контракта функции.

- `void func() noexcept;` — гарантирует, что функция не выбросит исключение.
- Если исключение всё же вылетит из `noexcept`-функции, программа немедленно вызовет `std::terminate()`. Раскрутка стека может не произойти (деструкторы не вызовутся).

Компилятор использует эту информацию для агрессивных оптимизаций. Если функция помечена `noexcept`, компилятору не нужно генерировать таблицы раскрутки стека для вызовов внутри неё.

Вектор и noexcept move-конструкторы

Это критически важный аспект для производительности контейнеров. Рассмотрим `std::vector::` Если вектору не хватает места (capacity), он должен:

1. Выделить новый блок памяти.
2. Перенести элементы из старого блока в новый.
3. Удалить старый блок.

При переносе элементов вектор предпочитает использовать *перемещение* (move constructor), так как это дешево (копирование указателей). Однако, если move-конструктор элемента может выбросить исключение, возникает проблема нарушения транзакционности.

Представьте, что мы перенесли 5 элементов из 10, и 6-й элемент выбросил исключение при перемещении.

- Старый вектор испорчен (из 5 элементов ресурсы украдены).
- Новый вектор недостроен.
- Восстановить исходное состояние невозможно.

Поэтому `std::vector` использует идиому `std::move_if_noexcept`:

```
// Псевдокод логики вектора при реаллокации
if (std::is_nothrow_move_constructible<T>::value) {
    // Безопасно муваем. Если упадем - значит terminate, терять нечего.
    move_elements();
} else {
    // Приходится копировать.
    // Если копия упадет, старый вектор останется нетронутым.
    copy_elements();
}
```

> **Важно!**
>
> Всегда помечайте перемещающие конструкторы и операторы присваивания как `noexcept`, иначе стандартные контейнеры деградируют до копирования, что убьет производительность.

Гарантии безопасности исключений (Exception Safety Guarantees)

При проектировании классов вы должны документировать, что произойдет с объектом, если его метод выбросит исключение. Абстракция не должна "течь" или оставлять систему в рассогласованном состоянии.

Выделяют четыре уровня гарантий (от слабой к сильной):

1. No-throw Guarantee (Гарантия отсутствия сбоев)

Функция никогда не выбрасывает исключений. Это обязательное требование для:

- **Деструкторов:** Выброс исключения из деструктора во время раскрутки стека (когда уже летит другое исключение) приводит к `std::terminate`.
- **Функций освобождения памяти** (`delete`, `free`).
- **Swap:** Функция обмена должна быть надежной, так как на ней строятся другие гарантии.

2. Strong Guarantee (Строгая гарантия)

Транзакционная семантика: "Commit or Rollback". Если функция завершилась с исключением, состояние программы остается таким же, каким было до вызова функции. Никаких побочных эффектов.

Пример реализации через идиому **Copy-and-Swap**:

```cpp
class Widget {
private:
    std::vector<int> data;
public:
    // Мы хотим заменить данные, но безопасно.
    void updateData(const std::vector<int>& newData) {
        // 1. Создаем временную копию (может бросить исключение).
        // Если бросит здесь - this не изменится.
        std::vector<int> temp = newData;

        // 2. Меняем состояние через небросающий swap.
        // std::vector::swap - noexcept.
        using std::swap;
        swap(this->data, temp);

        // 3. temp (со старыми данными) разрушается при выходе.
    }
};
```

3. Basic Guarantee (Базовая гарантия)

Если произошло исключение:

- Нет утечек ресурсов (память, файлы, мьютексы освобождены).
- Инварианты класса сохранены (внутреннее состояние согласовано).
- Однако данные могут быть изменены и непредсказуемы (но валидны).

Это минимальный уровень, требуемый от корректного C++ кода. Обычно обеспечивается использованием RAII (`std::unique_ptr`, `std::lock_guard`).

4. No Guarantee (Отсутствие гарантий)

Если произошло исключение, всё сломалось: память утекла, данные коррумпированы. Это недопустимо в качественном коде.

Инварианты класса

Инвариант — это логическое условие, которое всегда истинно для валидного объекта (кроме момента выполнения методов, меняющих состояние).

Пример класса, уязвимого к исключениям:

```cpp
class Buffer {
    int* ptr = nullptr;
    size_t size = 0;
public:
    Buffer(size_t n) {
        // ОПАСНО!
        size = n;            // 1. Установили размер
        ptr = new int[n];    // 2. Пытаемся выделить память
```

```
        // Если new бросит std::bad_alloc:
        // - Конструктор прервется.
        // - Деструктор НЕ будет вызван (объект не достроен).
        // - У нас остался "объект" с size > 0, но без памяти.
        // Хотя самого объекта технически нет, это нарушение логики построения.
    }
};
```

Более тонкий пример нарушения инварианта (Basic Guarantee):

```
void process() {
    size_t oldSize = size;
    size = 0;       // Ломаем инвариант "size соответствует данным"
    doWork();       // Если тут throw...
    size = oldSize; // ...восстановление не выполнится
    // Объект остался с size=0, но старыми данными.
}
```

Решение: использовать `ScopeGuard` или менять состояние в самом конце (после всех опасных операций).

Function-try-block

Специфическая конструкция C++, позволяющая перехватить исключения, возникающие в **списке инициализации конструктора** (Member Initializer List). Это единственный способ обработать ошибку в конструкторе базового класса или члена класса.

```
class Service {
    Resource res; // Конструктор Resource может бросить
public:
    Service() try : res(42) {
        // Тело конструктора
    }
    catch (const std::exception& e) {
        // Мы попадем сюда, если res(42) упадет.
        // ВАЖНО: Исключение здесь НЕЛЬЗЯ подавить.
       // Даже если мы не напишем throw;, оно будет перевыброшено автоматически.
        // Смысл блока только в логировании или трансляции исключения.
        std::cerr << "Init failed: " << e.what() << "\n";
    }
};
```

Использовать `function-try-block` для обычных функций не рекомендуется (это эквивалентно `try` внутри всего тела), а в конструкторах он нужен крайне редко, так как обычно лучше дать исключению улететь наверх, чем пытаться починить недостроенный объект.

Часть VII

Лекция 07 – Метапрограммирование

Глава 32

Препроцессор и Макромагия: От текстовой подстановки до кодогенерации

Метапрограммирование в C++ часто ассоциируется исключительно с шаблонами (templates), однако исторически первым и до сих пор широко используемым инструментом метапрограммирования является препроцессор.

Препроцессор языка C (и C++) — это утилита, которая обрабатывает исходный код **до** этапа компиляции. Он не знает синтаксиса C++, не понимает типов данных, областей видимости или классов. Для препроцессора ваш код — это просто поток текста (набор токенов), над которым производятся операции подстановки, склейки и условного включения.

Фундаментальные механики

Работа препроцессора управляется директивами, начинающимися с символа #. Эти директивы выполняются на самом раннем этапе трансляции, превращая исходный файл `.cpp` в единицу трансляции (translation unit), готовую к синтаксическому анализу компилятором.

Include Guards и условная компиляция

Одной из базовых задач препроцессора является управление зависимостями и платформозависимым кодом. Директива `#include` буквально копирует содержимое указанного файла в точку вызова. Это создает проблему множественного включения: если заголовочный файл `A.h` включен в `B.h` и `C.h`, а `main.cpp` включает и `B`, и `C`, то содержимое `A.h` попадет в `main.cpp` дважды, что приведет к ошибкам переопределения символов (ODR — One Definition Rule).

Традиционное решение — Include Guards:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

struct MyStruct { /* ... */ };

#endif // MY_HEADER_H
```

Также препроцессор используется для проверки окружения. Макросы позволяют включать или исключать куски кода в зависимости от ОС, компилятора или конфигурации сбор-

ки (Debug/Release).

```
#ifdef _WIN32
    #include <Windows.h>
#elif defined(__unix__)
    #include <unistd.h>
#else
    #error "Unknown platform"
#endif
```

> **Важно!**
>
> Директива `#error` прерывает компиляцию с указанным сообщением. Это полезно для гарантии того, что код не будет собран на неподдерживаемой архитектуре, вместо того чтобы получить тысячи непонятных синтаксических ошибок далее.

Предопределенные макросы

Компилятор предоставляет набор стандартных макросов, которые содержат информацию о текущем контексте компиляции. Они часто используются для логирования и отладки.

- `__LINE__` (int): Текущий номер строки в исходном файле.
- `__FILE__` (string literal): Имя текущего файла.
- `__func__` (string literal): Имя текущей функции (стандартизировано в C99/C++11). В GCC/Clang также доступен нестандартный `__PRETTY_FUNCTION__`, который выводит полную сигнатуру функции, включая типы аргументов и шаблонов.

Имена, начинающиеся с двух подчеркиваний (`__`) или с одного подчеркивания и заглавной буквы (`_Big`), зарезервированы стандартом для реализации. Определение собственных макросов с такими именами является Undefined Behavior.

Макросы как функции

Функциональные макросы позволяют выполнять текстовую подстановку аргументов. В отличие от шаблонов или `inline`-функций, макросы не проверяют типы и вычисляются путем простой замены токенов.

Опасность приоритета операций

При написании макросов критически важно оборачивать каждый аргумент и все выражение целиком в скобки. Поскольку препроцессор просто вставляет текст, операторы с низким приоритетом могут "захватить" соседние выражения.

Рассмотрим некорректный макрос:

```
#define MUL(a, b) a * b

int res = MUL(2 + 3, 4 + 5);
// Раскрывается в: 2 + 3 * 4 + 5
// Результат: 2 + 12 + 5 = 19
```

```
// Ожидалось: 5 * 9 = 45
```

Корректная реализация:

```
#define MUL(a, b) ((a) * (b))
```

Stringification (Оператор #)

Оператор решетки # используется для превращения аргумента макроса в строковый литерал. Это невозможно сделать средствами самого C++, так как имена переменных теряются после компиляции.

Это ключевой механизм для реализации `assert`-ов, которые выводят само проверяемое выражение при ошибке.

```
#include <iostream>
#include <cstdlib>

#define MY_ASSERT(expr) \
    if (!(expr)) { \
        std::cerr << "Assertion failed: " << #expr \
                  << ", file " << __FILE__ \
                  << ", line " << __LINE__ << std::endl; \
        std::abort(); \
    }

int main() {
    int x = 5;
    // Вывод: Assertion failed: x == 2, file main.cpp, line 15
    MY_ASSERT(x == 2);
}
```

Обратите внимание: препроцессор видит выражение `x == 2` как последовательность токенов, а оператор # превращает их в строку `"x == 2"`.

Идиома do-while(0)

При создании многострочных макросов возникает проблема корректного синтаксиса в управляющих конструкциях. Если просто обернуть код в фигурные скобки `{ }`, использование макроса в ветке `if` может сломать `else`.

Пример проблемы:

```
#define LOG_ERROR(msg) \
    { std::cerr << "Error: "; std::cerr << msg << std::endl; }

if (cond)
    LOG_ERROR("Fail"); // Точка с запятой здесь завершает if!
else                   // ОШИБКА: else без if
    /* ... */
```

Раскрытие кода:

```
if (cond)
    { std::cerr << "Error: "; std::cerr << "Fail" << std::endl; };
else // Синтаксическая ошибка: лишняя ; перед else
```

Решение — использование цикла `do { ... } while(0)`. Эта конструкция требует точки с запятой на конце, что делает вызов макроса синтаксически идентичным вызову обычной функции.

```
#define LOG_ERROR(msg) \
    do { \
        std::cerr << "Error: "; \
        std::cerr << msg << std::endl; \
    } while(0)
```

Variadic Macros и перегрузка

C99 и C++11 принесли поддержку макросов с переменным числом аргументов. Они обозначаются многоточием `...`, а доступ к аргументам осуществляется через идентификатор `__VA_ARGS__`.

Одной из самых мощных (и неочевидных) техник является перегрузка макросов по количеству аргументов. Поскольку препроцессор не поддерживает перегрузку функций напрямую, приходится использовать трюк с подменой макроса-диспетчера.

Механизм выбора N-го аргумента

Задача: реализовать макрос `ENSURE`, который можно вызывать как `ENSURE(cond)` или `ENSURE(cond, message)`.

Алгоритм: 1. Создать два макроса реализации: `ENSURE_1(cond)` и `ENSURE_2(cond, msg)`. 2. Создать макрос-селектор, который выбирает нужную реализацию в зависимости от количества переданных аргументов.

```
// Реализации
#define ENSURE_1(cond) \
    if (!(cond)) { std::cerr << "Check failed: " << #cond << std::endl;
    ↪  std::abort(); }

#define ENSURE_2(cond, msg) \
    if (!(cond)) { std::cerr << msg << std::endl; std::abort(); }

// Магия выбора
// Этот макрос принимает набор аргументов и возвращает 3-й аргумент
#define GET_3RD_ARG(arg1, arg2, arg3, ...) arg3

// Макрос-диспетчер
// Если передано 1 аргумент (x):
//   GET_3RD_ARG(x, ENSURE_2, ENSURE_1) -> вернет ENSURE_1
```

```
// Если передано 2 аргумента (x, y):
//   GET_3RD_ARG(x, y, ENSURE_2, ENSURE_1) -> вернет ENSURE_2
#define ENSURE_MACRO_CHOOSER(...) \
    GET_3RD_ARG(__VA_ARGS__, ENSURE_2, ENSURE_1)

// Итоговый макрос
#define ENSURE(...) \
    ENSURE_MACRO_CHOOSER(__VA_ARGS__)(__VA_ARGS__)
```

Этот механизм работает за счет сдвига аргументов. Макрос `GET_3RD_ARG` всегда возвращает третий элемент из списка. Подставляя в него пользовательские аргументы `__VA_ARGS__` в начало, мы сдвигаем служебные имена макросов (`ENSURE_2`, `ENSURE_1`) на нужную позицию.

X-Macros: Кодогенерация списков

Одной из самых частых проблем C++ является отсутствие рефлексии. Например, при объявлении `enum` для цветов, мы теряем текстовое представление имени цвета. Чтобы вывести имя цвета в лог, приходится писать функцию `ToString` с большим `switch-case`, дублируя имена констант.

Техника **X-Macros** позволяет определить данные один раз и генерировать различный код (enum, массив строк, switch-case) путем многократного включения списка.

Реализация X-Macro

Идея состоит в том, чтобы список элементов был определен как последовательность вызовов некоторого (пока не определенного) макроса `X`.

```
// colors.def (или просто #define внутри файла)
#define LIST_OF_COLORS(X) \
    X(Red) \
    X(Green) \
    X(Blue) \
    X(Yellow)
```

Теперь мы можем использовать этот список для генерации кода.

Шаг 1: Генерация enum Мы определяем макрос `X(name)` так, чтобы он разворачивался в `name,`.

```
enum class Color {
    #define X(name) name,
    LIST_OF_COLORS(X)
    #undef X
};
// Раскроется в:
// enum class Color {
//     Red,
//     Green,
//     Blue,
```

```
// Yellow,
// };
```

Шаг 2: Генерация функции ToString Мы переопределяем `X(name)` так, чтобы он генерировал `case`.

```
const char* ToString(Color c) {
    switch (c) {
        #define X(name) case Color::name: return #name;
        LIST_OF_COLORS(X)
        #undef X
    }
    return "Unknown";
}
// Раскроется в:
// switch (c) {
//     case Color::Red: return "Red";
//     case Color::Green: return "Green";
//     ...
// }
```

> **X-Macros**
>
> Это паттерн использования препроцессора, при котором данные отделяются от их представления. Данные описываются в виде списка вызовов макроса-заглушки, который переопределяется контекстно для генерации различных структур кода (объявлений, массивов, операторов switch).

Эта техника широко используется в крупных проектах (например, в LLVM или ClickHouse) для синхронизации конфигураций, кодов ошибок или настроек CLI, гарантируя, что добавление нового элемента в список автоматически обновит все связанные структуры данных.

Резюме раздела

- Препроцессор — мощный инструмент текстовой генерации, работающий до компилятора.
- Всегда используйте скобки вокруг аргументов макроса и идиому `do-while(0)` для многострочных макросов.
- Оператор # позволяет получить строковое представление кода (полезно для отладки и сериализации).
- X-Macros позволяют эмулировать рефлексию и избегать дублирования кода при работе с перечислениями и списками свойств.

Глава 33

Метапрограммирование на Шаблонах (TMP) и Constexpr

Метапрограммирование в C++ — это методика написания программ, которые выполняются компилятором и генерируют другие программы (или константы) в качестве своего вывода. Это Turing-complete подсистема языка, встроенная непосредственно в процесс компиляции.

Исторически метапрограммирование в C++ развивалось от случайного открытия возможности вычислений на шаблонах (Template Metaprogramming — TMP) до полноценной поддержки вычислений времени компиляции через механизм `constexpr`. В этой главе мы рассмотрим эволюцию этих подходов: от сложных рекурсивных структур к современному императивному коду, выполняемому на этапе трансляции.

Классический TMP: Шаблоны как функциональный язык

До стандарта C++11 шаблоны были единственным способом заставить компилятор выполнять произвольные вычисления. Этот подход (Legacy TMP) базируется на функциональной парадигме. В нем отсутствуют переменные (все данные иммутабельны) и циклы.

> **Template Metaprogramming (TMP)**
>
> Парадигма, где:
>
> - **Функции** представлены шаблонными структурами (`struct` или `class`).
> - **Аргументы** передаются как параметры шаблона (`<int N, typename T>`).
> - **Возвращаемые значения** — это вложенные константы (`static const value`) или определения типов (`typedef type`).
> - **Циклы** реализуются через рекурсию.
> - **Ветвление** реализуется через специализацию шаблонов.

Рекурсия и Специализация: Вычисление факториала

Классическим примером ("Hello World" от мира TMP) является вычисление факториала. Поскольку циклы `for` или `while` недоступны на уровне инстанцирования шаблонов, используется рекурсивное определение.

```cpp
#include <iostream>

// Общий шаблон: N! = N * (N-1)!
template <unsigned long N>
struct Factorial {
    // "Возвращаемое значение" вычисляется рекурсивно
    static const unsigned long value = N * Factorial<N - 1>::value;
};

// Условие выхода из рекурсии: специализация для 0
// 0! = 1
template <>
struct Factorial<0> {
    static const unsigned long value = 1;
};

int main() {
    // Вычисление происходит полностью во время компиляции.
    // В бинарный код попадает только константа 3628800.
    std::cout << Factorial<10>::value << std::endl;
    return 0;
}
```

Механика работы: Когда компилятор встречает `Factorial<10>`, он пытается инстанцировать эту структуру. Внутри он видит обращение к `Factorial<9>`. Это запускает цепную реакцию инстанцирования:

$$F\langle 10\rangle \to F\langle 9\rangle \to \cdots \to F\langle 1\rangle \to F\langle 0\rangle$$

На этапе `Factorial<0>` компилятор выбирает специализацию, где `value` жестко задано как 1. После этого цепочка сворачивается обратно, перемножая константы.

> **Важно!**
>
> Глубина рекурсии шаблонов ограничена компилятором (обычно 900-1024 уровня). Превышение этого лимита приводит к ошибке компиляции. Это одна из главных проблем классического TMP.

Ветвление времени компиляции (Static If)

В классическом TMP нельзя написать `if`, так как инструкции процессора не существуют на этапе компиляции. Ветвление реализуется через выбор одного из двух типов.

```cpp
// Базовый шаблон (не определен или false-ветка)
template <bool Condition, typename T, typename F>
struct If {
    using type = F; // По умолчанию выбираем False-тип
};

// Частичная специализация для true
template <typename T, typename F>
struct If<true, T, F> {
    using type = T; // Если Condition == true, выбираем True-тип
};
```

```
12
13 // Пример использования
14 using ResultType = If<sizeof(void*) == 8, long long, int>::type;
```

Здесь `ResultType` станет `long long` на 64-битной системе и `int` на 32-битной. Это решение является предшественником `std::conditional` из стандартной библиотеки.

Constexpr: Возвращение к императивному стилю

С выходом C++11, а затем C++14 и C++20, подход к метапрограммированию кардинально изменился. Ключевое слово `constexpr` сообщает компилятору, что выражение *может* быть вычислено во время компиляции.

Это позволяет писать метапрограммы на обычном C++, используя привычный синтаксис (циклы, переменные, условные операторы), вместо сложной магии шаблонов.

Эволюция Constexpr

1. **C++11**: Функции `constexpr` были крайне ограничены. Они могли содержать только один оператор `return`. Никаких переменных, циклов или `if`.

```
1 // C++11 стиль
2     constexpr int factorial(int n) {
3         return (n == 0) ? 1 : n * factorial(n - 1);
4     }
```

2. **C++14**: Ограничения сняты. Разрешены локальные переменные, циклы `for/while`, условия `if`, при условии, что они не модифицируют глобальное состояние и не вызывают non-constexpr код.

```
1 // C++14 стиль
2     constexpr int factorial(int n) {
3         int result = 1;
4         for (int i = 1; i <= n; ++i) {
5             result *= i;
6         }
7         return result;
8     }
```

Теперь тот же самый факториал вычисляется без создания сотен типов `struct`, что значительно снижает нагрузку на компилятор и ускоряет сборку.

Compile-time аллокация памяти (C++20)

До C++20 вся память в `constexpr` контексте должна была быть стековой (локальной). Динамическое выделение памяти (`new/delete`) было запрещено.

C++20 ввел понятие **Transient Allocation** (временная аллокация). Теперь внутри `constexpr` функции можно выделять память через `new`, **при условии**, что эта память будет освобожде-

на (`delete`) до завершения вычисления этой функции. Память "не может вытечь" из этапа компиляции в рантайм.

Это изменение позволило сделать методы `std::vector` и `std::string` `constexpr`. Теперь можно сортировать векторы и конкатенировать строки прямо во время компиляции.

```
#include <vector>
#include <numeric>
#include <algorithm>

constexpr int sum_squares(int n) {
    std::vector<int> v; // Аллокация памяти на этапе компиляции!
    for (int i = 0; i < n; ++i) v.push_back(i);

    // Использование STL алгоритмов
    int sum = 0;
    for (int x : v) sum += x * x;

    return sum; // Вектор уничтожается здесь, память освобождается.
}

// Результат вычисляется компилятором
constexpr int val = sum_squares(10);
```

If Constexpr (C++17)

Одним из важнейших дополнений C++17 стала конструкция `if constexpr`. В отличие от обычного `if`, условие в `if constexpr` должно быть известно на этапе компиляции. Главная особенность: **отбрасываемая ветка не инстанцируется**.

Это решает фундаментальную проблему обобщенного программирования: как написать код, который работает для типов с разными интерфейсами, не вызывая ошибок компиляции.

Проблема обычного if

Рассмотрим функцию, которая возвращает `.size()` контейнера, если метод существует, или 0, если это просто число.

```
template <typename T>
auto get_size_bad(const T& t) {
    if (std::is_integral_v<T>) {
        return 0;
    } else {
        return t.size(); // ОШИБКА КОМПИЛЯЦИИ для int!
    }
}
```

Даже если мы передадим `int`, компилятор обязан скомпилировать обе ветки обычного `if`. При попытке скомпилировать `int.size()` произойдет ошибка, несмотря на то, что эта ветка никогда не выполнится в рантайме.

Решение через if constexpr

Используя `if constexpr`, мы указываем компилятору полностью игнорировать код в неактивной ветке. Он проверяется только на базовый синтаксис, но не на корректность методов типа T.

```cpp
#include <type_traits>

template <typename T>
auto get_size_safe(const T& t) {
    if constexpr (std::is_integral_v<T>) {
        return 0;
    } else {
        // Эта строка компилируется ТОЛЬКО если T не интегральный тип.
        // Для int вызов .size() даже не будет проверяться.
        return t.size();
    }
}
```

Этот механизм позволяет писать компактный обобщенный код, заменяя сложные конструкции SFINAE (которые мы рассмотрим в следующей главе) на читаемые блоки условий.

На заметку

Код внутри `if constexpr` все равно должен быть синтаксически корректным C++. Вы не можете написать там случайный набор символов, даже если ветка отбрасывается.

Резюме раздела

- **Legacy TMP** использует рекурсию шаблонов и специализацию. Это мощный, но трудный для чтения и медленный при компиляции подход.
- **Constexpr** позволяет перенести обычную логику C++ (циклы, переменные) на этап компиляции.
- **C++20** разрешает динамическую память в `constexpr`, делая доступными `std::vector` и `std::string`.
- **if constexpr** позволяет исключать куски кода из компиляции в зависимости от свойств типов, упрощая написание шаблонов.

Глава 34

SFINAE и Концепты: Управление перегрузкой функций

Одной из самых сложных и мощных возможностей шаблонов C++ является управление выбором перегрузок. В обычном программировании перегрузка функций тривиальна: компилятор выбирает функцию, сигнатура которой лучше всего соответствует переданным аргументам.

Однако в шаблонном метапрограммировании возникают ситуации, когда <<лучшее>> с точки зрения компилятора совпадение оказывается семантически некорректным. Нам необходим механизм, позволяющий исключать определенные шаблоны из рассмотрения (из множества перегрузки — **Overload Set**) на основе свойств типов.

Исторически этим механизмом был SFINAE, а в C++20 ему на смену пришли Концепты (Concepts).

Проблема жадной перегрузки

Рассмотрим классическую проблему, с которой сталкиваются разработчики контейнеров, подобных `std::vector`. У вектора есть два похожих конструктора: 1. Конструктор заполнения: создает n элементов со значением val. 2. Конструктор диапазона: копирует элементы из диапазона итераторов $[first, last)$.

```cpp
#include <iostream>
#include <vector>

template <typename T>
class Vector {
public:
    // Конструктор 1: Заполнение (Fill Constructor)
    Vector(size_t n, const T& val) {
        std::cout << "Fill Constructor called" << std::endl;
    }

    // Конструктор 2: Диапазон (Range Constructor)
    template <typename InputIter>
    Vector(InputIter first, InputIter last) {
        std::cout << "Range Constructor called" << std::endl;
```

```
        // Представим, что здесь происходит разыменование
        // *first;
    }
};

int main() {
    // Case A: Работает ожидаемо
    Vector<int> v1(5, 10);
    // Аргументы: (int, int).
    // Конструктор 1 ждет (size_t, int). Требуется конверсия int -> size_t.
    // Конструктор 2 ждет (T, T). T выводится как int. Точное совпадение!

    // Компилятор выбирает Конструктор 2.
    // Внутри он пытается сделать *first (разыменовать число 5).
    // ОШИБКА КОМПИЛЯЦИИ: invalid type argument of unary '*'
}
```

В примере выше мы хотели создать вектор из 5 элементов со значением 10. Однако компилятор C++ следует строгим правилам разрешения перегрузки (Overload Resolution).

Анализ ситуации для `Vector<int> v1(5, 10)`:

- **Конструктор 1:** Ожидает `size_t, const int&`. Мы передаем `int, int`. Требуется стандартное преобразование типа (`int` → `size_t`).
- **Конструктор 2:** Шаблонный. Компилятор выводит тип `InputIter = int`. Сигнатура становится `(int, int)`. Это **точное совпадение** (Exact Match).

Точное совпадение всегда приоритетнее преобразования типов. Компилятор выбирает Конструктор 2. Затем он инстанцирует его тело. В теле мы пытаемся обращаться с `int` как с итератором (разыменовываем его). Это приводит к ошибке компиляции внутри тела функции.

Это **Hard Error**. Компиляция прерывается. Нам же нужно, чтобы компилятор, поняв, что `int` не может быть итератором, просто молча проигнорировал этот шаблон и перешел к следующему (Конструктору 1).

SFINAE: Substitution Failure Is Not An Error

Аббревиатура SFINAE (произносится как "сфинэ") расшифровывается как *"Ошибка подстановки не является ошибкой"*. Это правило, зашитое в ядро компилятора C++.

> **Принцип SFINAE**
>
> Если при подстановке выведенных типов в **объявление** (сигнатуру) шаблона функции или класса получается некорректный код, это не приводит к немедленной ошибке компиляции. Вместо этого, данный шаблон просто удаляется из множества кандидатов на перегрузку (Overload Set).

Критически важный нюанс: ошибка должна произойти именно в **заголовке** функции (типы аргументов, возвращаемое значение, параметры шаблона). Если ошибка происходит в **теле** функции (как в примере выше), SFINAE не работает, и мы получаем ошибку компиляции.

Инструмент std::enable_if

Чтобы воспользоваться SFINAE, нам нужно искусственно создать ошибку в заголовке функции, если условие не выполняется. Для этого используется метафункция `std::enable_if`.

```
// Упрощенная реализация enable_if
template <bool B, typename T = void>
struct enable_if {};

// Специализация для true
template <typename T>
struct enable_if<true, T> {
    using type = T;
};
```

Как это работает:

- Если `B == true`, структура имеет вложенный тип `type` (равный `T`).
- Если `B == false`, структура **пуста**. Попытка обратиться к `enable_if<false>::type` вызовет ошибку подстановки.

Применение SFINAE к конструктору

Мы хотим, чтобы шаблонный конструктор существовал только тогда, когда `InputIter` **не** является целочисленным типом.

Стандартный паттерн внедрения `enable_if` в конструкторы (у которых нет возвращаемого значения) — использование дефолтного шаблонного параметра.

```
#include <type_traits>

template <typename T>
class Vector {
public:
    Vector(size_t n, const T& val) { /* ... */ }

    // Этот конструктор будет рассматриваться только если
    // InputIter НЕ является интегральным типом.
    template <typename InputIter,
              typename = std::enable_if_t<!std::is_integral_v<InputIter>>>
    Vector(InputIter first, InputIter last) {
        std::cout << "Range Constructor" << std::endl;
    }
};
```

Разбор механики: 1. Вызов `Vector(5, 10)`. 2. Компилятор пытается инстанцировать шаблонный конструктор. 3. `InputIter` выводится как `int`. 4. Проверяется условие: `!std::is_integral_` Это `false`. 5. Вычисляется `std::enable_if_t<false>`. Такого типа `type` не существует. 6. Происходит ошибка подстановки в параметрах шаблона. 7. Благодаря SFINAE, компилятор **молча отбрасывает** этот конструктор. 8. Единственным оставшимся кандидатом является `Vector(size_t, const T&)`. 9. Аргументы конвертируются, код компилируется корректно.

Detection Idiom и void_t

Часто требуется проверить не просто тип (является ли он числом), а наличие определенного метода или вложенного типа. Например, "есть ли у типа метод `.size()`?".

Для этого используется идиома обнаружения (Detection Idiom) на базе `std::void_t`. `std::void_t<A` — это метафункция, которая всегда превращается в `void`, но требует, чтобы все аргументы `Args...` были валидными типами.

```
// Базовый шаблон: по умолчанию false
template <typename T, typename = void>
struct has_size : std::false_type {};

// Специализация: срабатывает, только если выражение валидно
template <typename T>
struct has_size<T, std::void_t<decltype(std::declval<T>().size())>>
    : std::true_type {};
```

Если у типа `T` есть метод `size()`, специализация становится валидной (второй аргумент `void`). Поскольку специализация более специфична, чем базовый шаблон, компилятор выбирает её, и мы получаем `true_type`. Если метода нет — SFINAE отбрасывает специализацию, и мы падаем в базовый шаблон (`false_type`).

Концепты (Concepts) в C++20

SFINAE — это мощный, но крайне неудобный инструмент. 1. Синтаксис ужасен (угловые скобки, `typename`, `::type`). 2. Сообщения об ошибках нечитаемы. Если ни одна перегрузка не подошла, компилятор вываливает простыню текста о том, почему не сработал `enable_if`. 3. Это "хак" системы типов, а не штатная возможность.

C++20 ввел **Концепты** — прямой способ ограничения шаблонов.

Синтаксис Requires

Вместо `std::enable_if` мы используем ключевое слово `requires` после списка параметров шаблона. Перепишем наш пример с вектором:

```
#include <concepts>

template <typename T>
class Vector {
public:
    Vector(size_t n, const T& val) { /* ... */ }

    // Читаемо и декларативно:
    template <typename InputIter>
    requires (!std::integral<InputIter>) // Ограничение
    Vector(InputIter first, InputIter last) {
        /* ... */
    }
};
```

Более того, стандартная библиотека предоставляет готовый концепт `std::input_iterator`, который проверяет, что тип ведет себя как итератор (поддерживает ++, *, !=).

```
template <std::input_iterator InputIter> // Сокращенный синтаксис
    Vector(InputIter first, InputIter last) { /* ... */ }
```

Если мы теперь попытаемся передать числа в этот конструктор, сообщение об ошибке будет гласить: *"template constraint not satisfied: int is not an input_iterator"*. Это на порядок понятнее, чем ошибки подстановки SFINAE.

Ad-hoc ограничения

Концепты позволяют проверять валидность произвольных выражений прямо по месту, не создавая отдельных структур (как в случае с `void_t`).

```
template <typename T>
requires requires (T x) {
    x.size();          // Должен быть метод size()
    typename T::value_type; // Должен быть вложенный тип value_type
    { x + x } -> std::convertible_to<int>; // Результат сложения приводим к int
}
void process(T obj) {
    // ...
}
```

Выражение `requires (T x) { ... }` создает область видимости, где мы описываем "пробный код". Компилятор проверяет, валиден ли этот код для типа `T`. Этот код никогда не исполняется, только проверяется.

Резюме раздела

- **Overload Resolution** всегда предпочитает точное совпадение типов конверсии. Это опасно для обобщенных конструкторов.
- **SFINAE** позволяет убрать функцию из списка перегрузок, если подстановка типов создает ошибку в сигнатуре.
- **std::enable_if** — основной инструмент SFINAE до C++20. Обычно применяется как дефолтный аргумент шаблона.
- **Concepts (C++20)** полностью заменяют SFINAE. Они делают код чище, а ошибки компиляции — понятными. Всегда предпочитайте `requires` использованию `enable_if` в современном коде.

Глава 35

Архитектура сборки и идиома Pimpl

C++ имеет архаичную модель компиляции, унаследованную от языка C. Программа состоит из набора единиц трансляции (Translation Units), которые компилируются независимо друг от друга, а затем связываются линковщиком. Заголовочные файлы (`.h`) работают через примитивную текстовую подстановку (`#include`).

Эта архитектура порождает проблему физических зависимостей. Любое изменение в заголовочном файле (даже в `private` секции класса) изменяет бинарный интерфейс (ABI) и контрольную сумму файла, вынуждая систему сборки перекомпилировать **все** `.cpp` файлы, которые (прямо или транзитивно) включают этот заголовок. В крупных проектах добавление одного поля `private int x;` в популярный хедер может вызвать "лавину пересборки" (Include Avalanche), занимающую часы.

Идиома Pimpl (Pointer to Implementation) — это архитектурный паттерн, разрывающий эту зависимость путем выноса деталей реализации в отдельный класс, скрытый внутри единицы трансляции.

Анатомия Pimpl

Суть идиомы заключается в замене всех приватных полей класса на единственный непрозрачный указатель (Opaque Pointer) на структуру реализации.

Рассмотрим класс `NetworkClient`, который использует тяжелую библиотеку `<asio.hpp>`.

Без Pimpl (NetworkClient.h):

```
// Проблема: ВЕСЬ asio.hpp попадает к каждому пользователю NetworkClient
#include <asio.hpp>

class NetworkClient {
public:
    void Connect(std::string_view url);
private:
    // Детали реализации "протекают" в интерфейс
    asio::io_context context;
    asio::ip::tcp::socket socket;
};
```

Любой файл, включающий `NetworkClient.h`, будет вынужден парсить тысячи строк `asio.hpp`.

С применением Pimpl:

Мы объявляем структуру `Impl`, но не определяем её в хедере (Forward Declaration).

```cpp
// NetworkClient.h
#include <memory>
#include <string_view>

class NetworkClient {
public:
    NetworkClient();
    ~NetworkClient(); // Важно: деструктор должен быть объявлен!

    void Connect(std::string_view url);

private:
    // Предварительное объявление (Forward Declaration)
    class Impl;

    // Указатель на неполный тип (Incomplete Type)
    std::unique_ptr<Impl> pImpl;
};
```

Теперь хедер ничего не знает о `asio`. Зависимости переносятся в `.cpp` файл.

