HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 12 – корутины

Глава 1

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 → Execute → Terminate

Coroutine:
- Start → Execute → **Suspend** (Save State) → Return Control
- ... (Time passes) ...
- **Resume** (Restore State) → Execute → 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

```
ef 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 2

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++):

```cpp
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*.

```cpp
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 3

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 4

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 5

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 6

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.