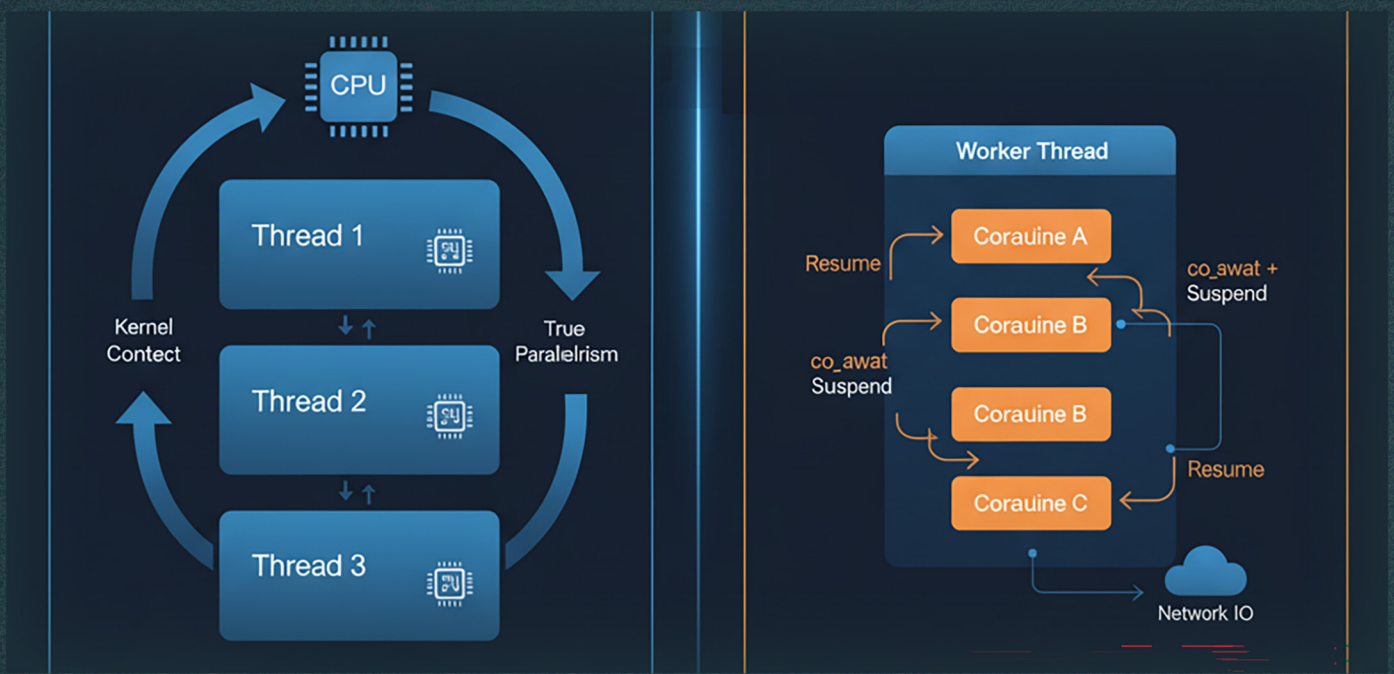
https://simplifyCpp.org

DRAFT

Mastering Coroutines in Modern C++



Prepared by Ayman Alheraki

First Edition

Mastering Coroutines in Modern C++

Prepared by Ayman Alheraki

simplifycpp.org

October 2025

Contents

Contents **2**

Author's Introduction **10**

Introduction to Coroutines **11**

Why Coroutines? 11

Difference between Coroutines, Traditional Functions, and Threads 18

What's New in C++20 and C++23 23

Practical Goals of This Guide 28

1 Deep Understanding of Coroutines **32**

1.1 Core Structure of Coroutines 32

1.1.1 `co_await` – Awaiting Asynchronous Results 32

1.1.2 `co_yield` – Generating Incremental Values 35

1.1.3 `co_return` – Returning from a Coroutine 37

1.1.4 How They Work Together 38

1.2 Promise Type and `std::coroutine_traits` 39

1.2.1 The Role of the Promise Type 39

1.2.2 `std::coroutine_traits` – Mapping Return Type to Promise Type 41

1.2.3 Key Points to Remember 42

1.2.4 Practical Example: Generator with Promise Type 43

1.3 Handle Type – `std::coroutine_handle` 45

1.3.1 What is `std::coroutine_handle`? 45

1.3.2 Basic Operations 45

1.3.3 Typed vs. Untyped Handles . . . 47
1.3.4 Lifetime Management . . . 47
1.3.5 Practical Example: Coroutine Scheduler . . . 48
1.3.6 Key Points . . . 49