```cpp
// NetworkClient.cpp
#include "NetworkClient.h"
#include <asio.hpp> // Тяжелый хедер инклюдится только здесь

// Полное определение класса реализации
class NetworkClient::Impl {
public:
    void ConnectInternal(std::string_view url) {
        // Логика работы с asio
    }

    asio::io_context context;
    asio::ip::tcp::socket socket{context};
};

// Конструктор: создаем реализацию
NetworkClient::NetworkClient() : pImpl(std::make_unique<Impl>()) {}

// Деструктор: нужен для unique_ptr (см. далее)
NetworkClient::~NetworkClient() = default;

// Проксирование вызовов
void NetworkClient::Connect(std::string_view url) {
    pImpl->ConnectInternal(url);
}
```

Проблема std::unique_ptr и неполных типов

При использовании сырых указателей (`Impl*`) код выше скомпилировался бы без проблем. Однако ручное управление памятью (`new/delete`) в современном C++ недопустимо. Стандартом де-факто является `std::unique_ptr`.

Здесь возникает тонкий момент, связанный с генерацией деструктора.

Если мы не объявим деструктор `~NetworkClient()` явно в хедере, компилятор попытается сгенерировать его автоматически как `inline` метод. Деструктор `std::unique_ptr<Impl>` вызывает `default_delete<Impl>`, который, в свою очередь, делает `delete ptr`.

Для безопасного вызова `delete` компилятор должен видеть **полное определение типа** `Impl`. Если тип неполный (только forward declaration), оператор `delete` может вызвать Undefined Behavior (если у класса есть нетривиальный деструктор), поэтому `static_assert` внутри `default_delete` выдаст ошибку компиляции: *"invalid application of sizeof to incomplete type"*.

> **Важно!**
>
> Если деструктор генерируется в хедере (автоматически или явно через `= default`), тип `Impl` еще не определен, и компиляция падает.

Решение: 1. Объявить деструктор в хедере: `~NetworkClient();` 2. Определить его в `.cpp` файле, **после** того как определен класс `Impl`: `NetworkClient::~NetworkClient() = default;`

В этой точке (в `.cpp`) тип `Impl` уже является полным (Complete Type), и `unique_ptr` может корректно сгенерировать код удаления.

Fast Pimpl: Избавление от аллокации

Классический Pimpl с `unique_ptr` имеет недостаток производительности: каждый объект требует динамической аллокации памяти (heap allocation). Это создает нагрузку на аллокатор и снижает локальность данных (Pointer Chasing).

Техника **Fast Pimpl** позволяет разместить объект реализации прямо внутри основного объекта (на стеке или внутри его layout), сохраняя инкапсуляцию.

Для этого используется буфер сырой памяти (`std::aligned_storage` или массив `std::byte`), размер и выравнивание которого совпадают с скрытым классом `Impl`.

Реализация Fast Pimpl

В хедере мы резервируем место "вслепую". Нам приходится угадывать размер реализации или фиксировать его константой.

```
// Header
#include <new> // для std::launder и placement new
#include <type_traits>

class FastWidget {
public:
    FastWidget();
    ~FastWidget();
```

```
    void DoWork();

private:
    struct Impl; // Только объявление

    // Константы подбираются экспериментально
    static constexpr size_t ImplSize = 64;
    static constexpr size_t ImplAlign = 8;

    // Сырой буфер памяти
    alignas(ImplAlign) std::byte storage[ImplSize];

    // Вспомогательный метод для каста
    Impl* GetImpl() {
        return reinterpret_cast<Impl*>(&storage);
    }
};
```

В файле реализации мы обязаны проверить, что наши догадки о размере верны, и инициализировать объект через *Placement New*.

```
// CPP file
class FastWidget::Impl {
    int data[10];
    // ... поля реализации
};

// Критическая проверка: если Impl вырастет, компиляция упадет
static_assert(sizeof(FastWidget::Impl) <= sizeof(FastWidget::storage),
              "Storage size is too small for Impl");
static_assert(alignof(FastWidget::Impl) <= alignof(FastWidget::storage),
              "Alignment mismatch");

FastWidget::FastWidget() {
    // Placement new: конструируем объект в буфере storage
    new (&storage) Impl();
}

FastWidget::~FastWidget() {
    // Явный вызов деструктора обязателен!
    GetImpl()->~Impl();
}

void FastWidget::DoWork() {
    GetImpl()->DoWorkInternal();
}
```

Trade-offs Fast Pimpl

- **Преимущество:** Нулевая аллокация в куче. Лучшая локальность кэша (данные лежат рядом).
- **Недостаток 1:** Сложность поддержки. При добавлении полей в `Impl` может сработать

`static_assert`, и придется вручную править константы размера в хедере.

- **Недостаток 2:** Работа с сырой памятью опасна. Забытый вызов деструктора приведет к утечке ресурсов (если `Impl` держит дескрипторы).
- **Недостаток 3:** Strict Aliasing. Использование `reinterpret_cast` требует осторожности. В C++17 для легального доступа к объекту, созданному через placement new, желательно использовать `std::launder`, хотя в данном простом случае доступ через указатель на storage работает на практике.

Резюме раздела

- **Pimpl** разрывает зависимость компиляции, скрывая реализацию за указателем. Это ускоряет сборку и обеспечивает стабильность ABI.
- Использование **std::unique_ptr** с Pimpl требует определения деструктора в `.cpp` файле, чтобы тип реализации был полным.
- **Fast Pimpl** использует буфер на стеке (`alignas`) вместо кучи, убирая оверхед аллокации, но требует ручного управления жизненным циклом (placement new, явный вызов деструктора) и контроля размеров (`static_assert`).

Глава 36

Динамический Полиморфизм: Vtables и RTTI

Полиморфизм — это способность объектов разных типов реагировать на один и тот же вызов метода специфическим для каждого типа образом. В C++ существуют два вида полиморфизма: статический (шаблоны, перегрузка), разрешаемый на этапе компиляции, и динамический (виртуальные функции), разрешаемый во время выполнения (runtime).

Динамический полиморфизм обеспечивает гибкость архитектуры, позволяя работать с объектами через указатель на базовый класс, не зная их реального типа. За эту гибкость приходится платить накладными расходами на вызов (runtime overhead) и память. В этой главе мы разберем низкоуровневую реализацию этого механизма.

Таблица виртуальных методов (vtable)

Ключевое слово `virtual` сообщает компилятору, что связывание вызова функции с ее реализацией должно происходить динамически. Для реализации этого механизма компиляторы (GCC, Clang, MSVC) используют структуру данных, называемую таблицей виртуальных методов (**vtable**).

Модификация раскладки объекта

Как только в классе появляется хотя бы одна виртуальная функция, компилятор неявно добавляет в него скрытое поле — указатель на таблицу виртуальных методов (**vptr**). Обычно `vptr` располагается в самом начале объекта (по смещению 0), чтобы механизм вызова был максимально быстрым.

Рассмотрим иерархию:

```
class Base {
public:
    virtual void func1() { /* Base::func1 */ }
    virtual void func2() { /* Base::func2 */ }
    int x;
};

class Derived : public Base {
public:
```

```
    // Переопределяем func1
    void func1() override { /* Derived::func1 */ }
    // func2 наследуется от Base
    // Добавляем новую виртуальную функцию
    virtual void func3() { /* Derived::func3 */ }
    int y;
};
```

Memory Layout для Base:

- `vptr` → указывает на `Base::vtable`
- `int x`

Содержимое Base::vtable:

- index 0: адрес `Base::func1`
- index 1: адрес `Base::func2`

Memory Layout для Derived:

- `vptr` → указывает на `Derived::vtable`
- `int x` (унаследовано)
- `int y` (собственное)

Содержимое Derived::vtable:

- index 0: адрес `Derived::func1` (переопределена)
- index 1: адрес `Base::func2` (наследуется оригинальная)
- index 2: адрес `Derived::func3` (новая)

Алгоритм диспетчеризации вызова

Когда компилятор встречает вызов `ptr→func1()`, где `ptr` имеет тип `Base*`, он генерирует следующий псевдокод на ассемблере:

1. **Загрузка vptr:** Прочитать значение по адресу, хранящемуся в `ptr` (получаем адрес начала vtable).
2. **Вычисление смещения:** Добавить к адресу vtable смещение, соответствующее индексу метода `func1` (в нашем случае index 0).
3. **Получение адреса функции:** Прочитать адрес из ячейки таблицы.
4. **Вызов (Indirect Call):** Перейти по полученному адресу, передав `ptr` в качестве аргумента `this`.

На заметку

Инициализация `vptr` происходит в конструкторе. Сначала вызывается конструктор `Base`, который устанавливает `vptr` на `Base::vtable`. Затем выполняется тело конструктора `Base`. После этого управление передается конструктору `Derived`, который перезаписывает `vptr` адресом `Derived::vtable`. Именно поэтому вызов виртуальной функции из конструктора никогда не является полиморфным — он вызывает версию текущего конструируемого класса.

Множественное наследование и Pointer Adjustment

Ситуация значительно усложняется при множественном наследовании. Если класс `Derived` наследуется от `Base1` и `Base2`, объект `Derived` должен содержать части обоих родителей, и каждая часть ожидает, что указатель `this` будет указывать на её начало.

```
struct Base1 {
    virtual void method1() {}
    int b1;
};

struct Base2 {
    virtual void method2() {}
    int b2;
};

struct Derived : Base1, Base2 {
    void method1() override {} // override Base1
    void method2() override {} // override Base2
    int d;
};
```

В памяти объект `Derived` будет выглядеть так:

1. **Subobject Base1**:
    - `vptr1` (для Derived-as-Base1)
    - `int b1`
2. **Subobject Base2**:
    - `vptr2` (для Derived-as-Base2)
    - `int b2`
3. **Members Derived**:
    - `int d`

Смещение this (Thunks)

При приведении `Derived*` к `Base2*`, адрес должен измениться. Он должен указывать не на начало всего объекта, а на начало подобъекта `Base2`.

```
Derived* d = new Derived();
Base1* b1 = d; // Адрес совпадает: (void*)b1 == (void*)d
Base2* b2 = d; // Адрес СМЕЩЕН: (void*)b2 == (char*)d + sizeof(Base1)
```

Самое сложное происходит при вызове виртуального метода через `Base2*`. Если мы вызываем `b2→method2()`, который переопределен в `Derived`, функция `Derived::method2` ожидает, что `this` указывает на начало `Derived`, но мы передаем ей указатель на `Base2` (смещенный).

Для решения этой проблемы компилятор генерирует специальную функцию-переходник, называемую **Thunk** (или Trampoline). В vtable для `Base2` лежит не адрес `Derived::method2`, а адрес thunk-а. Этот thunk вычитает смещение из указателя `this` и затем прыгает на реальную функцию `Derived::method2`.

RTTI и dynamic_cast

RTTI (Run-Time Type Information) — это механизм, позволяющий определить реальный тип объекта во время выполнения. Данные RTTI (например, имя типа для `typeid`) обычно хранятся в памяти по указателю, который лежит в специальном слоте vtable (часто по индексу -1).

Оператор dynamic_cast

`dynamic_cast<Target*>(source)` используется для безопасного приведения указателя базового класса к указателю производного класса (Downcast). В отличие от `static_cast`, который просто сдвигает указатель на константу времени компиляции, `dynamic_cast` выполняет проверку в рантайме.

Алгоритм работы: 1. Используя `vptr` объекта `source`, найти RTTI информацию. 2. Проверить, является ли `Target` наследником (или самим типом) реального типа объекта. Это может потребовать обхода дерева наследования. 3. Если приведение возможно, вернуть скорректированный указатель. 4. Если невозможно, вернуть `nullptr` (для указателей) или бросить исключение `std::bad_cast` (для ссылок).

Практический пример: Система событий

Рассмотрим типичный паттерн обработки событий, где полиморфизм используется для стирания типа (Type Erasure) при хранении, а RTTI — для восстановления типа при обработке.

```
#include <string>
#include <iostream>
#include <memory>
#include <vector>

// Интерфейс события.
// Наличие виртуального деструктора обязательно для работы dynamic_cast
// и корректного удаления через базовый указатель.
struct IEvent {
    virtual ~IEvent() = default;
```

```
};

struct ErrorEvent : IEvent {
    std::string message;
};

struct DataEvent : IEvent {
    int payload;
};

// Функция-обработчик не знает конкретных типов на этапе компиляции
void ProcessEvents(const std::vector<std::unique_ptr<IEvent>>& events) {
    for (const auto& event : events) {
        // Попытка интерпретировать событие как ошибку
        if (auto* err = dynamic_cast<ErrorEvent*>(event.get())) {
            std::cout << "[ERROR] " << err->message << std::endl;
            continue;
        }

        // Попытка интерпретировать событие как данные
        if (auto* data = dynamic_cast<DataEvent*>(event.get())) {
            std::cout << "[DATA] Payload: " << data->payload << std::endl;
            continue;
        }

        std::cout << "[INFO] Unknown event type" << std::endl;
    }
}
```

Важно!

`dynamic_cast` работает **только** с полиморфными классами (классами, имеющими хотя бы одну виртуальную функцию). Если в `IEvent` убрать виртуальный деструктор, код не скомпилируется, так как у объекта не будет vtable, а значит, неоткуда взять RTTI.

Чистые виртуальные функции

Если виртуальная функция объявлена с синтаксисом `= 0`, она называется чистой (pure virtual).

```
class Abstract {
public:
    virtual void doWork() = 0;
};
```

Такой класс становится абстрактным: создать его экземпляр невозможно. Однако `vtable` для него все равно может генерироваться (например, для вызова деструктора). В слоты, соответствующие чистым виртуальным функциям, компилятор обычно записывает указатель на служебную функцию-ловушку (например, `__cxa_pure_virtual` в GCC). Если каким-то образом (через UB в конструкторе) вызвать такую функцию, программа аварийно завершится с соответствующим сообщением.

Резюме раздела

- **Vtable** — механизм реализации динамического полиморфизма. Это массив указателей на функции.
- Каждый объект полиморфного класса несет скрытый указатель **vptr**, увеличивающий его размер (обычно на 8 байт).
- Вызов виртуальной функции требует разыменования указателя (индирекции), что может замедлить программу из-за промахов кэша процессора (Branch Prediction Miss).
- При **множественном наследовании** указатель `this` динамически корректируется при приведении типов и вызове методов.
- **dynamic_cast** использует RTTI из vtable для безопасного приведения типов, возвращая `nullptr` при неудаче. Это дорогая операция.

Глава 37

Паттерны проектирования и Идиомы C++

В языке C++ паттерны проектирования выходят за рамки классического объектно-ориентированно программирования (GoF). Благодаря мощной системе шаблонов и детерминированному управлению памятью, многие архитектурные решения реализуются через специфические идиомы (Idioms), недоступные в других языках (Java, C#). В этой главе мы рассмотрим современные реализации порождающих паттернов и статический полиморфизм.

Singleton (Одиночка)

Паттерн Singleton гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к нему. В C++ реализация этого паттерна прошла долгую эволюцию, связанную с проблемами многопоточной инициализации.

Meyers Singleton

До стандарта C++11 безопасная инициализация синглтона в многопоточной среде требовала сложных механизмов блокировок (Double-Checked Locking Pattern), которые часто реализовывались некорректно из-за перестановок инструкций процессором.

Скотт Мейерс предложил элегантное решение, опирающееся на локальные статические переменные.

```
class Database {
public:
    // Удаляем конструкторы копирования и перемещения
    Database(const Database&) = delete;
    Database& operator=(const Database&) = delete;

    // Глобальная точка доступа
    static Database& GetInstance() {
        // "Magic Static"
        static Database instance;
        return instance;
    }

    void Query(const char* sql) { /* ... */ }

private:
```

```
    Database() { /* Тяжелая инициализация подключения */ }
    ~Database() { /* Закрытие соединения */ }
};
```

Механика Magic Statics (Thread-Safe Initialization): Начиная с C++11, стандарт гарантирует: если управление входит в объявление блочной статической переменной (`static` внутри функции), и эта переменная еще не инициализирована, инициализация происходит **потокобезопасно**. Компилятор неявно окружает код инициализации блокировками (обычно через `std::call_once` или атомарные флаги). Если другой поток попытается выполнить `GetInstance()` в момент инициализации, он будет заблокирован до её завершения.

Singleton через CRTP

Чтобы не дублировать код метода `GetInstance` в каждом классе-одиночке, можно использовать CRTP (см. далее) для создания универсального базового класса.

```
template <typename T>
class Singleton {
public:
    static T& GetInstance() {
        static T instance;
        return instance;
    }
    // ... delete copy/move ...
protected:
    Singleton() = default;
    ~Singleton() = default;
};

// Использование:
class Logger : public Singleton<Logger> {
    // friend нужен, чтобы Singleton мог вызвать приватный конструктор Logger
    friend class Singleton<Logger>;
private:
    Logger() { /* ... */ }
};
```

Фабрики в эпоху Smart Pointers

Классические фабрики возвращают сырые указатели. В современном C++ это считается плохой практикой (Ownership Semantics неясна). Стандартом индустрии является возврат `std::unique_ptr`.

```
struct IFruit { virtual ~IFruit() = default; };
struct Apple : IFruit {};
struct Orange : IFruit {};

// Абстрактная фабрика
class FruitFactory {
```

```
public:
    // Возвращаем unique_ptr - передаем владение вызывающему
    static std::unique_ptr<IFruit> Create(std::string_view type) {
        if (type == "apple") return std::make_unique<Apple>();
        if (type == "orange") return std::make_unique<Orange>();
        return nullptr;
    }
};
```

Такой подход гарантирует, что созданный объект будет корректно удален, даже если клиент забудет про него или произойдет исключение.

CRTP: Статический Полиморфизм

Curiously Recurring Template Pattern (CRTP) — идиома, в которой класс `Derived` наследуется от шаблона класса `Base`, параметризованного самим `Derived`.

```
template <typename Derived>
class Base {
    // ...
};

class MyClass : public Base<MyClass> {
    // ...
};
```

Главная цель CRTP — достижение полиморфного поведения без использования виртуальных функций (Static Polymorphism). Базовый класс знает тип наследника на этапе компиляции и может приводить указатель `this` к типу `Derived*`.

```
template <typename Derived>
class BaseAPI {
public:
    void Interface() {
        // Статическое приведение к наследнику.
        // Безопасно, так как мы знаем, что this - это часть объекта Derived.
        static_cast<Derived*>(this)->Implementation();
    }

    // Дефолтная реализация (Compile-time check)
    void Implementation() {
        // Если наследник не переопределил метод, вызовется этот код
    }
};

class Service : public BaseAPI<Service> {
public:
    // "Переопределение" метода (без virtual)
    void Implementation() {
        /* Custom Logic */
    }
```

```
};
```

Преимущества перед `virtual`:

- **Отсутствие vtable:** Экономия памяти (нет vptr) и отсутствие лишней индирекции при вызове.
- **Инлайнинг:** Компилятор видит тело вызываемой функции наследника и может встроить его (Devirtualization), что критично для высоконагруженных циклов.

Пример: Полиморфное клонирование

В классическом ООП для создания копии объекта через базовый указатель требуется виртуальный метод `clone()`. Это приводит к дублированию шаблонного кода в каждом наследнике.

```
struct Shape {
    virtual std::unique_ptr<Shape> clone() const = 0;
    virtual ~Shape() = default;
};

// CRTP Mixin для автоматической реализации clone
template <typename Derived>
struct Cloneable : Shape {
    std::unique_ptr<Shape> clone() const override {
        // Копируем конкретный тип Derived
        return std::make_unique<Derived>(static_cast<const Derived&>(*this));
    }
};

struct Circle : Cloneable<Circle> {
    int radius;
};

struct Square : Cloneable<Square> {
    int side;
};
// Circle и Square автоматически получили корректную реализацию clone()
```

Policy-Based Design

Policy-Based Design (проектирование на основе стратегий) — это подход, популяризированный Андреем Александреску, позволяющий собирать сложные классы из независимых ортогональных поведений (политик) на этапе компиляции.

Вместо того чтобы жестко прописывать поведение внутри класса или использовать паттерн "Стратегия" с виртуальными функциями, мы передаем классы-повеления (Policies) как параметры шаблона.

Реализация умного указателя с политиками

Рассмотрим класс `SmartPtr`, поведение которого (проверка на `nullptr` при доступе) настраивается извне.

```
// Политика 1: Без проверок (для максимальной скорости)
struct NoCheck {
    template <typename T>
    static void Check(T* ptr) {} // Пусто
};

// Политика 2: Строгая проверка
struct EnforceNotNull {
    template <typename T>
    static void Check(T* ptr) {
        if (!ptr) throw std::runtime_error("Null pointer access");
    }
};

// Хост-класс
template <typename T, typename CheckingPolicy>
class SmartPtr : private CheckingPolicy { // Наследование для EBO
    T* ptr = nullptr;
public:
    T* operator->() {
        // Вызов метода политики
        CheckingPolicy::Check(ptr);
        return ptr;
    }
    // ... конструкторы ...
};

// Использование
using FastPtr = SmartPtr<int, NoCheck>;
using SafePtr = SmartPtr<int, EnforceNotNull>;
```

Этот подход позволяет генерировать код, идеально оптимизированный под конкретную задачу. `FastPtr::operator->` скомпилируется в одну инструкцию (чтение адреса), так как пустая функция `NoCheck::Check` будет удалена оптимизатором. `SafePtr` же будет содержать инструкции проверки.

Резюме раздела

- **Singleton** в современном C++ реализуется через локальную статическую переменную (Meyers Singleton), что гарантирует потокобезопасность без мьютексов в пользовательском коде.
- **CRTP** позволяет реализовать статический полиморфизм, заменяя runtime-оверхед виртуальных функций на compile-time разрешение типов.
- **Policy-Based Design** дает возможность создавать гибкие, конфигурируемые компоненты, комбинируя небольшие классы-стратегии через шаблоны.

Глава 38

Case Study: Разработка Интерпретатора Scheme

Разработка интерпретатора функционального языка (диалекта Lisp) является классической задачей, объединяющей все ключевые темы данного курса: полиморфизм, работу с динамической памятью, парсинг и алгоритмы обхода графов.

В этой главе мы спроектируем архитектуру интерпретатора языка Scheme, пройдя путь от токенизации до реализации собственного сборщика мусора (Garbage Collector), необходимого для решения проблемы циклических ссылок, с которой не справляются стандартные умные указатели C++.

Архитектура конвейера (Pipeline)

Процесс интерпретации разделяется на четыре изолированные стадии:

1. **Tokenizer (Lexer):** Преобразует сырой поток символов (`std::istream`) в поток атомарных лексем (токенов). На этом этапе отбрасываются комментарии и пробельные символы, а последовательности цифр группируются в числа.
2. **Parser:** Выполняет синтаксический анализ потока токенов и строит абстрактное синтаксическое дерево (AST). Для Lisp-подобных языков AST совпадает со структурой данных языка (S-expressions).
3. **Evaluation:** Рекурсивный обход AST с вычислением результатов. Здесь реализуется арифметика, вызовы функций и специальные формы (`if`, `define`, `lambda`).
4. **Memory Management:** Сквозной слой, отвечающий за время жизни объектов AST.

Представление данных и AST

В языке Scheme <<код есть данные>> (homoiconicity). Базовым строительным блоком является **Cons Cell** (пара) — структура, содержащая два указателя: `car` (голова) и `cdr` (хвост). Списки строятся как цепочки вложенных пар.

Для представления динамически типизированных объектов Scheme в статически типизированном C++ мы используем полиморфную иерархию классов.

```cpp
#include <memory>
#include <string>
#include <vector>

// Базовый класс для всех объектов Scheme
struct Object {
    virtual ~Object() = default;

    // Поддержка сборщика мусора (см. далее)
    bool marked = false;
};

// Числовой литерал
struct Number : Object {
    int value;
    Number(int v) : value(v) {}
};

// Символ (идентификатор переменной)
struct Symbol : Object {
    std::string name;
    Symbol(const std::string& n) : name(n) {}
};

// Пара (Cons Cell).
// Основной структурный элемент списков и деревьев.
struct Cell : Object {
    std::shared_ptr<Object> first;
    std::shared_ptr<Object> second;

    Cell(std::shared_ptr<Object> f, std::shared_ptr<Object> s)
        : first(f), second(s) {}
};
```

Использование `std::shared_ptr<Object>` кажется естественным выбором. Объекты в Lisp передаются по ссылке, могут использоваться в нескольких местах одновременно (structure sharing), и должны удаляться, когда на них больше никто не ссылается. Механизм подсчета ссылок (Reference Counting) идеально подходит... пока не появляются циклы.

Проблема циклических ссылок

Модель владения `shared_ptr` гарантирует удаление объекта только тогда, когда счетчик ссылок (refcount) достигает нуля. В функциональном программировании возможно создание структур, ссылающихся сами на себя.

Рассмотрим классический пример создания цикла через мутацию хвоста списка (функция `set-cdr!`):

```cpp
// Scheme:
// (define x (list 1 2)) ; x -> (1 . (2 . null))
// (set-cdr! (cdr x) x)  ; хвост списка теперь указывает на начало
```

На уровне C++ это выглядит так:

```
void CreateCycle() {
    auto n1 = std::make_shared<Number>(1);
    auto n2 = std::make_shared<Number>(2);

    // Создаем список (1 2)
    auto cell2 = std::make_shared<Cell>(n2, nullptr);
    auto cell1 = std::make_shared<Cell>(n1, cell2); // cell1 -> cell2

    // Замыкаем цикл: cell2 -> cell1
    // Теперь refcount у обоих объектов равен 2 (1 внешний + 1 внутренний)
    cell2->second = cell1;

    // При выходе из функции внешние указатели (cell1, cell2) уничтожаются.
    // Refcount обоих объектов уменьшается до 1.
    // Память НЕ освобождается. УТЕЧКА.
}
```

В языках без ручного управления памятью (как Scheme) программист не должен думать о разрыве циклов (в отличие от использования `weak_ptr` в C++). Единственным решением является реализация полноценного сборщика мусора.

Реализация Garbage Collector (Mark-and-Sweep)

Алгоритм Mark-and-Sweep (Пометь и Вымети) является классическим подходом к GC. Он не полагается на счетчики ссылок, а определяет <<живучесть>> объектов на основе их достижимости из корневого набора (Roots).

Архитектура меняется: 1. Мы отказываемся от `std::shared_ptr` в полях `Cell`. Используем сырые указатели `Object*`. 2. Создаем класс `Heap` (Куча), который владеет всеми созданными объектами (через `std::vector<std::unique_ptr<Object>>` или свой аллокатор).

Алгоритм

Фаза 1: Mark (Разметка) Обходим граф объектов, начиная с корней (переменные на стеке, глобальное окружение). Помечаем каждый посещенный объект флагом `marked = true`.

```
void Heap::Mark(Object* obj) {
    if (!obj || obj->marked) return;

    obj->marked = true;

    // Если это пара, рекурсивно помечаем детей
    if (auto cell = dynamic_cast<Cell*>(obj)) {
        Mark(cell->first);
        Mark(cell->second);
    }
}
```

Фаза 2: Sweep (Выметание) Проходим по всем объектам, зарегистрированным в куче.

- Если `marked == true`: объект достижим. Сбрасываем флаг в `false` (для следующего цикла GC).

- Если `marked == false`: объект недостижим (мусор). Удаляем его.

```
void Heap::Sweep() {
    auto it = allocated_objects.begin();
    while (it != allocated_objects.end()) {
        Object* obj = *it;
        if (obj->marked) {
            obj->marked = false; // Сброс для следующего GC
            ++it;
        } else {
            delete obj; // Освобождение памяти
            // Удаление из списка живых объектов (swap-and-pop для O(1))
            *it = allocated_objects.back();
            allocated_objects.pop_back();
        }
    }
}
```

Этот подход корректно обрабатывает любые циклы, так как недостижимый циклический граф просто не будет помечен в фазе Mark и будет целиком удален в фазе Sweep.

Синтаксический анализ: Recursive Descent

Парсинг Lisp-выражений упрощается благодаря их префиксной скобочной структуре. Метод рекурсивного спуска идеально ложится на грамматику.

Грамматика (упрощенно):

- *Expression* $\rightarrow$ *Atom* | *List*
- *List* $\rightarrow$ '(' *Elements* ')'
- *Elements* $\rightarrow$ *Expression Elements* | ϵ

Реализация парсера:

```
Object* Read(Tokenizer& tokenizer) {
    Token token = tokenizer.Next();

    if (token.type == Token::NUMBER) {
        return new Number(token.int_val);
    }

    if (token.type == Token::OPEN_PAREN) {
        return ReadList(tokenizer);
    }

    if (token.type == Token::SYMBOL) {
        return new Symbol(token.str_val);
    }

    throw SyntaxError("Unexpected token");
}

```

```
Object* ReadList(Tokenizer& tokenizer) {
    Token peek = tokenizer.Peek();
    if (peek.type == Token::CLOSE_PAREN) {
        tokenizer.Next(); // Поглощаем ')'
        return nullptr;   // Пустой список
    }

    Object* head = Read(tokenizer);
    Object* tail = ReadList(tokenizer);

    return new Cell(head, tail);
}
```

Функция `ReadList` рекурсивно вызывает `Read` для считывания головы списка, а затем саму себя для считывания хвоста, автоматически формируя цепочку `Cell`-ов.

Резюме раздела

- Интерпретатор Scheme демонстрирует необходимость использования полиморфизма для реализации динамической типизации в C++.
- **std::shared_ptr** непригоден для графов объектов с циклами без ручного разрыва связей.
- **Mark-and-Sweep** — фундаментальный алгоритм сборки мусора, решающий проблему циклов через анализ достижимости графа.
- Парсинг S-выражений тривиально реализуется методом рекурсивного спуска, отражая рекурсивную природу самого списка.

Часть VIII

Лекция 08 – Baby thread

Глава 39

Аппаратные основы многопоточности и роль ОС

Современное программирование на C++ невозможно рассматривать в отрыве от аппаратной архитектуры. Производительность кода, особенно в многопоточных приложениях, определяется не столько сложностью алгоритма, сколько эффективностью использования иерархии памяти и конвейера процессора. В этой главе мы рассмотрим эволюцию вычислительных систем от одноядерных процессоров до современных многоядерных архитектур с NUMA, а также разберем фундаментальные физические ограничения, известные как Memory Wall.

Эволюция архитектуры процессоров

От одноядерной парадигмы к CMP

На протяжении десятилетий доминировала парадигма последовательного исполнения кода. Программисты писали инструкции, полагая, что они будут выполнены процессором одна за другой в строгом порядке. Рост производительности обеспечивался увеличением тактовой частоты и усложнением микроархитектуры (суперскалярность, предсказание переходов).

В начале 2000-х годов этот подход достиг физического предела. Дальнейшее повышение частоты приводило к экспоненциальному росту тепловыделения и энергопотребления. Решением стал переход от наращивания частоты одного ядра к увеличению количества ядер на кристалле (Chip Multiprocessors, CMP).

> **SMP и CMP**
>
> **SMP (Symmetric Multiprocessing)** — архитектура, в которой два и более одинаковых процессора подключены к общей памяти.
> **CMP (Chip Multiprocessor)** — реализация SMP, где несколько ядер размещены на одном кристалле кремния. В 2005 году появились первые массовые многоядерные процессоры.

Переход к многоядерности изменил парадигму разработки программного обеспечения. "Бесплатное" ускорение программ с выходом нового процессора прекратилось. Теперь для утилизации вычислительной мощности требуется явное распараллеливание задач.

Проблема Memory Wall

Фундаментальным ограничением производительности современных систем является диспропорция между скоростью процессора и скоростью доступа к оперативной памяти. Этот феномен получил название **Memory Wall**.

Физические ограничения и латентность

За последние 30 лет производительность процессоров выросла на порядки, в то время как латентность (задержка доступа) памяти осталась практически неизменной.

> **Важно!**
>
> **Ключевые показатели латентности:**
>
> - **1 такт CPU (3-4 ГГц):** $\approx 0.25 - 0.3$ нс.
> - **Доступ в L1 Cache:** ≈ 1 нс (3-4 такта).
> - **Доступ в RAM:** $\approx 70 - 100$ нс.

Это означает, что один промах мимо кэша (cache miss) стоит процессору сотен тактов простоя. Если данные находятся в оперативной памяти, процессор вынужден ждать их получения, не выполняя полезной работы.

Физическая причина этого ограничения кроется в скорости света. Сигнал в вакууме проходит 30 см за 1 нс. В среде (медь, кремний) скорость ниже. Учитывая размеры кристалла и расстояние до модулей памяти (DIMM), сделать память одновременно большой и быстрой физически невозможно. Большая память требует длинных шин адреса и данных, что увеличивает задержку.

Иерархия кэшей

Для смягчения эффекта Memory Wall используется многоуровневая иерархия кэш-памяти.

- **L1 Cache (Level 1):** Самый маленький (обычно 32-64 КБ), расположен непосредственно в ядре. Разделен на кэш инструкций (L1i) и кэш данных (L1d). Латентность: единицы тактов.
- **L2 Cache (Level 2):** Больше по объему (256 КБ - 1 МБ), чуть медленнее. Обычно является приватным для ядра.
- **L3 Cache (Level 3):** Большой (десятки МБ), общий для всех ядер на кристалле (Last Level Cache, LLC). Латентность: десятки тактов.

Задача кэша — хранить данные, к которым процессор обращался недавно (временная локальность) или которые находятся рядом с ними (пространственная локальность). Когда данные загружаются из памяти, они загружаются не побайтово, а целыми блоками — **кэш-линиями** (cache lines), обычно размером 64 байта.

> **На заметку**
>
> Эффективное программирование на C++ подразумевает работу с данными, которые хорошо укладываются в кэш. Последовательный доступ к массиву (vector) на порядки быстрее, чем доступ к связному списку (list), узлы которого разбросаны по куче.

Hyper-threading (SMT)

В попытках скрыть латентность памяти и максимально загрузить исполнительные устройства (ALU, FPU) была разработана технология SMT (Simultaneous Multithreading), наиболее известная в реализации Intel как Hyper-threading.

Концепция виртуальных ядер

Идея SMT заключается в том, что одно физическое ядро представляется операционной системе как два (или более) логических процессора. Ядро дублирует архитектурное состояние (регистры, счетчик команд), но исполнительные устройства (ALU, FPU, L1 кэш) остаются общими.

> **Важно!**
>
> **Hyper-threading не удваивает производительность!** Два потока на одном ядре конкурируют за одни и те же вычислительные блоки.

Механизм эффективен в сценариях, когда один поток ожидает данные из памяти (memory stall). В этот момент ядро может переключиться на исполнение инструкций второго потока, загружая простаивающие ALU. Если же оба потока выполняют интенсивные вычисления (compute-bound) и не ждут памяти, выигрыш от SMT будет минимальным или даже отрицательным из-за конкуренции за кэш.

Для операционной системы и программиста это выглядит как наличие $2N$ ядер, где N — число физических ядер. Например, 8 физических ядер видны как 16 логических потоков.

Роль Операционной Системы

Операционная система (ОС) выступает абстракцией над аппаратным обеспечением, предоставляя программам иллюзию монопольного владения процессором.

Планирование и Квант Времени

ОС использует **вытесняющую многозадачность** (preemptive multitasking). Каждому активному потоку выделяется интервал времени — **квант** (time slice), в течение которого он исполняется на процессоре.

- Длительность кванта в Linux (CFS scheduler) варьируется, но порядок величины — 10-100 мс.
- Это значение огромно по меркам процессора (100 мс $= 10^8$ нс $\approx 4 \cdot 10^8$ тактов).
- Человеческий глаз не замечает переключений с такой частотой, создавая иллюзию параллельной работы множества приложений.

Когда квант истекает или поток блокируется (например, на вводе-выводе), происходит аппаратное прерывание таймера, и управление передается планировщику ОС.

Переключение контекста (Context Switch)

Процедура смены исполняемого потока называется переключением контекста.

1. Сохраняется состояние текущего потока: значения регистров общего назначения (RIP, RSP, RAX и др.), флагов, регистров FPU/AVX.
2. Выбирается следующий поток из очереди готовых к исполнению (runqueue).
3. Восстанавливается состояние нового потока (загрузка регистров).

Объем сохраняемых данных невелик (сотни байт), и сама операция быстрая. Однако косвенная стоимость переключения контекста может быть высокой из-за "остывания" кэша: новый поток, скорее всего, будет работать с другими данными, что приведет к серии кэш-промахов.

Миграция потоков и Processor Affinity

Планировщик ОС старается удерживать поток на одном и том же ядре (Processor Affinity), чтобы сохранить "прогретый" кэш (L1/L2). Миграция потока на другое ядро — дорогая операция, так как данные придется загружать в кэш нового ядра заново (из L3 или RAM).

NUMA (Non-Uniform Memory Access)

В многопроцессорных системах (серверах с несколькими сокетами) используется архитектура NUMA. Память физически разделена между процессорами.

- У каждого процессора есть "своя" (локальная) память, подключенная напрямую. Доступ к ней быстрый.
- Доступ к памяти другого процессора (удаленная память) происходит через межпроцессорную шину (например, Intel QPI/UPI) и является более медленным.

Игнорирование NUMA-топологии в высокопроизводительных приложениях может привести к существенной деградации производительности из-за трафика по межпроцессорной шине и повышенной латентности.

Резюме раздела

- Производительность упирается в память (Memory Wall). Доступ к RAM стоит сотни тактов.
- Иерархия кэшей критически важна. Локальность данных определяет скорость программы.
- Hyper-threading позволяет скрыть задержки памяти, утилизируя простаивающие такты ядра.
- ОС переключает потоки раз в квант времени (десятки мс), сохраняя регистровый контекст.
- Миграция потоков между ядрами нежелательна из-за потери горячего кэша.

Глава 40

Управление потоками в C++: API и стоимость абстракций

Понимание аппаратной части многопоточности, рассмотренное в предыдущей главе, позволяет нам перейти к программным абстракциям. Стандарт C++11 ввел в язык нативную поддержку потоков, предоставив класс `std::thread` как обертку над системными вызовами (pthreads в Linux, WinAPI Threads в Windows). В этой главе мы разберем не только синтаксис, но и цену, которую приходится платить за создание потоков, а также распространенные архитектурные ошибки, приводящие к деградации производительности.

Базовый API: std::thread

Класс `std::thread` (заголовочный файл `<thread>`) представляет собой объект, владеющий системным потоком исполнения. Конструктор `std::thread` принимает функцию (или любой Callable объект), которую поток начнет выполнять сразу после создания.

Запуск и передача аргументов

При создании потока аргументы передаются в конструктор. Здесь действует семантика, схожая с `std::bind`: аргументы копируются во внутреннее хранилище потока (обычно в кучу или стек нового потока).

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

void worker(int id, const std::string& data) {
    // std::this_thread::get_id() возвращает уникальный идентификатор
    std::cout << "Thread " << id
              << " (" << std::this_thread::get_id() << ") processing: "
              << data << std::endl;
}

int main() {
    int id = 1;
    std::string message = "Hello";

```

```
    // message будет скопирован!
    std::thread t1(worker, id, message);

    // Если нужно передать ссылку, используем std::ref / std::cref
    std::thread t2(worker, 2, std::cref(message));

    t1.join();
    t2.join();
    return 0;
}
```

Важно!

