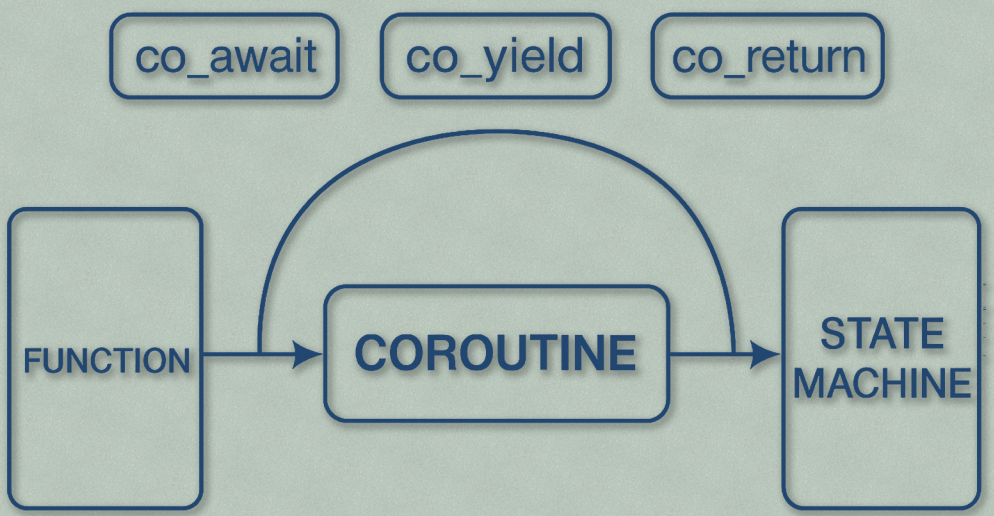
https://simplifycpp.org

Coroutines

The Comprehensive Concise Guide



Prepared By Ayman Alheraki

Coroutines

The Comprehensive Concise Guide

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Preface

Coroutines represent one of the most significant evolutionary steps in the modern history of the C++ programming language. For decades, C++ developers have relied on threads, callbacks, state machines, and complex control-flow structures to express asynchronous computations or incremental data pipelines. While these mechanisms were powerful, they often resulted in scattered logic, hard-to-maintain code, and performance compromises.

With the introduction of coroutines in C++20, the language finally gained a standardized, efficient, and expressive model for writing suspendable and resumable computations. Unlike other programming languages that provide coroutines as high-level constructs, C++ introduces a low-level, high-performance coroutine foundation that allows developers to design their own execution abstractions: asynchronous tasks, generators, cooperative schedulers, pipelines, state machines, and more. This open-ended design grants unmatched flexibility while preserving the efficiency guarantees that distinguish C++ in systems programming.

This booklet was written to provide a **concise but comprehensive guide** to modern coroutine development. It focuses exclusively on the standards, techniques, and patterns emerging after C++20—reflecting the practices of post-2020 modern C++ development. Each chapter blends theory with practical, fully functional examples that compile on major compilers such as GCC, Clang, and MSVC. Exercises and solutions at the end of every chapter reinforce the foundations and guide the reader through deeper understanding.

The content is crafted for:

- Experienced C++ developers seeking clarity on coroutine internals
- Engineers building concurrent or asynchronous systems
- Professionals wanting correct, modern, standards-based guidance
- Students and researchers looking for an authoritative introduction

Coroutines open a new paradigm for designing C++ software. They encourage writing code that mirrors natural control flow while avoiding the overhead and complexity of threads or callback chains. They unify asynchronous and incremental processing in a way that is both elegant and performant.
My goal in this booklet is to equip you with the understanding and confidence to use coroutines as a fundamental tool in your modern C++ development practice—to write clearer code, build more scalable systems, and take full advantage of what C++20 and later standards offer.

Ayman Alheraki

Chapter 1

Understanding the C++ Coroutine Model

The C++ coroutine machinery is built around a set of well-defined components. Unlike threads, coroutines do not run concurrently. Instead, they represent an **execution state that can be suspended and resumed**.
This chapter explains:

- What a coroutine is in C++20
- How compiler transformation works
- Coroutine frame: lifetime and memory model
- Awaiter and Awaitable concepts
- Promise type and handle mechanics

1.1 What Is a Coroutine?

A coroutine is simply a function that may suspend execution and later resume from the same point. In C++20, a function becomes a coroutine if it uses:

- `co_await`
- `co_yield`
- `co_return`

Example: The simplest coroutine function:

```
#include <coroutine>
#include <iostream>

struct SimpleTask {
    struct promise_type {
        SimpleTask get_return_object() { return {}; }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };
};

SimpleTask myCoroutine() {
    std::cout << "Inside coroutine\n";
    co_return;
}

int main() {
    myCoroutine();
}
```

Even though this coroutine never suspends, the compiler still generates a coroutine frame and transforms the function.