1.4 Practical Example – A Simple Coroutine Printing a Sequence of Values with Suspension and Resumption . . . 50
1.4.1 The Concept . . . 50
1.4.2 Implementation . . . 50
1.4.3 Explanation . . . 52
1.4.4 Practical Insights . . . 53
1.4.5 Real-World Use Cases . . . 53

2 Coroutine Life Cycle **55**
2.1 Initial Suspend and Final Suspend . . . 55
2.1.1 `initial_suspend` – Controlling the Start of a Coroutine . . . 55
2.1.2 `final_suspend` – Controlling Suspension at the End . . . 57
2.1.3 Combining Initial and Final Suspend . . . 59
2.1.4 Practical Insights . . . 59
2.2 Resumption and Destruction . . . 61
2.2.1 Resumption – Controlling Coroutine Execution . . . 61
2.2.2 Destruction – Cleaning Up Coroutine Frames . . . 63
2.2.3 Combining Resumption and Destruction . . . 64
2.2.4 Practical Insight . . . 65
2.3 Illustration – Life Cycle Diagram . . . 66
2.3.1 Coroutine Life Cycle States . . . 66
2.3.2 Life Cycle Diagram (Text Representation) . . . 66
2.3.3 Step-by-Step Explanation . . . 67
2.3.4 Practical Example Illustrating Life Cycle . . . 68
2.3.5 Insights from the Diagram . . . 69
2.4 Practical Example – Coroutine Managing Multiple Tasks in a Defined Sequence . . . 71
2.4.1 Concept . . . 71
2.4.2 Implementation . . . 71
2.4.3 Explanation . . . 73
2.4.4 Output . . . 74