По умолчанию аргументы копируются. Если передаваемый объект тяжелый или не копируемый (например, `std::unique_ptr`), его необходимо перемещать с помощью `std::move`, а если функция ожидает ссылку — явно оборачивать в `std::ref`.

Проблема времени жизни: join() и detach()

После запуска потока основной поток (родитель) обязан определить судьбу дочернего потока до того, как объект `std::thread` будет уничтожен (выдет из области видимости).

- **join()**: Родитель блокируется и ждет завершения дочернего потока. Это точка синхронизации. После возврата из `join()` поток считается завершенным, а ресурсы ОС очищенными.
- **detach()**: Поток "отпускается" в свободное плавание. Связь с объектом `std::thread` разрывается. Поток продолжит работать в фоне, пока не завершится сам или пока не завершится программа.

Ловушка деструктора std::thread

Одним из самых спорных решений в дизайне C++11 является поведение деструктора `std::thread`. Если на момент вызова деструктора поток все еще "присоединен" (joinable) — то есть для него не был вызван ни `join()`, ни `detach()` — программа аварийно завершается.

```
void dangerous_code() {
    std::thread t([]{
        std::this_thread::sleep_for(std::chrono::seconds(1));
    });

    // Забыли t.join()!
    // При выходе из функции вызывается ~thread().
    // Так как t.joinable() == true, вызывается std::terminate().
}
```

Это сделано намеренно. Если бы деструктор делал неявный `join()`, это могло бы привести к неочевидным зависаниям программы (деструктор ждет вечно). Если бы он делал `detach()`, поток продолжил бы обращаться к локальным переменным уже уничтоженного стека родительской функции, что привело бы к UB. Комитет стандартизации выбрал "fail fast" — немедленное падение, сигнализирующее об ошибке в логике.

RAII для потоков: std::jthread

В стандарте C++20 был добавлен класс `std::jthread` (joining thread), который исправляет неудобство обычного `std::thread`. Он реализует идиому RAII: в своем деструкторе он автоматически вызывает `join()` (если поток еще работает), либо ничего не делает (если поток уже завершен).

```
#include <thread>

void safe_code() {
    // std::jthread автоматически присоединится при выходе из scope
    std::jthread t([]{
        // ... работа ...
    });
} // Здесь будет вызван t.join()
```

Кроме того, `std::jthread` поддерживает механизм токенов отмены (`std::stop_token`), позволяя вежливо попросить поток завершиться, но это тема отдельного разговора.

Стоимость создания потока

Частой ошибкой новичков является отношение к потокам как к "легковесным" сущностям (по аналогии с горутинами в Go или файберами). В C++ поток `std::thread` обычно отображается 1:1 на поток операционной системы (kernel thread).

Создание потока — дорогая операция. Рассмотрим, что происходит "под капотом" в Linux (системный вызов `clone`):

1. **Аллокация стека:** По умолчанию размер стека составляет около 2-8 МБ (зависит от `ulimit -s`). Даже если физическая память выделяется лениво (по мере обращения к страницам), это все равно нагрузка на менеджер виртуальной памяти (VMA).
2. **Структуры ядра:** Создается запись в таблице процессов (`task_struct`), выделяются дескрипторы.
3. **Планировщик:** Новый поток нужно добавить в очередь планировщика.

Время создания потока на современном железе измеряется **десятками микросекунд** (на порядок больше, чем просто вызов функции).

Бенчмарк: Создание потоков в цикле

Рассмотрим пример неэффективного кода, который создает новый поток для каждой микрозадачи (вычисление корня).

```
// Плохой паттерн: создание потока на каждую итерацию
void SlowSum() {
    double sum = 0;
    for (int i = 0; i < 100000; ++i) {
        std::thread t([i, &sum] {
            sum += std::sqrt(i);
        });
```

```
        t.join(); // Ждем завершения сразу же
    }
}
```

Запуск такого кода через профилировщик (`perf` или `strace -T`) покажет, что программа проводит больше времени в системных вызовах `clone` и `mmap` (выделение памяти под стек), чем в полезных вычислениях.

> **Важно!**
>
> Если ваша задача выполняется быстрее, чем время создания потока (10-50 мкс), распараллеливание через создание новых потоков **замедлит** программу.

Проблема Oversubscription

Другая крайность — запуск слишком большого количества потоков одновременно. Если количество активных (выполняющих работу) потоков значительно превышает количество физических ядер (`std::thread::hardware_concurrency()`), возникает явление **Oversubscription**.

Операционная система вынуждена часто переключать контекст между потоками, чтобы дать каждому квант времени. Как мы обсуждали в Главе 1, переключение контекста портит кэши. В предельном случае (Context Switch Storm) процессор тратит почти все время на переключение задач, а не на их выполнение.

Ловушка std::async

В C++11 была предпринята попытка сделать высокоуровневую абстракцию для асинхронных вычислений — `std::async`. Она возвращает объект `std::future`, через который можно получить результат.

Однако реализации стандартной библиотеки (в частности, libstdc++ в GCC и Clang) часто реализуют политику запуска `std::launch::async` самым примитивным способом: **всегда создается новый поток**.

```
// Опасно: может положить систему
std::vector<std::future<double>> futures;
for (int i = 0; i < 100000; ++i) {
    // Каждая итерация порождает новый поток ОС!
    futures.push_back(std::async(std::launch::async, [i]{
        return std::sqrt(i);
    }));
}
```

Этот код может привести к исчерпанию ресурсов (ошибка `std::system_error` или `Resource temporarily unavailable`), так как он попытается создать 100 000 потоков одновременно. В MSVC есть внутренний глобальный тред-пул, но полагаться на это в кроссплатформенном коде нельзя.

Решение: Thread Pool

Чтобы избежать накладных расходов на создание потоков и проблем с oversubscription, в промышленном коде используется паттерн **Thread Pool** (Пул потоков).

Идея проста:

1. При старте программы создается фиксированное количество потоков (обычно равное числу ядер).
2. Потоки крутятся в бесконечном цикле, ожидая задачи из потокобезопасной очереди.
3. Основной поток не создает новые потоки, а кладет задачи (функции/лямбды) в эту очередь.

Это позволяет платить цену за создание потоков только один раз — при инициализации. Мы разберем реализацию очереди и пула в следующих главах, когда изучим примитивы синхронизации.

Резюме раздела

- Используйте `std::jthread` (C++20) или убедитесь, что всегда вызываете `join()` для `std::thread`.
- Создание потока — тяжелая операция (аллокация стека, syscall). Не создавайте потоки в горячих циклах.
- Избегайте `std::async` с политикой `launch::async` в циклах, так как это неявное создание потоков.
- Оптимальное число рабочих потоков для вычислений равно числу ядер процессора.

Глава 41

Гонки данных и модель памяти

Введение многопоточности в программу фундаментально меняет модель её исполнения. Если в однопоточном коде поведение строго детерминировано (инструкции выполняются последовательно), то в многопоточном окружении порядок исполнения инструкций разных потоков относительно друг друга не определен. Это приводит к двум классам проблем: состоянию гонки (Race Condition) и гонке данных (Data Race). Хотя эти термины часто используют как синонимы, в C++ между ними существует принципиальная разница: первое — это логическая ошибка, второе — неопределенное поведение (Undefined Behavior).

Race Condition vs Data Race

Для начала разграничим понятия.

> **Race Condition (Состояние гонки)**
>
> Ситуация, когда результат работы программы (или её состояние) зависит от того, в каком порядке планировщик ОС решит исполнять потоки. Это семантическая ошибка. Программа технически корректна с точки зрения языка, но делает не то, что ожидал программист.

> **Data Race (Гонка данных)**
>
> Ситуация, когда два и более потока одновременно обращаются к одной и той же ячейке памяти, при этом:
>
> 1. Минимум один из потоков выполняет запись (модификацию).
> 2. Между обращениями нет синхронизации (atomic, mutex).
>
> В стандарте C++ это строго классифицируется как **Undefined Behavior**.

Пример 1: Вывод в консоль (Race Condition)

Рассмотрим классический пример, с которого начинают изучение потоков.

```
#include <thread>
#include <iostream>

```

```
void print_hello(int id) {
    // Оператор << вызывается цепочкой
    std::cout << "Thread " << id << " says hello!" << std::endl;
}

int main() {
    std::thread t1(print_hello, 1);
    std::thread t2(print_hello, 2);
    t1.join();
    t2.join();
}
```

При запуске этой программы вы можете получить корректный вывод, а можете увидеть что-то вроде: `Thread 1 says Thread 2 says hello!hello!`

Это **Race Condition**. Стандартная библиотека C++ гарантирует, что глобальные объекты вроде `std::cout` потокобезопасны в том смысле, что их внутреннее состояние не будет разрушено (программа не упадет). Однако атомарность гарантируется только на уровне одного вызова функции (одного оператора <<).

Выражение `std::cout << "A" << "B"` — это два последовательных вызова функции. Планировщик может прервать первый поток после вывода "A", переключиться на второй, который выведет свои данные, и только потом вернуться к первому.

Решение: std::osyncstream

В C++20 появился синхронизируемый поток вывода `std::osyncstream` (заголовок `<syncstream>`). Он накапливает вывод во внутреннем буфере и атомарно сбрасывает его в выходной поток при уничтожении или вызове `emit()`.

```
#include <syncstream>

void safe_print(int id) {
    std::osyncstream(std::cout) << "Thread " << id << " says hello!\n";
    // При выходе из выражения временный объект osyncstream разрушается
    // и атомарно пишет в cout.
}
```

Пример 2: Инкремент (Data Race)

Теперь рассмотрим ситуацию, которая является фатальной ошибкой. Пусть у нас есть глобальный счетчик, который инкрементируют несколько потоков.

```
#include <thread>
#include <vector>
#include <iostream>

int counter = 0; // Разделяемый ресурс

void worker() {
```

```
    for (int i = 0; i < 100000; ++i) {
        counter++; // Data Race!
    }
}

int main() {
    std::thread t1(worker);
    std::thread t2(worker);
    t1.join();
    t2.join();
    std::cout << "Result: " << counter << std::endl;
}
```

Ожидаемый результат: 200 000. Реальный результат: случайное число, например, 142 583. Или 200 000 (если повезет).

Анатомия гонки

Операция инкремента `counter++` не является атомарной. На уровне процессора (x86) она раскладывается на три этапа (Read-Modify-Write):

1. `MOV EAX, [addr]` — Загрузить значение из памяти в регистр.
2. `INC EAX` — Увеличить значение в регистре.
3. `MOV [addr], EAX` — Записать значение обратно в память.

Если два потока выполняют эти инструкции одновременно, происходит следующее:

1. Поток 1 читает 0 в свой регистр.
2. Поток 2 читает 0 в свой регистр (так как Поток 1 еще не записал результат).
3. Поток 1 увеличивает 0 до 1 и пишет 1 в память.
4. Поток 2 увеличивает 0 до 1 и пишет 1 в память.

Два инкремента превратились в один. Потеря данных произошла.

Почему это Undefined Behavior?

Может показаться, что проблема только в "потере" инкрементов. Но стандарт C++ объявляет это UB, что дает компилятору право на агрессивные оптимизации.

Компилятор видит, что переменная `counter` не является атомарной. Согласно модели памяти C++ (которая подразумевает single-threaded execution, если нет явной синхронизации), он может предположить, что переменную никто другой не меняет. Он может загрузить `counter` в регистр в начале цикла, прибавить к нему 100 000, и записать обратно только в самом конце. Или вообще выкинуть промежуточные записи.

В многопоточной среде такие оптимизации ломают логику программы полностью. Именно поэтому использование обычных переменных (`int`, `bool`) для синхронизации без мьютексов или атомиков — это всегда ошибка.

ThreadSanitizer (TSan)

Поиск гонок данных вручную крайне сложен, так как они проявляются недетерминированно. Ошибка может не воспроизводиться на машине разработчика, но "стрелять" в продакшене под нагрузкой.

Для автоматического детектирования используется инструмент **ThreadSanitizer (TSan)**. Это модуль для компиляторов Clang и GCC.

Для использования нужно скомпилировать код с флагом: `-fsanitize=thread -g`

TSan инструментирует код, отслеживая все обращения к памяти во время исполнения. Если он видит два обращения к одному адресу из разных потоков, одно из которых — запись, и между ними не было "happens-before" связи (захвата мьютекса, join потока), он выдает подробный отчет об ошибке.

> **Важно!**
>
> TSan замедляет выполнение программы в 5-15 раз и потребляет много памяти. Используйте его на этапах тестирования и в CI, но не в релизной сборке для клиентов.

Ловушка std::vector<bool>

Существует особый случай гонки данных, который неочевиден для многих разработчиков. Это использование `std::vector<bool>` в многопоточной среде.

```
std::vector<bool> flags(16, false);

// Поток 1
std::thread t1([&]{
    flags[0] = true;
});

// Поток 2
std::thread t2([&]{
    flags[1] = true;
});
```

Казалось бы, потоки обращаются к **разным** элементам вектора (индексы 0 и 1). Логически пересечения нет. Однако, согласно спецификации, `std::vector<bool>` — это псевдоконтейнер, который специализируется для экономии памяти. Он упаковывает 8 булевых значений в один байт.

- Процессор не умеет адресовать отдельные биты. Минимальная адресуемая единица — байт.
- Чтобы записать `true` в 0-й бит, процессор должен: прочитать весь байт, наложить битовую маску (OR), записать байт обратно.
- Поток 1 читает байт, меняет 0-й бит.
- Поток 2 одновременно читает **тот же самый байт**, меняет 1-й бит.

В результате возникает классический Data Race на уровне байта. Изменения одного из потоков могут быть потеряны, или байт превратится в мусор.

Резюме раздела

- **Race Condition** — ошибка логики (порядок действий), **Data Race** — ошибка доступа к памяти (UB).
- Любая запись в общую память без синхронизации — потенциальный Data Race.
- Используйте **ThreadSanitizer** для поиска гонок.
- Остерегайтесь `std::vector<bool>`: разные индексы не гарантируют безопасность потоков, так как они могут делить один байт памяти.

Глава 42

Синхронизация: Мьютексы и Взаимные блокировки

Рассмотрев опасности гонок данных, мы приходим к необходимости механизмов защиты разделяемых ресурсов. Самым базовым и распространенным примитивом синхронизации в C++ (и в большинстве других языков) является мьютекс (Mutual Exclusion — взаимное исключение). В этой главе мы изучим не только API `std::mutex`, но и правильные паттерны его использования (RAII), а также разберем одну из самых сложных проблем многопоточности — взаимную блокировку (Deadlock) и способы борьбы с ней.

std::mutex и Критическая секция

Класс `std::mutex` (заголовок `<mutex>`) предоставляет два основных метода:

- `lock()`: Поток пытается захватить мьютекс. Если мьютекс свободен, поток захватывает его и продолжает выполнение. Если мьютекс уже захвачен другим потоком, текущий поток **блокируется** (уходит в состояние ожидания, sleeping/waiting), пока мьютекс не освободится.
- `unlock()`: Поток освобождает мьютекс, позволяя одному из ожидающих потоков захватить его.

Участок кода между вызовами `lock()` и `unlock()` называется **критической секцией**. В любой момент времени внутри критической секции может находиться только один поток.

Почему "сырые" lock() и unlock() опасны

Использование методов `lock()` и `unlock()` напрямую крайне не рекомендуется в современном C++. Рассмотрим пример:

```
std::mutex mtx;
int shared_data = 0;

void bad_practice() {
    mtx.lock();
    // ... работа с shared_data ...

    if (some_error) {
```

```
        // Забыли unlock()! Мьютекс остался захвачен навсегда.
        return;
    }

    // Если здесь вылетит исключение, unlock() тоже не вызовется.
    func_that_may_throw();

    mtx.unlock();
}
```

Если поток завершится (через `return` или исключение), не вызвав `unlock()`, мьютекс останется в заблокированном состоянии. Все остальные потоки, пытающиеся его захватить, зависнут навечно (Deadlock).

RAII: lock_guard и unique_lock

Для решения проблемы управления временем жизни блокировки используется идиома RAII (Resource Acquisition Is Initialization). Стандартная библиотека предоставляет обертки, которые захватывают мьютекс в конструкторе и освобождают в деструкторе.

std::lock_guard

Самая простая и легкая обертка. Захватывает мьютекс при создании, освобождает при выходе из области видимости. Не копируется, не перемещается.

```
void good_practice() {
    // Конструктор вызывает mtx.lock()
    std::lock_guard<std::mutex> guard(mtx);

    // ... безопасная работа ...

    // При любом выходе (return, exception) вызовется деструктор guard,
    // который вызовет mtx.unlock().
}
```

Начиная с C++17, шаблонный тип можно не указывать (CTAD): `std::lock_guard guard(mtx);`.

std::unique_lock

Более мощная и тяжелая обертка. В отличие от `lock_guard`, она позволяет:

- Отложить блокировку (стратегия `std::defer_lock`).
- Временно разблокировать мьютекс (`unlock()`) и снова заблокировать (`lock()`) внутри одного скоупа.
- Передавать владение блокировкой (перемещаемый тип).

`std::unique_lock` необходим при работе с условными переменными (`std::condition_variable`), так как они требуют возможности атомарно отпускать мьютекс при ожидании.

Shared Mutex (Reader-Writer Lock)

Часто возникает ситуация, когда данные редко меняются, но часто читаются (например, конфигурация, кэш DNS). Использование обычного `std::mutex` заставит читателей выстраиваться в очередь, хотя они могли бы читать данные параллельно, не мешая друг другу.

Для этого сценария в C++17 был добавлен `std::shared_mutex` (заголовок `<shared_mutex>`). Он поддерживает два режима захвата:

1. **Эксклюзивный (Writer):** Аналог обычного `lock()`. Блокирует всех (и читателей, и писателей). Используется для изменения данных.
2. **Разделяемый (Reader):** Метод `lock_shared()`. Позволяет нескольким потокам одновременно владеть мьютексом в режиме чтения. Блокирует только писателей.

Для удобства использования существуют соответствующие RAII-обертки:

- `std::lock_guard<std::shared_mutex>` или `std::unique_lock` — для эксклюзивного захвата (писатель).
- **`std::shared_lock<std::shared_mutex>`** — для разделяемого захвата (читатель).

```
#include <shared_mutex>

class ThreadSafeConfig {
    std::map<std::string, std::string> settings;
    mutable std::shared_mutex mtx; // mutable, чтобы использовать в const методах

public:
    std::string get(const std::string& key) const {
        // Множество потоков могут вызывать get() одновременно
        std::shared_lock lock(mtx);
        auto it = settings.find(key);
        return (it != settings.end()) ? it->second : "";
    }

    void set(const std::string& key, const std::string& value) {
        // Только один поток может писать, блокируя всех читателей
        std::lock_guard lock(mtx);
        settings[key] = value;
    }
};
```

Deadlock (Взаимная блокировка)

Использование мьютексов вводит новый класс ошибок — взаимные блокировки. Классический пример — транзакция перевода денег между двумя банковскими счетами. Чтобы операция была атомарной, нужно заблокировать оба аккаунта.

Проблема порядка блокировки

Рассмотрим наивную реализацию функции `transfer`:

```cpp
struct Account {
    std::mutex m;
    int balance;
};

void transfer(Account& from, Account& to, int amount) {
    std::lock_guard<std::mutex> lock1(from.m);
    // Имитация задержки, увеличивающая вероятность Deadlock
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lock2(to.m);

    from.balance -= amount;
    to.balance += amount;
}
```

Представим, что два потока одновременно выполняют встречные переводы:

- **Поток 1:** `transfer(AccA, AccB, 100)`
- **Поток 2:** `transfer(AccB, AccA, 50)`

Сценарий катастрофы:

1. Поток 1 захватывает мьютекс `AccA`.
2. Поток 2 захватывает мьютекс `AccB`.
3. Поток 1 пытается захватить `AccB` и засыпает, так как он занят Потоком 2.
4. Поток 2 пытается захватить `AccA` и засыпает, так как он занят Потоком 1.

Оба потока ждут друг друга. Программа зависает.

Решение 1: Иерархия блокировок

Можно установить правило: всегда захватывать мьютексы в определенном глобальном порядке. Например, по адресу памяти объекта `Account` (от меньшего к большему) или по уникальному ID аккаунта.

```cpp
void safe_transfer_manual(Account& from, Account& to, int amount) {
    if (&from < &to) {
        std::lock_guard<std::mutex> lock1(from.m);
        std::lock_guard<std::mutex> lock2(to.m);
    } else {
        std::lock_guard<std::mutex> lock1(to.m); // Сначала меньший адрес
        std::lock_guard<std::mutex> lock2(from.m);
    }
    // ...
}
```

Это работает, но сложно поддерживать в больших системах.

Решение 2: std::lock и std::scoped_lock

Стандартная библиотека C++ предоставляет алгоритмы для безопасного захвата нескольких мьютексов без риска дедлока (обычно внутри используется тот же алгоритм try-and-

backoff или сортировка по адресам).

В C++11 есть функция `std::lock(m1, m2, ...)`, которая блокирует мьютексы, но не управляет их освобождением (нужно передавать их в `lock_guard` с флагом `std::adopt_lock`).

В C++17 появился идеальный инструмент — **`std::scoped_lock`**. Это вариадик шаблон, который принимает произвольное количество мьютексов, захватывает их безопасным алгоритмом в конструкторе и освобождает в деструкторе.

```
void safe_transfer_cpp17(Account& from, Account& to, int amount) {
    // Блокирует оба мьютекса безопасно. Порядок аргументов не важен.
    std::scoped_lock lock(from.m, to.m);

    from.balance -= amount;
    to.balance += amount;
}
```

Резюме раздела

- Никогда не вызывайте `lock()` и `unlock()` вручную. Используйте RAII: `std::lock_guard` или `std::unique_lock`.
- Используйте `std::shared_mutex` и `std::shared_lock` для сценариев "много читателей, один писатель".
- Захват нескольких мьютексов одновременно — риск Deadlock.
- Для захвата нескольких мьютексов всегда используйте `std::scoped_lock` (C++17).

Глава 43

Атомики и низкоуровневая синхронизация

Мьютексы, рассмотренные нами ранее, являются механизмами синхронизации уровня операционной системы. Когда поток не может захватить мьютекс, он "усыпляется" ядром ОС, что влечет за собой накладные расходы на системные вызовы и переключение контекста. Однако для простейших операций, таких как инкремент счетчика или установка флага, эти расходы могут быть неприемлемо высоки.

В этой главе мы спустимся на уровень ниже — к атомарным операциям и инструкциям процессора. Мы разберем, как работает `std::atomic`, развеем опасный миф о `volatile` и напишем собственный примитив синхронизации — спинлок (Spinlock).

Миф о volatile

Прежде чем переходить к правильным инструментам, необходимо разобрать одну из самых живучих ошибок в C++ программировании. Многие разработчики, пришедшие из мира микроконтроллеров или старых версий Java, ошибочно полагают, что ключевое слово `volatile` обеспечивает потокобезопасность.

> **Важно!**
>
> **volatile в C++ не имеет отношения к многопоточности!** Оно не гарантирует атомарность. Оно не создает барьеров памяти. Оно не предотвращает гонки данных.

Ключевое слово `volatile` сообщает компилятору лишь одно: "значение этой переменной может измениться извне, поэтому не кэшируй его в регистрах". Это необходимо для работы с **MMIO** (Memory-Mapped I/O) — когда адрес в памяти на самом деле является портом ввода-вывода устройства (например, датчика температуры).

Рассмотрим пример Data Race с использованием `volatile`:

```
volatile int counter = 0;

void increment() {
    // ОШИБКА: Это все еще Read-Modify-Write операция.
    // volatile лишь заставит процессор каждый раз читать из памяти,
    // но не запрещает другому потоку вклиниться между чтением и записью.
    counter++;
}
```

Если запустить этот код в несколько потоков, результат будет неверным. Единственное применение `volatile` в современном C++ — это взаимодействие с железом или обработчиками сигналов UNIX. Для потоков используйте `std::atomic`.

std::atomic

Шаблон `std::atomic<T>` (заголовок `<atomic>`) предоставляет интерфейс для работы с данными, операции над которыми гарантированно выполняются атомарно.

Базовые операции: load и store

Атомики запрещают компилятору и процессору переупорядочивать инструкции (reordering) опасным образом и гарантируют, что чтение или запись переменной произойдет целиком.

```
std::atomic<int> flag = 0;

void writer() {
    flag.store(1); // Атомарная запись
}

void reader() {
    // Атомарное чтение. Гарантируется, что мы не прочитаем
    // "частично записанное" значение (torn read).
    int val = flag.load();
}
```

Read-Modify-Write (RMW)

Самая мощная возможность атомиков — это операции, которые читают, изменяют и записывают значение как единое неделимое действие.

- `fetch_add(val)`: прибавляет значение, возвращает *старое*.
- `fetch_sub(val)`: вычитает значение.
- `exchange(val)`: записывает новое значение, возвращает *старое*.

```
std::atomic<int> counter = 0;

void worker() {
    // Эквивалентно fetch_add(1).
    // На x86 компилируется в инструкцию LOCK XADD.
    counter++;
}
```

Важно понимать разницу между атомарностью самой переменной и последовательности действий.

На заметку

`atomic_val = atomic_val + 1` — **НЕ** атомарно! Это последовательность: (1) атомарное чтение, (2) локальное сложение, (3) атомарная запись. Между (1) и (3) может вклиниться другой поток. Для атомарного инкремента используйте только `fetch_add` или оператор `++`.

Compare-And-Swap (CAS)

Фундаментом всех lock-free структур данных (очередей, списков без мьютексов) является операция Compare-And-Swap. В C++ она представлена методами `compare_exchange_strong` и `compare_exchange_weak`.

Логика CAS следующая: "Проверь, равно ли текущее значение `expected`. Если да, запиши `desired` и верни true. Если нет, обнови `expected` текущим значением и верни false".

```
std::atomic<int> current_head;

void push(int new_val) {
    int old_head = current_head.load();
    // Пытаемся обновить head, только если он не изменился с момента чтения.
    // Если изменился (кто-то другой успел запушить), повторяем цикл.
    while (!current_head.compare_exchange_weak(old_head, new_val)) {
        // Тело цикла может быть пустым, old_head обновляется автоматически
        ↪ внутри CAS
    }
}
```

- **weak**: Может вернуть `false`, даже если значение равно ожидаемому (spurious failure). Это особенность архитектур типа ARM и PowerPC. Используется в циклах.
- **strong**: Гарантирует успех, если значения равны. Используется, когда цикл неудобен, но стоит дороже на некоторых платформах.

Реализация Spinlock

Используя атомик, мы можем реализовать собственный примитив синхронизации — спинлок. В отличие от мьютекса, спинлок не усыпляет поток, а заставляет его крутиться в цикле ("spin"), ожидая освобождения флага.

Для реализации идеально подходит `std::atomic_flag`. Это единственный тип, для которого стандарт гарантирует lock-free реализацию на любой платформе.

```
#include <atomic>
#include <thread>

class Spinlock {
    // ATOMIC_FLAG_INIT устарело в C++20, конструктор по умолчанию ставит в
    ↪ false (clear)
    std::atomic_flag flag;
```

```
7
8  public:
9      void lock() {
10         // test_and_set атомарно ставит флаг в true и возвращает ПРЕДЫДУЩЕЕ
           ↪  значение.
11         // Если вернулось true -> значит флаг уже был занят -> продолжаем
           ↪  крутиться.
12         // Если вернулось false -> мы успешно захватили спинлок.
13         while (flag.test_and_set(std::memory_order_acquire)) {
14             // Hint для процессора, что мы в цикле ожидания (снижает
               ↪  энергопотребление)
15             // В C++20: std::atomic::wait
16             #if defined(__x86_64__) || defined(_M_X64)
17                 _mm_pause();
18             #endif
19         }
20     }
21
22     void unlock() {
23         flag.clear(std::memory_order_release);
24     }
25 };
```

Spinlock vs Mutex: Цена решения

Когда стоит использовать спинлок, а когда мьютекс?

Spinlock:

- **Плюсы:** Не требует системных вызовов. Захват происходит за десятки наносекунд (если свободен). Не сбрасывает кэш процессора (нет переключения контекста).
- **Минусы:** Если спинлок занят долго, ожидающий поток сжигает 100% процессорного времени впустую (busy-wait).
- **Применение:** Критические секции длиной в несколько инструкций (например, инкремент счетчика или перестановка указателя).

Mutex:

- **Плюсы:** Если ресурс занят, поток уступает ядро другим задачам. Экономит энергию.
- **Минусы:** Захват/освобождение требуют обращения к ядру ОС, что долго.
- **Применение:** Длительные операции (ввод-вывод, сложная логика).

> **Важно!**
>
> Линус Торвальдс и многие эксперты считают, что использование спинлоков в userspace (пользовательском пространстве) вредно. Причина — **Priority Inversion** и планировщик ОС. Если поток, захвативший спинлок, будет вытеснен планировщиком (кончился квант времени), все остальные потоки будут крутиться впустую целый квант, ожидая его возвращения. Современные реализации `std::mutex` (futex в Linux) гибридны: они немного "крутятся" в userspace перед тем, как уйти в сон.

Модель памяти (Memory Ordering)

В примерах выше мы видели странные аргументы `std::memory_order_acquire/release`. Это управление моделью памяти. По умолчанию C++ использует **Sequential Consistency** (`seq_cst`). Это самая строгая модель: она гарантирует, что все потоки видят все изменения в одном и том же глобальном порядке. Это безопасно, но может быть медленно, так как требует блокировки шины или сброса буферов записи процессора.

Более слабые модели (`acquire`, `release`, `relaxed`) позволяют процессору переупорядочивать инструкции ради производительности, сохраняя лишь необходимые гарантии видимости. Программирование с weak ordering — это высший пилотаж C++, где любая ошибка приводит к неуловимым багам. До тех пор, пока профайлер не покажет, что узкое место именно в синхронизации кэшей, используйте дефолтный `seq_cst`.

Резюме раздела

- **volatile** не для потоков. Никогда не используйте его для синхронизации.
- **std::atomic** — единственный корректный способ работы с общими простыми типами без мьютексов.
- **RMW операции** (`fetch_add`, `exchange`) позволяют менять данные безопасно.
- **Spinlock** эффективен для сверхкоротких блокировок, но опасен в общем случае из-за сжигания CPU.
- **CAS (Compare-And-Swap)** — основа lock-free алгоритмов.

Глава 44

Оптимизация под железо: False Sharing

В предыдущих главах мы добились логической корректности многопоточных программ, используя мьютексы и атомики. Мы устранили гонки данных (Data Race) и неопределенное поведение. Однако в высокопроизводительных системах логической корректности недостаточно.

Существует класс проблем, при которых программа работает абсолютно правильно, не имеет блокировок (lock-free), использует все ядра процессора на 100%, но при этом выполняется в десятки раз медленнее, чем однопоточная версия. Причина кроется в физической организации подсистемы памяти, а именно в явлении, называемом **False Sharing** (Ложное разделение).

Физика кэш-линий

Процессор никогда не читает из оперативной памяти (RAM) отдельные байты. Это было бы крайне неэффективно с точки зрения пропускной способности шины. Обмен данными между оперативной памятью и кэшами процессора (L1/L2/L3) происходит блоками фиксированного размера, которые называются **кэш-линиями** (Cache Lines).

> **На заметку**
>
> На большинстве современных архитектур (x86_64, многие ARM) размер кэш-линии составляет **64 байта**.

Это означает, что если вы читаете переменную типа `int` (4 байта) по адресу `0x1000`, процессор загрузит в кэш весь блок от `0x1000` до `0x103F`. Вместе с нужной вам переменной в кэш попадут и соседние данные ("хвост" длиной 60 байт).

Протоколы когерентности (MESI)

В многоядерных системах каждый процессор (ядро) имеет свой собственный L1 и L2 кэш. Если два ядра закэшировали одну и ту же область памяти, возникает проблема синхронизации (когерентности). Если одно ядро изменит данные у себя в L1, другое ядро должно узнать об этом, иначе оно будет работать с устаревшим значением.

Для решения этой задачи существуют аппаратные протоколы когерентности, такие как **MESI** (Modified, Exclusive, Shared, Invalid). Упрощенно логика выглядит так:

1. Чтобы ядро могло записать данные в кэш-линию, оно должно получить её в эксклюзивное владение (состояние *Exclusive* или *Modified*).
2. Если эта линия находится в кэшах других ядер (состояние *Shared*), то перед записью инициатор посылает сигнал инвалидации (*Invalidate*).
3. Другие ядра помечают свои копии линии как невалидные (*Invalid*) и выбрасывают их.
4. При следующей попытке чтения эти ядра вынуждены заново загружать линию из памяти (или из кэша пишущего ядра), что очень дорого.