1.2 Compiler Transformation Overview

The compiler turns:

- Local variables → stored in a heap-allocated or optimizable frame
- Function calls → state machine transitions
- `co_await` expressions → awaiter protocol calls

A conceptual expansion of a `co_await` expression:

```
co_await expr;
```

Becomes something like:

```
auto awaiter = expr.operator co_await();
// or expr if it already matches the awaiter interface
if (!awaiter.await_ready()) {
    suspend_coroutine();
    awaiter.await_suspend(handle);
}
awaiter.await_resume();
```

This knowledge is important for debugging and performance tuning.

1.3 Awaitable and Awaiter

An awaitable must support one of the following:

- a member `operator co_await()`
- a free `operator co_await(expr)`

- providing the three methods:
  - `await_ready()`
  - `await_suspend()`
  - `await_resume()`

Example awaiter:

```
struct MyAwaiter {
    bool await_ready() noexcept { return false; }
    void await_suspend(std::coroutine_handle<>) noexcept {
        // logic
    }
    int await_resume() noexcept {
        return 42;
    }
};
```

1.4 Promise Type

Every coroutine has exactly one **promise type**. Lifetime:

- Created when coroutine starts
- Owns coroutine frame
- Destroyed on final suspend or external destruction

Skeleton:

```
struct promise_type {
    ReturnObject get_return_object();
```

```
    std::suspend_always initial_suspend();
    std::suspend_always final_suspend() noexcept;
    void return_void(); // or return_value(...)
    void unhandled_exception();
};
```

1.5 Exercises (Chapter 1)

Exercise 1 Explain in your own words why a function using `co_return` is still a coroutine even if it never suspends.

Exercise 2 Write a custom awaiter that always suspends and resumes immediately.

Exercise 3 Draw a diagram of a coroutine frame and label all components.

1.6 Solutions

Solution 1 Because the compiler injects coroutine machinery, allocates a frame, and uses the promise type, even if suspension never happens, it is still treated as a coroutine.

Solution 2

```
struct AlwaysSuspend {
    bool await_ready() noexcept { return false; }
    void await_suspend(std::coroutine_handle<>) noexcept {}
    void await_resume() noexcept {}
};
```

Solution 3 A correct diagram shows:

- promise object
- saved state

- local variables
- suspended-point index
- return object

Chapter 2

Writing Custom Awaitables in Modern C++

Coroutines in C++ are extremely flexible because developers can define their own types that participate in the `co_await` protocol. This chapter explains how to design a fully custom awaitable and awaiter using modern patterns, focusing on post-2020 C++ conventions.
We cover:

- The Awaiter interface: `await_ready`, `await_suspend`, `await_resume`
- Designing Awaitables using operator overloading
- Using free-standing `operator co_await`
- Returning values from awaitables
- Practical patterns for asynchronous programming

2.1 The Awaiter Interface

An **awaiter** is the object that the coroutine interacts with when it encounters a `co_await` expression.
To be a valid awaiter, a type must provide:

```
bool await_ready();
auto await_suspend(std::coroutine_handle<>);
auto await_resume();
```

Example minimal awaiter:

```
struct SimpleAwaiter {
    bool await_ready() const noexcept { return false; }

    void await_suspend(std::coroutine_handle<>) const noexcept {
        // logic executed when the coroutine suspends
    }

    int await_resume() const noexcept {
        return 123; // value returned to co_await expression
    }
};
```

2.2 Awaitables

An **awaitable** is simply a type that produces an awaiter.
Valid ways:

1. Providing a member `operator co_await()`
2. Providing a free `operator co_await(T)`

3. Directly being considered an awaiter (if it implements the awaiter interface)

Example (member `operator co_await`):

```
struct MyAwaitable {
    struct Awaiter {
        bool await_ready() const noexcept { return false; }
        void await_suspend(std::coroutine_handle<>) const noexcept {}
        int await_resume() const noexcept { return 77; }
    };

    Awaiter operator co_await() const noexcept { return {}; }
};
```

Used in a coroutine:

```
MyAwaitable getValue() { return {}; }

SimpleTask foo() {
    int x = co_await getValue();
    // x == 77
}
```

2.3 Using Free-Standing `operator co_await`

C++20 allows developers to customize await behavior without modifying the type:

```
struct ExternalType {
    int value;
};

// Free operator co_await
auto operator co_await(const ExternalType& obj) {
```

```
    struct Awaiter {
        const ExternalType& ref;

        bool await_ready() const noexcept { return false; }
        void await_suspend(std::coroutine_handle<>) const noexcept {}
        int await_resume() const noexcept { return ref.value * 2; }
    };
    return Awaiter{obj};
}
```

This is useful for adapting third-party types.