2.4.5 Practical Insights . . . 74

3 Awaitables and Awaiters **76**

3.1 Awaitable Concept . . . 76

3.1.1 Definition of Awaitable . . . 76

3.1.2 Built-in Awaitables in C++23 . . . 77

3.1.3 Example: Custom Awaitable . . . 77

3.1.4 Practical Insights . . . 79

3.2 Custom Awaiters . . . 81

3.2.1 Anatomy of an Awaiter . . . 81

3.2.2 Example: Timer Awaiter . . . 81

3.2.3 Explanation . . . 83

3.2.4 Benefits of Custom Awaiters . . . 84

3.2.5 Advanced Tips . . . 84

3.3 Await Transformations . . . 86

3.3.1 Concept of Await Transformation . . . 86

3.3.2 Example: Logging Await Transform . . . 86

3.3.3 Explanation . . . 88

3.3.4 Benefits of Await Transformation . . . 89

3.3.5 Practical Use Cases . . . 89

3.4 Practical Example – Implementing a Custom Awaitable for Waiting on a Timer or Resource . . . 90

3.4.1 Concept . . . 90

3.4.2 Timer Awaitable Example . . . 90

3.4.3 Explanation . . . 92

3.4.4 Resource Awaitable Example . . . 92

3.4.5 Explanation . . . 93

3.4.6 Practical Insights . . . 94

4 Generators – Producing Data with Coroutines **96**

4.1 Using `co_yield` to Create Data Generators . . . 96

4.1.1 Concept of `co_yield` . . . 96

4.1.2 Basic Generator Example . . . 97

4.1.3 Explanation . . . 99

4.1.4 Practical Insights . . . 99
4.1.5 Advanced Example: Fibonacci Generator . . . 100
4.2 Generators vs Traditional Solutions (Performance Comparison) . . . 101
4.2.1 Traditional Approaches . . . 101
4.2.2 Generator-Based Approach . . . 102
4.2.3 Performance Comparison . . . 103
4.2.4 Practical Example: Comparing Execution . . . 104
4.2.5 When to Prefer Generators . . . 105
4.3 Practical Example – Number Sequence Generator or Reading Data from File/Network . . . 107
4.3.1 Number Sequence Generator . . . 107
4.3.2 Reading Data from a File Using a Generator . . . 108
4.3.3 Reading Data from a Network Source . . . 110
4.3.4 Practical Insights . . . 110

5 Tasks and Async Operations 112
5.1 Creating Asynchronous Tasks . . . 112
5.1.1 Concept . . . 112
5.1.2 Simple Task Example . . . 113
5.1.3 Using `co_await` for True Asynchronous Behavior . . . 114
5.1.4 Practical Insights . . . 115
5.1.5 Summary . . . 116
5.2 Integration with `std::future` and `std::task` (C++23) . . . 117
5.2.1 `std::future` Integration . . . 117
5.2.2 `std::task` (C++23 Proposal) . . . 118
5.2.3 Comparison: `std::future` vs `std::task` . . . 119
5.2.4 Practical Example: Combining `std::future` with Coroutines . . . 119
5.2.5 Practical Insights . . . 120
5.3 Practical Example – Performing an Async Network I/O Request Using Coroutines . . . 121
5.3.1 Concept . . . 121
5.3.2 Custom Awaitable for Async Network Read . . . 121
5.3.3 Coroutine Performing Async Network I/O . . . 122
5.3.4 Key Advantages . . . 123
5.3.5 Extending to Real-World Applications . . . 124
5.3.6 Summary . . . 124

5.4 Using `co_await` for Managing Multiple Tasks . . . 125
5.4.1 Concept . . . 125
5.4.2 Sequential Awaiting . . . 125
5.4.3 Concurrent Awaiting . . . 126
5.4.4 Practical Patterns . . . 128
5.4.5 Error Handling . . . 128
5.4.6 Summary . . . 129

6 Real-World Applications 130
6.1 Concurrent Data Pipelines . . . 130
6.1.1 Concept . . . 130
6.1.2 Building a Simple Concurrent Pipeline . . . 131
6.1.3 Enhancing Concurrency . . . 133
6.1.4 Practical Insights . . . 134
6.1.5 Summary . . . 134
6.2 Game Loops Using Coroutines . . . 135
6.2.1 Traditional Game Loop . . . 135
6.2.2 Coroutine-Based Game Loop . . . 135
6.2.3 Benefits of Using Coroutines in Game Loops . . . 137
6.2.4 Advanced Example: Concurrent Coroutines in Game Loop . . . 137
6.2.5 Summary . . . 138
6.3 Network & I/O-Heavy Applications . . . 139
6.3.1 The Challenge with Network & I/O . . . 139
6.3.2 Asynchronous Network Request with Coroutines . . . 139
6.3.3 Handling Multiple I/O Tasks Concurrently . . . 141
6.3.4 File I/O with Coroutines . . . 141
6.3.5 Real-World Integration . . . 142
6.3.6 Best Practices . . . 142
6.3.7 Summary . . . 143
6.4 Practical Example – Data Stream Pipeline with Intelligent Suspensions . . . 144
6.4.1 Concept . . . 144
6.4.2 Pipeline Components . . . 144
6.4.3 Full Pipeline Execution . . . 146
6.4.4 Advanced Enhancements . . . 147