Феномен False Sharing

Ложное разделение возникает, когда два или более потока одновременно модифицируют **логически разные** переменные, которые случайно оказались на **одном и том же** физическом носителе — в одной кэш-линии.

Сценарий "Пинг-понг"

Представим структуру, содержащую два атомарных счетчика:

```
struct SharedData {
    std::atomic<int> a; // 4 байта
    std::atomic<int> b; // 4 байта
};
// sizeof(SharedData) == 8 байт. Оба поля лежат в одной кэш-линии.
```

Пусть Поток 1 (Ядро 0) инкрементирует `a`, а Поток 2 (Ядро 1) инкрементирует `b`.

1. **Ядро 0** хочет записать в `a`. Оно запрашивает линию эксклюзивно.
2. **Ядро 1** вынуждено сбросить свою копию линии (Invalidate), даже если оно работало только с `b`.
3. Ядро 0 меняет `a`. Линия в состоянии Modified на Ядре 0.
4. **Ядро 1** хочет записать в `b`. Промах кэша (L1 Miss). Оно запрашивает линию у Ядра 0.
5. Ядро 0 сбрасывает свои изменения в общий кэш/память (Write Back) и отдает владение. Линия на Ядре 0 становится Invalid.
6. Ядро 1 меняет `b`.

Происходит "пинг-понг" кэш-линии между ядрами. Вместо того чтобы работать на скорости L1 кэша (1 нс), ядра ждут синхронизации через шину (десятки нс). Производительность может упасть в 10-50 раз по сравнению с теоретическим максимумом.

Термин "ложное" (False) означает, что разделения данных на логическом уровне нет (потоки не трогают данные друг друга), но процессор этого не знает — он видит разделение на уровне 64-байтного блока.

Демонстрация и диагностика

Рассмотрим пример кода, демонстрирующий этот эффект. Мы запустим несколько потоков, каждый из которых инкрементирует свой собственный счетчик в массиве.

```cpp
#include <thread>
#include <vector>
#include <atomic>
#include <new> // для hardware_destructive_interference_size

// Плохая структура: данные упакованы плотно
struct PackedCounter {
    std::atomic<long> value{0};
};

// Хорошая структура: принудительное выравнивание
struct AlignedCounter {
    alignas(64) std::atomic<long> value{0};
};

template <typename CounterType>
void benchmark() {
    std::vector<CounterType> counters(4);
    std::vector<std::thread> threads;

    // Каждый поток работает строго со своим индексом!
    // Логического пересечения нет.
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back([&counters, i] {
            for (int j = 0; j < 100'000'000; ++j) {
                counters[i].value.fetch_add(1, std::memory_order_relaxed);
            }
        });
    }

    for (auto& t : threads) t.join();
}
```

Анализ памяти

В случае `PackedCounter`: `sizeof(atomic<long>)` равен 8 байтам. Четыре счетчика займут 32 байта. Они гарантированно поместятся в одну 64-байтную кэш-линию. Четыре ядра будут драться за право владения этой единственной линией.

В случае `AlignedCounter`: спецификатор `alignas(64)` заставляет компилятор размещать каждый объект по адресу, кратному 64. Размер структуры станет 64 байта (8 байт полезных данных + 56 байт padding). Каждый счетчик гарантированно окажется в своей уникальной кэш-линии.

> **Важно!**
>
> Профилирование такого кода (например, через `perf c2c` в Linux) покажет огромное количество событий **Hitm** (Hit Modified) — ситуаций, когда ядро запрашивает данные, которые были модифицированы в кэше другого ядра.

Решение: Padding и alignas

Единственный способ борьбы с False Sharing — разнести часто изменяемые данные (hot spots) по разным кэш-линиям. Это делается путем добавления "мусорных" данных (padding) между полезными переменными.

В старом C++ это делалось вручную добавлением массивов `char pad[64]`. В современном C++ (с C++11) используется спецификатор выравнивания `alignas`.

```cpp
// Гарантируем, что атомики не попадут в одну линию
struct SafeMetrics {
    alignas(64) std::atomic<int> success_count;
    alignas(64) std::atomic<int> error_count;
};
```

Это увеличивает потребление оперативной памяти (trade-off: память в обмен на скорость процессора), но в контексте глобальных счетчиков или структур синхронизации это ничтожная плата за многократный прирост производительности.

Проблема переносимости и ABI

Возникает вопрос: почему мы пишем `alignas(64)`? Откуда взялось число 64?

Хотя 64 байта — стандарт де-факто для x86_64 (Intel/AMD), другие архитектуры могут иметь другие размеры линии (например, 128 байт на некоторых PowerPC или Apple Silicon). Если мы скомпилируем код с `alignas(64)` для процессора с линией 128 байт, False Sharing вернется, так как два 64-байтных блока попадут в одну 128-байтную линию.

В C++17 в заголовок `<new>` были добавлены константы:

- `std::hardware_destructive_interference_size`: минимальный отступ, гарантирующий отсутствие False Sharing (размер кэш-линии).
- `std::hardware_constructive_interference_size`: максимальный размер непрерывных данных, которые гарантированно влезут в одну линию (для оптимизации локальности).

Дилемма ABI

К сожалению, использование этих констант сопряжено с проблемами ABI (Application Binary Interface). Если библиотека скомпилирована с одним значением этой константы (на старом компиляторе/железе), а приложение с другим, размеры структур не совпадут, что приведет к краху программы.

Из-за этого, например, компилятор GCC долгое время выдавал предупреждение при использовании этих констант, а в некоторых стандартных библиотеках они реализованы просто как жестко заданные значения.

Резюме раздела

- Процессор обменивается данными кэш-линиями (обычно 64 байта).
- **False Sharing** возникает, когда разные потоки пишут в разные переменные, лежащие в одной кэш-линии.
- Симптомом является резкое падение производительности при высокой нагрузке на CPU без видимых блокировок.
- Решение — использование `alignas(64)` (или больше) для разнесения "горячих" данных в памяти.
- Это классический пример трейдоффа: мы тратим лишнюю память (padding), чтобы выиграть в скорости шины.

Глава 45

Практикум: Управление памятью и Smart Pointers в многопоточности

Мы завершаем наш цикл, объединяя знания о многопоточности с фундаментальной темой C++ — управлением памятью. Умные указатели (smart pointers) стали стандартом де-факто для владения ресурсами, но их поведение в многопоточной среде часто понимается превратно. Является ли `std::shared_ptr` потокобезопасным? Ответ "да, но..." требует детального разбора.

Кроме того, мы рассмотрим реальный кейс из разработки интерпретаторов (на примере учебного Scheme) — реализацию сборщика мусора (Garbage Collector) по алгоритму Mark and Sweep, где вопросы достижимости объектов и управления графом становятся критическими.

Потокобезопасность std::shared_ptr

Вопрос "потокобезопасен ли shared_ptr?" — один из самых популярных на собеседованиях. Правильный ответ состоит из двух частей, так как `std::shared_ptr` управляет двумя сущностями:

1. **Control Block (Управляющий блок):** Хранит счетчик ссылок (`ref_count`) и счетчик слабых ссылок (`weak_count`).
2. **Managed Object (Управляемый объект):** Тот самый `T*`, на который указывает умный указатель.

Гарантии стандарта

Стандарт C++ дает следующие гарантии:

- **Счетчик ссылок — потокобезопасен.** Операции инкремента (копирование `shared_ptr`) и декремента (уничтожение) являются атомарными. Вы можете безопасно копировать `shared_ptr`, указывающий на один объект, в разных потоках. Гонки данных на счетчике не будет.
- **Управляемый объект — НЕ потокобезопасен.** Если два потока имеют доступ к одному объекту через разные копии `shared_ptr`, и один из них пишет в объект — нужна внешняя синхронизация (мьютекс).

- **Сам экземпляр shared_ptr — НЕ потокобезопасен.** Если два потока пытаются изменить **один и тот же экземпляр** `shared_ptr` (например, глобальный указатель, присваивая ему новое значение), это гонка данных.

```cpp
std::shared_ptr<int> global_ptr = std::make_shared<int>(42);

void thread1() {
    // Безопасно: создается локальная копия.
    // Счетчик атомарно увеличивается.
    std::shared_ptr<int> local = global_ptr;

    // Опасно! Чтение объекта без мьютекса (если кто-то пишет в *local)
    std::cout << *local;
}

void thread2() {
    // ОПАСНО! Гонка данных на самом global_ptr.
    // Присваивание изменяет внутренности глобального объекта (ptr и control
    ↪  block).
    // thread1 может прочитать "разорванное" значение.
    global_ptr = std::make_shared<int>(100);
}
```

Для безопасной модификации самого экземпляра `shared_ptr` в C++20 введены `std::atomic<std::`
До C++20 приходилось использовать внешние функции `std::atomic_store(&ptr, ...)`, что было неудобно и опасно.

Thread Local Storage (TLS)

Иногда нам вообще не нужно делить данные между потоками. Напротив, нам нужно, чтобы у каждого потока была своя, уникальная копия переменной. Для этого используется спецификатор `thread_local`.

```cpp
#include <random>

// Глобальный генератор случайных чисел? Плохая идея (мьютекс убьет скорость).
// thread_local создает генератор для каждого потока отдельно.
int get_random() {
    thread_local std::mt19937 gen(std::clock());
    std::uniform_int_distribution<int> dist(0, 100);
    return dist(gen);
}
```

Переменные `thread_local`:

- Инициализируются при первом обращении к ним в потоке (lazy initialization).
- Уничтожаются при завершении потока.
- Живут в специальной области памяти (TLS segment), доступ к которой обычно осуществляется через регистр сегмента (например, `fs` на x86_64), что очень быстро.

Это идеальное решение для кэшей, буферов (например, строковых билдеров) и генераторов случайных чисел в многопоточных сервисах.

Case Study: Сборщик мусора (Mark and Sweep)

В курсе мы разрабатываем интерпретатор Scheme. Языки Lisp-семейства славятся тем, что в них легко создаются циклические структуры данных (списки, ссылающиеся сами на себя, замыкания).

Почему shared_ptr не подходит?

Если использовать `std::shared_ptr` для узлов графа объектов Scheme, мы столкнемся с проблемой циклических ссылок. Объект A ссылается на B, B ссылается на A. Их `ref_count` никогда не станет нулем, даже если они оба недостижимы из остальной программы. Это классическая утечка памяти.

Решение — реализовать собственный Garbage Collector (GC). Самый простой и надежный алгоритм — **Mark and Sweep** (Пометь и Удали).

Алгоритм

Идея алгоритма базируется на понятии **достижимости** (reachability). Объект "жив", если до него можно добраться по ссылкам от "корней" (Roots).

1. **Root Set (Корневое множество):** Это объекты, которые гарантированно живы. В нашем случае это Глобальный Скоуп (Global Scope) интерпретатора и, возможно, временные переменные на стеке вычислений.
2. **Фаза Mark (Пометка):** Мы запускаем обход графа (DFS/BFS) от корней. Каждый посещенный объект помечается флагом `is_marked = true`. Если мы встречаем уже помеченный объект, останавливаемся (защита от зацикливания).
3. **Фаза Sweep (Удаление):** Мы проходим по **всем** выделенным объектам в куче (нам нужен список всех аллокаций).
    - Если у объекта `is_marked == true`, мы сбрасываем флаг в `false` (готовим к следующему циклу) и оставляем его жить.
    - Если у объекта `is_marked == false`, значит он недостижим (мусор). Мы вызываем `delete` и удаляем его из списка аллокаций.

```
// Упрощенная модель объекта
struct Object {
    bool is_marked = false;
    std::vector<Object*> children; // Ссылки на другие объекты
    virtual ~Object() = default;
};

void mark(Object* obj) {
    if (!obj || obj->is_marked) return;

    obj->is_marked = true;
    for (auto* child : obj->children) {
        mark(child);
    }
}

```

```
void sweep(std::vector<Object*>& all_objects) {
    auto it = all_objects.begin();
    while (it != all_objects.end()) {
        if ((*it)->is_marked) {
            (*it)->is_marked = false; // Сброс для следующего раза
            ++it;
        } else {
            delete *it; // Удаление мусора
            // Удаление из списка всех объектов (swap-and-pop для вектора)
            *it = all_objects.back();
            all_objects.pop_back();
            // Итератор не инкрементируем, проверяем новый элемент на этом месте
        }
    }
}
```

Этот подход решает проблему циклов: если группа объектов замкнута сама на себя, но недостижима от Global Scope, алгоритм `mark` до них не дойдет, их флаги останутся `false`, и `sweep` их удалит.

Инструментарий: Leak Sanitizer

Для проверки корректности работы GC в задачах мы используем **Leak Sanitizer** (LSan). Это часть набора санитайзеров (как и TSan/ASan). Он запускается в конце работы программы и проверяет, остались ли в куче аллокации, которые не были освобождены.

Если ваш GC работает корректно, на выходе LSan молчит. Если вы забыли удалить циклические ссылки, LSan выдаст отчет об утечке, показав стек вызовов, где была выделена память.

Резюме раздела

- `std::shared_ptr` гарантирует атомарность счетчика, но не безопасность данных внутри.
- Модификация одного экземпляра `shared_ptr` из разных потоков — гонка данных.
- `thread_local` позволяет создавать уникальные копии данных для потоков без мьютексов.
- Для сложных структур с циклами (графы, интерпретаторы) `shared_ptr` недостаточен. Требуется полноценный Garbage Collection (Mark and Sweep).

Часть IX

Лекция 09 – Condition variable

Глава 46

От активного ожидания к поддержке ядра: Проблема синхронизации

Одной из фундаментальных задач многопоточного программирования является координация действий между потоками. Часто возникает сценарий producer-consumer, где один поток готовит данные, а второй должен дождаться их готовности перед обработкой. В этой главе мы рассмотрим эволюцию подходов к ожиданию события: от наивного активного цикла до использования механизмов ядра операционной системы.

Активное ожидание (Polling)

Самый очевидный способ заставить поток ждать изменения состояния — это запустить бесконечный цикл проверки условия. Этот подход называется активным ожиданием (busy wait) или поллингом (polling).

```cpp
#include <atomic>
#include <chrono>
#include <thread>
#include <iostream>

std::atomic<bool> is_ready(false);

void producer() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    is_ready.store(true);
}

void consumer() {
    // Busy wait цикл
    while (!is_ready.load()) {
        // Поток бесконечно проверяет переменную
    }
    std::cout << "Data processed\n";
}

int main() {
    std::thread t(producer);
    consumer();
```

```
    t.join();
}
```

С точки зрения логики программы этот код корректен: поток `consumer` действительно дождется установки флага. Однако с точки зрения архитектуры системы это решение неприемлемо.

Цикл `while` в `consumer` представляет собой непрерывный поток инструкций для процессора. Планировщик операционной системы (OS Scheduler) видит, что поток готов к исполнению, и выделяет ему квант времени (time slice). Поток тратит этот квант на бесконечную проверку `false`, не выполняя полезной работы.

> **Важно!**
>
> Активное ожидание приводит к загрузке ядра процессора на 100% ("греет воздух"). Это увеличивает энергопотребление, нагрев CPU и может приводить к троттлингу частот. Кроме того, в условиях конкуренции за ресурсы (например, на одноядерной машине или при большом количестве потоков), активный поток отбирает время у потока-продюсера, замедляя наступление ожидаемого события (Starvation).

Попытка смягчить проблему с помощью `std::this_thread::yield()` не решает ее фундаментально:

```
while (!is_ready.load()) {
    std::this_thread::yield();
}
```

`yield()` сообщает планировщику, что поток готов отдать остаток своего кванта времени. Однако, если в очереди планировщика нет других готовых к исполнению потоков (или их приоритет ниже), управление немедленно вернется к текущему потоку, и цикл продолжится. В нагруженной системе это лишь увеличивает накладные расходы на переключение контекста (Context Switch Overhead).

Наивное решение: Сон (Sleep)

Следующая итерация попытки решить проблему без специальных примитивов синхронизации — добавление задержки в цикл проверки.

```
while (!is_ready.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(2));
}
```

Здесь поток добровольно уходит в состояние ожидания, и планировщик исключает его из очереди на исполнение на указанное время. Загрузка CPU падает почти до нуля. Однако появляются две критические проблемы:

1. Латентность (Latency)

Если событие (`is_ready = true`) произойдет через 0.1 мс после начала сна, поток-потребитель узнает об этом только через 1.9 мс, когда проснется. В высокопроизводительных системах

(HFT, Real-Time Systems) задержка в миллисекунды недопустима. Уменьшение времени сна приближает нас обратно к активному ожиданию и лишним переключениям контекста.

2. Зависимость от планировщика

Аргумент функции `sleep_for` — это *минимальное* время сна. Реальное время зависит от гранулярности системного таймера и загруженности планировщика.

- В Windows дефолтная гранулярность таймера может достигать 15.6 мс. Запрос на сон в 1 мс фактически усыпит поток на 15 мс.
- Операционная система не гарантирует мгновенного пробуждения. Поток проснется, попадет в очередь готовых к исполнению, и только когда до него дойдет очередь, он получит процессорное время.

Использование "магических констант" (например, 2 мс) — признак плохого дизайна. Время выполнения операций может варьироваться на порядки в зависимости от железа, и подобранная константа будет либо слишком большой (латентность), либо слишком маленькой (нагрузка на CPU).

Атомарные операции и Livelock

Иногда разработчики пытаются реализовать механизмы синхронизации на базе сложных атомарных операций, таких как `compare_exchange_weak` (CAS), надеясь избежать использования "тяжелых" мьютексов.

```
void wait_for_flag(std::atomic<int>& counter) {
    int expected = 1;
    // Пытаемся атомарно изменить значение, если оно равно expected
    while (!counter.compare_exchange_weak(expected, 0)) {
        expected = 1; // Сброс expected, так как CAS обновляет его при неудаче
        // Busy wait
    }
}
```

Хотя `std::atomic` обеспечивает корректность с точки зрения гонок данных (Data Race Free), с точки зрения использования ресурсов это тот же самый busy loop. Атомарные операции (особенно RMW — Read-Modify-Write) значительно дороже обычных операций чтения/записи, так как требуют блокировки шины памяти или использования протоколов когерентности кэшей (MESI) между ядрами.

В таком сценарии возможен **Livelock**. В отличие от Deadlock, где потоки заблокированы и не двигаются, в Livelock потоки активно выполняют инструкции, меняют свои локальные состояния, но глобальный прогресс системы отсутствует или крайне мал из-за постоянной конкуренции за кэш-линию (cache contention).

Необходимость поддержки ядра

Все рассмотренные выше методы работают в пространстве пользователя (User Space) и пытаются эмулировать ожидание. Эффективное решение невозможно без участия ядра операционной системы.

Нам нужен механизм, который позволяет:

1. Перевести поток в состояние блокировки (Blocked/Waiting), полностью убрав его из очереди планировщика.
2. Гарантировать, что поток будет разбужен **только** тогда, когда наступит определенное событие (или придет сигнал).
3. Выполнить пробуждение максимально быстро после наступления события.

В Linux для этого используется системный вызов `futex` (fast userspace mutex), в Windows — `WaitOnAddress` или объекты ядра (Event, Semaphore). В стандарте C++ эти механизмы абстрагированы в примитив **Condition Variable** (условная переменная), который мы рассмотрим в следующей главе.

Глава 47

Механизм Condition Variable: Корректное ожидание

Как мы выяснили в предыдущей главе, для эффективного ожидания событий необходима поддержка со стороны операционной системы. В стандартной библиотеке C++ таким механизмом является `std::condition_variable`. Это примитив синхронизации, который позволяет одному или нескольким потокам заблокироваться (уйти в сон) до тех пор, пока другой поток не отправит уведомление о том, что состояние разделяемых данных изменилось.

Триада синхронизации

Для корректной работы с условными переменными всегда требуются три компонента, работающие в связке:

1. **Разделяемое состояние** (Shared State) — данные, изменения которых мы ждем (например, флаг `bool is_ready` или очередь задач).
2. **Мьютекс** (`std::mutex`) — для защиты доступа к разделяемому состоянию.
3. **Условная переменная** (`std::condition_variable`) — механизм сигнализации.

Важная особенность интерфейса `std::condition_variable::wait` заключается в том, что он принимает не просто мьютекс, а `std::unique_lock<std::mutex>`. Использование `std::lock_gua` здесь невозможно. Это продиктовано механикой работы: `wait` должен иметь возможность программно разблокировать и снова заблокировать мьютекс, чем `lock_guard` (владеющий мьютексом до конца области видимости) не управляет.

Рассмотрим канонический пример взаимодействия Producer и Consumer:

```
#include <mutex>
#include <condition_variable>
#include <iostream>
#include <thread>

std::mutex m;
std::condition_variable cv;
bool ready = false; // Разделяемое состояние
int data = 0;

```

```cpp
void producer() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        // 1. Захватываем мьютекс для модификации состояния
        std::lock_guard<std::mutex> lk(m);
        data = 42;
        ready = true;
    } // Мьютекс освобождается здесь

    // 2. Уведомляем ожидающий поток
    cv.notify_one();
}

void consumer() {
    // 1. Захватываем мьютекс через unique_lock
    std::unique_lock<std::mutex> lk(m);

    // 2. Ждем выполнения условия
    // wait принимает блокировку и предикат
    cv.wait(lk, []{ return ready; });

    // 3. Здесь мьютекс снова захвачен, а ready == true
    std::cout << "Data received: " << data << "\n";
}
```

Механика метода wait

Что именно происходит внутри вызова `cv.wait(lk)`? Это сложная составная операция, которую можно разложить на следующие этапы:

1. **Проверка предиката (опционально):** Если передан предикат (лямбда `[]{ return ready; }`), он выполняется. Если он возвращает `true`, метод немедленно возвращает управление, поток продолжает работу.
2. **Атомарное освобождение и сон:** Если предикат вернул `false` (или не был передан), `condition_variable` атомарно выполняет два действия:
   - Разблокирует мьютекс (`lk.unlock()`).
   - Переводит текущий поток в список ожидания на данной условной переменной и усыпляет его (Syscall `futex wait` в Linux).

   Атомарность здесь критически важна: между освобождением мьютекса и уходом в сон не может "вклиниться" другой поток и отправить уведомление, которое мы бы пропустили (проблема Lost Wakeup).
3. **Ожидание:** Поток спит. Он не потребляет CPU.
4. **Пробуждение:** Поток просыпается по сигналу (`notify`) или системному событию.
5. **Захват мьютекса:** Перед тем как вернуть управление пользовательскому коду, `wait` обязан снова захватить мьютекс (`lk.lock()`). Если мьютекс занят другим потоком, наш поток блокируется на мьютексе.

Ложные пробуждения (Spurious Wakeups)

Один из самых неочевидных и опасных аспектов работы с условными переменными — это феномен ложных пробуждений.

> **Важно!**
>
> Поток, заблокированный в `cv.wait()`, может проснуться **даже если никто не вызывал** `notify_one()` или `notify_all()`.

Это не баг стандартной библиотеки C++, а особенность реализации планировщиков операционных систем (в частности, POSIX Threads). Причины могут быть разными: обработка сигналов процессом, оптимизации реализации условных переменных (например, использование широковещательного пробуждения вместо точечного в некоторых гонках внутри ядра).

Из-за этого факта использование `if` для проверки условия является грубой ошибкой:

Некорректный код (Anti-pattern)

```
std::unique_lock<std::mutex> lk(m);
if (!ready) {
    cv.wait(lk); // ОШИБКА!
}
// Если произошло ложное пробуждение, мы попадем сюда,
// хотя ready все еще false.
process(data); // Обработка невалидных данных
```

Единственно верный паттерн использования — цикл `while`. Стандартный метод `cv.wait(lock, predicate)` является синтаксическим сахаром для следующего цикла:

Корректная реализация цикла ожидания

```
while (!ready) {
    cv.wait(lk);
}
```

Алгоритм работы потребителя с учетом ложных пробуждений выглядит как диаграмма состояний:

1. Захватить мьютекс.
2. Проверить условие. Если `true` — выйти из цикла.
3. Если `false` — освободить мьютекс и уснуть.
4. ... (сон) ...
5. Проснуться (по любой причине).
6. Захватить мьютекс.
7. Перейти к шагу 2.

Таким образом, даже если ОС разбудит поток "просто так", цикл снова проверит переменную `ready`, увидит `false` и снова отправит поток спать.

Проблема потерянного уведомления (Lost Wakeup)

Почему так важно изменять разделяемую переменную (`ready = true`) под мьютексом? Рассмотрим сценарий без мьютекса:

Сценарий гонки (Race Condition)

```
// Поток 1 (Producer)          // Поток 2 (Consumer)
                               while (!ready) {
ready = true;                      // 1. Прочитал ready (false)
cv.notify_one();                   // 2. Отправил сигнал (в пустоту)
                                   cv.wait(lk); // 3. Уснул навечно
                               }
```

Если уведомление отправляется в момент между проверкой условия (шаг 1) и входом в состояние сна (шаг 3), оно "теряется". Условная переменная не хранит состояние (в отличие от Семафора или Event в Windows), она работает как мгновенный сигнал. Если в момент свистка на платформе никого не было, никто этот свисток не услышит и позже не узнает о нем.

Мьютекс гарантирует, что проверка условия и переход в режим ожидания являются атомарной последовательностью относительно изменения условия.

Стратегии уведомления: notify_one vs notify_all

`std::condition_variable` предоставляет два метода пробуждения:

- `notify_one()`: Будит *один* из ожидающих потоков. Какой именно — не определено стандартом (зависит от планировщика). Используется, когда событие подразумевает эксклюзивную обработку (один элемент в очереди — один рабочий поток).
- `notify_all()`: Будит *все* ожидающие потоки. Используется, когда событие касается всех (например, сигнал остановки приложения или открытие "ворот" барьера).

Использование `notify_all()` в сценарии, когда задачу может выполнить только один поток, приводит к проблеме **Thundering Herd** ("Эффект разорвавшейся бомбы" или "Стадо бизонов"). Представьте 100 потоков, ждущих на одном мьютексе. При `notify_all()` все 100 просыпаются. Все 100 пытаются захватить один и тот же мьютекс. Один выигрывает, 99 блокируются снова (теперь уже на мьютексе, а не на CV). Это вызывает огромный всплеск переключений контекста и деградацию производительности.

Оптимизация: Уведомление вне блокировки

Рассмотрим внимательно код продюсера:

```
{
    std::lock_guard<std::mutex> lk(m);
    ready = true;
    cv.notify_one(); // (A) Уведомление под мьютексом
}
```

В варианте (A) происходит следующее:

1. Продюсер будит консьюмера (сигнал внутри ядра).
2. Консьюмер просыпается в ядре, планировщик ставит его на исполнение.
3. Консьюмер пытается выйти из `wait`, для чего ему нужно захватить мьютекс `m`.
4. Но мьютекс `m` все еще захвачен продюсером!
5. Консьюмер снова блокируется, теперь уже ожидая освобождения мьютекса.
6. Продюсер выходит из скоупа, освобождает мьютекс.
7. Консьюмер снова просыпается и захватывает мьютекс.

Это называется "Hurry up and Wait". Чтобы избежать лишнего переключения контекста, можно вынести уведомление из-под мьютекса:

Оптимизированный Producer

```
{
    std::lock_guard<std::mutex> lk(m);
    ready = true;
} // Мьютекс освобожден
cv.notify_one(); // Уведомление
```

Теперь, когда консьюмер проснется, мьютекс будет свободен, и он сможет захватить его сразу.

На заметку

Эта оптимизация безопасна в большинстве случаев, но требует осторожности, если объект `cv` может быть уничтожен сразу после разблокировки мьютекса (например, если ожидающий поток отвечает за удаление структуры данных, содержащей CV). Однако в стандартных сценариях долгоживущих очередей или пулов потоков это рекомендуемый паттерн.

В следующей главе мы используем эти знания для построения более высокоуровневых примитивов синхронизации, таких как События и Семафоры.

Глава 48

Построение абстракций синхронизации: Event, Semaphore, Latch

Механизм `std::condition_variable`, рассмотренный ранее, является низкоуровневым строительным блоком. В прикладном коде прямое использование связки `mutex + cv + bool` часто приводит к дублированию кода и ошибкам.

Для решения типовых задач синхронизации строятся высокоуровневые абстракции, инкапсулирующие управление состоянием и блокировками. В этой главе мы реализуем три фундаментальных примитива: Событие (Event), Семафор (Semaphore) и Защелку (Latch), разбирая их внутреннюю механику и сценарии использования.

Событие (Event)

Абстракция **Event** (в терминологии Windows API) или "одноразовый сигнал" позволяет одному потоку уведомить другой (или несколько других) о наступлении определенного факта. В отличие от сырой условной переменной, класс `Event` хранит свое состояние внутри.

Существует два основных типа событий:

- **Manual Reset Event:** После перехода в сигнальное состояние остается в нем до явного сброса. Все потоки, пришедшие в `Wait`, проходят сквозь него немедленно.
- **Auto Reset Event:** После пробуждения одного потока событие автоматически сбрасывается в несигнальное состояние.

Реализуем вариант с ручным сбросом (Manual Reset), так как он наиболее наглядно демонстрирует работу с разделяемым флагом.

Реализация класса Event

```
class Event {
public:
    // Перевод события в сигнальное состояние
    void Signal() {
        {
            // Захват мьютекса обязателен для изменения флага
            std::lock_guard lock{mtx_};
            ready_ = true;
        }
```

```
        // Оптимизация: уведомление после разблокировки мьютекса
        // позволяет проснувшимся потокам сразу захватить его
        cv_.notify_all();
    }

    // Блокирующее ожидание события
    void Wait() {
        std::unique_lock lock{mtx_};
        // Стандартный паттерн ожидания с предикатом
        cv_.wait(lock, [this] { return ready_; });
    }

    // Неблокирующая проверка состояния
    bool Ready() {
        std::lock_guard lock{mtx_};
        return ready_;
    }

    // Сброс события (для повторного использования)
    void Reset() {
        std::lock_guard lock{mtx_};
        ready_ = false;
    }

private:
    std::mutex mtx_;
    std::condition_variable cv_;
    bool ready_ = false;
};
```

В методе `Signal` мы используем `notify_all`, так как событие подразумевает, что "информация стала доступна", и это может быть интересно множеству подписчиков.

Семафор (Semaphore)

Если мьютекс обеспечивает **эксклюзивный** доступ (в критической секции находится ровно 1 поток), то семафор управляет пулом ресурсов, допуская одновременный доступ для N потоков.

Семафор содержит внутренний счетчик доступных "токенов" (разрешений).

- **Acquire (P-операция):** Если счетчик > 0, уменьшает его и продолжает работу. Если счетчик $== 0$, поток блокируется.
- **Release (V-операция):** Увеличивает счетчик и будит один из ожидающих потоков.

Сценарий использования: Ограничение количества одновременных подключений к базе данных или внешнему API (Rate Limiting). Например, если внешний сервис разрешает не более 10 параллельных запросов, семафор с начальным значением 10 обеспечит соблюдение этого лимита на уровне архитектуры приложения.

Реализация Counting Semaphore

```
class Semaphore {
public:
    explicit Semaphore(int initial_count) : counter_{initial_count} {}

    // Захват ресурса (аналог lock)
    void acquire() {
        std::unique_lock lock{mutex_};
        // Ждем, пока появятся свободные слоты (counter > 0)
        not_empty_.wait(lock, [this] { return counter_ > 0; });
        --counter_;
    }

    // Освобождение ресурса (аналог unlock)
    void release() {
        {
            std::unique_lock lock{mutex_};
            ++counter_;
        }
        // Будим только один поток, так как освободился всего один слот
        not_empty_.notify_one();
    }

private:
    int counter_;
    std::mutex mutex_;
    std::condition_variable not_empty_;
};
```

Важное отличие от мьютекса: у семафора нет концепции "владельца". Поток, вызвавший `acquire`, не обязан вызывать `release`. Разрешение может быть возвращено любым другим потоком (хотя в классическом RAII-подходе это происходит в деструкторе гарда).

Начиная с C++20, в стандарте доступны `std::counting_semaphore` и `std::binary_semaphore`, которые могут быть реализованы более эффективно (через атомики и футексы без тяжелых мьютексов), но приведенная реализация демонстрирует их логическую суть.

Защелка (Latch)

Защелка (Latch) — это примитив синхронизации обратного отсчета. Она инициализируется значением N. Потоки могут уменьшать это значение (`Arrive`). Потоки могут ждать, пока значение не станет равным нулю (`Wait`). В отличие от `Barrier`, защелка одноразовая: после достижения нуля она не сбрасывается.

Сценарий использования: Ожидание завершения инициализации N сервисов перед запуском основного цикла обработки, или ожидание завершения N параллельных подзадач в алгоритме MapReduce.

Наивная реализация

Latch на базе мьютекса

```
class Latch {
public:
    explicit Latch(int count) : count_{count} {}

    void Arrive() {
        std::lock_guard lock{mtx_};
        if (--count_ == 0) {
            cv_.notify_all(); // Будим всех ожидающих
        }
    }

    void Wait() {
        std::unique_lock lock{mtx_};
        cv_.wait(lock, [this] { return count_ == 0; });
    }

private:
    std::mutex mtx_;
    std::condition_variable cv_;
    int count_;
};
```

В этой реализации есть проблема производительности. Если 100 рабочих потоков завершают работу почти одновременно и вызывают `Arrive`, они создают высокую конкуренцию (contention) на одном мьютексе `mtx_`. Каждое уменьшение счетчика требует захвата и освобождения мьютекса, что дорого.

Оптимизация с использованием std::atomic

Заметим, что мьютекс нам нужен по факту только для одного события: уведомления ожидающего потока (когда счетчик переходит из 1 в 0). Все остальные декременты (из 100 в 99, из 99 в 98 и т.д.) можно выполнить атомарно без полной блокировки.

Оптимизированный Latch

```
#include <atomic>

class Latch {
public:
    explicit Latch(int count) : count_{count} {}

    void Arrive() {
        // Атомарный декремент. fetch_sub возвращает СТАРОЕ значение.
        // Блокировка не захватывается для декремента.
        if (count_.fetch_sub(1) == 1) {
            // Если старое значение было 1, значит теперь стало 0.
            // Именно этот поток "выключил свет" и должен уведомить остальных.

            // Формально notify_all можно вызывать без лока, но
            // в некоторых реализациях CV это может быть безопасно
```

```
            // только если мы уверены в времени жизни объекта.
            // Для строгости стандарта захватим лок или используем atomic notify
            ↪ (C++20).
            std::lock_guard lock{mtx_};
            cv_.notify_all();
        }
    }

    void Wait() {
        std::unique_lock lock{mtx_};
        // Предикат читает атомик
        cv_.wait(lock, [this] { return count_.load() == 0; });
    }

private:
    std::mutex mtx_;
    std::condition_variable cv_;
    std::atomic<int> count_; // Теперь это atomic
};
```

В оптимизированном варианте 99 из 100 потоков выполнят лишь одну процессорную инструкцию `lock xadd` (на x86) и продолжат выполнение, не затрагивая мьютекс и планировщик ОС. Лишь последний поток заплатит цену за синхронизацию. Это существенно снижает накладные расходы при большом количестве воркеров.

> **Важно!**
>
> C++20 вводит `std::latch` и `std::barrier`, которые реализованы максимально эффективно (часто вообще без использования `std::mutex` и `std::condition_variable`, опираясь только на атомики и системные вызовы типа futex).

Глава 49

Управление потоками: Blocking Queue и Thread Pool

В предыдущих главах мы рассматривали механизмы синхронизации на примере взаимодействия небольшого количества потоков. Однако в реальных высоконагруженных системах создание нового потока (`std::thread`) на каждую задачу является архитектурной ошибкой.

Стоимость создания потока в современных операционных системах не является нулевой. Она включает в себя:

- Аллокацию стека (обычно 1–8 МБ адресного пространства).
- Создание структур данных ядра (Kernel Thread Control Block).
- Системные вызовы для регистрации потока в планировщике.