2.4 Returning Values from Awaitables

Awaitables can return:

- **void**
- **simple values (int, float, etc.)**
- **reference types**
- **objects by value**
- **move-only types**

Example returning a move-only result:

```
struct MoveResult {
    std::unique_ptr<int> ptr;
};

struct MoveAwaiter {
```

```
    bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<>) const noexcept {}

    MoveResult await_resume() {
        return MoveResult{std::make_unique<int>(999)};
    }
};
```

2.5 Practical Example: Delayed Resume

A realistic delay simulation using `std::jthread` and an awaiter:

```
#include <coroutine>
#include <chrono>
#include <thread>

struct Delay {
    std::chrono::milliseconds duration;

    struct Awaiter {
        std::chrono::milliseconds duration;

        bool await_ready() noexcept { return false; }

        void await_suspend(std::coroutine_handle<> h) {
            std::jthread([h, d = duration] {
                std::this_thread::sleep_for(d);
                h.resume();
            });
        }

        void await_resume() noexcept {}
    };
```

```
    Awaiter operator co_await() const noexcept {
        return Awaiter{duration};
    }
};
```

Usage:

```
Task sleepDemo() {
    std::cout << "Sleeping...\n";
    co_await Delay{500ms};
    std::cout << "Awake!\n";
}
```

2.6 Chapter Summary

In this chapter, we learned:

- The structure and role of awaiters
- How awaitables expose suspendable behavior
- How to customize await using operator overloading
- How to integrate external types using free `operator co_await`
- Practical patterns for asynchronous operations

2.7 Exercises (Chapter 2)

Exercise 1 Write a custom awaitable called `DoubleValue` that takes an integer and, when awaited, returns its value multiplied by 2.

Exercise 2 Modify the `Delay` awaitable so that it accepts a callback function executed after the delay finishes.

Exercise 3 Explain when you should use a free `operator co_await` instead of a member `operator co_await`.

2.8 Solutions

Solution 1

```
struct DoubleValue {
    int value;

    struct Awaiter {
        int v;
        bool await_ready() const noexcept { return false; }
        void await_suspend(std::coroutine_handle<>) const noexcept {}
        int await_resume() const noexcept { return v * 2; }
    };

    Awaiter operator co_await() const noexcept {
        return Awaiter{value};
    }
};
```

Solution 2

```
struct DelayWithCallback {
    std::chrono::milliseconds dur;
    std::function<void()> callback;

    struct Awaiter {
        std::chrono::milliseconds dur;
```

```
        std::function<void()> callback;

        bool await_ready() noexcept { return false; }

        void await_suspend(std::coroutine_handle<> h) {
            std::jthread([h, d = dur, cb = callback] {
                std::this_thread::sleep_for(d);
                if (cb) cb();
                h.resume();
            });
        }

        void await_resume() noexcept {}
    };

    Awaiter operator co_await() const noexcept {
        return Awaiter{dur, callback};
    }
};
```

Solution 3 Use a free `operator co_await` when:

- you cannot modify the original type (third-party or const type),
- you want different await behaviors for the same type,
- you want separation between coroutine logic and data definition,
- you need SFINAE-based customization of await behavior.

Chapter 3

Coroutine Return Types: Task, Generator, Lazy, and Beyond

The return type of a coroutine defines how the coroutine behaves, how the execution is resumed, and how results are retrieved. Modern C++ allows developers to design their own coroutine return abstractions, enabling patterns such as:

- **Task**-based asynchronous operations
- **Generator**-style lazy sequence generation
- **Lazy evaluation**
- **Event-driven coroutines**
- **Detached coroutines**
- **Scheduler-bound coroutines**

Understanding these patterns is crucial for modern, professional C++ development—especially when building frameworks, libraries, or high-performance systems.

3.1 Anatomy of a Coroutine Return Type

A coroutine return type must:

1. Provide a nested **promise_type**
2. Provide a method **get_return_object()** inside the promise
3. Optionally provide:
    - value-returning APIs
    - continuation handling
    - destruction behavior
    - scheduler hooks

A minimal coroutine return type:

```
struct SimpleTask {
    struct promise_type {
        SimpleTask get_return_object() { return {}; }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
};
```

3.2 Task<T>: A Future-Like Coroutine Return Type

A **Task** conceptually represents an operation whose result will be available in the future. It behaves similarly to a lightweight, coroutine-based alternative to `std::future`, but with

far greater flexibility and zero-cost suspension semantics. This makes **Task**-style abstractions one of the most widely used patterns in modern asynchronous C++ frameworks, libraries, and runtimes.

Below is a modern, simplified implementation of **Task**<**T**> using C++20 coroutines:

```
template<typename T>
struct Task {
    struct promise_type {
        T value;
        std::exception_ptr error;

        Task get_return_object() {
            return Task{
                std::coroutine_handle<promise_type>::from_promise(*this)
            };
        }

        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }

        void return_value(T v) noexcept { value = v; }

        void unhandled_exception() {
            error = std::current_exception();
        }
    };

    using handle_t = std::coroutine_handle<promise_type>;
    handle_t coro;

    Task(handle_t h) : coro(h) {}
    ~Task() { if (coro) coro.destroy(); }
```

```
    T get() {
        coro.resume();
        if (coro.promise().error)
            std::rethrow_exception(coro.promise().error);
        return coro.promise().value;
    }
};
```

Usage:

```
Task<int> compute() {
    co_return 42;
}

int main() {
    auto t = compute();
    std::cout << t.get(); // prints 42
}
```

3.3 Generator<T>: Lazy Iteration via Coroutines

The **generator** pattern yields values one-by-one, lazily.