6.4.5 Key Insights . . . 148
6.4.6 Summary . . . 148

7 Best Practices **149**
7.1 Exception Management Inside Coroutines . . . 149
7.1.1 How Exceptions Work in Coroutines . . . 149
7.1.2 Exception Handling in Promise Types . . . 150
7.1.3 Exception Propagation with `co_await` . . . 151
7.1.4 Best Practices for Exception Management . . . 151
7.1.5 Practical Example: Async File Read with Exception Handling . . . 152
7.1.6 Summary . . . 153
7.2 Memory Management . . . 154
7.2.1 How Coroutines Use Memory . . . 154
7.2.2 Coroutine Handle and Memory Lifetime . . . 154
7.2.3 Custom Allocators for Coroutines . . . 155
7.2.4 RAII and Coroutine Memory Safety . . . 156
7.2.5 Practical Recommendations . . . 157
7.2.6 Advanced Memory Management Patterns . . . 157
7.2.7 Summary . . . 157
7.3 Avoiding Common Pitfalls – Dangling Handles and Undefined Behavior . . . 159
7.3.1 Dangling Coroutine Handles . . . 159
7.3.2 Undefined Behavior due to Suspensions . . . 160
7.3.3 Handling Exceptions Safely . . . 160
7.3.4 Avoiding Resource Leaks . . . 161
7.3.5 Thread Safety Concerns . . . 161
7.3.6 Practical Guidelines . . . 162
7.3.7 Example: Safe Coroutine Wrapper . . . 162
7.3.8 Summary . . . 162
7.4 Practical Example – Resilient Coroutine with Exception Handling . . . 164
7.4.1 Scenario . . . 164
7.4.2 Resilient Awaitable . . . 164
7.4.3 Resilient Coroutine . . . 165
7.4.4 Execution . . . 166
7.4.5 Enhancements for Real-World Applications . . . 167

7.4.6 Key Takeaways . . . 168

8 Debugging & Performance Optimization 169

8.1 Tracking Resumption and Suspension . . . 169
8.1.1 Why Track Resumptions and Suspensions . . . 169
8.1.2 Instrumenting Coroutine Promises . . . 170
8.1.3 Tracking Suspensions in Awaitables . . . 171
8.1.4 Counters and Profiling . . . 172
8.1.5 Visualizing Resumptions and Suspensions . . . 172
8.1.6 Best Practices . . . 173
8.1.7 Summary . . . 173
8.2 Tools and Libraries for Profiling . . . 174
8.2.1 Built-in Timing and Profiling Techniques . . . 174
8.2.2 External Profiling Tools . . . 175
8.2.3 Custom Logging Libraries . . . 176
8.2.4 Best Practices for Coroutine Profiling . . . 177
8.2.5 Summary . . . 177
8.3 Practical Example – Performance Analysis for a Large Data Generator Using Coroutines . 178
8.3.1 Scenario . . . 178
8.3.2 Coroutine-Based Generator . . . 178
8.3.3 Measuring Performance . . . 179
8.3.4 Comparison with Traditional Approaches . . . 180
8.3.5 Profiling Suspensions . . . 180
8.3.6 Key Takeaways . . . 181

9 Advanced C++23 Features 182

9.1 Awaitable Improvements . . . 182
9.1.1 Motivation for Awaitable Improvements . . . 182
9.1.2 Simplified Awaitable Conventions . . . 183
9.1.3 Standard Library Integration . . . 184
9.1.4 Lazy and Composable Awaitables . . . 185
9.1.5 Best Practices . . . 185
9.1.6 Summary . . . 185
9.2 Composable Coroutines . . . 187

9.2.1 What is Coroutine Composability? . . . 187
9.2.2 Basic Example of Coroutine Composition . . . 187
9.2.3 Advantages of Composable Coroutines . . . 188
9.2.4 Composable Awaitables in C++23 . . . 189
9.2.5 Parallel Composition . . . 189
9.2.6 Best Practices for Composable Coroutines . . . 190
9.2.7 Summary . . . 190
9.3 Integration with `std::expected` . . . 191
9.3.1 Motivation . . . 191
9.3.2 Using `std::expected` with Coroutines . . . 191
9.3.3 Advantages . . . 192
9.3.4 Composable `std::expected` Example . . . 192
9.3.5 Best Practices . . . 193
9.3.6 Summary . . . 193
9.4 Practical Example – Integrated Chain of Coroutines with Error Handling Using `std::expected` . . . 195
9.4.1 Scenario . . . 195
9.4.2 Coroutine Implementation . . . 195
9.4.3 Execution . . . 197
9.4.4 Key Concepts Illustrated . . . 197
9.4.5 Best Practices for Integrated Pipelines . . . 197
9.4.6 Summary . . . 198