Замеры показывают, что запуск потока может занимать от десятков до сотен микросекунд. Если полезная нагрузка задачи сопоставима с этим временем (например, вычисление квадратного корня или короткий сетевой запрос), система будет тратить больше времени на администрирование потоков, чем на вычисления.

Решением является паттерн **Thread Pool** (Пул потоков): создание фиксированного набора потоков при старте приложения и переиспользование их для выполнения множества задач. Центральным элементом этой архитектуры выступает **Blocking Queue** (Блокирующая очередь).

Блокирующая очередь (Blocking Queue)

Блокирующая очередь — это потокобезопасный буфер, реализующий паттерн Producer-Consumer.

- Метод `Put` добавляет элемент в очередь и уведомляет ожидающие потоки.
- Метод `Take` извлекает элемент. Если очередь пуста, вызывающий поток блокируется до появления данных.

Ключевой сложностью при реализации очереди является не сама синхронизация доступа (она тривиальна через мьютекс), а механизм корректного завершения работы (Graceful Shutdown).

Проблема остановки (Shutdown)

Рассмотрим ситуацию, когда приложение завершается. Пул потоков должен остановиться. Если мы просто вызовем деструкторы потоков или `std::terminate`, мы рискуем прервать задачи на середине или получить утечки ресурсов. Нам нужен механизм, который сообщает потокам-потребителям: "Больше данных не будет, доделывайте текущую работу и выходите".

Этот механизм реализуется через внутренний флаг `stopped_` и изменение сигнатуры метода `Take`. Вместо `T` он должен возвращать `std::optional<T>`:

- `std::nullopt` — сигнал о том, что очередь закрыта и пуста.
- `Value` — валидная задача для исполнения.

Реализация UnboundedBlockingQueue

Ниже представлена реализация неограниченной блокирующей очереди с поддержкой механизма остановки.

Реализация блокирующей очереди

```
template <typename T>
class UnboundedBlockingQueue {
public:
    bool Put(T value) {
        std::lock_guard<std::mutex> guard(mutex_);
        if (stopped_) {
            return false; // Нельзя добавлять в закрытую очередь
        }
        buffer_.push_back(std::move(value));
        not_empty_.notify_one(); // Будим одного воркера
        return true;
    }

    std::optional<T> Take() {
        std::unique_lock<std::mutex> guard(mutex_);

        // КРИТИЧЕСКИЙ МОМЕНТ: Предикат ожидания
        // Мы ждем, пока:
        // 1. Очередь остановлена (stopped_ == true)
        // ИЛИ
        // 2. В очереди есть данные (!buffer_.empty())
        not_empty_.wait(guard, [this] {
            return stopped_ || !buffer_.empty();
        });

        // После пробуждения проверяем:
        // Если очередь остановлена И пуста -> возвращаем nullopt
        if (stopped_ && buffer_.empty()) {
            return std::nullopt;
        }

        // Иначе забираем элемент
        T result = std::move(buffer_.front());
        buffer_.pop_front();
```

```
        return result;
    }

    // Graceful Shutdown: доработать оставшиеся задачи
    void Close() {
        CloseImpl(/*clear=*/false);
    }

    // Hard Shutdown: выбросить все задачи и остановиться
    void Cancel() {
        CloseImpl(/*clear=*/true);
    }

private:
    void CloseImpl(bool clear) {
        std::lock_guard<std::mutex> guard(mutex_);
        stopped_ = true;
        if (clear) {
            buffer_.clear();
        }
        // Важно: будим ВСЕХ, чтобы они проснулись, увидели stopped_
        // и корректно завершили свои циклы
        not_empty_.notify_all();
    }

private:
    std::mutex mutex_;
    std::condition_variable not_empty_;
    bool stopped_{false};
    std::deque<T> buffer_; // std::deque эффективнее vector для FIFO
};
```

Важно!

Обратите внимание на вызов `not_empty_.notify_all()` в методе `CloseImpl`. Если 10 потоков спят в методе Take, и мы закроем очередь, нам нужно разбудить их всех, чтобы они могли проверить флаг `stopped_` и вернуть `std::nullopt`. Если использовать `notify_one`, 9 потоков останутся спать навечно, и программа зависнет при выходе (deadlock при `join`).

Пул потоков (Thread Pool)

Пул потоков представляет собой владельца рабочих потоков (Workers) и очереди задач. Для хранения произвольных задач используется `std::function<void()>`. Это техника стирания типа (Type Erasure), позволяющая хранить в одном контейнере лямбда-выражения, функторы и указатели на функции с разным контекстом захвата, при условии, что они имеют сигнатуру вызова `void()`.

Реализация ThreadPool

```cpp
#include <vector>
#include <thread>
#include <functional>

class ThreadPool {
public:
    using Task = std::function<void()>;

    explicit ThreadPool(size_t num_threads) {
        workers_.reserve(num_threads);
        for (size_t i = 0; i < num_threads; ++i) {
            // Запускаем потоки, каждый из которых исполняет метод Run
            workers_.emplace_back([this] { Run(); });
        }
    }

    // Добавление задачи
    void Submit(Task task) {
        queue_.Put(std::move(task));
    }

    // Явное ожидание завершения
    void Join() {
        queue_.Close(); // Сообщаем потокам о закрытии
        for (auto& t : workers_) {
            if (t.joinable()) {
                t.join(); // Ждем завершения каждого потока
            }
        }
    }

    // Деструктор обеспечивает безопасность (RAII)
    ~ThreadPool() {
        Join();
    }

private:
    // Цикл рабочего потока
    void Run() {
        // Цикл работает, пока Take() возвращает значение
        while (auto task = queue_.Take()) {
            try {
                (*task)(); // Выполняем задачу
            } catch (...) {
                // В простейшем пуле исключения в задачах
                // не должны убивать рабочий поток.
                // В реальном коде здесь нужно логирование.
            }
        }
        // Выход из цикла означает, что очередь закрыта и пуста
    }

private:
```

```
    UnboundedBlockingQueue<Task> queue_;
    std::vector<std::thread> workers_;
};
```

Анализ архитектуры

Данная реализация демонстрирует классическую схему "One Queue - Multiple Workers".

Достоинства

1. **Балансировка нагрузки:** Свободный поток сразу забирает следующую задачу. Нет простоя, пока есть работа.
2. **Простота:** Вся синхронизация инкапсулирована в `BlockingQueue`. Код пула минималистичен.

Недостатки и ограничения

1. **Contention (Конкуренция):** Все потоки борются за один мьютекс внутри очереди. При очень большом количестве потоков (>32-64) и коротких задачах мьютекс становится узким местом. В таких случаях используют *Work Stealing Queue* (очередь на каждый поток + кража задач).
2. **Исключения:** Если задача выбросит исключение, которое не будет поймано внутри `Run`, функция потока завершится аварийно, и `std::terminate` убьет все приложение. Поэтому вызов задачи обернут в `try-catch`.
3. **Блокирующие задачи:** Если все потоки пула заняты задачами, которые ожидают ввода-вывода или других событий (например, спят), пул встанет (Thread Starvation). Для блокирующих операций рекомендуется использовать отдельные пулы или асинхронный I/O.

Резюме раздела

Пул потоков с блокирующей очередью — это стандарт де-факто для параллельной обработки CPU-bound задач в C++. Он позволяет амортизировать накладные расходы на создание потоков и контролировать уровень параллелизма в системе.

Часть X

Лекция 10 – Advanced thread

Глава 50

Anatomy of a Thread: Cost, Kernel & Scheduling

Многопоточность в C++ — это не просто абстракция `std::thread`. За каждым объектом стандартной библиотеки скрывается сложный механизм операционной системы, имеющий свою цену в тактах процессора и байтах оперативной памяти. Понимание физического устройства потока, алгоритмов планировщика ядра (OS Scheduler) и накладных расходов на переключение контекста является фундаментом для построения высоконагруженных систем.

В этой главе мы деконструируем понятие "поток", разберем стоимость его создания и эксплуатации, а также проанализируем архитектурные ошибки, возникающие при наивном использовании примитивов стандарта.

Физическая структура потока

> **Контекст исполнения (Execution Context)**
>
> Поток (thread) физически представляет собой совокупность состояния регистров процессора и выделенного региона памяти под стек. В многопоточной среде все потоки одного процесса разделяют общее адресное пространство (кучу, сегмент кода, глобальные переменные), но имеют изолированный стек и набор регистров.

С точки зрения ядра Linux, поток — это "легковесный процесс" (LWP — Light Weight Process). Для планировщика нет принципиальной разницы между процессом и потоком, кроме того факта, что потоки делят таблицу страниц памяти (Page Table).

Минимальный набор данных, определяющий поток, включает:

1. **Регистры общего назначения (GPR):** RAX, RBX, RCX и т.д. в архитектуре x86_64.
2. **Указатель инструкций (Instruction Pointer, RIP):** Адрес текущей исполняемой команды.
3. **Указатель стека (Stack Pointer, RSP):** Адрес вершины стека данного потока.
4. **Стек (Stack):** Область памяти для локальных переменных, адресов возврата и аргументов функций.

При переключении контекста ядро ОС обязано сохранить текущее состояние регистров уходящего потока в специальную структуру в Kernel Space и загрузить значения регистров приходящего потока.

Стоимость создания потока

Создание потока — операция дорогая. В отличие от вызова функции, который занимает наносекунды (просто `call` и `push` в стек), создание потока (`std::thread`) требует системного вызова (`clone` в Linux), выделения структур ядра и настройки виртуальной памяти.

Эмпирическая оценка времени создания потока на современном железе составляет **от 5 до 50 микросекунд**. Это на несколько порядков медленнее, чем создание объекта в куче.

Бенчмарк: Thread vs Function

Сравним стоимость выполнения пустой работы через вызов функции и через создание нового потока.

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>

void dummy_work() {
    // Имитация минимальной работы
    volatile int x = 0;
    x++;
}

int main() {
    const int N = 10000;

    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        dummy_work();
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "Function calls: "
              << std::chrono::duration_cast<std::chrono::microseconds>(end -
              ↪  start).count()
              << " us\n";

    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        std::thread t(dummy_work);
        t.join();
    }
    end = std::chrono::high_resolution_clock::now();
    std::cout << "Thread creation: "
              << std::chrono::duration_cast<std::chrono::microseconds>(end -
              ↪  start).count()
              << " us\n";
}
```

Результаты на типичном сервере показывают, что создание потоков занимает в тысячи раз больше времени. Если ваша задача выполняется быстрее, чем время создания потока (единицы микросекунд), использование `std::thread` напрямую приведет к деградации производительности.

Memory Overhead: Стек и Виртуальная память

По умолчанию в Linux размер стека для главного потока и создаваемых потоков (`pthread_create`) составляет **8 МБ**. Это значение можно проверить командой `ulimit -s`.

> **Важно!**
>
> Поток не потребляет 8 МБ физической памяти (RAM) мгновенно. Стек выделяется в *виртуальном адресном пространстве*. Физические страницы (frames) выделяются ядром лениво (Demand Paging) по мере обращения к адресам стека.

Однако, даже виртуальная память — ресурс исчерпаемый. Рассмотрим задачу C10K (обслуживание 10 000 соединений). Если мы используем модель "один поток на соединение" (Thread-per-connection):

$$10\,000 \text{ threads} \times 8 \text{ MB} \approx 80 \text{ GB Virtual Memory}$$

Для 32-битных систем это гарантированный крах (адресное пространство всего 4 ГБ). Для 64-битных систем это создает огромную нагрузку на TLB (Translation Lookaside Buffer) процессора и структуры ядра, управляющие памятью (VMA — Virtual Memory Areas).

Если программа под нагрузкой реально использует стек (глубокая рекурсия или большие буферы на стеке), то потребление физической памяти также станет колоссальным, что приведет к OOM (Out Of Memory) Killer.

Планировщик Linux (CFS)

В современных ОС используется **вытесняющая многозадачность** (preemptive multitasking). Разработчик не контролирует, когда поток начнет исполняться и когда будет прерван. Этим занимается планировщик ядра.

В Linux используется **CFS (Completely Fair Scheduler)**.

Механика работы CFS

CFS моделирует "идеальный многозадачный процессор" на реальном железе.

- Каждому потоку присваивается `vruntime` (virtual runtime) — время, которое поток уже провел на процессоре, взвешенное на его приоритет (nice value).
- Потоки организованы в **красно-черное дерево** (Red-Black Tree), отсортированное по `vruntime`.
- Планировщик всегда выбирает поток с наименьшим `vruntime` (самый левый узел дерева).

Квант времени и Переключение контекста

Минимальный интервал перепланирования (latency target) в Linux может составлять, например, 6-24 мс, но реальные кванты времени динамические.

Переключение контекста (Context Switch) — это не только сохранение регистров. Это:

1. Смена режима процессора (User mode → Kernel mode).
2. Загрязнение кэша инструкций и данных (Cache Pollution). Когда новый поток начинает исполнение, нужные ему данные, скорее всего, отсутствуют в L1/L2 кэшах.
3. Накладные расходы на обновление структур планировщика.

На заметку

Типичный сервер может иметь 256 логических ядер (например, AMD EPYC). Балансировка нагрузки и миграция потоков между NUMA-нодами в таких условиях — сложнейшая задача. Миграция потока на другое ядро аннулирует его прогретый кэш.

Антипаттерн: Thread per Request

Исторически веб-серверы писались по модели: "Пришел запрос → Создали поток → Обработали → Уничтожили поток".

Почему это плохо в современном HighLoad: 1. **Unbounded Concurrency:** При всплеске трафика (DDoS или хабраэффект) создается бесконечное число потоков. 2. **Thrashing:** Система тратит больше времени на переключение контекста между тысячами потоков, чем на полезную работу. 3. **Latency Tail:** Время ответа начинает непредсказуемо расти из-за очередей в планировщике.

> **Thread Pool (Пул потоков)**
>
> Архитектурный паттерн, при котором создается фиксированное число долгоживущих потоков (обычно равное числу аппаратных ядер), разбирающих задачи из общей очереди.

Это унифицирует место создания потоков и ограничивает их максимальное число, предотвращая перегрузку системы.

Проблемы стандартных абстракций C++

Стандарт C++ предлагает высокоуровневые инструменты, которые часто вводят в заблуждение своей простотой.

std::execution::par (C++17)

Алгоритмы с политиками исполнения, например `std::for_each(std::execution::par, ...)`, обещают автоматическое распараллеливание.

```
std::vector<int> data = ...;
std::for_each(std::execution::par, data.begin(), data.end(), [](int& x) {
    heavy_computation(x);
});
```

Проблема: Стандарт не специфицирует, как именно реализуется параллелизм.

- Реализация может создать новый поток на каждый вызов алгоритма.

- Если вы вызовете такой алгоритм внутри уже запущенного параллельного кода, произойдет комбинаторный взрыв числа потоков (Oversubscription).
- Отсутствует контроль над тем, в каком пуле исполняются задачи.

В продакшен-коде использование `std::execution::par` без глубокого понимания реализации стандартной библиотеки конкретного компилятора (libstdc++, libc++) считается небезопасным.

std::async и std::future

`std::async` может запускать задачу синхронно (в том же потоке) или асинхронно, в зависимости от флагов и реализации. Возвращаемый `std::future` в деструкторе блокирует поток до завершения задачи, что часто приводит к неожиданным последовательным исполнениям там, где ожидалась параллельность.

Резюме раздела

- Создание потока — дорогая операция (>5 мкс). Избегайте частого создания/уничтожения потоков.
- Каждый поток резервирует 8 МБ виртуального адресного пространства. 100k потоков недопустимы.
- Переключение контекста стоит дорого из-за cache misses.
- Используйте Thread Pool для контроля конкурентности.
- Осторожно с `std::async` и `std::execution::par` — они скрывают детали управления потоками, что опасно для HighLoad.

Глава 51

Building a Production-Grade ThreadPool

Осознав стоимость создания потоков и ограничения операционной системы в предыдущей главе, мы приходим к необходимости архитектурного паттерна **ThreadPool**. Идея проста: мы создаем фиксированный набор потоков (worker threads) на старте приложения и переиспользуем их для выполнения множества задач.

Однако реализация надежного, готового к продакшену пула потоков — задача нетривиальная. Она требует глубокого понимания примитивов синхронизации, управления жизненным циклом потоков и обработки исключительных ситуаций. В этой главе мы шаг за шагом спроектируем и реализуем ThreadPool на современном C++.

Фундамент: Блокирующая очередь (Blocking Queue)

Сердцем любого пула потоков является очередь задач. Воркеры (потребители) должны извлекать задачи из очереди, а клиенты (производители) — добавлять их.

Ключевое требование к такой очереди — она должна быть **блокирующей** для потребителей. Если задач нет, поток-воркер не должен крутиться в цикле (`while(empty) `), потребляя 100% CPU (busy wait). Он должен "уснуть" на уровне ядра ОС, освободив ресурсы процессора, и проснуться только тогда, когда появится новая задача или поступит сигнал остановки.

Для реализации этого механизма нам потребуются три компонента:

1. `std::deque<T>` или `std::queue<T>` — контейнер для хранения данных.
2. `std::mutex` — для обеспечения взаимного исключения (mutual exclusion) при доступе к контейнеру.
3. `std::condition_variable` — для организации ожидания и уведомления потоков.

Реализация UnboundedBlockingQueue

Рассмотрим реализацию неограниченной (unbounded) очереди. "Неограниченная" означает, что метод `Push` никогда не блокируется из-за переполнения (пока есть оперативная память).

```
#include <mutex>
#include <condition_variable>
```

```cpp
#include <deque>
#include <optional>

template <typename T>
class UnboundedBlockingQueue {
public:
    void Push(T item) {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            buffer_.push_back(std::move(item));
        }
        // Уведомляем один спящий поток (если есть)
        not_empty_.notify_one();
    }

    std::optional<T> Pop() {
        std::unique_lock<std::mutex> lock(mutex_);

        // Ожидаем, пока очередь не станет непустой ИЛИ не будет закрыта
        not_empty_.wait(lock, [this]() {
            return !buffer_.empty() || is_closed_;
        });

        if (buffer_.empty() && is_closed_) {
            return std::nullopt; // Очередь закрыта и пуста
        }

        T item = std::move(buffer_.front());
        buffer_.pop_front();
        return item;
    }

    void Close() {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            is_closed_ = true;
        }
        // Будим ВСЕ потоки, чтобы они увидели флаг is_closed_
        not_empty_.notify_all();
    }

private:
    std::deque<T> buffer_;
    std::mutex mutex_;
    std::condition_variable not_empty_;
    bool is_closed_ = false;
};
```

Разбор механики синхронизации

1. Lock Guard vs Unique Lock

В методе `Push` используется `std::lock_guard`, так как нам нужно просто захватить мьютекс на время скоупа. В методе `Pop` используется `std::unique_lock`. Это критически важ-

но, так как `std::condition_variable::wait` требует именно `unique_lock`. Внутри `wait` атомарно происходит следующее: 1. Мьютекс разблокируется. 2. Поток переводится в режим ожидания (sleep). При пробуждении мьютекс снова захватывается.

2. Spurious Wakeups (Ложные пробуждения)

Обратите внимание на конструкцию вызова `wait`:

```
not_empty_.wait(lock, [this]() {
    return !buffer_.empty() || is_closed_;
});
```

Это эквивалентно циклу:

```
while (!(!buffer_.empty() || is_closed_)) {
    not_empty_.wait(lock);
}
```

> **Важно!**
>
> Никогда не используйте `if` с `wait`. POSIX Threads (и стандарт C++) допускают **Spurious Wakeups** — ситуация, когда поток просыпается без вызова `notify`. Если не проверить условие (предикат) повторно в цикле, поток может попытаться извлечь элемент из пустой очереди, что приведет к UB или исключению.

3. Graceful Shutdown (Закрытие очереди)

Метод `Close` устанавливает флаг `is_closed_` и вызывает `notify_all()`. Это необходимо для корректного завершения работы. Представим ситуацию: 5 воркеров спят в методе `Pop`, ожидая задач. Если мы просто перестанем посылать задачи, воркеры никогда не проснутся и программа зависнет при выходе. Вызов `notify_all()` будит их всех, они проверяют предикат `is_closed_`, видят `true` и корректно выходят, возвращая `std::nullopt`.

Архитектура ThreadPool

Теперь, имея надежную очередь, мы можем реализовать сам пул.

Интерфейс задач

Для простоты будем считать, что задача — это `std::function<void()>`. Это стирает тип функтора (лямбда, указатель на функцию, объект с `operator()`), позволяя хранить разнородные задачи в одной очереди.

> **На заметку**
>
> В реальных высокопроизводительных системах `std::function` может быть заменена на кастомную реализацию delegate без аллокаций памяти (SBO — Small Buffer Optimization), так как `std::function` может аллоцировать память в куче.

Полная реализация ThreadPool

```cpp
#include <vector>
#include <thread>
#include <functional>
#include <iostream>

// Используем UnboundedBlockingQueue, описанную выше
using Task = std::function<void()>;

class ThreadPool {
public:
    explicit ThreadPool(size_t num_threads) {
        Start(num_threads);
    }

    ~ThreadPool() {
        Stop();
    }

    // Запрещаем копирование и перемещение для простоты
    ThreadPool(const ThreadPool&) = delete;
    ThreadPool& operator=(const ThreadPool&) = delete;

    void Submit(Task task) {
        queue_.Push(std::move(task));
    }

private:
    void Start(size_t num_threads) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this]() {
                WorkerRoutine();
            });
        }
    }

    void Stop() {
        // 1. Сообщаем очереди, что больше задач не будет
        // и будим всех спящих воркеров.
        queue_.Close();

        // 2. Ждем завершения каждого потока.
        for (auto& worker : workers_) {
            if (worker.joinable()) {
                worker.join();
            }
        }
    }

    void WorkerRoutine() {
        while (true) {
            // Блокируемся в ожидании задачи
            auto task_opt = queue_.Pop();
```

```
53
54              // Если вернулся nullopt, значит очередь закрыта и пуста -> выход
55              if (!task_opt.has_value()) {
56                  break;
57              }
58
59              auto& task = task_opt.value();
60              try {
61                  task(); // Выполнение задачи
62              } catch (const std::exception& e) {
63                  // Критически важно ловить исключения!
64                  // Иначе исключение покинет поток и вызовет std::terminate
65                  std::cerr << "Worker exception: " << e.what() << std::endl;
66              } catch (...) {
67                  std::cerr << "Worker unknown exception" << std::endl;
68              }
69          }
70      }
71
72  private:
73      std::vector<std::thread> workers_;
74      UnboundedBlockingQueue<Task> queue_;
75  };
```

Анализ Corner Cases и Ошибок

1. Исключения в воркерах

В методе `WorkerRoutine` блок `try-catch` обязателен. Стандарт C++ гласит: если исключение покидает функцию верхнего уровня потока (в данном случае лямбду, переданную в конструктор `std::thread`), вызывается `std::terminate()`, который аварийно завершает **весь процесс**, а не только один поток.

Без `try-catch`, одна сбойная задача ("poison pill") убила бы всё приложение.

2. Порядок остановки (Destruction Order)

В деструкторе (или методе `Stop`) порядок действий критичен:

1. Сначала **Queue Close**. Это устанавливает флаг и будит потоки.
2. Только потом **Thread Join**.

Если поменять местами или забыть `Queue Close`, вызов `worker.join()` заблокируется навечно (deadlock), так как главный поток будет ждать завершения воркера, а воркер будет спать внутри `queue.Pop()`, ожидая задач, которые никогда не придут.

3. Невозможность принудительного убийства (Thread Cancellation)

Новички часто спрашивают: "Как убить задачу, которая зависла?". В стандартном C++ (да и в большинстве ОС) безопасного способа принудительно остановить поток извне **не существует**.

- `pthread_cancel` или `TerminateThread` (WinAPI) существуют, но их использование опасно.
- Если убить поток, пока он держит `std::mutex`, этот мьютекс останется заблокированным навсегда (abandoned mutex). Любой другой поток, попытавшийся его захватить, зависнет.
- Также не будут вызваны деструкторы стековых объектов (RAII не сработает), что приведет к утечкам ресурсов.

Единственный способ прервать задачу — кооперативная отмена, когда сама задача периодически проверяет флаг `std::atomic<bool> should_stop`.

4. Data Races при проверке состояния

Рассмотрим распространенную ошибку реализации очереди без мьютексов при проверке на пустоту:

```
// ОШИБКА!
bool IsEmpty() const {
    return buffer_.empty(); // Чтение без мьютекса!
}
```

`std::deque::empty()` не является атомарной операцией. Чтение состояния контейнера параллельно с его модификацией (в методе `Push` из другого потока) — это классическая **Data Race**, приводящая к Undefined Behavior. Любой доступ к разделяемым данным должен быть защищен тем же мьютексом, что и операции записи.

Резюме раздела

Мы построили корректный ThreadPool, который:

- Использует фиксированное число потоков.
- Эффективно ожидает задачи без busy-wait (благодаря `condition_variable`).
- Корректно обрабатывает завершение работы (Graceful Shutdown).
- Устойчив к исключениям внутри пользовательских задач.

Этот код является минимальным базисом. В реальных системах (например, в игровых движках или веб-серверах) его усложняют, добавляя приоритеты задач, work-stealing (кражу задач между очередями разных потоков) и локальные очереди для уменьшения конкуренции за единый мьютекс.

Глава 52

User-Space Concurrency: Implementing Fibers

В предыдущих главах мы работали с потоками операционной системы (`std::thread`). Мы выяснили, что это тяжеловесные объекты: переключение контекста требует вмешательства ядра (syscall), загрязняет кэш и занимает микросекунды.

Для систем, требующих обработки сотен тысяч одновременных соединений (C100K+), модель "один поток на соединение" становится узким местом. Решением является **User-Space Concurrency** — реализация многозадачности силами самого приложения, без участия ядра ОС в планировании конкретных задач.

В этой главе мы реализуем библиотеку **Файберов** (Fibers, также известны как Green Threads или Coroutines) с нуля. Мы опустимся на самый низкий уровень — уровень ассемблера и регистров процессора.

Кооперативная многозадачность

В отличие от вытесняющей (preemptive) многозадачности, где ОС насильно прерывает потоки по таймеру, файберы используют **кооперативную** (cooperative) модель.

> **Файбер (Fiber)**
>
> Легковесный поток исполнения, управляемый планировщиком в пространстве пользователя (User Space). Переключение между файберами происходит только тогда, когда сам файбер явно отдает управление (вызывает `Yield`).

Модель M:N Threading: Мы запускаем M файберов поверх N физических потоков ОС (обычно N равно числу ядер). Если файбер блокируется (ждет I/O или мьютекс), физический поток не блокируется, а переключается на выполнение другого файбера.

Архитектура Файбера

Чтобы превратить функцию в "поток", который можно остановить и продолжить, нам нужно сохранить его состояние. Что определяет состояние исполнения?

1. **Стек (Stack):** Локальные переменные и цепочка вызовов. У файбера должен быть свой собственный стек, выделенный в куче.

2. **Регистры процессора (Context):** Значения переменных, находящихся в данный момент на "верхушке" вычислений.
3. **Указатель инструкций (RIP):** Место в коде, где мы остановились.

Инфраструктура: Стек

Стек — это просто непрерывный блок памяти. Стек растет "вниз" (от старших адресов к младшим) на архитектуре x86_64.

```
class Stack {
public:
    explicit Stack(size_t size)
        : stack_{std::make_unique<char[]>(size)}
        , top_{stack_.get() + size} // Указатель на КОНЕЦ массива (старший адрес)
    {
        // Выравнивание стека по границе 16 байт (требование ABI)
        top_ = reinterpret_cast<char*>(
            reinterpret_cast<uintptr_t>(top_) & ~0xF
        );
    }

    void* Top() const { return top_; }
private:
    std::unique_ptr<char[]> stack_;
    char* top_;
};
```

Deep Dive: Assembly & Calling Conventions

Самая сложная часть реализации — переключение контекста. C++ не имеет стандартных средств для прямой записи в регистры `RSP` (Stack Pointer) и `RIP` (Instruction Pointer). Нам придется использовать ассемблер.

Согласно **System V AMD64 ABI** (соглашение о вызовах в Linux/Unix), регистры делятся на две группы:

- **Caller-saved (Volatile):** `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`-`r11`. Функция может менять их как угодно. Если вызывающему коду они нужны, он сам их сохраняет *перед* вызовом.
- **Callee-saved (Non-volatile):** `rbx`, `rbp`, `r12`-`r15`. Вызываемая функция **обязана** сохранить их значения и восстановить перед возвратом.

> **Важно!**
>
> Для реализации переключения контекста нам достаточно сохранять только **Callee-saved** регистры. Почему? Потому что переключение контекста выглядит как вызов функции `SaveContext`. Компилятор C++, генерирующий код вызова этой функции, *уже* сохранил все нужные ему Volatile-регистры на стеке (или они ему не нужны). Наша задача — сохранить только те регистры, которые компилятор ожидает увидеть неизменными после возврата из функции.

Структура контекста:

```
struct Context {
    void* rip; // Instruction Pointer (адрес возврата)
    void* rsp; // Stack Pointer

    // Callee-saved регистры
    void* rbp; // Base Pointer
    void* rbx;
    void* r12;
    void* r13;
    void* r14;
    void* r15;
};
```

Магия переключения: SaveContext и JumpContext

Мы реализуем две функции на ассемблере.

1. SaveContext

Сохраняет текущее состояние в структуру `Context` и возвращает `0`. Сигнатура: `extern "C" int SaveContext(Context* ctx);`

> **На заметку**
>
> В System V ABI первый аргумент функции (в данном случае указатель `ctx`) передается в регистре `RDI`. Возвращаемое значение — в `RAX`.

```
.global SaveContext
SaveContext:
    // RDI содержитуказательна   Context

    // 1. Сохраняемадресвозврата   (RIP).
    // Текущий RIP лежитнаверхушкестекаположила    ( инструкция call).
    mov rax, [rsp]
    mov [rdi + 0], rax  // ctx->rip = [rsp]

    // 2. Сохраняемуказательстека   (RSP).
    // Намнужен  RSP довызова  call, поэтому +8 байтпропускаем ( адресвозврата )
    lea rax, [rsp + 8]
    mov [rdi + 8], rax  // ctx->rsp = rsp + 8

    // 3. Сохраняемостальныерегистры
    mov [rdi + 16], rbp
    mov [rdi + 24], rbx
    mov [rdi + 32], r12
    mov [rdi + 40], r13
    mov [rdi + 48], r14
    mov [rdi + 56], r15

    // 4. Возвращаем 0 (ESaveContextResult::Saved)
    xor rax, rax
    ret
```

2. JumpContext

Восстанавливает состояние из структуры и передает управление. Эта функция **не возвращает управление** туда, откуда её вызвали. Она "возвращается" туда, где был сохранен контекст. Сигнатура: `extern "C" void JumpContext(Context* ctx);`

```
.global JumpContext
JumpContext:
    // RDI содержитуказательна   Context

    // 1. Восстанавливаем Callee-saved регистры
    mov rbp, [rdi + 16]
    mov rbx, [rdi + 24]
    mov r12, [rdi + 32]
    mov r13, [rdi + 40]
    mov r14, [rdi + 48]
    mov r15, [rdi + 56]

    // 2. ВосстанавливаемСтек ! Самый( важныймомент )
    mov rsp, [rdi + 8]

    // 3. Подготавливаем "возврат".
    // Мыхотим , чтобыпотокпродолжилвыполнениетак    , будто
    // SaveContext толькочтовернулуправление   , носдругимрезультатом   .
    // Загружаемсохраненный  RIP встек , чтобыинструкция  ret егозабрала .

    mov rax, [rdi + 0] // Загружаемцелевой  RIP
    push rax           // Кладемегонастек

    // 4. Устанавливаемвозвращаемоезначение   1 (ESaveContextResult::Resumed)
    mov rax, 1

    // 5. Прыжок!
    ret // Снимет RIP состекаипрыгнеттуда
```

> **Важно!**
>
> Атрибут `__attribute__((returns_twice))` для `SaveContext` критически важен. Он сообщает компилятору, что функция может вернуть управление дважды (как `setjmp`), запрещая агрессивные оптимизации (например, хранение переменных только в регистрах без сброса на стек перед вызовом).

Трамплин и Запуск файбера

Мы научились переключаться. Но как запустить новый файбер в первый раз? У него нет сохраненного контекста.

Мы должны *сфабриковать* контекст вручную.

1. Выделяем стек.
2. Устанавливаем `context.rsp` на вершину этого стека.
3. Устанавливаем `context.rip` на адрес функции-трамплина.

Зачем нужен трамплин? Мы не можем просто указать `rip` на пользовательскую функцию. Когда пользовательская функция завершится (`ret`), процессору нужно знать, куда вернуться. В пустом стеке нет адреса возврата → Segfault. Трамплин — это обертка, которая вызывает пользовательский код, а после его завершения вызывает `Scheduler::Terminate()`, корректно завершая жизнь файбера.

```
// static
void Fiber::Trampoline() {
    // Получаем текущий файбер (установленный шедулером перед прыжком)
    Fiber* fiber = GetCurrentFiber();
    fiber->RunUserRoutine(); // Вызов пользовательской лямбды

    // Если мы здесь, значит задача выполнена.
    // Файбер нельзя просто удалить, нужно уйти в шедулер.
    GetCurrentScheduler()->Terminate();
}

Fiber::Fiber(std::function<void()> routine)
    : stack_(kStackSize), routine_(std::move(routine)) {

    context_.rsp = stack_.Top();
    context_.rip = reinterpret_cast<void*>(Trampoline);
}
```

Планировщик (Scheduler)

Планировщик управляет очередью готовых файберов (`Runnable`) и распределяет процессорное время. Поскольку мы пишем однопоточный планировщик (для простоты), он работает в главном потоке программы.

Intrusive List

Чтобы избежать аллокаций памяти при каждом добавлении файбера в очередь планировщика (что убило бы производительность), мы используем **Интрузивный список**. Файбер сам является узлом списка:

```
class Fiber : public IntrusiveListItem<Fiber> { ... };
```

Это позволяет перемещать файбер между очередями (Ready Queue ↔ Wait Queue мьютекса) простым переписыванием указателей `prev/next` внутри объекта файбера, без `new/delete`.

Цикл планирования

```
void Scheduler::Run() {
    while (!queue_.Empty()) {
        Fiber* fiber = queue_.PopFront();
        current_fiber_ = fiber;

```

```
        // Магия переключения:
        // 1. Сохраняем контекст планировщика в main_context_
        // 2. Загружаем контекст файбера
        Switch(&main_context_, fiber->GetContext());

        // Сюда мы попадаем, когда файбер вернул управление (Yield/Suspend)

        if (fiber->GetState() == EFiberState::Finished) {
            delete fiber;
        } else if (fiber->GetState() == EFiberState::Runnable) {
            queue_.PushBack(fiber);
        }
        // Если Suspended, то ничего не делаем (он хранится в wait queue
        ↪ примитива)
    }
}
```

Примитивы синхронизации: Yield и Suspend

В мире файберов нельзя использовать `std::mutex`, так как он заблокирует весь физический поток, остановив все файберы планировщика. Нам нужны свои примитивы.

Yield (Добровольная уступка)

Файбер говорит: "Я еще не закончил, но готов уступить место другим".

1. Состояние файбера остается `Runnable`.
2. Переключаемся в контекст шедулера.
3. Шедулер кладет файбер в конец очереди.

Suspend (Блокировка)

Используется в мьютексах и condition variables.

1. Файбер меняет состояние на `Suspended`.
2. Переключаемся в контекст шедулера.
3. **Важно:** Шедулер НЕ кладет файбер обратно в очередь `RunQueue`. Файбер "исчезает" для планировщика.
4. Ответственность за возвращение файбера к жизни берет на себя тот, кто его заблокировал (например, `Mutex::Unlock` вызовет `Schedule(fiber)`).

Резюме раздела

Мы реализовали файберы — механизм User-Space многозадачности.

- Мы контролируем стек и регистры вручную.
- Переключение контекста стоит десятки наносекунд (просто `mov` регистров), в отличие от микросекунд у потоков ОС.
- Мы используем только Callee-saved регистры, полагаясь на System V ABI.
- Кооперативная многозадачность требует явного `Yield` или ожидания на примитивах синхронизации, чтобы система была отзывчивой.

Эта технология лежит в основе Goroutines в Go, Coroutines в C++20 и Fiber API в Windows.

Глава 53

Linux Low-Level Sync: The Futex

До сих пор мы рассматривали два полюса синхронизации:

1. **Спинлоки (Spinlocks):** Работают полностью в User Space, используя атомарные инструкции. Они феноменально быстры при отсутствии конкуренции, но при ожидании "сжигают" процессорное время впустую, нагревая воздух.
2. **Тяжелые мьютексы (Kernel Mutexes):** Старые реализации (например, в ранних версиях Linux) всегда обращались к ядру ОС для блокировки. Это экономит CPU (поток спит), но системный вызов (`syscall`) стоит сотни наносекунд, даже если конкуренции нет.

Истина, как всегда, посередине. Большую часть времени мьютекс свободен, и захватывать его нужно быстро (как спинлок). Но если он занят, поток должен честно уснуть (как системный мьютекс).

Именно эту гибридную модель реализует **Futex** (Fast Userspace Mutex) — фундаментальный примитив синхронизации в Linux, на котором построены `std::mutex`, `pthread_mutex` и механизмы синхронизации в JVM и Go runtime.

Философия Futex

Основная идея Futex звучит так: "**Не тревожь ядро, если нет конкуренции**".

> **Futex (Fast Userspace Mutex)**
>
> Механизм, позволяющий потокам синхронизироваться через общую область памяти в пространстве пользователя, прибегая к системным вызовам только в случае необходимости ожидания (блокировки) или пробуждения других потоков.

Технически, futex состоит из двух компонентов:

1. **Aligned Integer в User Space:** Обычная 32-битная переменная (обычно `std::atomic<int32_t>`), хранящая состояние блокировки.
2. **Очередь ожидания в Kernel Space:** Структура внутри ядра, где хранятся спящие потоки, привязанные к физическому адресу этой переменной.

Системный вызов futex()

В Linux нет отдельных сисколлов `futex_wait` или `futex_wake`. Есть один мультиплексированный вызов `sys_futex`. Стандартная библиотека C++ не предоставляет прямого доступа к нему, поэтому для реализации собственных примитивов приходится использовать функцию `syscall`.