Minimal example:

```
template<typename T>
struct Generator {
    struct promise_type {
        T current;

        Generator get_return_object() {
            return Generator{
                std::coroutine_handle<promise_type>::from_promise(*this)};
        }
```

```
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }

        std::suspend_always yield_value(T v) noexcept {
            current = v;
            return {};
        }

        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    using handle_t = std::coroutine_handle<promise_type>;
    handle_t coro;

    Generator(handle_t h) : coro(h) {}
    ~Generator() { if (coro) coro.destroy(); }

    bool next() {
        if (!coro.done()) coro.resume();
        return !coro.done();
    }

    T value() { return coro.promise().current; }
};
```

Usage:

```
Generator<int> counter(int n) {
    for (int i = 1; i <= n; ++i)
        co_yield i;
}
```

```
int main() {
    auto g = counter(5);
    while (g.next())
        std::cout << g.value() << " ";
}
```

3.4 Lazy<T>: Computation Deferred Until First Use

Lazy evaluation saves work by delaying computation until needed.

Example:

```
template<typename T>
struct Lazy {
    struct promise_type {
        T value;

        Lazy get_return_object() {
            return Lazy{
                std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_value(T v) noexcept { value = v; }
        void unhandled_exception() { std::terminate(); }
    };

    using handle_t = std::coroutine_handle<promise_type>;
    handle_t coro;

    Lazy(handle_t h) : coro(h) {}
    ~Lazy() { if (coro) coro.destroy(); }
```

```
    T get() {
        if (!coro.done()) coro.resume();
        return coro.promise().value;
    }
};
```

Usage:

```
Lazy<int> expensive() {
    std::cout << "Computing...\n";
    co_return 5000;
}

int main() {
    auto v = expensive();
    std::cout << v.get(); // prints "Computing..." then 5000
}
```

3.5 Detached Coroutines

Some coroutines do not return results—they simply run and detach.

```
struct FireAndForget {
    struct promise_type {
        FireAndForget get_return_object() { return {}; }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
};
```

Used for event loops, background monitoring, etc.

3.6 Scheduler-Bound Tasks (Advanced)

In real systems, we often want:

- resume on a specific thread
- queue to a thread-pool
- execute on an event-loop

Modern schedulers inject logic inside `await_suspend`.
Example: resume on thread-pool:

```
struct ResumeOnPool {
    ThreadPool& pool;

    bool await_ready() noexcept { return false; }

    void await_suspend(std::coroutine_handle<> h) {
        pool.enqueue([h] { h.resume(); });
    }

    void await_resume() noexcept {}
};
```

3.7 Chapter Summary

We covered five powerful coroutine return abstractions:

- **Task**<**T**> — async return value
- **Generator**<**T**> — lazy sequences

- **Lazy<T>** — deferred computation
- **Fire-and-forget** — detached execution
- **Scheduler-aware designs** — advanced concurrency

3.8 Exercises (Chapter 3)

Exercise 1

Extend the `Task<T>` implementation so that it supports multiple `co_await` points.

Exercise 2

Write a generator that yields Fibonacci numbers lazily.

Exercise 3

Design a `Lazy<std::string>` coroutine that constructs a string only on first access.

Exercise 4

Explain the difference between **Task<T>** and **Lazy<T>**.

3.9 Solutions

Solution 1 — Multi-step Task

Add a suspend point in the coroutine:

```
co_await std::suspend_always{};
```

This is enough for multi-step execution.

Solution 2 — Fibonacci Generator

```
Generator<long long> fibonacci(int n) {
    long long a = 0, b = 1;
    for (int i = 0; i < n; ++i) {
```

```
        co_yield a;
        auto next = a + b;
        a = b;
        b = next;
    }
}
```

Solution 3 — Lazy String

```
Lazy<std::string> makeString() {
    std::string s = "Coroutine-generated string!";
    co_return s;
}
```

Solution 4 — Conceptual Difference

- **Task**<**T**> represents an asynchronous operation that may have multiple resumptions.
- **Lazy**<**T**> represents a deferred computation evaluated exactly once on demand.

Chapter 4

Deep Internals of `co_await`, `co_yield`, `co_return`

This chapter provides a deep, compiler-level exploration of Modern C++ coroutine mechanics. Unlike traditional stack-based functions, a coroutine compiles into a resumable state machine with its own permanent activation record (coroutine frame). We analyze how the compiler lowers coroutine syntax, how the `promise_type` communicates values between caller and callee, how suspension points work, and what happens at initial and final suspension.

This is the layer where C++ coroutines differ fundamentally from languages like Python, C#, or Kotlin: in C++, the coroutine transformation is *purely a compiler rewrite* with *zero mandatory runtime*, making it one of the most powerful and customizable coroutine systems in any mainstream language.