Conclusion **199**
Quick Summary of Learned Concepts . . . 199
Practical Steps to Implement Coroutines in Large Projects . . . 203
Tips for Professionals to Reduce Complexity and Maximize Efficiency . . . 207

Appendices **210**
Appendix A : Quick Reference – All Keywords and Types Used in Coroutines . . . 210
Appendix B: Cheatsheet for Awaitables and Generators Examples . . . 214
Appendix C: Additional Resources and References . . . 219

Author's Introduction

I recognize the transformative impact of the C++20 and C++23 additions for structured concurrency and threading—most notably, Modern C++ Coroutines. However, many experienced C++ developers still find coroutines complex and intimidating. This book is designed to bridge that gap: a focused, practical handbook dedicated entirely to this powerful feature. It explains every aspect in depth, simplifies difficult concepts, and demonstrates real-world usage through clear, proven examples and patterns.

My goal is to make coroutines accessible for professional developers. Before publication, the manuscript will undergo extensive review, meticulous proofreading, and rigorous testing of every example to ensure it serves as a reliable, production-ready resource for the C++ community.

For more discussions and valuable content about

Mastering Coroutines in Modern C++,

I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit the company website:

https://simplifycpp.org

Wishing everyone success and prosperity.

`Ayman Alheraki`

Introduction to Coroutines

Why Coroutines?

In modern software systems, concurrency, asynchronicity, and efficient resource usage are no longer optional — they are essential. C++ has long provided runnable threads, futures/promises, asynchronous I/O frameworks, and manual event loops, but each approach has trade-offs in complexity, performance, and readability. Coroutines (added in C++20 and evolved further by C++23) offer a middle ground: a lightweight, expressive mechanism for writing asynchronous and cooperative logic in a sequential style, without undue overhead.
Here is a structured argument for *why* coroutines matter, what they bring to C++ development, and when they make sense.

The problem space: what we lack without coroutines

Before coroutines, common idioms for concurrency/asynchrony in C++ include:

- **Threads + synchronization**: create threads, join them, use mutexes, condition variables, atomic operations, etc. This gives true parallelism but incurs overhead, synchronization complexity, and risks of deadlock, data racing, and priority inversion.
- **Callbacks / continuation-passing style (CPS)**: you hand a callback to an async function, and when the operation completes, the callback is invoked. This style can lead to *callback hell*, tangled control flow, and error-handling difficulties.
- **State machines or custom dispatch loops**: embed an explicit state variable (e.g. an `enum`) and in each "step" of the loop decide what to do next. This is verbose, brittle, and hard to maintain.

- **Futures / promises / `std::async` / `std::future`**: these provide a composition of asynchronous operations, but they are often blocking (waiting), limited in expressiveness, and don't always integrate cleanly with event loops or scheduling frameworks.

Each of these approaches introduces friction:

- The logical flow of a multi-step asynchronous operation (e.g. "read, process, write") gets split across multiple callbacks or chained futures, scattering state across closures or lambda captures.
- Error and exception forwarding through callbacks or promise chains is tricky.
- Overhead: spawning threads or context switches can be expensive; managing many threads for many small tasks doesn't scale.
- Readability and maintainability suffer when the programmer has to intertwine scheduling logic, error handling, and core domain logic.

Coroutines aim to alleviate or eliminate these friction points.

What is a coroutine — and what makes it different?