```
#include <linux/futex.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <atomic>

// Обертка над системным вызовом
long sys_futex(void* addr1, int op, int val1,
               const struct timespec* timeout, void* addr2, int val3) {
    return syscall(SYS_futex, addr1, op, val1, timeout, addr2, val3);
}
```

Рассмотрим две главные операции.

FUTEX_WAIT

Сигнатура: `futex(addr, FUTEX_WAIT, expected_val, ...)`

Семантика: "Проверь атомарно, что по адресу `addr` все еще лежит значение `expected_val`. Если это так — усыпи текущий поток. Если значение изменилось — верни управление немедленно (код ошибки `EAGAIN`)."

Эта атомарная проверка критически важна. Рассмотрим сценарий гонки без нее (Lost Wakeup Problem):

1. Поток А видит, что мьютекс занят (`val == 1`).
2. Поток А решает уснуть.
3. *В этот момент происходит переключение контекста.*
4. Поток Б освобождает мьютекс (`val = 0`) и вызывает `WAKE`. Но никто еще не спит, уведомление уходит в пустоту.
5. *Управление возвращается к А.*
6. Поток А вызывает "безусловный sleep" и засыпает навсегда, хотя мьютекс уже свободен.

Futex решает это, выполняя проверку значения и засыпание как единую транзакцию внутри ядра (под спинлоком планировщика).

FUTEX_WAKE

Сигнатура: `futex(addr, FUTEX_WAKE, count, ...)`

Семантика: "Разбуди `count` потоков, ожидающих по этому адресу". Обычно используют:

- `count = 1`: аналог `notify_one` (для мьютексов).
- `count = INT_MAX`: аналог `notify_all` (для cond_var или `barrier`).

Реализация Mutex на базе Futex

Напишем простейший мьютекс. Мы будем использовать три состояния переменной `state`:

- **0:** Разблокировано (Unlocked).
- **1:** Заблокировано, нет ожидающих (Locked, no waiters).
- **2:** Заблокировано, есть ожидающие (Locked, with waiters).

Состояние "2" необходимо, чтобы оптимизировать `Unlock`. Если при разблокировке мы видим "1", значит, будить никого не надо, и можно не делать дорогой сисколл `FUTEX_WAKE`.

```
class FutexMutex {
public:
    void lock() {
        int c;
        // 1. Fast Path: Попытка атомарно сменить 0 -> 1.
        // Если успешно, мы захватили мьютекс без системных вызовов.
        if ((c = cmpxchg(0, 1)) != 0) {

            // 2. Slow Path: Мьютекс занят.
            // Если состояние было 1, меняем его на 2 (сигнализируя о наличии ждущего).
            if (c != 2) {
                c = xchg(2);
            }

            // Крутимся в цикле, пока не захватим
            while (c != 0) {
                // Пытаемся уснуть.
                // Ядро усыпит нас ТОЛЬКО если state == 2.
                // Если state изменился (кто-то сделал unlock), мы не уснем.
                sys_futex(&state, FUTEX_WAIT, 2, nullptr, nullptr, 0);

                // Проснулись (или не спали). Пробуем захватить 2 -> 2
                // (фактически просто check, но нам нужно атомарно убедиться)
                // В реальной реализации тут стоит повторить попытку 0 -> 2
                c = xchg(2);
            }
        }
    }

    void unlock() {
        // 1. Fast Path: Атомарно меняем любое значение на 0.
        // fetch_sub(1) было бы оптимизацией, чтобы отличить 1 от 2,
        // но здесь для простоты используем exchange.
        if (state.exchange(0) == 2) {
            // 2. Slow Path: Будим одного ждущего, только если старое значение было 2.
            sys_futex(&state, FUTEX_WAKE, 1, nullptr, nullptr, 0);
        }
    }

private:
    // Вспомогательные обертки над std::atomic
```

```
    int cmpxchg(int expected, int desired) {
        int expected_copy = expected;
        state.compare_exchange_strong(expected_copy, desired);
        return expected_copy;
    }

    int xchg(int desired) {
        return state.exchange(desired);
    }

    std::atomic<int> state{0}; // Требует выравнивания (обычно 4 байта)
};
```

Внутри ядра: Kernel Wait Queues

Что происходит, когда вы зовете `FUTEX_WAIT`?

1. **Hash Table Lookup:** Ядро не может просто взять виртуальный адрес `&state`, так как у разных процессов разные адресные пространства (а futex может быть shared между процессами). Ядро транслирует виртуальный адрес в физический (или в уникальный ключ inode+offset для memory-mapped файлов). 2. По этому ключу вычисляется хэш, и выбирается соответствующая цепочка (bucket) в глобальной хэш-таблице `futex_queues`. 3. **Spinlock:** Блокируется бакет хэш-таблицы. 4. **Atomic Check:** Ядро читает значение из user-space памяти. Если оно не совпадает с `expected`, блокировка снимается, и функция возвращает ошибку. 5. **Sleep:** Если значения совпадают, создается объект ожидания, добавляется в очередь, и поток переводится в состояние `TASK_INTERRUPTIBLE`. Спинлок отпускается, вызывается планировщик (`schedule()`).

Этот механизм гарантирует, что futex является самым эффективным способом блокировки в Linux, сочетая скорость атомиков (в 99% случаев) и корректность системного ожидания при высокой нагрузке.

На заметку

Понимание futex необходимо для следующей главы, так как многие Lock-Free алгоритмы используют его как "fallback mechanism" (механизм отката). Нельзя бесконечно крутиться в CAS-цикле, если ожидание затягивается; рано или поздно поток нужно усыпить.

Глава 54

Lock-Free Programming: Atomic Foundations

В предыдущих главах мы использовали блокировки (мьютексы, футексы), чтобы защитить общие данные. Это подход **пессимистичной** синхронизации: "Я предполагаю, что кто-то помешает мне, поэтому я закрою дверь на замок, прежде чем что-то делать".

Lock-Free программирование — это подход **оптимистичной** синхронизации. Потоки не блокируют друг друга. Вместо этого они пытаются выполнить операцию, и если обнаруживают конфликт (кто-то другой изменил данные быстрее), они повторяют попытку.

Определения и Гарантии

Часто термин "Lock-Free" используют неправильно, называя так любой код без `std::mutex`. Формальное определение основано на гарантиях прогресса системы.

> **Lock-Free (Свобода от блокировок)**
>
> Алгоритм называется Lock-Free, если гарантируется, что хотя бы один поток в системе будет продвигаться вперед (делать полезную работу) за конечное число шагов, даже если другие потоки замедлились или остановились.

> **Wait-Free (Свобода от ожидания)**
>
> Более сильная гарантия. Каждый поток гарантированно завершает свою операцию за конечное число шагов, независимо от действий других потоков.

> **Важно!**
>
> Lock-Free не означает "быстрее". Lock-Free алгоритмы часто потребляют больше CPU из-за циклов повторных попыток (CAS-loops) и могут быть медленнее качественных мьютексов при высокой конкуренции (contention). Их главная цель — устойчивость к остановке потоков (например, если поток убит или прерван планировщиком, он не держит блокировку, мешая другим).

Атомики в C++

Стандарт C++11 ввел модель памяти и тип `std::atomic<T>`.

Операции над атомиками неделимы. Невозможно увидеть атомарную переменную в "частично измененном" состоянии. Основные операции:

- `load()`: Атомарное чтение.
- `store()`: Атомарная запись.
- `exchange(val)`: Записывает новое значение и возвращает старое (RMW — Read-Modify-Write).
- `fetch_add(val)`: Прибавляет значение и возвращает старое.

Compare-And-Swap (CAS)

Фундаментом всех Lock-Free структур данных является операция **Compare-And-Swap**. В C++ она представлена двумя методами: `compare_exchange_weak` и `compare_exchange_strong`.

Сигнатура:

```
bool compare_exchange_strong(T& expected, T desired);
```

Логика (псевдокод, выполняемый атомарно):

```
if (*this == expected) {
    *this = desired;
    return true;
} else {
    expected = *this; // Обновляем expected актуальным значением!
    return false;
}
```

Паттерн CAS-Loop

Рассмотрим простейшую задачу: атомарное изменение переменной по произвольной формуле $X_{new} = f(X_{old})$. Почему нельзя просто написать `atomic = f(atomic.load())`? Потому что между чтением (`load`) и записью (`store`) может вклиниться другой поток и изменить значение. Наш `store` перезатрет его результат. Это классическая гонка (Lost Update).

Правильный подход — использование цикла CAS:

```
std::atomic<int> value{0};

void atomic_update(int operand) {
    int old_val = value.load(); // 1. Снэпшот состояния
    int new_val;
    do {
        // 2. Вычисление нового значения на основе локальной копии
        new_val = old_val + operand; // Здесь может быть любая функция f(old_val)

        // 3. Попытка публикации
        // Если value все еще равно old_val, то записать new_val.
```

```
        // Если нет (кто-то успел изменить value), то обновить old_val и вернуть
        ↪ false.
    } while (!value.compare_exchange_weak(old_val, new_val));
}
```

Strong vs Weak CAS

- `compare_exchange_strong`: Гарантирует успех, если значение равно ожидаемому. Обычно реализуется инструкцией процессора (например, `LOCK CMPXCHG` на x86).
- `compare_exchange_weak`: Разрешает **Spurious Failures** (ложные отказы). Может вернуть `false`, даже если значение равно ожидаемому.

Зачем нужен `weak`? На некоторых архитектурах (ARM, PowerPC) CAS реализуется через пару инструкций `LL/SC` (Load-Linked / Store-Conditional). Если между ними произошло прерывание или обращение к кэш-линии, `SC` может не сработать. `weak` версия позволяет избежать лишних циклов внутри самой инструкции CAS, перекладывая ответственность на внешний цикл пользователя. В циклах (`while`) всегда предпочтительнее использовать `weak`, так как цикл все равно перезапустится.

Пример: Lock-Free Stack (Treiber Stack)

Реализуем классический Lock-Free стек (LIFO). Это одна из простейших структур, так как все изменения происходят только на верхушке (`head`).

Узлы стека:

```
template <typename T>
struct Node {
    T data;
    Node* next;

    Node(const T& d) : data(d), next(nullptr) {}
};
```

Операция Push

```
std::atomic<Node<T>*> head{nullptr};

void Push(const T& data) {
    Node<T>* new_node = new Node<T>(data);

    // Фаза 1: Читаем текущую голову
    new_node->next = head.load(std::memory_order_relaxed);

    // Фаза 2: Пытаемся подменить голову на нашу новую ноду
    // Если head изменился (кто-то успел сделать push/pop),
    // CAS обновит new_node->next актуальным значением head
    // и мы попробуем снова.
```

```
    while (!head.compare_exchange_weak(new_node->next, new_node,
                                       std::memory_order_release,
                                       std::memory_order_relaxed)) {
        // Тело цикла пустое, вся работа в условии
    }
}
```

Это идеально работает. Новый узел невидим для других потоков, пока CAS не увенчается успехом. Как только CAS прошел, узел мгновенно становится новой головой.

Операция Pop и утечки памяти

```
std::optional<T> Pop() {
    Node<T>* old_head = head.load(std::memory_order_acquire);

    while (old_head && !head.compare_exchange_weak(old_head, old_head->next,
                                                   std::memory_order_acquire,
                                                   std::memory_order_relaxed)) {
        // Если CAS не прошел, old_head обновился. Проверяем, не стал ли он
        ↪ nullptr.
    }

    if (!old_head) return std::nullopt; // Стек пуст

    T res = old_head->data;

    // ПРОБЛЕМА: Когда удалять old_head?
    // delete old_head; // <--- ОПАСНО!

    return res;
}
```

Проблема безопасного удаления памяти

В коде выше строка `delete old_head` закомментирована не случайно. Это главная проблема Lock-Free структур в C++.

Представим сценарий: 1. Поток А читает `head` (адрес 0x100) в локальную переменную `old_head`. 2. Поток А засыпает перед CAS. 3. Поток Б делает `Pop`, успешно меняет `head`, забирает ноду 0x100 и вызывает `delete` (возвращает память ОС). 4. Поток А просыпается. Он пытается обратиться к `old_head->next` внутри CAS или просто сравнить значение. 5. **Use-After-Free:** Память по адресу 0x100 уже освобождена и, возможно, выделена под другой объект. Чтение `old_head->next` приводит к Segfault или чтению мусора.

В языках с Garbage Collector (Java, Go, C#) этой проблемы нет — GC не удалит объект, пока на него есть ссылки (включая локальную ссылку в потоке А). В C++ нам нужно реализовать собственный механизм управления памятью для Lock-Free.

Эта проблема и её решения (Hazard Pointers, RCU) настолько объемны, что им посвящена отдельная глава.

Спинлок (Spinlock) на атомиках

Для полноты картины реализуем примитивнейший мьютекс (спинлок) на `std::atomic_flag`. Это единственный тип, который гарантированно является Lock-Free на всех архитектурах (даже самых простых).

```
class Spinlock {
    // ATOMIC_FLAG_INIT устанавливает флаг в состояние false (clear)
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        // test_and_set устанавливает флаг в true и возвращает предыдущее
        ↪ значение.
        // Мы крутимся, пока предыдущее значение было true (значит, было занято).
        while (flag.test_and_set(std::memory_order_acquire)) {
            // Оптимизация для процессора: "я в цикле ожидания"
            // Снижает энергопотребление и позволяет SMT-потокам работать
            ↪ быстрее.
#if defined(__x86_64__) || defined(_M_X64)
            _mm_pause();
#elif defined(__aarch64__)
            asm volatile("yield");
#endif
        }
    }

    void unlock() {
        flag.clear(std::memory_order_release);
    }
};
```

Резюме раздела

- **Атомики** обеспечивают неделимость операций и видимость изменений между потоками (Memory Ordering).
- **CAS (Compare-And-Swap)** — универсальный примитив для построения Lock-Free алгоритмов.
- **CAS-Loop** позволяет применять произвольные функции к разделяемым данным оптимистично.
- Основная сложность Lock-Free не в алгоритмах изменения данных, а в **управлении памятью** (когда безопасно удалить узел?).

Глава 55

Advanced Lock-Free: ABA & Memory Reclamation

В предыдущей главе мы реализовали Lock-Free стек и столкнулись с фундаментальной проблемой отсутствия Garbage Collector в C++: мы не знаем, когда безопасно вызывать `delete` для узла, исключенного из структуры данных.

Однако проблема управления памятью в Lock-Free алгоритмах глубже, чем просто утечки или Segfault. Повторное использование памяти (memory recycling) может привести к логическому разрушению структуры данных, даже если все указатели валидны. Этот феномен известен как проблема ABA.

В этой главе мы разберем механику ABA на уровне состояний памяти, изучим аппаратные методы решения (Tagged Pointers) и реализуем программный алгоритм безопасной рекламации памяти — Hazard Pointers. Также мы рассмотрим более сложную структуру — Lock-Free очередь Майкла-Скотта.

Проблема ABA

Название ABA происходит от последовательности изменения состояний: значение было A, стало B, затем снова стало A. Для наивного наблюдателя, использующего CAS (Compare-And-Swap), кажется, что значение не менялось, хотя мир вокруг изменился кардинально.

Анатомия катастрофы (Сценарий на Стеке)

Рассмотрим наш Lock-Free стек (LIFO). Состояние стека: `Top` $\rightarrow$ `A` $\rightarrow$ `B` $\rightarrow$ `C`.

Поток 1 (Жертва):

- Планирует выполнить `Pop`.
- Читает текущую голову: `old_head = A` (адрес 0x1000).
- Читает следующий элемент: `next = A`$\rightarrow$`next` (узел B, адрес 0x2000).
- *Поток прерывается планировщиком ОС.*

Поток 2 (Разрушитель):

- Выполняет `Pop`: Успешно удаляет A. Стек: `Top` $\rightarrow$ `B` $\rightarrow$ `C`.
- Удаляет узел A (`delete A`). Память по адресу 0x1000 возвращается аллокатору.

- Выполняет `Pop`: Успешно удаляет B. Стек: `Top` $\rightarrow$ `C`.
- Удаляет узел B (`delete B`). Адрес 0x2000 свободен.
- Выполняет `Push`: Создает новый узел. Аллокатор, оптимизируя работу, возвращает только что освобожденный адрес 0x1000.
- Записывает в новый узел (по адресу 0x1000) какие-то данные. Его `next` указывает на C.
- Стек: `Top` $\rightarrow$ `A' (0x1000)` $\rightarrow$ `C`.

Поток 1 (Продолжение):

- Просыпается. Готовится выполнить CAS для смены головы.
- Аргументы CAS:
  - Адрес переменной: `&Top`.
  - Ожидаемое значение: 0x1000 (узел A).
  - Новое значение: 0x2000 (узел B, сохраненный в переменной `next`).
- **CAS Проверка:** Текущий `Top` равен 0x1000? Да, равен (это новый узел A').
- **CAS Успех:** `Top` меняется на 0x2000 (узел B).

Результат: Голова стека указывает на адрес 0x2000. Но узел по этому адресу был удален Потоком 2! Стек сломан. Следующий `Pop` приведет к чтению освобожденной памяти (Use-After-Free) или, что хуже, к чтению данных другого объекта, который аллокатор разместил по адресу 0x2000.

Решение 1: Tagged Pointers (Указатели с версиями)

Проблема ABA возникает из-за того, что указатель (адрес) не уникален во времени. Адрес 0x1000 сегодня — это узел стека, завтра — текстура видеокарты, послезавтра — снова узел стека.

Чтобы сделать указатель уникальным, нужно добавить к нему **счетчик версий** (Tag). Каждое обновление указателя (CAS) должно инкрементировать этот счетчик. Тогда последовательность $A \rightarrow B \rightarrow A$ превратится в $A_1 \rightarrow B_2 \rightarrow A_3$. CAS, ожидающий A_1, провалится, увидев A_3.

Реализация на x86_64

В 64-битных процессорах виртуальные адреса используют только младшие 48 бит. Старшие 16 бит обычно должны быть нулями (или копией 47-го бита). Мы можем использовать эти 16 бит для хранения счетчика.

```
// Упрощенная концепция (требует битхаков)
struct TaggedPointer {
    uint64_t raw; // 48 бит адрес + 16 бит тег

    static const uint64_t TAG_MASK = 0xFFFF000000000000;
    static const uint64_t PTR_MASK = 0x0000FFFFFFFFFFFF;

```

```
    Node* get_ptr() const {
        return reinterpret_cast<Node*>(raw & PTR_MASK);
    }

    uint16_t get_tag() const {
        return (raw & TAG_MASK) >> 48;
    }

    // При создании нового ptr инкрементируем tag
    static TaggedPointer make(Node* ptr, uint16_t old_tag) {
        return { (uint64_t(ptr) & PTR_MASK) | (uint64_t(old_tag + 1) << 48) };
    }
};
```

В C++ для этого удобно использовать двойную ширину CAS (`cmpxchg16b` на x86_64), который оперирует парой 64-битных чисел (итого 128 бит). Стандартный `std::atomic<shared_ptr>` часто реализован именно так, но это дорого.

Решение 2: Hazard Pointers (Опасные указатели)

Tagged Pointers решают ABA, но не решают проблему преждевременного `delete`. Самым надежным чисто программным решением в C++ (без использования GC) является метод **Hazard Pointers** (HP), предложенный Майклом (Maged Michael).

Идея: Каждый поток-читатель имеет свой личный список "опасных указателей" (обычно 1-2 слота). Перед тем как работать с указателем, поток записывает его в свой HP-слот, объявляя всем остальным: "Я читаю этот объект, не удаляйте его!".

Поток-писатель, желающий удалить объект, сначала сканирует списки HP всех потоков.

- Если адрес найден в чьем-то HP — удалять нельзя. Адрес добавляется в список "на пенсию" (Retire List).
- Если адрес нигде не найден — можно безопасно вызвать `delete`.

Алгоритм работы с HP в Pop

Это сложнее, чем кажется, из-за гонок данных. Мы не можем просто записать указатель в HP, потому что к моменту записи он уже мог быть удален.

```
// Глобальный массив Hazard Pointers (по одному слоту на поток)
std::atomic<Node*> HP[MAX_THREADS];

std::optional<T> Pop(int thread_id) {
    Node* old_head;
    Node* next_node;

    while (true) {
        // 1. Читаем текущую голову
        old_head = head.load(std::memory_order_acquire);
        if (!old_head) return std::nullopt;

```

```
        // 2. Объявляем указатель "опасным" (защищаем его)
        HP[thread_id].store(old_head, std::memory_order_seq_cst);

        // 3. КРИТИЧЕСКАЯ ПРОВЕРКА!
        // За время между шагом 1 и 2 old_head мог быть удален.
        // Если head изменился, значит, наш HP защищает уже "мертвый" объект.
        // Нужно начать сначала.
        if (head.load(std::memory_order_acquire) != old_head) {
            HP[thread_id].store(nullptr, std::memory_order_relaxed); // Снимаем
            ↪  защиту
            continue;
        }

        // 4. Теперь указатель безопасно защищен. Можно читать поля.
        next_node = old_head->next; // Безопасно!

        // 5. Пытаемся удалить из стека
        if (head.compare_exchange_weak(old_head, next_node)) {
            break; // Успех! Мы извлекли элемент.
        }

        // Неудача: снимаем защиту и пробуем снова
        HP[thread_id].store(nullptr, std::memory_order_relaxed);
    }

    HP[thread_id].store(nullptr, std::memory_order_release); // Мы закончили

    // 6. Отправляем old_head на удаление
    RetireNode(old_head);

    return old_head->data;
}
```

RetireNode и Scan

Функция `RetireNode` не делает `delete` сразу. Она добавляет указатель в локальный буфер потока (`thread_local vector`). Когда буфер заполняется (например, достигает размера $2 \cdot N$, где N — число потоков), запускается процедура `Scan`.

`Scan`: 1. Собирает все ненулевые указатели из глобального массива `HP` в `std::set` (или сортированный вектор). 2. Проходит по локальному буферу "пенсионеров". 3. Если указатель из буфера ЕСТЬ в множестве `HP` — оставляем его на потом. 4. Если указателя НЕТ в множестве `HP` — делаем реальный `delete`.

Это гарантирует, что мы никогда не удалим объект, который кто-то читает.

Решение 3: RCU (Read-Copy-Update)

Для структур, где чтений много, а записей мало, HP может быть слишком дорогим из-за необходимости писать в `atomic` (это сбивает кэш-линии). Альтернатива — RCU.

Идея: разделить время на "эпохи" (generations). 1. У нас есть глобальный счетчик эпох. 2. Писатель, удаляющий данные, "публикует" новую версию структуры данных, а старую

откладывает в список удаления текущей эпохи. 3. Писатель ждет, пока *все* читатели, начавшие работу в старой эпохе, завершат свои операции (Quiescent State). 4. После этого старую эпоху можно безопасно чистить.

В user-space RCU сложен в реализации (нужно знать, когда читатель "вышел" из критической секции), но библиотека `liburcu` предоставляет готовые решения.

Практика: Lock-Free Очередь (Queue)

Очередь (FIFO) сложнее стека, так как имеет две точки изменения: `Head` (откуда забираем) и `Tail` (куда добавляем). Классический алгоритм — **Michael-Scott Queue**.

Проблемы двух указателей

Если мы просто храним `atomic<Node*> Head` и `Tail`, мы не можем обновить их оба атомарно одной инструкцией. Вставка в конец требует двух шагов:

1. `Tail->next = new_node` (линковка).
2. `Tail = new_node` (продвижение хвоста).

Если поток заснет между 1 и 2, `Tail` останется указывать на предпоследний элемент. Следующий поток, пришедший делать `Push`, увидит, что `Tail->next != nullptr`.

Идея помощи (Helping): Lock-Free алгоритм очереди устроен так, что если поток видит "отставший" `Tail` (у которого `next` не null), он не ждет, а **помогает** завершить операцию, продвигая `Tail` вперед с помощью CAS, и только потом делает свою вставку.

Dummy Node (Фиктивный узел)

Чтобы избежать гонок на пустой очереди (когда `Head == Tail == nullptr`), очередь Майкла-Скотта всегда содержит хотя бы один "фиктивный" узел. Изначально: `Head = Tail = new Node(dummy)`.

- `Push`: Добавляет после `Tail`, двигает `Tail`.
- `Pop`: Читает `Head->next`. Если есть, возвращает данные, и делает `Head = Head->next`. Старый `Head` (бывший dummy) удаляется (через HP), а узел, который содержал данные, становится новым dummy.

Резюме раздела

- Проблема ABA разрушает логику Lock-Free алгоритмов при переиспользовании памяти.
- **Tagged Pointers** решают ABA аппаратно, добавляя счетчик версий в неиспользуемые биты указателя.
- **Hazard Pointers** — де-факто стандарт для управления памятью в C++ Lock-Free. Они гарантируют безопасность доступа, но требуют накладных расходов на `store-load` барьеры при каждом чтении.
- Реализация очередей требует кооперации потоков: если один "застрял" посередине операции, другие должны ему помочь, чтобы сохранить свойство Lock-Free.

Часть XI

Лекция 11 – lock free

Глава 56

Анатомия синхронизации: Mutex vs Atomic и цена ожидания

Многопоточное программирование в C++ строится на примитивах синхронизации, предоставляемых стандартной библиотекой. В основе всех *lock-free* и *wait-free* алгоритмов лежит фундаментальный строительный блок — `std::atomic`.

Атомарные переменные: базовый API

Атомарная переменная (`std::atomic<T>`) гарантирует отсутствие гонок данных (*data race*) при одновременном доступе из нескольких потоков. Это достигается за счет использования специальных инструкций процессора (например, `LOCK XADD` на x86) или барьеров памяти.

Ключевая особенность атомиков в том, что операции над ними (загрузка, сохранение, инкремент) неделимы. Невозможна ситуация, когда один поток прочитал "половину" записанного значения или когда два потока одновременно инкрементировали переменную, но результат увеличился только на 1.

Базовые операции

```cpp
#include <atomic>

std::atomic<int> counter{0};

void worker() {
    // 1. Атомарная запись (Store)
    counter.store(10);

    // 2. Атомарное чтение (Load)
    int val = counter.load();

    // 3. Атомарная модификация (Read-Modify-Write)
    // Возвращает ПРЕДЫДУЩЕЕ значение
    int old_val = counter.fetch_add(1);

    // Эквивалентно fetch_add(1), но возвращает
```

```
    // результат в зависимости от перегрузки (post/pre)
    counter++;
}
```

API `std::atomic` различается для разных типов данных:

- **Целочисленные типы:** Поддерживают арифметику `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor`.
- **Указатели:** Поддерживают адресную арифметику (например, `fetch_add` сдвигает указатель на размер типа).
- **Пользовательские структуры:** Поддерживают только `load`, `store` и `compare_exchange` (CAS).

Сравнительный анализ: Atomic vs Mutex

Существует распространенное заблуждение, что атомарные операции всегда быстрее, чем использование `std::mutex`. Это утверждение верно только для определенных сценариев и часто ложно при высокой конкуренции (*high contention*).

Бенчмарк: Инкремент счетчика

Рассмотрим классический бенчмарк: N потоков инкрементируют одну общую переменную.

Вариант 1: Mutex

```
std::mutex mtx;
int shared_data = 0;

void mutex_inc() {
    std::lock_guard<std::mutex> lock(mtx);
    shared_data++;
}
```

Вариант 2: Atomic

```
std::atomic<int> shared_data{0};

void atomic_inc() {
    shared_data.fetch_add(1, std::memory_order_relaxed);
}
```

При малом количестве потоков (1-4) атомарная версия работает быстрее, так как отсутствует накладной расход на захват блокировки. Однако, по мере роста числа потоков (например, до 16 и выше на 8-ядерном процессоре), производительность атомиков деградирует.

Физика процесса: Атомарный инкремент требует эксклюзивного владения кэш-линией (*cache coherency protocol*, например, MESI). Когда множество ядер пытаются записать в одну и

ту же переменную, кэш-линия начинает "метаться" между ядрами (cache ping-pong). Шина процессора насыщается трафиком синхронизации кэшей.

В случае `std::mutex`, потоки, не сумевшие захватить блокировку, уходят в режим ожидания (sleep) на уровне ядра ОС. Они перестают конкурировать за шину памяти, позволяя владельцу мьютекса быстро выполнить работу.

Резюме раздела

`std::mutex` может оказаться эффективнее `std::atomic` при высокой конкуренции, так как он сериализует доступ и убирает паразитную нагрузку на шину памяти от ожидающих потоков.

Анатомия ожидания: Spinlock, Futex и Throttling

Понимание того, как реализован `std::mutex` "под капотом", критически важно для выбора инструмента синхронизации.

Spinlock (Спинлок)

Простейшая реализация блокировки на атомиках — это Spinlock. Поток крутится в цикле, постоянно проверяя флаг, пока не получит доступ.

> **Spinlock**
>
> Механизм синхронизации, при котором ожидающий поток не приостанавливает свое выполнение, а находится в цикле активного ожидания (*busy wait*), потребляя такты процессора.

```
class Spinlock {
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        // test_and_set возвращает true, если флаг УЖЕ был установлен.
        // Если вернул false — мы успешно установили флаг (захватили лок).
        // memory_order_acquire — для корректной синхронизации.
        while (flag.test_and_set(std::memory_order_acquire)) {
            // Busy wait: поток сжигает CPU впустую
            // На x86 здесь часто ставят инструкцию _mm_pause()
        }
    }

    void unlock() {
        flag.clear(std::memory_order_release);
    }
};
```

Проблема Spinlock в Userspace: Использование спинлоков в прикладном коде (userspace) — опасный анти-паттерн.

1. **CPU Burning:** Поток занимает ядро на 100%, не выполняя полезной работы. Это тратит энергию и греет процессор.
2. **Priority Inversion:** Если высокоприоритетный поток ждет спинлок, захваченный низкоприоритетным потоком, а планировщик не дает времени владельцу, система может зависнуть.

Проблема троттлинга (Throttling)

Особая опасность спинлоков возникает в облачных средах (Docker, Kubernetes) с лимитами по CPU.

Представьте ситуацию:

1. Поток А захватывает спинлок.
2. Планировщик ОС (или гипервизор) прерывает выполнение потока А, так как его квант времени (timeslice) истек.
3. Поток Б пытается захватить тот же спинлок. Он начинает крутиться в `while`, сжигая свой квант времени.
4. Поток Б не может продвинуться, так как Поток А "спит" и не может освободить лок.

Если квота CPU исчерпана, поток А может быть "заморожен" на десятки миллисекунд (throttling latency). Все это время остальные потоки, ожидающие спинлок, просто сжигают процессорное время впустую.

Реализация Mutex через Futex

Современный `std::mutex` (например, в Linux через glibc/pthreads) использует гибридную схему, основанную на системном вызове **futex** (Fast Userspace Mutex).

Алгоритм работы `std::mutex::lock()`:

1. **Fast Path (Userspace):** Поток пытается атомарно (CAS) изменить состояние мьютекса с "свободен" на "занят". Если удается — блокировка захвачена без системного вызова. Это очень быстро (десятки наносекунд).
2. **Slow Path (Kernel Space):** Если мьютекс занят, поток делает системный вызов `futex_wait`. Ядро ОС переводит поток в состояние `BLOCKED` и убирает его из очереди планирования. Поток перестает потреблять CPU.

Когда владелец вызывает `unlock()`, он сначала освобождает флаг, а затем (если есть ожидающие) делает `futex_wake`, чтобы ядро разбудило один из спящих потоков.

> **Важно!**
>
> `std::mutex` эффективен, потому что ожидание переносится в ядро (*kernel space*). Spinlock же оставляет ожидание в пространстве пользователя, сжигая ресурсы.

Аппаратные ограничения и std::atomic<T>

Не любой тип `T` может быть обработан атомарно на аппаратном уровне. Процессоры обычно гарантируют атомарность только для типов, помещающихся в машинное слово (64 бита) или двойное слово (128 бит, инструкция `CMPXCHG16B` на x64).

is_always_lock_free

Стандарт C++ предоставляет механизм проверки, является ли атомик "настоящим" (lock-free) или эмулируется библиотекой.

```
struct LargeData {
    char buffer[400]; // Слишком большой для регистра CPU
};

std::atomic<LargeData> data;

// Вернет false, так как процессор не умеет атомарно писать 400 байт
bool is_fast = std::atomic<LargeData>::is_always_lock_free;
```

Если `is_always_lock_free` ложно, компилятор реализует `std::atomic` через скрытый глобальный мьютекс (хеш-таблица мьютексов). В этом случае использование `std::atomic` теряет смысл с точки зрения производительности и свойства lock-free.

> **На заметку**
>
> Перед использованием `std::atomic` со структурами всегда проверяйте `is_always_lock_free` через `static_assert`.

Эволюция ожидания в C++20

До C++20 у разработчиков не было стандартного способа реализовать эффективное ожидание на атомарной переменной без спинлоков или использования сторонних мьютексов/condition variables.

В C++20 добавлены методы `wait`, `notify_one` и `notify_all` непосредственно в `std::atomic`. Это фактически предоставляет интерфейс футексов на уровне языка.