4.1 The Coroutine State Machine

When the compiler encounters any coroutine keyword (`co_await`, `co_yield`, or `co_return`), it transforms the function into:

1. A **coroutine frame**, allocated on the heap or via custom allocators.
2. A **promise object** stored inside that frame.
3. A **state machine** that jumps between suspension points.
4. A hidden **dispatcher function** that resumes execution based on state.

Coroutine Frame Layout (Conceptual)

```
+-----------------------------------------------------------+
|                   Coroutine Frame                         |
+-----------------------------------------------------------+
| promise_type promise;                                     |
|                                                           |
| // Local variables that survive across suspension         |
| T  local1;                                                |
| U  local2;                                                |
|                                                           |
| // State machine bookkeeping                              |
| int state;                                                |
| std::exception_ptr exception;                             |
|                                                           |
| // Internal compiler-managed fields                       |
| ...                                                       |
```

```
+-----------------------------------------------------------+
```

The frame persists for the entire lifetime of the coroutine and is destroyed when:

- the coroutine reaches `final_suspend` and is destroyed by the caller,
- or it is manually destroyed via its `std::coroutine_handle`.

4.1.1 Lowering of `co_await`

When the compiler sees:

```
co_await expr;
```

it expands it roughly as:

```
auto&& awaitable = expr;
auto&& awaiter = get_awaiter(awaitable);

if (!awaiter.await_ready()) {
    state = NEXT_STATE;
    if (auto continuation = awaiter.await_suspend(coro_handle))
        return; // COROUTINE SUSPENDED
}

auto result = awaiter.await_resume();
```

This is called the **awaiter protocol**.

4.2 Detailed Awaiter Contract

An expression `expr` is awaitable if it can be converted into an **awaiter**, which satisfies three mandatory operations:

```
bool await_ready();
auto await_suspend(std::coroutine_handle<>);
auto await_resume();
```

Semantics

- `await_ready()` Determines whether the coroutine suspends. If it returns `true`, suspension is skipped.
- `await_suspend(handle)` Transfers control to the awaiter. It may:
  - schedule the coroutine on another thread,
  - resume another coroutine,
  - enqueue the handle in a scheduler,
  - return `void`, `bool`, or a coroutine handle.
- `await_resume()` Produces the final value of the await expression and may throw.

Awaiter Discovery Rules

The compiler uses ADL + transformation rules:

1. If `expr` is already an awaiter (provides the 3 functions), use it.
2. If `promise.await_transform(expr)` exists, use that.
3. If `expr.operator co_await()` exists, call it.
4. If a free `operator co_await(expr)` exists, call it.
5. Otherwise, `expr` is not awaitable.

4.3 Full Expansion of `co_await`

The compiler expansion is equivalent to:

```
{
    auto&& a = expr;
    auto&& awaiter = get_awaiter(a);

    if (!awaiter.await_ready()) {
        state = SUSPENDED_POINT;

        if (awaiter.await_suspend(coro_handle))
            return; // COROUTINE SUSPENDED
    }

    auto result = awaiter.await_resume();
}
```

Key notes:

- `await_ready()` prevents unnecessary suspension.
- `await_suspend()` determines exact control transfer.
- `await_resume()` is executed upon resumption and may throw.

4.4 Deep Expansion of `co_yield`

`co_yield expr;` is rewritten as:

```
co_await promise.yield_value(expr);
```

Meaning:

1. The `promise.yield_value()` method is invoked.
2. It returns an awaitable.
3. `co_await` is applied to that awaitable using the same rules as above.

Performance Note

Because `co_yield` is just `co_await`, generators have:

- no special syntax,
- no additional runtime overhead,
- a zero-cost abstraction for lazy sequences.

4.5 Deep Expansion of `co_return`

There are two variants:

Void-returning coroutines

```
co_return;
```

expands to:

```
promise.return_void();
goto FINAL_SUSPEND;
```

Value-returning coroutines

```
co_return value;
```

expands to:

```
promise.return_value(value);
goto FINAL_SUSPEND;
```

When is the coroutine actually finished?

The coroutine does *not* truly finish at `co_return`. It finishes only after the **final suspend point** completes.
This allows:

- deterministic destruction,
- safe continuation chaining,
- scheduler-based finalization.

4.6 Exception Propagation and Error Semantics

If a coroutine throws and the exception is not handled internally, the runtime invokes:

```
void promise_type::unhandled_exception();
```

Inside this function, common strategies include:

- storing `std::current_exception()`,
- rethrowing inside `await_resume()`,
- terminating the program.

Why exceptions do not automatically unwind the coroutine frame

C++ coroutine frames are heap-allocated or custom-allocated and require manual cleanup. This is why exceptions do not implicitly destroy the coroutine. Instead, destruction occurs when the handle's owner destroys it.
This avoids accidental leaks or premature destruction.

4.7 Initial and Final Suspension

Every coroutine inserts two mandatory suspension points:

Initial Suspension

Controls whether the coroutine begins execution immediately:

- `std::suspend_always` → coroutine starts in suspended state.
- `std::suspend_never` → coroutine starts running immediately.