A *coroutine* is a routine whose execution can be **suspended** at certain points (called *suspension points*) and **resumed** later, while preserving its local state across those suspensions. In other words, coroutines generalize ordinary subroutines by allowing control to exit and re-enter the function multiple times, rather than running start-to-finish in one shot.
Key characteristics:

- **State preservation**: local variables, parameters, control position (instruction pointer) are preserved across suspension and resumption.
- **Suspension & resumption**: control may yield out (via `co_await`, `co_yield`, or implicit suspension) and later resume where it left off.
- **Lightweight**: unlike full threads, coroutines don't require OS threads or heavy context switching. The "stack frame" or activation object is managed by the coroutine machinery (often on the heap or via compiler-managed allocation).
- **Composable control flow**: coroutines can await other coroutines, yield values, or be coordinated by schedulers or executors.

Because C++'s coroutine support is *stackless* (i.e. you cannot arbitrarily suspend deep inside nested call frames unless those frames are coroutine-enabled), the language gives you a *low-level plumbing* (promise types, awaiters, handles). But that plumbing is exactly what lets you build flexible, performant abstractions.
With C++23, the standard library finally adds a first-class coroutine type `std::generator`, making it easier to build simple generator-based flows without reinventing the promise machinery. This reduces boilerplate for common use cases. (Before C++23 you often had to rely on third-party libraries or hand-written boilerplate.)

Benefits of using coroutines in modern C++ (C++23)

Here are the primary advantages that make coroutines compelling:

1. **Expressive, sequential-style asynchronous code**

   Coroutines allow writing asynchronous or non-blocking operations in a natural, linear style:

   ```
   Task<int> fetch_and_process()
   {
       auto data = co_await async_read();   // suspend until read completes
       auto processed = do_process(data);
       co_await async_write(processed);
       co_return processed_sum(processed);
   }
   ```

   This looks and feels like synchronous code, but under the hood it is non-blocking. You avoid nested callbacks or awkward `.then()` chains, making the logic easier to follow, maintain, and extend.

2. **Low-overhead suspension**

   Switching among coroutines is far cheaper than switching OS threads: no kernel context switch, no heavy scheduler invocation, and minimal stack-copying. The coroutine's state is stored in a promise object (or associated frame), and resumption is effectively a jump/resume of that state. This allows large numbers of concurrent coroutines (thousands or millions) with low memory/CPU overhead.

3. **Composability & modular concurrency**

Because coroutines can `co_await` other coroutine-based tasks or awaitables, you can build modular components (e.g. `read_file()`, `transform()`, `write_file()`) and combine them in higher-level orchestrators. This composability works well with modern patterns of *structured concurrency*. You can propagate cancellation, error semantics, and lifetime control more cleanly than in callback-based systems.

4. **Cleaner error and exception propagation**

   Within coroutines, you can use ordinary `try/catch` logic. If an awaited coroutine throws (or reports an error), the exception can propagate naturally up the call chain. This is often simpler to manage than error propagation across chained callbacks or futures.

5. **Fewer synchronization concerns**

   Because coroutines execute cooperatively (they yield explicitly), you don't typically need mutexes or locks among coroutines running in the same thread. Shared mutable state still needs care, but concurrency hazards are reduced when many tasks run in a single event loop context. You avoid many pitfalls of preemptive multithreading (race conditions, priority inversion, deadlocks) in this cooperative paradigm.

6. **Better memory and resource efficiency**

   Because you avoid spawning heavyweight threads or allocating large stacks, coroutines allow better scaling for I/O-heavy, event-driven systems. You can overlap I/O, compute, and orchestration with minimal idle time and low resource footprint.

   Moreover, in high-performance domains, coroutines can be used as building blocks to achieve advanced optimization techniques. For example:

   - Some coroutine-based parallel libraries (e.g. *libfork*) use coroutines to implement lock-free *continuation stealing* to realize efficient fork-join parallelism with lower memory overhead.
   - In database engines (e.g. *CoroBase*), coroutines are used to hide memory stalls by interleaving I/O/data prefetch with computation, improving throughput.