```
#include <atomic>
#include <thread>

std::atomic<int> signal_flag{0};

void consumer() {
    // Эффективное ожидание:
    // Блокирует поток, пока signal_flag == 0.
    // Не потребляет CPU в цикле!
    int value = 0;
    while ((value = signal_flag.load()) == 0) {
        signal_flag.wait(0);
    }
    // ... работаем ...
}

void producer() {
    signal_flag.store(1);
    signal_flag.notify_one(); // Будим ожидающий поток
```

```
20 }
```

Метод `wait(old_val)` работает следующим образом:

1. Атомарно сравнивает текущее значение атомика с `old_val`.
2. Если значения не равны — немедленно возвращает управление (значение уже изменилось).
3. Если равны — поток усыпляется (как на мьютексе/футексе) до вызова `notify`.

Это позволяет строить эффективные механизмы синхронизации, сочетающие скорость атомиков (fast path) и энергоэффективность блокировок (slow path).

Глава 57

Теория гарантий прогресса: Lock-free и Wait-free

В разработке многопоточных систем термины *lock-free* и *wait-free* часто используются ошибочно как синонимы "высокой производительности". Однако в строгом академическом смысле это не метрики скорости, а гарантии прогресса системы (*progress guarantees*).

Эти гарантии описывают поведение алгоритма в условиях максимальной конкуренции (*contention*) или при нештатном поведении планировщика (например, приостановка потоков).

Lock-freedom: Глобальный прогресс

> **Lock-free (LF)**
>
> Алгоритм называется **lock-free**, если гарантируется, что при непрерывном выполнении системы хотя бы один поток сделает полезный прогресс за конечное число шагов.

Свойство *lock-freedom* является глобальным (system-wide). Оно не гарантирует, что конкретный поток T_1 когда-либо завершит свою операцию. Оно лишь гарантирует, что система в целом не зависнет. Если T_1 не может завершить операцию, это происходит только потому, что какой-то другой поток T_2 *успешно* завершил свою операцию, изменив состояние разделяемой структуры.

Критерий приостановки (Suspension Criterion)

Интуитивный способ проверки алгоритма на lock-freedom — мысленный эксперимент с "заморозкой" потока.

> **Важно!**
>
> Если мы произвольно остановим (suspend) любой поток (например, в отладчике или из-за троттлинга ОС) в любой точке выполнения, другие потоки **должны** иметь возможность продолжать выполнение и завершать свои операции.

Если остановка одного потока приводит к тому, что остальные потоки начинают бесконечно ожидать (как в случае с захваченным мьютексом), алгоритм **не является** lock-free.

Wait-freedom: Локальный прогресс

> **Wait-free (WF)**
>
> Алгоритм называется **wait-free**, если гарантируется, что **каждый** поток сделает прогресс (завершит операцию) за конечное, ограниченное число шагов, независимо от действий других потоков.

Это более строгое требование. Wait-free алгоритм исключает не только взаимные блокировки (deadlocks), но и голодание (starvation). Никакая конкуренция не может заставить поток выполняться бесконечно долго.

Иерархия гарантий выглядит так:

$$\text{Wait-free} \subset \text{Lock-free} \subset \text{Blocking (с блокировками)}$$

Любой wait-free алгоритм является lock-free, но обратное неверно.

Классификация алгоритмов синхронизации

Рассмотрим классические примитивы через призму гарантий прогресса.

1. Mutex и Spinlock (Blocking)

Использование `std::mutex` или самописного спинлока автоматически выводит алгоритм из категории lock-free.

```
std::mutex mtx;

void blocked_op() {
    mtx.lock();
    // CRITICAL SECTION
    // Если поток будет остановлен здесь (OS scheduler, crash),
    // то НИКТО больше не сможет войти в секцию.
    // Прогресс системы равен нулю.
    mtx.unlock();
}
```

Спинлок, реализованный на атомиках (как в Главе 1), технически использует lock-free инструкции процессора (CAS, test-and-set), но *алгоритмически* является блокирующим. Ожидание в цикле `while` зависит от действий другого потока (владельца), что нарушает определение lock-freedom.

2. CAS-Loop (Lock-free)

Классический паттерн для реализации атомарных транзакций (например, добавление в стек) — цикл Compare-And-Swap.

```
void lock_free_op(std::atomic<int>& shared_data) {
    int old_val = shared_data.load();
    int new_val;
```

```
4
5       do {
6           new_val = complex_calculation(old_val);
7           // Пытаемся атомарно обновить:
8           // Если shared_data == old_val, то пишем new_val и выходим (true).
9           // Если нет, в old_val записывается актуальное значение, повторяем
            ↪  (false).
10      } while (!shared_data.compare_exchange_weak(old_val, new_val));
11  }
```

Анализ прогресса:

- Представим, что 10 потоков одновременно зашли в этот цикл.
- Они конкурируют за `compare_exchange`.
- В конкретный момент времени только **один** поток выиграет гонку (у него CAS вернет `true`). Он совершит прогресс и выйдет из функции.
- Остальные 9 потоков получат `false`, обновят свои локальные копии `old_val` и пойдут на следующую итерацию.
- **Результат:** Система совершила прогресс (одна операция выполнена), несмотря на то, что 9 потоков "потратили время впустую".

Этот алгоритм — **Lock-free**, но не **Wait-free**, так как теоретически возможен сценарий, где "поток-неудачник" бесконечно проигрывает гонку CAS более быстрым потокам (livelock/starvation). Однако система в целом продолжает работать.

3. Атомарные Load/Store (Wait-free)

Простейшие операции над `std::atomic` (если они аппаратно поддерживаются) являются wait-free.

```
1  std::atomic<int> flag{0};
2
3  void set_flag() {
4      // Выполняется за фиксированное число инструкций процессора.
5      // Не содержит циклов.
6      // Не зависит от других потоков.
7      flag.store(1);
8  }
```

Wait-free алгоритмы сложнее в реализации для комплексных структур данных, так как требуют механизмов помощи (*helping*), когда один поток, видя, что другой застрял, помогает ему завершить операцию.

Зачем нужны эти гарантии?

Частое заблуждение: "Lock-free код быстрее". Как показано в бенчмарках (Глава 1), это не всегда так из-за высокой стоимости CAS-инструкций и cache-contention. Главная ценность lock-free не в пропускной способности (*throughput*), а в надежности и предсказуемости задержек (*latency*).

1. Устойчивость к инверсии приоритетов (Priority Inversion)

В системах с блокировками возможна ситуация, когда низкоприоритетный поток захватывает мьютекс и прерывается планировщиком. Высокоприоритетный поток, ожидающий этот мьютекс, вынужден ждать, пока низкоприоритетный получит процессорное время. В lock-free алгоритмах высокоприоритетный поток может просто выполнить свою операцию (через CAS), "перебив" или завершив работу за медленный поток.

2. Устойчивость к сбоям (Fault Tolerance)

Если поток, владеющий мьютексом, аварийно завершается (segfault, kill), мьютекс остается "зависшим" (в несогласованном состоянии), что приводит к дедлоку всей системы (если не использовать специальные *robust mutexes*). Lock-free структуры данных всегда остаются в согласованном состоянии: сбой потока посередине CAS-цикла не портит данные, так как изменение либо произошло целиком, либо не произошло вовсе.

Резюме раздела

- **Lock-free:** Гарантирует отсутствие взаимных блокировок (deadlock). Система всегда движется вперед, но отдельные потоки могут голодать.
- **Wait-free:** Гарантирует отсутствие голодания (starvation). Время выполнения операции ограничено сверху.
- Lock-free алгоритмы необходимы там, где недопустима блокировка всей системы из-за остановки одного потока (системы реального времени, обработчики сигналов, ядра ОС).

Глава 58

Паттерн CAS-Loop и сложные атомарные операции

Стандартная библиотека C++ предоставляет ограниченный набор атомарных операций модификации: сложение (`fetch_add`), вычитание (`fetch_sub`) и побитовые операции (`and`, `or`, `xor`). Однако на практике часто требуются более сложные транзакции, например, атомарное умножение, вычисление максимума или обновление битовых полей по маске.

Для реализации любой произвольной операции Read-Modify-Write (RMW) используется универсальный примитив: **Compare-And-Swap (CAS)**.

Анатомия Compare-And-Swap

Инструкция CAS — это самая "тяжелая" и мощная операция в арсенале атомиков. На ней строятся все lock-free структуры данных.

В C++ она представлена семейством методов `compare_exchange`.

Сигнатура и семантика

```
bool compare_exchange_weak(T& expected, T desired, ...);
bool compare_exchange_strong(T& expected, T desired, ...);
```

Ключевая особенность, которую часто упускают новички: первый аргумент `expected` передается **по ссылке**. Он играет двойную роль: входного параметра (ожидание) и выходного (реальность).

Логика работы метода (псевдокод):

```
// Атомарно внутри процессора:
if (this->value == expected) {
    this->value = desired; // Успех: записали новое значение
    return true;
} else {
    expected = this->value; // Неудача: обновили expected реальным значением
    return false;
}
```

Это избавляет программиста от необходимости делать повторный `load()` при неудаче. Метод сам сообщает: "Ты ожидал А, но там лежит Б. Я записал Б в твою переменную `expected`, попробуй еще раз".

Weak vs Strong: Цена гарантий

Стандарт C++ разделяет CAS на две версии.

> **Spurious Failure (Ложный отказ)**
>
> Ситуация, когда `compare_exchange` возвращает `false`, и значение не обновляется, даже если текущее значение атомика **равно** `expected`.

compare_exchange_weak

Разрешает ложные отказы. На некоторых архитектурах (ARM, PowerPC) атомарные операции реализуются через пару инструкций LL/SC (Load Linked / Store Conditional). Если между ними произошло прерывание или другой поток тронул кэш-линию, запись может не пройти, даже если значение не менялось.

- **Преимущество:** Генерирует более эффективный машинный код на RISC-архитектурах.
- **Использование:** Всегда используется в циклах.

compare_exchange_strong

Гарантирует отсутствие ложных отказов. Возвращает `false` только если значение реально не совпадает.

- **Цена:** На RISC-архитектурах компилятор генерирует вложенный цикл для сокрытия ложных отказов.
- **Использование:** Когда операция выполняется однократно (вне цикла) и повтор вычисления `desired` слишком дорог.

> **Важно!**
>
> В 99% случаев при написании lock-free алгоритмов вы будете использовать паттерн цикла. В цикле **всегда** используйте `weak` версию. Использование `strong` в цикле приводит к двойной вложенности циклов на уровне ассемблера.

Реализация Atomic Fetch-Max

Рассмотрим задачу: реализовать атомарную функцию, обновляющую переменную значением `max(current, value)`. Стандартного `fetch_max` нет.

Мы используем паттерн **CAS-Loop**.

```
#include <atomic>
#include <algorithm>

```

```cpp
void atomic_fetch_max(std::atomic<int>& atom, int val) {
    // 1. Предварительная загрузка текущего состояния (Snapshot)
    // Можно использовать memory_order_relaxed, так как порядок пока не важен
    int prev = atom.load(std::memory_order_relaxed);

    // 2. Цикл попыток (CAS Loop)
    while (prev < val) {
        // 3. Попытка атомарной подмены
        // Если atom == prev, то записать val. Вернуть true.
        // Если atom ≠ prev, то prev = atom. Вернуть false.
        if (atom.compare_exchange_weak(prev, val,
                                       std::memory_order_release,
                                       std::memory_order_relaxed)) {
            break; // Успех!
        }

        // 4. Неявная ветвь else (при неудаче):
        // compare_exchange_weak УЖЕ обновил переменную 'prev'
        // актуальным значением из атомика.
        // Мы просто идем на следующую итерацию while.
        // Там условие (prev < val) проверится заново с новым prev.
    }
}
```

Построчный разбор механики

1. `int prev = atom.load(...)`: Мы делаем локальную копию значения. Это наше предположение о состоянии мира.
2. `while (prev < val)`: Оптимизация. Если текущее значение в атомике уже больше или равно `val`, нам не нужно ничего писать. Мы выходим без дорогих операций записи.
3. `compare_exchange_weak(prev, val, ...)`: Это точка синхронизации.
   - **Сценарий Успеха:** Между `load` (или предыдущей попыткой) и этой инструкцией никто не трогал `atom`. Значение успешно обновлено.
   - **Сценарий Провала:** Другой поток успел записать свое значение. В `atom` теперь лежит условное число 100, а мы думали там 50.
4. **Автоматическое обновление:** При провале переменная `prev` по ссылке получает значение 100. На следующей итерации цикла мы проверим `100 < val` и попробуем сделать CAS(100, val).

Доказательство свойства Lock-free

Почему этот алгоритм является Lock-free, но не Wait-free?

1. Не Wait-free: Теоретически возможна ситуация, когда поток-неудачник бесконечно крутится в цикле `while`. Это происходит, если между моментом проверки условия и вызовом CAS всегда вклинивается другой поток, изменяющий переменную. Время выполнения операции для конкретного потока не ограничено сверху.

2. Является Lock-free: Цикл продолжается только тогда, когда CAS возвращает `false`. CAS

возвращает `false` (игнорируя spurious failure) только тогда, когда значение в атомике **изменилось**. Значение меняется только в том случае, если какой-то другой поток успешно выполнил свой CAS (или `store`).

Резюме раздела

Следовательно, бесконечное выполнение цикла в одном потоке означает бесконечное количество успешных обновлений данных в других потоках. Глобальный прогресс системы гарантирован.

На заметку

CAS-loop — это оптимистичная блокировка. Мы предполагаем, что конфликта не будет, делаем работу локально, а в конце пытаемся "закоммитить" результат. Если конфликт был — откатываемся и повторяем. Это выгоднее мьютексов при низкой и средней конкуренции.

Глава 59

Модели памяти: Sequentially Consistent vs Relaxed

При работе с атомарными переменными программист обязан явно или неявно выбирать **модель памяти** (Memory Order). Этот выбор определяет, какие гарантии компилятор и процессор предоставляют относительно порядка выполнения инструкций и видимости изменений между потоками.

Ошибочный выбор модели памяти приводит к гейзенбагам, которые невозможно воспроизвести в отладчике, но которые гарантированно возникают под высокой нагрузкой на специфическом оборудовании (например, на архитектуре ARM).

Sequentially Consistent (SC)

Модель по умолчанию в C++ — `std::memory_order_seq_cst` (Sequentially Consistent).

> **Sequential Consistency**
>
> Модель исполнения, при которой результат работы многопоточной программы эквивалентен некоторому последовательному чередованию (*interleaving*) инструкций всех потоков на одноядерном процессоре.

Это самая сильная ("строгая") модель. Она интуитивно понятна человеку, так как сохраняет причинно-следственные связи.

Гарантии SC:

1. **Total Order:** Существует единый глобальный порядок всех модификаций всех SC-атомиков. Все потоки видят изменения в одном и том же порядке.
2. **Отсутствие переупорядочивания:** Инструкции исходного кода не могут "перепрыгивать" через SC-операции ни вверх, ни вниз.

Цена абстракции

За простоту приходится платить производительностью. Чтобы обеспечить SC, компилятор обязан:

- Отключить многие оптимизации (перестановка инструкций).

- Вставить аппаратные барьеры памяти (*memory fences/barriers*). На x86 это часто инструкции `MFENCE` или `LOCK prefixed instructions`, которые сбрасывают конвейер процессора и принудительно синхронизируют буферы записи.

Аппаратная реальность: Store Buffers

Чтобы понять необходимость более слабых моделей, нужно рассмотреть архитектуру процессора. Запись в оперативную память (RAM) — операция экстремально медленная (сотни тактов CPU).

Чтобы не останавливать конвейер на каждой инструкции записи, процессоры используют **Store Buffer** (буфер записи).

Механика:

1. Ядро выполняет инструкцию записи `x = 1`.
2. Значение помещается в локальный Store Buffer ядра, а не в кэш L1/RAM. Процессор продолжает выполнение следующих инструкций **мгновенно**.
3. Спустя время буфер сбрасывается (*flush*) в когерентный кэш, и только тогда изменение становится видимым другим ядрам.

> **Важно!**
>
> Наличие Store Buffer приводит к тому, что порядок *выполнения* инструкций (Program Order) не совпадает с порядком *видимости* изменений (Visibility Order) для других ядер.

Relaxed Ordering (Ослабленная модель)

Модель `std::memory_order_relaxed` снимает все ограничения на синхронизацию и порядок видимости, оставляя только одну гарантию: **атомарность** самой операции над переменной.

- **Нет Happends-Before:** Запись `x.store(1, relaxed)` в одном потоке не гарантирует, что другой поток увидит это значение "своевременно" относительно других переменных.
- **Свобода компилятора:** Компилятор может менять местами Relaxed-инструкции с обычными операциями, если это не ломает однопоточную логику.

Это самая дешевая модель. На большинстве архитектур она компилируется в обычные инструкции `MOV` без барьеров.

Парадокс Store Buffer: "Невозможный" результат

Рассмотрим классический пример, демонстрирующий разницу между SC и Relaxed (пример Деккера).

У нас есть две атомарные переменные `x` и `y`, инициализированные нулями.

```cpp
std::atomic<int> x{0}, y{0};

// Поток 1
void thread_1() {
    x.store(1, std::memory_order_relaxed); // (A)
    int r1 = y.load(std::memory_order_relaxed); // (B)
}

// Поток 2
void thread_2() {
    y.store(1, std::memory_order_relaxed); // (C)
    int r2 = x.load(std::memory_order_relaxed); // (D)
}
```

Анализ в Sequential Consistency

Если бы мы использовали `seq_cst`, возможны разные чередования, например:

- $A \to B \to C \to D \implies r1 = 0, r2 = 1$
- $C \to D \to A \to B \implies r1 = 1, r2 = 0$
- $A \to C \to B \to D \implies r1 = 1, r2 = 1$

Исход $r1 = 0, r2 = 0$ **невозможен**. Чтобы $r1$ был 0, (B) должно выполниться до (C). Чтобы $r2$ был 0, (D) должно выполниться до (A). Это создает цикл противоречий.

Анализ в Relaxed (Store Buffer Reality)

При использовании `relaxed` исход $r1 = 0, r2 = 0$ становится **возможным**.

Физическое объяснение:

1. Поток 1 выполняет (A): `x=1`. Значение попадает в Store Buffer ядра 1. В основной памяти `x` все еще 0.
2. Поток 1 сразу выполняет (B): читает `y`. В памяти `y` равно 0. Результат: `r1=0`.
3. Поток 2 выполняет (C): `y=1`. Значение попадает в Store Buffer ядра 2.
4. Поток 2 сразу выполняет (D): читает `x`. Ядро 2 не видит буфер ядра 1. В памяти `x` все еще 0. Результат: `r2=0`.
5. Спустя время буферы сбрасываются в память, но слишком поздно — потоки уже прочитали нули.

С точки зрения внешнего наблюдателя, произошло переупорядочивание: чтение (Load) выполнилось раньше, чем запись (Store) — эффект **StoreLoad Reordering**.

Резюме раздела

- Используйте `seq_cst` по умолчанию. Это безопасно и предсказуемо.
- Используйте `relaxed` только для счетчиков статистики или когда порядок видимости переменной абсолютно не влияет на логику других потоков.
- Никогда не используйте `relaxed` для реализации механизмов публикации данных (флаг готовности, указатель на объект), так как данные могут стать видимы *позже*, чем флаг готовности.

Глава 60

Практикум: Разработка Lock-free Stack и его фатальные ошибки

Разработка lock-free контейнеров кардинально отличается от разработки их блокирующих аналогов. Если в `std::stack` под мьютексом мы можем использовать `std::vector`, то в lock-free мире динамический массив практически неприменим. Переаллокация памяти (resizing) требует эксклюзивного доступа ко всему массиву, что противоречит идее lock-free.

Поэтому стандартом де-факто для lock-free стека является односвязный список (Treiber Stack).

Базовая архитектура

Структура данных предельно проста: атомарный указатель на вершину списка (`head`).

```
#include <atomic>

template <typename T>
struct Node {
    T data;
    Node* next;

    Node(const T& d) : data(d), next(nullptr) {}
};

template <typename T>
class LockFreeStack {
    std::atomic<Node<T>*> head;

public:
    LockFreeStack() : head(nullptr) {}
    // ... методы ...
};
```

Реализация операции Push

Операция вставки элемента является безопасной и относительно простой. Мы создаем новый узел (который пока невидим для других потоков) и атомарно подменяем `head`.

```
void push(const T& data) {
    Node<T>* new_node = new Node<T>(data);

    // 1. Загружаем текущую голову
    new_node->next = head.load();

    // 2. CAS-цикл
    // Мы пытаемся сделать new_node новой головой.
    // Если head изменился за это время, CAS обновит new_node->next
    // (через параметр expected) и мы попробуем снова.
    while (!head.compare_exchange_weak(new_node->next, new_node)) {
        // Тело цикла пустое.
        // new_node->next автоматически обновлен актуальным значением head.
    }
}
```

Этот код корректен и является lock-free. Новый узел является локальным для потока до момента успешного CAS, поэтому гонок данных на нем нет.

Наивная реализация Pop

Кажется, что удаление элемента симметрично вставке. Нам нужно прочитать `head`, запомнить следующий элемент и атомарно переставить `head` на следующий.

```
// ВНИМАНИЕ: ЭТОТ КОД СОДЕРЖИТ КРИТИЧЕСКИЕ ОШИБКИ
bool pop(T& result) {
    Node<T>* old_head = head.load();

    while (old_head &&
           !head.compare_exchange_weak(old_head, old_head->next)) {
        // Если CAS не удался, old_head обновлен.
        // Крутимся, пока не удалим или пока стек не станет пустым.
    }

    if (old_head == nullptr) {
        return false; // Стек пуст
    }

    result = old_head->data; // Извлечение данных
    delete old_head;         // Освобождение памяти
    return true;
}
```

Данная реализация иллюстрирует "проблему наивного lock-free". Она компилируется, проходит простые тесты, но неизбежно приведет к падению (segfault) и коррупции данных под нагрузкой.

Фатальная ошибка №1: Use-After-Free

В многопоточной среде нельзя просто так разыменовать указатель, полученный из общей структуры данных. Между моментом чтения указателя и доступом к его полям объект мог быть удален другим потоком.

В строке `compare_exchange_weak(old_head, old_head→next)` происходит обращение к `old_head→next`. Рассмотрим сценарий краха:

1. **Поток A:** Читает `head`. Пусть это адрес `0x1000`.
2. **Поток A:** Прерывается планировщиком ОС сразу после чтения, но ДО вызова CAS.
3. **Поток B:** Вызывает `pop()`. Успешно удаляет узел `0x1000`.
4. **Поток B:** Вызывает `delete 0x1000`. Память возвращена ОС или аллокатору.
5. **Поток A:** Просыпается. Пытается вычислить второй аргумент для CAS: `old_head→next`.
6. **Результат:** Обращение к освобожденной памяти (`0x1000`). Segmentation Fault.

> **Важно!**
>
> В lock-free алгоритмах с динамической памятью **нельзя** использовать `delete` сразу после извлечения узла, если есть вероятность, что другие потоки всё ещё держат на него ссылки.

Фатальная ошибка №2: Проблема ABA

Даже если мы решим проблему чтения мусора (например, не будем удалять память, а положим её в список переиспользования), нас поджидает проблема ABA (ABA Problem). Она связана с тем, что CAS проверяет равенство *значений* (адресов), а не *идентичность* объектов во времени.

Пусть стек выглядит так: **Top -> A -> B -> C**.

1. **Поток 1:** Начинает `pop`.
   - Читает `old_head = A`.
   - Читает `next = A→next` (это **B**).
   - Засыпает перед CAS. Он "думает", что хочет заменить A на B.
2. **Поток 2:** Вытесняет Поток 1 и работает агрессивно.
   - `pop()`: Удаляет A. Стек: **B -> C**. Узел A освобожден.
   - `pop()`: Удаляет B. Стек: **C**. Узел B освобожден.
   - `push(D)`: Добавляет D. Стек: **D -> C**.
   - `push(E)`: Аллокатор памяти, видя, что адрес от узла A свободен, **переиспользует** его для нового узла E.
   - Теперь адрес E равен адресу A.
   - Стек: **A(новый) -> D -> C**.
3. **Поток 1:** Просыпается.

- Выполняет `CAS(expected=A, desired=B)`.
- Процессор сравнивает текущий `head` и `expected`. Оба равны адресу A. **CAS успешен!**
- Поток 1 устанавливает `head` в **B**.

Катастрофа: Текущее состояние стека: **Top -> B**. Но узел B был давно удален! Узлы D и C потеряны (утечка памяти), структура списка сломана, `head` указывает на освобожденную память.

Стратегии решения

Написание корректного lock-free стека (MPMC — Multi-Producer Multi-Consumer) требует сложных схем управления памятью (Memory Reclamation).

1. Tagged Pointers (Версионирование)

В 64-битных процессорах используются не все 64 бита адреса (обычно только 48). Мы можем упаковать счетчик обновлений (tag) в неиспользуемые биты указателя. При каждом изменении `head` мы инкрементируем счетчик. Теперь CAS сравнивает пару `{ptr, counter}`. В примере с ABA: адрес будет тот же (A), но счетчик изменится. CAS провалится.

2. Hazard Pointers

Потоки публикуют список указателей ("Hazard Pointers"), которые они сейчас читают, в глобальном массиве. Поток, желающий удалить узел, сначала сканирует массивы Hazard Pointers всех потоков. Если на узел кто-то смотрит — удаление откладывается.

3. Ослабление требований (MPSC)

Если алгоритм допускает наличие только одного потребителя (Single Consumer), то проблемы ABA и Use-After-Free при `pop` исчезают, так как конфликт возникает только между методами `pop` и `pop`. `push` остается безопасным для многих потоков.

Резюме раздела

Lock-free программирование — это не просто расстановка атомиков. Это борьба с моделью памяти, аллокатором и самой природой конкурентности. Без применения Hazard Pointers или RCU написание корректного связного списка на C++ невозможно.

Часть XII

Лекция 12 – корутины

Глава 61

Кризис конкурентности и абстракция Корутины

В основе современной архитектуры высоконагруженных систем лежит проблема эффективной утилизации вычислительных ресурсов. Процессор (CPU) — дорогостоящий ресурс, который должен выполнять инструкции максимально плотно. Однако анализ работы большинства программ показывает парадоксальную картину: CPU редко загружен на 100%, при этом пропускная способность системы (throughput) не растет.

В этой главе мы рассмотрим физические причины простоя процессора, недостатки классической модели многопоточности и архитектурный переход к кооперативной многозадачности в User Space через абстракцию корутин.

Проблема утилизации CPU

Программы делятся на два класса по типу нагрузки:

1. **Compute-bound (Вычислительно-емкие):** Хэширование, перемножение матриц, рендеринг. Утилизируют CPU полностью.
2. **I/O-bound (Зависимые от ввода-вывода):** Работа с сетью, диском, базой данных.

Большинство прикладного ПО относится ко второму типу. Рассмотрим цикл работы I/O-bound потока: 1. Выполнение логики запроса (CPU busy). 2. Отправка запроса в сеть или на диск (System Call). 3. Синхронное ожидание ответа (Blocked state).

В момент ожидания (пункт 3) поток операционной системы (OS Thread) заблокирован. С точки зрения планировщика ОС, поток не готов к исполнению. Ядро процессора переключается на другой поток. Если активных потоков, готовых к вычислениям, нет, процессор переходит в режим ожидания (idle).

> **Важно!**
>
> В синхронной модели ввода-вывода поток бездействует большую часть времени жизни, но продолжает занимать физические ресурсы операционной системы (стек, структуры ядра).

Кризис модели «Thread per Request»

В начале 2000-х годов (эпоха Apache HTTP Server) стандартом архитектуры была модель **Thread per Request**: на каждое входящее соединение создавался отдельный поток ОС.

Этот подход масштабируется до определенного предела. При росте нагрузки (C10K problem — 10 000 одновременных соединений) система деградирует из-за накладных расходов.

1. Расход памяти (Stack Memory)

Каждый поток ОС требует выделения стека.

- Стандартный размер стека в Linux: 8 МБ (configurable, `ulimit -s`).
- Минимальный разумный размер: 64 КБ – 2 МБ.

При 10 000 потоков даже с минимальным стеком в 256 КБ потребление памяти составит $10^4 \times 256$ КБ ≈ 2.5 ГБ. Это память, выделенная только под инфраструктуру, без учета полезных данных приложения.

2. Переключение контекста (Context Switching)

Планировщик ОС (OS Scheduler) работает в режиме вытесняющей многозадачности (Preemptive Multitasking). Переключение между потоками — дорогая операция.

- **Kernel Space Transition:** Процессор переходит в привилегированный режим (Ring 0).
- **Cache Pollution:** Сброс L1/L2 кэшей, TLB (Translation Lookaside Buffer).
- **Latency:** Одно переключение занимает микросекунды (1-5 мкс). При тысячах активных потоков процессор тратит существенное время (overhead) не на полезную работу, а на сам процесс переключения.

На заметку

Операционная система не обладает контекстом приложения (Application Knowledge). Она не знает, какой поток получит данные первым или какой поток выполняет критически важную задачу. Планирование происходит «вслепую» на основе приоритетов и квантов времени.

User Space Scheduling

Решение проблемы — отказ от маппинга «один запрос — один поток». Вместо этого используется пул потоков (Thread Pool), количество которых равно количеству физических ядер CPU.

> **User Space Scheduling**
>
> Планирование задач выполняется внутри процесса пользователя, без участия ядра ОС.

Задачи (Tasks) становятся легкими объектами. Когда задача блокируется на I/O, она не блокирует поток ОС. Вместо этого она **приостанавливается** (Suspend), её состояние сохраняется, а поток берет следующую задачу из очереди.

Эту модель называют **Кооперативной многозадачностью**: задачи сами отдают управление, когда им нужно подождать.

Анатомия Корутины

Корутина (Coroutine) — это обобщение понятия функции.

Для обычной функции (Subroutine) определены две операции:

1. **Call (Invoke):** Создание стекового кадра, передача аргументов, переход на первую инструкцию.
2. **Return:** Уничтожение стекового кадра, возврат результата, переход к точке вызова.

Корутина добавляет две новые операции:

1. **Suspend (Приостановка):** Сохранение текущего состояния исполнения (локальные переменные, instruction pointer) в выделенную область памяти (не на стек потока!). Управление возвращается вызывающей стороне (Caller) или планировщику.
2. **Resume (Возобновление):** Восстановление состояния из памяти и продолжение исполнения с точки остановки.

Операции управления потоком

Subroutine:
- Start $\rightarrow$ Execute $\rightarrow$ Terminate

Coroutine:
- Start $\rightarrow$ Execute $\rightarrow$ **Suspend** (Save State) $\rightarrow$ Return Control
- ... (Time passes) ...
- **Resume** (Restore State) $\rightarrow$ Execute $\rightarrow$ Terminate

Прототипирование ментальной модели (Python)

Прежде чем переходить к C++, рассмотрим семантику корутин на языке Python. Механизм генераторов в Python концептуально идентичен тому, что мы хотим получить в C++, хотя реализация кардинально отличается (интерпретатор vs компилятор).

Задача: реализовать генератор последовательности чисел (аналог `range`), который не хранит весь массив в памяти, а вычисляет следующее число по требованию.

d

```
def range_gen(max_val):
    num = 0
    while True:
        # yield возвращает значение и "замораживает" функцию
        yield num
        num += 1
        if num >= max_val:
            return

# Использование
gen = range_gen(3)

print(next(gen)) # Вывод: 0. Функция дошла до yield num
print(next(gen)) # Вывод: 1. Функция продолжилась с num += 1
print(next(gen)) # Вывод: 2
# print(next(gen)) # StopIteration
```

Разбор механики

Ключевое слово `yield` выполняет роль **Suspend + Output**.

1. При первом вызове `next(gen)` функция запускается, инициализирует `num = 0`.
2. Доходит до `yield num`.
3. Интерпретатор сохраняет состояние (значение `num`, позицию инструкции) в объект генератора `gen`.
4. Функция «возвращает» 0 и приостанавливается.
5. При следующем вызове `next(gen)` состояние восстанавливается. Исполнение продолжается **сразу после** `yield`. Выполняется `num += 1`.
6. Цикл `while` отправляет исполнение снова на `yield`.

> **Важно!**
>
> С точки зрения вызывающего кода (Caller), корутина — это объект, который можно опрашивать. С точки зрения самой корутины — это непрерывный поток выполнения, который иногда ставится на паузу.

В C++20 мы стремимся к аналогичной семантике, но с требованием **Zero Overhead**. Компилятор должен преобразовать код с точками остановки в конечный автомат (State Machine) на этапе компиляции, избегая динамической аллокации тяжелых структур интерпретатора.

Резюме раздела

- Классические потоки неэффективны для I/O-нагрузки из-за накладных расходов на контекст и память.
- Корутины реализуют User Space Scheduling, позволяя «останавливать» вычисления без блокировки ядра.
- Основные примитивы корутин: Suspend (сохранить состояние) и Resume (восстановить состояние).
- Генераторы Python — хорошая ментальная модель для понимания потока управления (Control Flow) в корутинах.

Глава 62

Архитектурный выбор: Stackful vs Stackless

При разработке стандарта C++20 перед комитетом стоял фундаментальный выбор архитектуры корутин. Существует два полярных подхода к реализации асинхронности: **Stackful** (с собственным стеком) и **Stackless** (бесстековые).

Этот выбор определяет не только синтаксис, но и характеристики производительности, потребление памяти и возможности оптимизации. C++ пошел по пути *Stackless*, что отличает его от Go (Goroutines) или Java (Project Loom). В этой главе мы разберем технические детали обоих подходов и причины выбора C++.

Stackful Coroutines (Fibers)

Stackful корутины (часто называемые Файберами или Зелеными потоками) — это прямая проекция модели потоков операционной системы в пространство пользователя.

Механика работы

Каждая Stackful корутина при создании аллоцирует **собственный непрерывный блок памяти под стек**. Этот стек используется для хранения:

- Локальных переменных текущей функции.
- Адресов возврата (Return Addresses) при вложенных вызовах функций.
- Аргументов функций.

Переключение контекста (Context Switch) между двумя файберами выглядит так:

1. Сохранить регистры процессора (CPU Registers: RIP, RSP, RBP и регистры общего назначения) в текущий стек или специальную структуру контекста.
2. Подменить указатель стека (`RSP`) на стек целевого файбера.
3. Восстановить регистры целевого файбера.
4. Выполнить инструкцию перехода (`JMP` или `RET`).

С точки зрения исполняемого кода, файбер ничем не отличается от потока. Функция может уйти в глубокую рекурсию, вызвать десять вложенных функций, и где-то на глубине 11-

го уровня вызвать `yield()`. Поскольку у файбера есть свой стек, всё состояние цепочки вызовов сохраняется автоматически.

Достоинства и Недостатки

Плюсы:

- **Прозрачность для кода:** Можно взять старую синхронную библиотеку, запустить её внутри файбера, и если она использует блокирующие вызовы (переопределенные через `yield`), она станет асинхронной без переписывания кода.
- **Произвольная вложенность:** Приостановка возможна в любой точке стека вызовов.

Минусы:

- **Накладные расходы на память:** Это главная проблема. Сколько памяти выделить под стек?
  - Если выделить мало (4 КБ), возможен *Stack Overflow* при глубокой рекурсии.
  - Если выделить много (1 МБ) с запасом, то миллион корутин потребует 1 ТБ виртуальной памяти.
- **Segmented Stacks / Stack Copying:** Чтобы решить проблему размера, языки вроде Go используют динамически растущие стеки. Это вносит оверхед: при каждом вызове функции нужно проверять, хватает ли стека, и при необходимости переаллоцировать его и копировать данные. Это нарушает ABI C++ и несовместимо с указателями на локальные переменные (адреса меняются при перемещении стека).

Stackless Coroutines (C++20)

C++20 выбрал модель Stackless. В этой модели корутина **не имеет собственного стека**. Она использует стек вызывающего потока (Thread Stack) для выполнения кода между точками приостановки.

> **Stackless Coroutine**
>
> Функция, которая может быть приостановлена только на верхнем уровне своего тела (в точках использования ключевых слов `co_await`/`co_yield`). Состояние сохраняется не на стеке, а в специально сгенерированном объекте в куче.

State Machine Transformation

Компилятор преобразует функцию-корутину в конечный автомат (State Machine).

1. Анализируется тело функции.
2. Все локальные переменные, время жизни которых пересекает точку приостановки (`suspend point`), переносятся из стека в поля класса-автомата.
3. Точки входа в корутину размечаются через `switch/case` или `goto` метки.

Это фундаментальное отличие. Вместо сохранения *всего* стека, сохраняется только *минимально необходимое* состояние конкретной функции.

Under the hood: Трансформация кода

Рассмотрим, как концептуально происходит трансформация.

Исходный код корутины (C++):

```
generator<int> sequence(int start) {
    int x = start;       // Локальная переменная
    int y = 42;          // Переменная, живущая "сквозь" yield

    // Точка остановки 1
    co_yield x;

    x += y;

    // Точка остановки 2
    co_yield x;
}
```

Результат компиляции (Псевдокод C++):

Компилятор генерирует структуру, часто называемую *Coroutine Frame*.