Used for:

- lazy generators,
- scheduler-controlled coroutine startup,
- custom pipeline initialization.

Final Suspension

Final suspend controls what happens after `co_return`. It returns an awaiter, typically:

- `std::suspend_always` → caller must destroy the coroutine.

- `std::suspend_never` → coroutine frame is destroyed automatically.

Final suspend is critical because it:

- triggers continuation chains,
- notifies schedulers,
- ensures deterministic cleanup.

4.8 Low-Level Execution Flow of a Coroutine

Consider this simple coroutine:

```
Task<int> foo() {
    int a = co_await computeSomething();
    co_return a + 1;
}
```

The runtime execution flow becomes:

1. Allocate coroutine frame.
2. Construct `promise`.
3. Capture parameters.
4. Call `initial_suspend()`.
5. Evaluate `computeSomething()` into an awaiter.
6. Check `await_ready()`.
7. Possibly suspend and resume later.

8. On resume, call `await_resume()` → produce `a`.
9. Execute `co_return a + 1`.
10. Call `return_value()`.
11. Enter `final_suspend()`.
12. Destroy coroutine when caller decides.

4.9 Lifetime of a Coroutine in Memory

```
Creation:
    allocate frame
    construct promise
    run initial_suspend

Execution:
    resume/suspend many times

Completion:
    final_suspend
    caller destroys coroutine handle -> free frame
```

Coroutines provide deterministic lifetime by giving the user control over the coroutine handle.

4.10 Compiler Optimizations and Performance Considerations

Coroutines are designed to be zero-cost abstractions, but performance depends on:

1. Frame Allocation

Avoid unnecessary heap allocation. Options:

- **Allocator customization**: override `operator new/delete`.
- **Small Buffer Optimization**: place frames on the stack.
- **Fixed-size arenas**: ideal for real-time systems.

2. Promise Size

Keep the promise small:

- avoid storing large objects,
- avoid heavy containers,
- store references instead of copies where possible.

3. Suspension Frequency

- Prefer `await_ready() == true` whenever possible.
- Reduce the number of suspension points inside tight loops.

4. Inlining and LTO

- Always compile coroutine-heavy code with `-O2` or `-O3`.
- LTO enables whole-program optimization.

5. Avoid Exception-Based Control Flow

Exceptions interrupt coroutine destruction flow and should be rare.

4.11 Chapter Summary

This chapter explained the most internal aspects of C++ coroutine execution:

- how the compiler rewrites coroutine keywords into a state machine,
- how awaitables and awaiters form the suspension protocol,
- how `co_yield` is syntactic sugar over `co_await`,
- how `co_return` interacts with `return_value` and `return_void`,
- how exceptions propagate through `unhandled_exception()`,
- how initial and final suspension define the lifecycle,
- how allocation, promise layout, and suspension frequency affect performance.

Understanding these internals is a prerequisite for building advanced coroutine frameworks, custom schedulers, async runtimes, and high-performance pipelines.

Chapter 5

Generators, Async Pipelines, and Thread-Pool Integration

In earlier chapters we explored coroutine internals, awaiters, promise types, and return-type mechanics. We now move from foundational theory to practical, high-level coroutine patterns used in modern C++ systems: lazy iterators, pull- and push-based pipelines, real asynchronous execution, and thread-pool integration.

This chapter focuses on how coroutine-based architectures form the backbone of asynchronous data processing, real-time event systems, game loops, and modern networking frameworks. The goal is to show how coroutines can be composed into larger systems that are both elegant and high-performance.

5.1 The Modern Generator Pattern

A **generator** is a coroutine that produces a sequence of values lazily. Unlike traditional iterators or containers, a generator does not compute the entire sequence in advance. Instead, each value is produced exactly when needed. This is similar in spirit to Python generators

or C#'s `yield return`, but in C++ the coroutine machinery is lower-level, giving the developer more control and performance.

5.1.1 Design Goals of a Generator

A C++20 generator typically satisfies these:

- Produce values one at a time using `co_yield`.
- Lazily evaluate sequences, saving memory and CPU.
- Resume where it left off on each pull.
- Allow for efficient composition into pipelines.
- Enable infinite streams without consuming memory.

5.1.2 Minimal Generator Implementation

Below is a fully functional generator template using C++20 coroutines:

```
template<typename T>
struct Generator {
    struct promise_type {
        T current;

        Generator get_return_object() {
            return Generator{
                std::coroutine_handle<promise_type>::from_promise(*this)};
        }

        std::suspend_always initial_suspend() noexcept { return {}; }
```

```
        std::suspend_always yield_value(T value) noexcept {
            current = std::move(value);
            return {};
        }

        std::suspend_always final_suspend() noexcept { return {}; }

        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    using handle_t = std::coroutine_handle<promise_type>;
    handle_t h;

    Generator(handle_t h) : h(h) {}
    Generator(const Generator&) = delete;

    Generator(Generator&& other) noexcept : h(other.h) { other.h = nullptr;
    ↪  }

    ~Generator() {
        if (h) h.destroy();
    }

    bool next() {
        if (!h || h.done()) return false;
        h.resume();
        return !h.done();
    }

    T value() const {
        return h.promise().current;
    }
```

```
};
```

5.1.3 Using the Generator

```
Generator<int> countTo(int n) {
    for (int i = 1; i <= n; ++i)
        co_yield i;
}

int main() {
    auto gen = countTo(5);
    while (gen.next())
        std::cout << gen.value() << "\n";
}
```

5.1.4 Advantages of Generators

- Avoid allocating large arrays.
- Support infinite sequences (e.g., Fibonacci, primes, random numbers).
- Integrate easily with pipelines.
- Very low overhead—optimized by compilers.

5.2 Multi-Stage Pipelines

A powerful use of generators is building pipelines: multi-step transformations where each stage processes values from the previous stage.

5.2.1 Pipeline Overview

A typical coroutine-based pipeline looks like:

```
auto stage1 = source();
auto stage2 = filter(std::move(stage1));
auto stage3 = transform(std::move(stage2));
```

Each stage pulls data from the previous one lazily.

5.2.2 Stage 1: Source

```
Generator<int> numbers(int n) {
    for (int i = 0; i < n; ++i)
        co_yield i;
}
```

5.2.3 Stage 2: Filter

```
Generator<int> filterEven(Generator<int> g) {
    while (g.next()) {
        if (g.value() % 2 == 0)
            co_yield g.value();
    }
}
```

5.2.4 Stage 3: Transform

```
Generator<int> multiplyBy10(Generator<int> g) {
    while (g.next())
        co_yield g.value() * 10;
}
```

5.2.5 Combining the Pipeline

```
auto g = multiplyBy10(filterEven(numbers(20)));

while (g.next())
    std::cout << g.value() << "\n";
```

This style produces readable, composable logic with high efficiency.

5.2.6 Pipeline Performance Notes

- Every stage uses minimal memory.
- No intermediate containers.
- Values are generated only when needed.
- Ideal for large-scale data processing or real-time streaming.

5.3 Push-Based Pipelines

Generators are **pull-based**. Sometimes, you need **push-based** processing—for example:

- GUI event propagation
- Real-time reactive systems
- Network packet sinks
- Asynchronous event delivery

Push pipelines use awaiters that transfer execution to the next coroutine:

```
struct PushAwaiter {
    std::coroutine_handle<> next;

    bool await_ready() noexcept { return false; }

    std::coroutine_handle<> await_suspend(std::coroutine_handle<>) noexcept
    ↪ {
        return next; // transfer control
    }

    void await_resume() noexcept {}
};
```

This pattern leads to high-performance reactive engines.

5.4 Combining Generators with Async Tasks

Generators produce values. Async tasks execute latency-bound work. Coroutines allow mixing them naturally.
Example: delayed generation:

```
Task<int> delayedValue(int x, std::chrono::milliseconds d) {
    co_await Delay{d};
    co_return x;
}

Generator<Task<int>> delayedSequence() {
    for (int i = 1; i <= 5; ++i)
        co_yield delayedValue(i, 300ms);
}
```

Consumers can then:

```
auto g = delayedSequence();

while (g.next()) {
    Task<int> t = g.value();
    std::cout << t.get() << "\n";
}
```

5.5 Thread-Pool Integration

Coroutines integrate naturally with thread pools, enabling asynchronous parallel processing.

5.5.1 Thread Pool Implementation

```
class ThreadPool {
public:
    ThreadPool(size_t n = std::thread::hardware_concurrency()) {
        for (size_t i = 0; i < n; ++i) {
            workers.emplace_back([this] {
                for (;;) {
                    std::function<void()> task;
                    {
                        std::unique_lock lk(mtx);
                        cv.wait(lk, [&]{ return stopped || !tasks.empty();
                        ↪  });
                        if (stopped && tasks.empty()) return;
                        task = std::move(tasks.front());
                        tasks.pop();
                    }
                    task();
                }
            });
        }
```

```
    }

    ~ThreadPool() {
        {
            std::lock_guard lk(mtx);
            stopped = true;
        }
        cv.notify_all();
        for (auto& t : workers) t.join();
    }

    void enqueue(std::function<void()> f) {
        {
            std::lock_guard lk(mtx);
            tasks.push(std::move(f));
        }
        cv.notify_one();
    }

private:
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool stopped = false;
};
```

5.5.2 Awaitable: Schedule Work on Pool

```
struct ScheduleOnPool {
    ThreadPool& pool;

    bool await_ready() noexcept { return false; }
```

```
    void await_suspend(std::coroutine_handle<> h) {
        pool.enqueue([h] { h.resume(); });
    }

    void await_resume() noexcept {}
};
```

5.5.3 Coroutine Using Scheduler

```
Task<int> computeAsync(ThreadPool& pool, int x) {
    co_await ScheduleOnPool{pool};
    co_return x * x;
}
```