```
struct Sequence_Frame {
    // 1. Служебные поля (Promise, Resume Point)
    int _resume_point = 0; // Индекс точки возобновления
    promise_type _promise;

    // 2. Сохраненные аргументы
    int start;

    // 3. Сохраненные локальные переменные
    // 'x' и 'y' нужны после resume, поэтому они здесь.
    // Если бы была переменная 'z', которая умирает до co_yield,
    // она бы осталась на стеке и не попала в Frame.
    int x;
    int y;

    void resume() {
        switch (_resume_point) {
            case 0: goto STATE_0;
            case 1: goto STATE_1;
            case 2: goto STATE_2;
        }

    STATE_0:
        x = start;
        y = 42;

        // co_yield x -> suspend
        _promise.value = x;
        _resume_point = 1;
        return; // Возврат управления вызывающему (Caller)

```

```
    STATE_1:
        x += y; // Восстановили контекст, выполняем операцию

        // co_yield x -> suspend
        _promise.value = x;
        _resume_point = 2;
        return;

    STATE_2:
        // Завершение корутины
        _promise.return_void();
    }
};
```

Анализ эффективности

- **Размер:** Структура `Sequence_Frame` занимает ровно столько байт, сколько нужно для хранения `int x, int y` и служебных полей. Это десятки байт, а не килобайты стека.
- **Аллокация:** Фрейм выделяется в куче (Heap Allocation) один раз при старте корутины. C++ позволяет оптимизировать это через HALO (Heap Allocation Elision), если компилятор видит, что время жизни корутины полностью вложено в вызывающую функцию.
- **Переключение:** Вызов `resume()` — это непрямой вызов функции (indirect function call) плюс `switch`. Это дешевле, чем подмена регистров и стека процессора.

Почему C++ выбрал Stackless?

Выбор в пользу Stackless был продиктован философией C++: "**Zero Overhead Abstraction**".

1. **Масштабируемость (Scalability):** Stackless корутины потребляют память пропорционально количеству реальных данных, а не глубине резервируемого стека. Это позволяет создавать **миллиарды** корутин на одной машине. В случае Stackful, даже 4 КБ стека на корутину ограничили бы нас 250 000 корутин на 1 ГБ памяти.
2. **Отсутствие магического рантайма:** Stackful корутины требуют сложного менеджера памяти для стеков (сборка мусора стеков, сплит-стеки). Stackless корутины компилируются в простые структуры данных, с которыми можно работать стандартными аллокаторами.
3. **Взаимодействие с C ABI:** Stackless корутина при возобновлении использует обычный стек потока. Это значит, что внутри корутины можно вызывать обычный C-код, использовать указатели на стек и все оптимизации компилятора. Stackful реализации часто ломают совместимость с существующим кодом из-за перемещения стеков в памяти.

Резюме раздела

- **Stackful (Fibers):** Эмуляция потоков. Просто писать код, но дорого по памяти. Требует отдельного стека.
- **Stackless (C++20):** Синтаксический сахар над конечными автоматами. Нет своего стека.
- C++ использует **Stackless** для минимизации оверхеда. Компилятор "нарезает" функцию на куски и сохраняет локальные переменные в объект-фрейм в куче.

Глава 63

Механика C++20: Триада Promise, Handle, Return Object

C++20 предоставляет самый мощный, но и самый низкоуровневый API для корутин среди популярных языков программирования. В то время как Python или C# скрывают механизмы управления состоянием за простыми ключевыми словами, C++ заставляет разработчика самостоятельно проектировать поведение корутины.

Это сделано намеренно: стандарт C++20 определяет не «корутины» как готовую фичу (feature), а «фреймворк для создания корутин». В этой главе мы разберем анатомию этого фреймворка, состоящего из трех взаимосвязанных сущностей: **Promise Type**, **Coroutine Handle** и **Return Object**.

Синтаксис и ключевые слова

Корутина в C++ определяется не сигнатурой функции, а её телом. Если в теле функции встречается хотя бы одно из следующих ключевых слов, компилятор автоматически считает её корутиной и применяет трансформацию в конечный автомат:

- `co_await <expr>` — приостановить исполнение до завершения асинхронной операции.
- `co_yield <expr>` — вернуть промежуточное значение и приостановиться (генератор).
- `co_return <expr>` — завершить выполнение корутины и вернуть финальный результат.

На заметку

Почему префикс `co_`? При разработке стандарта (C++20) возникла проблема обратной совместимости. Миллионы строк кода уже использовали переменные с именами `await`, `yield` или `return`. Введение таких ключевых слов сломало бы существующий код. Комитет выбрал префикс `co_` (от coroutine). Существует шутка, что это также напоминает химическую формулу угарного газа (CO), намекая на опасность неправильного использования этого механизма.

Архитектура Триады

Чтобы функция могла стать корутиной, её возвращаемый тип должен удовлетворять особому контракту. Этот контракт связывает три объекта.

1. Promise Type (Обещание)

Это «мозг» корутины. Пользовательский тип, который живет внутри фрейма корутины.

- Хранит входные аргументы и локальные переменные (если они перемещены во фрейм).
- Аккумулирует результаты (`co_yield` или `co_return` пишут данные сюда).
- Определяет точки остановки при старте и завершении.
- Обрабатывает исключения, вылетевшие из тела корутины.

2. Return Object (Возвращаемый объект)

Это «интерфейс» для вызывающего кода (Caller). То, что возвращает функция-корутина.

- Обычно реализует паттерн RAII: владеет корутиной и уничтожает её в деструкторе.
- Предоставляет методы для внешнего управления: `next()`, `get()`, `future.wait()`.
- Создается внутри Promise в самом начале работы.

3. Coroutine Handle (Ручка)

Это низкоуровневый «указатель» (`void*` wrapper) на фрейм корутины. Предоставляется стандартной библиотекой (`std::coroutine_handle`).

- Позволяет возобновить (`resume`) или уничтожить (`destroy`) корутину.
- Позволяет получить доступ к Promise извне (`handle.promise()`).
- Передается между Promise и Return Object как связующее звено.

Схема взаимодействия

Compiler $\xrightarrow{\text{creates}}$ **Frame (Heap)**
Внутри Frame живет **Promise**.
Promise $\xrightarrow{\text{creates}}$ **Return Object**.
Return Object содержит **Handle**, указывающий на Frame.
Caller владеет **Return Object**.

Алгоритм запуска корутины (Under the hood)

Когда вы вызываете корутину `my_coro()`, компилятор генерирует следующий код (упрощенно):

```
// Псевдокод того, что делает компилятор
RetType my_coro_transformed(Args... args) {
    // 1. Выделение памяти под фрейм (через operator new)
    // Размер вычисляется компилятором (Promise + Locals + Save points)
    void* frame_mem = allocate_frame(sizeof(Frame));

    // 2. Конструирование Promise в этой памяти
    using P = RetType::promise_type; // Магия поиска типа!
    P* promise = new (frame_mem) P(args...);

    // 3. Создание Return Object, который уйдет наружу
    RetType return_val = promise->get_return_object();

    // 4. Начальная приостановка
    try {
        co_await promise->initial_suspend();
        // Если suspend_always -> возврат управления caller'у сразу.
        // Если suspend_never -> выполнение тела до первого реального await.

        // ... выполнение тела корутины ...

    } catch (...) {
        promise->unhandled_exception();
    }

    // 5. Финальная стадия
    co_await promise->final_suspend();

    return return_val; // Возврат объекта, созданного на шаге 3
}
```

Этот алгоритм жестко зашит в компилятор. Мы не можем его изменить, но мы можем настроить каждый шаг, реализуя методы в `promise_type`.

Реализация Hello World (Resumable)

Напишем минимальную корутину, которая выводит "Hello", приостанавливается, а при возобновлении выводит "World".

Для этого нам нужно определить тип возвращаемого значения `Resumable`. Компилятор будет искать в нем вложенный тип `promise_type`.

```
#include <coroutine>
#include <iostream>
#include <exception>

// 1. Return Object - то, что видит пользователь
struct Resumable {
    // Обязательное объявление promise_type
    struct promise_type;

    // Храним handle для управления
```

```
    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle_;

    Resumable(handle_type h) : handle_(h) {}

    // RAII: Уничтожаем корутину при выходе
    ~Resumable() {
        if (handle_) handle_.destroy();
    }

    // Метод для ручного возобновления
    void resume() {
        if (handle_ && !handle_.done()) {
            handle_.resume(); // Переход управления внутрь корутины
        }
    }

    // 2. Promise Type - внутренняя кухня
    struct promise_type {
        // Шаг 3 алгоритма: создание Resumable
        Resumable get_return_object() {
            return Resumable{handle_type::from_promise(*this)};
        }

        // Шаг 4: Start Policy
        // suspend_always = Ленивый старт. Корутина создается, но тело не
        ↪  запускается.
        std::suspend_always initial_suspend() { return {}; }

        // Шаг 5: End Policy
        // suspend_always = Не уничтожать фрейм автоматически после завершения.
        // Это важно, чтобы мы могли проверить .done() снаружи.
        std::suspend_always final_suspend() noexcept { return {}; }

        // Обработка исключений
        void unhandled_exception() { std::terminate(); }

        // Обработка co_return (для void корутин)
        void return_void() {}
    };
};

// Сама функция-корутина
Resumable hello_coro() {
    std::cout << "Hello, ";
    co_await std::suspend_always{}; // Явная приостановка
    std::cout << "World!" << std::endl;
} // Здесь неявный co_return, вызывающий promise.return_void()

int main() {
    // 1. Создание корутины. Выполняется initial_suspend.
    Resumable coro = hello_coro();

    std::cout << "Main: Coroutine created." << std::endl;

```

```
65      // 2. Первый resume. Выполняется "Hello, " и остановка на co_await.
66      coro.resume();
67
68      std::cout << "Main: Coroutine suspended." << std::endl;
69
70      // 3. Второй resume. Выполняется "World!" и выход.
71      coro.resume();
72
73      std::cout << "Main: Coroutine finished." << std::endl;
74
75      return 0; // Деструктор ~Resumable уничтожит фрейм
76  }
```

Анализ Boilerplate-кода

Разберем ключевые методы `promise_type`, которые мы реализовали.

get_return_object()

Этот метод вызывается до начала исполнения тела корутины. Его задача — создать объект `Resumable`, связав его с текущим просимом.

- `std::coroutine_handle<promise_type>::from_promise(*this)` — магический метод, который вычисляет адрес начала фрейма, зная адрес просиса (они лежат в памяти рядом).

initial_suspend()

Определяет стратегию запуска.

- Возврат `std::suspend_always`: Корутина создается в замороженном состоянии. Тело не выполняется ни на шаг. Это типично для генераторов (ленивые вычисления).
- Возврат `std::suspend_never`: Корутина сразу начинает выполнение (Eager execution) до первого `co_await`. Это типично для асинхронных задач (Tasks), которые должны начать работу немедленно (например, отправить сетевой запрос).

final_suspend()

Определяет поведение при достижении закрывающей фигурной скобки `}`.

- `suspend_never`: Фрейм корутины уничтожается автоматически сразу после завершения тела. Это опасно, если у нас остался `handle` снаружи — он станет висячим (dangling). Используется для "Fire and Forget" корутин.
- `suspend_always`: Корутина "зависает" в финальной точке. Фрейм и переменные внутри Promise остаются живыми. Это позволяет вызывающему коду безопасно считать результат или проверить статус `.done()`. Обязанность уничтожить фрейм ложится на `handle.destroy()`.

std::coroutine_handle

Это типизированная обертка над `void*`.

```
template <typename Promise = void>
struct coroutine_handle;
```

- `resume()`: Восстанавливает регистры и переходит по адресу приостановки (хранится во фрейме). Если корутина завершена (`done() == true`), вызов `resume()` — это Undefined Behavior (часто краш).
- `done()`: Проверяет, находится ли корутина в точке `final_suspend`.
- `destroy()`: Принудительно вызывает деструкторы объектов во фрейме и освобождает память.
- `promise()`: Возвращает ссылку на `Promise` объект. Доступно только для типизированного хендла (`Promise != void`).

> **Важно!**
>
> `coroutine_handle` не является RAII-объектом. Он ведет себя как сырой указатель. Если вы потеряете хендл и не вызовете `destroy()`, произойдет утечка памяти (фрейм останется в куче). Именно поэтому мы оборачиваем его в класс `Resumable`.

Резюме раздела

C++20 предоставляет конструктор, а не готовое здание.

1. **Promise** определяет семантику (ленивая/жадная, генератор/задача).
2. **Return Object** инкапсулирует владение (RAII).
3. **Handle** обеспечивает механику переключения.

Правильная реализация `initial_suspend` и `final_suspend` критически важна для корректного управления памятью и временем жизни корутины.

Глава 64

Жизненный цикл и типичные ошибки (Live Coding Analysis)

Управление памятью в корутинах C++20 кардинально отличается от привычной семантики функций. Если в обычных функциях стековый кадр уничтожается автоматически при возврате (`ret`), то время жизни фрейма корутины управляется сложным конечным автоматом, который программист настраивает через `promise_type`.

Непонимание работы методов `initial_suspend` и `final_suspend` приводит к двум классам критических ошибок: утечкам памяти (Memory Leaks) и использованию памяти после освобождения (Use-After-Free). В этой главе мы разберем классический пример падения программы (SEGFAULT), который часто демонстрируется на собеседованиях и лекциях.

Точки приостановки (Suspend Points)

Компилятор вставляет вызовы `await_suspend` в две ключевые точки жизненного цикла корутины: перед началом исполнения тела и после его завершения.

1. initial_suspend: Жадность против Лени

Метод `promise.initial_suspend()` вызывается сразу после создания фрейма, но до входа в пользовательский код.

- **Lazy Start (`std::suspend_always`):** Корутина создается в приостановленном состоянии. Это стандарт для генераторов (sequence generators). Вычислительные ресурсы не тратятся, пока пользователь явно не вызовет `resume()`.
- **Eager Start (`std::suspend_never`):** Корутина начинает выполнение немедленно. Это стандарт для задач (Tasks), которые запускают фоновую работу (например, сетевой запрос) сразу при создании.

2. final_suspend: Кто убивает фрейм?

Это самая опасная точка настройки. Метод вызывается, когда исполнение доходит до закрывающей фигурной скобки `}` или `co_return`.

- **Автоматическое самоуничтожение (`std::suspend_never`):** Корутина считает, что её работа закончена, и никто снаружи не ждет результатов. Фрейм немедленно деаллоцируется.

- **Продление жизни (`std::suspend_always`):** Корутина приостанавливается "на пороге смерти". Фрейм остается в памяти. Это необходимо, если внешний код (Caller) хочет считать результат из `promise` или проверить состояние. Обязанность вызвать `destroy()` ложится на внешний код.

Анатомия краша: Case Study

Рассмотрим код, который компилируется без предупреждений, но гарантированно падает (или вызывает UB) при запуске.

Сценарий: Мы хотим написать корутину, которая выполняется до конца, а мы снаружи проверяем её статус через `.done()`.

```
#include <coroutine>
#include <iostream>

struct BrokenTask {
    struct promise_type {
        BrokenTask get_return_object() {
            return {std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; } // Lazy start

        // ОШИБКА ЗДЕСЬ: Мы разрешаем фрейму умереть сразу
        std::suspend_never final_suspend() noexcept { return {}; }

        void unhandled_exception() {}
        void return_void() {}
    };

    std::coroutine_handle<promise_type> h;

    // В деструкторе ничего не делаем, надеясь на автоматику?
    ~BrokenTask() {}
};

BrokenTask my_coro() {
    std::cout << "Coro: working..." << std::endl;
    co_return; // Точка завершения
}

int main() {
    BrokenTask task = my_coro();

    task.h.resume(); // Запускаем. Выводит "Coro: working..."

    // КРИТИЧЕСКАЯ ОШИБКА: Use-After-Free
    if (task.h.done()) {
        std::cout << "Done!" << std::endl;
    }

    return 0;
}
```

Разбор механики падения

Почему строка `task.h.done()` вызывает неопределенное поведение? Проследим хронологию событий в памяти:

1. `main` вызывает `task.h.resume()`.
2. Управление переходит в `my_coro`.
3. `my_coro` выполняет тело и доходит до `co_return`.
4. Вызывается `promise.return_void()`.
5. Вызывается `co_await promise.final_suspend()`.
6. `final_suspend` возвращает `suspend_never` («не останавливайся»).
7. Поскольку остановки не произошло, выполняется процедура уничтожения корутины:
   - Вызывается деструктор `promise_type`.
   - Освобождается память фрейма (Heap Deallocation).
8. Управление возвращается из `resume()` обратно в `main`.
9. В `main` выполняется `task.h.done()`. Хендл `h` хранит адрес памяти, которая **уже освобождена** на шаге 7.

Это классический **Dangling Pointer**. Доступ к освобожденной памяти может вернуть мусор, вызвать Segmentation Fault или (что хуже) вернуть `true`/`false` случайным образом, создавая плавающий баг.

> **Важно!**
>
> Если вы используете `std::suspend_never` в `final_suspend`, ваш `coroutine_handle` становится невалидным сразу после возврата управления из последнего `resume()`. Вы не имеете права вызывать на нем никакие методы, включая `done()`.

Паттерн безопасного завершения

Чтобы безопасно опрашивать корутину после завершения (например, чтобы забрать результат вычислений), фрейм должен пережить само тело функции.

Исправление Promise Type

Изменим `final_suspend`:

```
struct promise_type {
    // ...
    // ТЕПЕРЬ ПРАВИЛЬНО: Зависаем в конце
    std::suspend_always final_suspend() noexcept { return {}; }
};
```

Теперь хронология меняется:

1. `co_return` вызывает `final_suspend`.
2. Возвращается `suspend_always`.

3. Корутина приостанавливается в состоянии «завершена, но жива». Фрейм в памяти.
4. Управление возвращается в `main`.
5. `task.h.done()` обращается к живому фрейму и корректно возвращает `true`.

Проблема утечки памяти

Теперь у нас новая проблема. Так как корутина "зависла" в конце, она сама себя не удалила. Если мы просто выйдем из `main`, память фрейма утечет (Memory Leak).

Мы обязаны вызвать `handle.destroy()` вручную. Лучшее место для этого — деструктор RAII-обертки.

```
struct SafeTask {
    // ... promise_type с final_suspend = suspend_always ...

    std::coroutine_handle<promise_type> h;

    SafeTask(std::coroutine_handle<promise_type> handle) : h(handle) {}

    // Запрещаем копирование (чтобы не удалить дважды)
    SafeTask(const SafeTask&) = delete;

    // Разрешаем перемещение
    SafeTask(SafeTask&& other) noexcept : h(other.h) {
        other.h = nullptr;
    }

    ~SafeTask() {
        // Если хендл валиден, мы обязаны его уничтожить
        if (h) h.destroy();
    }
};
```

Fire-and-Forget корутины

Существует единственный сценарий, где `final_suspend` должен возвращать `suspend_never`. Это корутины, которые не возвращают никакого объекта управления (возвращаемый тип `void` или отвязанный тип).

Пример: корутина, которая запускает анимацию или логирование и о которой вызывающий код сразу забывает.

```
struct Detached {
    struct promise_type {
        Detached get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; } // Сразу старт
        std::suspend_never final_suspend() noexcept { return {}; } // Сама умрет
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };
```

```
};

Detached log_async() {
    // ... какая-то работа ...
    co_return; // Фрейм уничтожится здесь автоматически
}
```

В таком случае у нас нет `coroutine_handle` снаружи, поэтому риск Use-After-Free отсутствует. Но теряется возможность контроля и обработки ошибок.

Резюме раздела

Правила выживания:

1. Если у вас есть доступ к `coroutine_handle` снаружи, `final_suspend` **обязан** возвращать `std::suspend_always`.
2. В этом случае вы **обязаны** вызвать `handle.destroy()`, иначе будет утечка.
3. Используйте RAII-обертки (как `SafeTask`), чтобы автоматизировать вызов `destroy()`.
4. Никогда не копируйте сырые `coroutine_handle`, используйте семантику перемещения (move semantics).

Глава 65

Генераторы данных: Трансформация co_yield

До сих пор мы рассматривали корутины как механизм управления потоком исполнения (Control Flow). Однако в большинстве прикладных задач (Python generators, C# `yield return`) корутины используются как источники данных.

Оператор `co_yield` превращает корутину в генератор — функцию, которая производит последовательность значений лениво (lazy evaluation), возвращая управление вызывающей стороне после генерации каждого элемента. В этой главе мы реализуем полноценный класс `Generator<T>`, совместимый с range-based for циклами C++.

Семантика co_yield

Ключевое слово `co_yield` — это синтаксический сахар. Компилятор C++20 преобразует выражение `co_yield expr` в цепочку вызовов, проходящую через `promise_type`.

Выражение:

```
co_yield some_value;
```

Трансформируется в:

```
co_await promise.yield_value(some_value);
```

Это означает, что механизм передачи данных полностью контролируется методом `yield_value` внутри промиса.

Архитектура Генератора

Чтобы корутина работала как генератор, нам нужно решить две задачи:

1. **Exfiltration (Экфильтрация данных):** Как передать значение из локальной переменной корутины во внешний мир?
2. **Iteration (Итерация):** Как интегрировать низкоуровневые методы `resume()` и `done()` в стандартный интерфейс итераторов C++ (`begin`, `end`, `operator++`).

1. Promise как буфер обмена

Поскольку `promise_type` живет внутри фрейма корутины, он является идеальным местом для временного хранения "выброшенного" значения.

```
template<typename T>
struct Generator {
    struct promise_type {
        // Буфер для значения. Используем указатель для эффективности
        // (избегаем лишних копирований) и возможности вернуть nullptr.
        T* current_value = nullptr;

        // Точка расширения для co_yield
        std::suspend_always yield_value(T& value) noexcept {
            current_value = std::addressof(value); // Сохраняем адрес
            return {}; // Приостанавливаем корутину!
        }

        // Стандартный boilerplate
        Generator get_return_object() {
            return
             ↪  Generator{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; } // Ленивый старт
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };
    // ...
};
```

Важно!

Обратите внимание на возвращаемый тип `yield_value`. Мы возвращаем `std::suspend_always`. Это критически важно.

- Если бы мы вернули `suspend_never`, корутина записала бы значение в `current_value` и **немедленно** продолжила бы выполнение.
- В цикле генерации это привело бы к перезаписи значения следующим элементом до того, как потребитель (Consumer) успел бы его прочитать.

Остановка (`suspend`) гарантирует, что управление перейдет к потребителю, который прочитает значение и только потом запросит следующее.

2. Итераторный интерфейс

Чтобы использовать генератор в диапазонах (`for (auto v : gen)`), класс `Generator` должен предоставлять методы `begin()` и `end()`. Итератор будет оберткой над `coroutine_handle`.

```
// Внутри класса Generator<T>

    struct Iterator {
```

```
        std::coroutine_handle<promise_type> handle;

        // operator++: Продвижение вперед = resume()
        Iterator& operator++() {
            handle.resume();
            if (handle.done()) {
                // Если корутина завершилась, разыменовывать больше нельзя
                handle = nullptr;
            }
            return *this;
        }

        // operator*: Доступ к данным = чтение из promise
        T& operator*() const {
            return *handle.promise().current_value;
        }

        // Сравнение с sentinel (end iterator)
        bool operator!=(std::default_sentinel_t) const {
            return handle != nullptr && !handle.done();
        }
    };

    Iterator begin() {
        if (handle_) {
            handle_.resume(); // Первый шаг, чтобы дойти до первого yield
            if (handle_.done()) return end();
        }
        return Iterator{handle_};
    }

    std::default_sentinel_t end() { return {}; }
```

Полная реализация и пример использования

Соберем всё вместе в законченный класс. Мы используем `std::default_sentinel_t` для упрощения логики завершения итерации (C++20 feature).

```
#include <coroutine>
#include <iostream>
#include <memory>

template<typename T>
struct Generator {
    struct promise_type {
        const T* current_value = nullptr;

        Generator get_return_object() {
            return
             ↪  Generator{std::coroutine_handle<promise_type>::from_promise(*this)};
        }

```

```
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }

        // Поддержка co_yield val;
        std::suspend_always yield_value(const T& value) noexcept {
            current_value = std::addressof(value);
            return {};
        }

        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };

    using Handle = std::coroutine_handle<promise_type>;
    Handle handle_;

    explicit Generator(Handle h) : handle_(h) {}

    ~Generator() {
        if (handle_) handle_.destroy();
    }

    // Запрет копирования (Generator владеет ресурсом)
    Generator(const Generator&) = delete;
    Generator& operator=(const Generator&) = delete;

    // Перемещение
    Generator(Generator&& other) noexcept : handle_(other.handle_) {
        other.handle_ = nullptr;
    }

    // Итераторная часть
    struct Iterator {
        Handle handle;

        void operator++() {
            handle.resume();
            if (handle.done()) handle = nullptr;
        }

        const T& operator*() const {
            return *handle.promise().current_value;
        }

        bool operator!=(std::default_sentinel_t) const {
            return handle != nullptr && !handle.done();
        }
    };

    Iterator begin() {
        if (handle_) {
            handle_.resume();
            if (handle_.done()) return end();
        }
        return Iterator{handle_};
```

```
    }

    std::default_sentinel_t end() { return {}; }
};

// --- Пользовательский код ---

Generator<int> fibonacci(int limit) {
    int a = 0, b = 1;
    while (a < limit) {
        co_yield a; // Здесь происходит магия

        int t = a;
        a = b;
        b = t + b;
    }
}

int main() {
    std::cout << "Fibonacci sequence: ";

    // Range-based for loop работает прозрачно
    for (int num : fibonacci(100)) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Анализ потока управления (Control Flow)

Разберем пошагово, что происходит в строке `for (int num : fibonacci(100))`.

1. **Инициализация:** Вызывается `fibonacci(100)`. Создается фрейм, promise, возвращается объект `Generator`. Корутина стоит на `initial_suspend`.
2. **begin():** Цикл вызывает `gen.begin()`. Внутри вызывается `handle.resume()`.
3. **Вход в корутину:** Исполнение переходит внутрь `fibonacci`. Инициализируются `a=0`, `b=1`.
4. **Первый yield:** Доходим до `co_yield a`.
   - Вызывается `promise.yield_value(0)`.
   - Поле `promise.current_value` указывает на локальную переменную `a` (во фрейме).
   - Возвращается `suspend_always`. Корутина приостанавливается.
5. **Возврат в begin():** `resume()` возвращает управление. `begin()` возвращает итератор.
6. **Тело цикла:** Разыменование итератора `*it` читает значение `0` из промиса. `std::cout` печатает `0`.
7. **Инкремент:** Цикл вызывает `operator++`. Он вызывает `handle.resume()`.

8. **Продолжение корутины:** Исполнение продолжается сразу после `co_yield`.
9. Вычисляются новые `a` и `b`.
10. Цикл `while` переходит на новую итерацию.
11. **Второй yield:** Снова `co_yield a` (теперь `1`). Процесс повторяется.

Сравнение сложности

Вспомним, как приходилось писать итераторы до C++20 (пример из первой главы). Нам нужно было вручную создавать класс, хранить состояние (`current_a`, `current_b`), реализовывать операторы. Логика генерации была "размазана" по методам класса.

С использованием корутин:

- **Логика:** Написана линейно, как обычный алгоритм (функция `fibonacci`).
- **Состояние:** Автоматически сохраняется компилятором во фрейме.
- **Инфраструктура:** Код класса `Generator<T>` пишется **один раз** в библиотеке. Прикладной программист пишет только бизнес-логику.

Резюме раздела

- Оператор `co_yield` — это способ коммуникации корутины с её промисом.
- Метод `yield_value` должен возвращать `awaitable` (обычно `suspend_always`), чтобы дать потребителю время обработать данные.
- Превращение корутины в `InputRange` требует реализации стандартного паттерна итератора, который внутри себя дергает `resume()` и читает данные из промиса.
- Этот подход позволяет писать сложные алгоритмы генерации последовательностей (обход графов, бесконечные потоки) в линейном стиле, сохраняя производительность конечного автомата.

Глава 66

Асинхронное ожидание: Концепция Awaitable

Если `co_yield` — это инструмент для *исходящего* потока данных (Output), то `co_await` предназначен для *ожидания* событий (Input/Completion). Это самый сложный и мощный оператор в C++20, превращающий синхронный код в асинхронный конечный автомат.

Именно `co_await` позволяет корутине "уснуть" в ожидании завершения сетевого запроса, таймера или задачи в другом потоке, и "проснуться" ровно в тот момент, когда результат готов. В этой главе мы разберем механику трансформации этого оператора и напишем свой примитив синхронизации.

Механика трансформации co_await

Оператор `co_await <expr>` — это унарный оператор. Компилятор преобразует его не в вызов одной функции, а в сложную последовательность действий, известную как **Await Protocol**.

Пусть у нас есть выражение:

```
auto result = co_await <expr>;
```

Компилятор генерирует следующий код (псевдокод):

```
{
    // 1. Получение "Awaiter" - объекта, который знает, как ждать
    auto&& awaiter = get_awaiter(<expr>);

    // 2. Оптимизация горячего пути (Fast Path)
    if (!awaiter.await_ready()) {

        // 3. Сохранение состояния корутины (Suspend)
        <suspend-coroutine-state>

        // 4. Интеграция с внешним миром
        // handle - это дескриптор текущей, уже остановленной корутины
        using Handle = std::coroutine_handle<P>;

```

```
        // Результат определяет, что делать дальше
        auto suspend_result = awaiter.await_suspend(Handle::from_promise(p));

        // Логика обработки результата await_suspend (см. далее)
        if (suspend_result == false) {
            <resume-immediately>
        }

        // Точка, где корутина физически возвращает управление
        // своему вызывающему (Caller) или переходит в другую корутину
        <return-to-caller-or-jump>
    }

    // 5. Точка возобновления (Resume Point)
    // Сюда мы попадаем после handle.resume()

    // 6. Получение результата
    result = awaiter.await_resume();
}
```

Концепт Awaitable

Чтобы объект мог быть операндом `co_await`, он (или результат оператора `operator co_await`) должен реализовывать три метода.

1. await_ready() → bool

Это механизм оптимизации. Метод вызывается **до** реальной остановки корутины.

- Возвращает `true`: "Результат уже готов". Компилятор пропускает шаги 3 и 4 (Suspend и `await_suspend`). Корутина продолжает исполнение синхронно. Это экономит циклы процессора на сохранение регистров и переключение контекста.
- Возвращает `false`: "Результат не готов, нужно ждать". Запускается процедура приостановки.

Пример: Если вы делаете `co_await TryLockAsync()`, и мьютекс свободен, нет смысла усыплять корутину. `await_ready` вернет `true`.

2. await_suspend(handle) → void | bool | handle

Это самый важный метод. Он вызывается, когда корутина **уже остановлена** (все регистры сохранены во фрейм). Аргумент `handle` — это "пульт управления" текущей корутиной.

Именно здесь происходит передача ответственности. Мы должны сохранить `handle` куда-то, откуда его потом вызовут (в очередь ThreadPool, в callback сетевой библиотеки, в структуру таймера).

Варианты возвращаемого значения:

- `void`: Безусловная остановка. Управление возвращается тому, кто вызвал/возобновил эту корутину.

- `bool`: Условная остановка. Если вернуть `false`, корутина немедленно просыпается (как будто `await_ready` вернул `true`). Это нужно для разрешения гонок (Race Conditions), когда результат появился ровно в момент засыпания.
- `coroutine_handle`: **Symmetric Transfer**. Управление передается не вызывающему (Caller), а той корутине, чей хендл мы вернули. Это позволяет делать "хвостовые вызовы" корутин (Tail Call Optimization) и избегать переполнения стека при переключении между множеством корутин.

3. await_resume() → T

Вызывается при возобновлении. Результат этого метода становится результатом всего выражения `co_await`. Если в процессе ожидания произошла ошибка (например, разрыв соединения), здесь принято выбрасывать исключение.

Практика: Переключение потоков (Thread Switcher)

Напишем кастомный Awaitable, который переносит исполнение корутины в фоновый поток. Это база для реализации Thread Pool.

```
#include <coroutine>
#include <thread>
#include <iostream>

struct ResumeOnNewThread {
    // 1. Всегда останавливаемся, чтобы сменить поток
    bool await_ready() const noexcept {
        return false;
    }

    // 2. Логика переключения
    void await_suspend(std::coroutine_handle<> h) const {
        // Создаем новый поток и передаем ему ответственность за handle.
        // В реальном коде здесь была бы очередь задач (Task Queue).
        std::thread([h]() {
            // Эмулируем задержку или работу
            std::cout << "Thread " << std::this_thread::get_id()
                      << ": Resuming coroutine..." << std::endl;

            // Возобновляем корутину УЖЕ в этом новом потоке
            h.resume();

        }).detach(); // Внимание: detach опасен, но здесь для примера
    }

    // 3. Ничего не возвращаем
    void await_resume() const noexcept {}
};

// Тестовая корутина
struct Task {
    struct promise_type {
```

```
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };
};

Task async_op() {
    std::cout << "Step 1 on thread " << std::this_thread::get_id() << std::endl;

    // МАГИЯ ЗДЕСЬ
    co_await ResumeOnNewThread{};

    // Этот код выполнится уже в другом потоке
    std::cout << "Step 2 on thread " << std::this_thread::get_id() << std::endl;
}

int main() {
    async_op();
    // Ждем, чтобы detach-поток успел отработать
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 0;
}
```

Under the hood: Стандартные Awaitable

Теперь мы можем понять, как реализованы стандартные заглушки.

> **std::suspend_always**
>
> Awaitable, который всегда останавливает корутину.
>
> - `await_ready` возвращает `false`.
> - `await_suspend` ничего не делает (возвращает void).
> - `await_resume` ничего не делает.
>
> Используется в `initial_suspend` (для ленивого старта) и `yield_value`.

> **std::suspend_never**
>
> Awaitable, который никогда не останавливает корутину.
>
> - `await_ready` возвращает `true`.
> - Остальные методы никогда не вызываются (и могут быть пустыми).
>
> Используется, когда синтаксис требует `co_await` (например, в `final_suspend` для автоматического удаления), но реальная остановка не нужна.

```
// Реализация из libstdc++ (упрощенно)
struct suspend_always {
```

```
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

Symmetric Transfer (Симметричная передача)

В сложных асинхронных системах (например, state-машинах парсеров или планировщиках) одна корутина часто будит другую.

Если делать это наивно через `h.resume()` внутри `await_suspend`, возникает рекурсия вызовов на стеке: `Coroutine A → await_suspend → Coroutine B → await_suspend → Coroutine C...` Это быстро приведет к **Stack Overflow**, так как стек каждого `await_suspend` остается в памяти.

Решение — **Tail Call Optimization** для корутин. Если `await_suspend` возвращает `coroutine_handle`, компилятор генерирует код вида:

```
// Псевдокод Symmetric Transfer
auto next_handle = current_awaiter.await_suspend(me);
if (next_handle) {
    // JUMP вместо CALL!
    // Текущий стек очищается, происходит переход на next_handle
    jump_to(next_handle);
}
```

Это позволяет создавать бесконечные цепочки переключений между корутинами без потребления стековой памяти.

Резюме раздела

- `co_await` разбивает исполнение функции на три этапа: проверка готовности (`await_ready`), регистрация ожидания (`await_suspend`) и получение результата (`await_resume`).
- `await_suspend` получает полный контроль над остановленной корутиной через `handle`.
- Этот механизм позволяет интегрировать корутины C++ с любой асинхронной подсистемой: от `select/poll` в Linux до GUI-циклов событий в Windows.
- Стандартные типы `suspend_always/never` — это простейшие реализации этого концепта.