5.5.4 Pipeline Example

```
ThreadPool pool(4);

auto t1 = computeAsync(pool, 10);
auto t2 = computeAsync(pool, 20);

std::cout << t1.get() << "\n"; // 100
std::cout << t2.get() << "\n"; // 400
```

5.6 Asynchronous Multi-Stage Pipeline (Advanced)

Now we combine:

- Lazy generators
- Asynchronous thread-pool processing

- Coroutine chaining

5.6.1 Stage 1: Lazy Source

```
Generator<int> source(int count) {
    for (int i = 1; i <= count; ++i)
        co_yield i;
}
```

5.6.2 Stage 2: Async Mapper

```
Task<int> asyncMap(ThreadPool& pool, int x) {
    co_await ScheduleOnPool{pool};
    co_return x * 100;
}
```

5.6.3 Stage 3: Coroutine Consumer

```
Task<void> runPipeline(ThreadPool& pool) {
    auto g = source(10);

    while (g.next()) {
        int value = g.value();
        int mapped = co_await asyncMap(pool, value);
        std::cout << "Mapped: " << mapped << "\n";
    }
}
```

5.7 Practical Use Cases of Coroutine Pipelines

- Streaming large data sets (GB-scale logs, sensor data)

- Video processing frames lazily
- Network packet filtering and transformation
- Game engine update loops
- Event-driven simulations
- Financial tick processing
- Real-time telemetry pipelines

5.8 When to Use Coroutines for Pipelines

Use coroutine-based pipelines when:

- You want zero-overhead control flow.
- You need lazy evaluation.
- You need readable async code.
- You want to avoid temporary allocations.
- You want to mix sync and async stages seamlessly.

5.9 Chapter Summary

This chapter showed how coroutines enable:

- Lazy, memory-efficient generators
- Pull-based and push-based pipelines

- Asynchronous thread-pool processing
- Seamless combination of sync and async flows
- Modular and composable pipeline architectures
- Real-world applications across many domains

Coroutines provide the foundation for building high-performance, cleanly-structured data flows in Modern C++20/23.

Final Conclusion

Coroutines represent one of the most transformative additions to Modern C++. Introduced in C++20 and expanded steadily through C++23 and beyond, the coroutine model gives developers access to a unique level of expressive power: a low-level, zero-cost abstraction that allows suspension, resumption, lazy generation, asynchronous execution, and stateful computations—all without compromising performance or control.

Throughout this booklet, we have carefully explored the complete coroutine ecosystem from its foundations to its most advanced patterns. Beginning with the core machinery—the promise type, coroutine frame, awaiter protocol, and compiler transformation—we built a robust understanding of how C++20 coroutines function at the lowest level. This was followed by a practical and progressive journey through:

- writing custom awaitables and awaiters,
- implementing generators and lazy pipelines,
- designing Task-like return types for asynchronous work,
- integrating coroutines with thread pools and schedulers,
- chaining continuations with final suspend semantics,
- handling exceptions and error propagation across await chains,

- and finally, customizing coroutine behavior using `await_transform` and sender/receiver bridges.

Mastering coroutines is not simply about learning a new syntax—it is about embracing a new architecture for thinking about programs. The concepts explored in this booklet are the foundation for the next generation of idiomatic C++ software design.

Ayman Alheraki

References

- ISO/IEC 14882:2020 — *Programming Languages — C++*. International Organization for Standardization (ISO), 2020.
- ISO/IEC 14882:2023 — *Programming Languages — C++*. International Organization for Standardization (ISO), 2023.
- Gor Nishanov — *P0912R5: A Unified Coroutine Model for C++*. ISO C++ Committee Paper, 2020.
- Gor Nishanov — *P1056R1: C++ Coroutine Theory and Practice*. ISO C++ Committee Paper, 2021.
- Lewis Baker, Michael Garland — *P2168R3: std::generator — Lazy Generator for C++*. ISO C++ Committee Paper, 2022.
- Lewis Baker — *P0664R10: C++ Awaitable and Awaiter Design*. ISO WG21 Paper, 2021.
- cppreference.com — *Coroutines (C++20)*. Continuously updated, accessed 2020–2025.
- Microsoft C++ Team — *Understanding C++20 Coroutines*. Microsoft Developer Documentation, 2020–2024.

- LLVM Project — *Coroutines Support in Clang*. LLVM/Clang Documentation, 2021–2024.
- GNU Project — *GCC Coroutine Support Notes*. GCC Documentation, 2020–2024.
- Lewis Baker — *cppcoro: C++ Coroutine Library*. GitHub Repository, active 2020–2025.
- Facebook Engineering — *Folly Coroutine Extensions*. Meta Open Source, 2020–2025.
- Christopher Kohlhoff — *Boost.Asio Coroutine and Awaitable Support*. Boost Documentation, 2021–2025.
- Microsoft C++ Libraries — *Concurrency and Coroutines in Modern MSVC*. 2020–2025.