   These examples illustrate that coroutines are not just a convenience — they can be enablers of new, high-performance designs.

When (and when not) to use coroutines

Coroutines are a powerful tool, but they are not a universal panacea. It is important to know when to use them and when other abstractions are more suitable.

Good use cases for coroutines include:

1. **I/O-bound asynchronous tasks** (networking, file I/O, timers): coroutines elegantly manage waiting for I/O without blocking threads.
2. **Generator / lazy sequences**: e.g. streaming data, reading lines, infinite sequences. The introduction of `std::generator` in C++23 simplifies that pattern.
3. **State-machine style logic**: where a function needs to pause and resume at multiple points, e.g. protocol handling, parsing, streaming pipelines.
4. **Orchestration / structured concurrency**: combining multiple asynchronous subtasks, propagating cancellation, and managing lifetimes.
5. **High-concurrency event systems**: for example, handling many lightweight tasks concurrently without creating a thread per task.

Limitations or caveats:

- **Learning curve / boilerplate**: building custom coroutine types (promise types, awaitables, scheduling) can be complex, especially initially. Many early adopters regard C++ coroutines as a "low-level library-building tool" more than a direct user-facing convenience.
- **Lack of higher-level standard library support**: until C++23, the standard library did not provide many coroutine abstractions beyond the plumbing. Even now, ecosystem support (executors, scheduling, cancellation tokens) is still maturing.
- **Stackless restriction**: because C++ coroutines are stackless, you cannot suspend out of arbitrary call frames unless those frames are coroutine-enabled (i.e. you can't `co_await` deep within non-coroutine calls). This can constrain how you structure your code.
- **Parallelism vs concurrency**: coroutines enable concurrency (interleaved execution) inside a thread, but they do not automatically provide parallel execution across multiple cores. If you truly need parallel CPU-bound workloads, you still require threads, tasks, or external parallelism frameworks.

- **Debugging and tooling support**: while support is improving, stepping through suspension points and visualizing promise state may sometimes be less straightforward than traditional code. Some IDEs and debuggers are still catching up in coroutine awareness.

A small illustrative example in "realistic style"

Here's a more concrete example (C++23 style) that demonstrates the advantage of coroutines. Suppose we need to fetch multiple web resources, process them, and store results, *concurrently* (but without blocking threads). A coroutine-based orchestrator might look like:

```
// A simplified awaitable representing an HTTP fetch.
struct HttpFetch {
    std::string url;

    bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) const {
        // schedule asynchronous fetch; when done, resume h
        schedule_http_fetch(url, [h](std::string result){
            // store result somewhere accessible to promise
            h.resume();
        });
    }
    std::string await_resume() const {
        return get_fetched_result(url);
    }
};

Task<void> fetch_and_store(std::string url) {
    try {
        auto body = co_await HttpFetch{url};
        auto processed = process_response(body);
        co_await async_store(processed);
    } catch (std::exception &e) {
        log_error(e);
    }
}

Task<void> fetch_all(std::vector<std::string> urls) {
    std::vector<Task<void>> tasks;
    for (auto &u : urls) {
        tasks.push_back(fetch_and_store(u));
```

```
    }
    // co_await all tasks (pseudo-syntax)
    co_await when_all(tasks);
}
```

The key points:

- The control flow reads naturally: fetch, then process, then store.
- You avoid explicit callback chaining; error handling is local.
- The coroutine machinery handles suspension, resuming, and combining multiple tasks.
- Because of lightweight suspension, many such fetch tasks can run concurrently, limited mainly by I/O, not thread count.

In a contrast with callback-based code, you'd have to write lambdas inside lambdas, propagate errors manually, and manage lifetimes of callback scopes. Coroutines let you factor logic more cleanly.

Summary (Why use coroutines?)

To wrap up, here's a distilled "why" checklist:

- You want **asynchronous, non-blocking workflows** but want to write them in a clear, sequential style.
- You want **low-overhead, scalable concurrency** without spawning many OS threads.
- You seek **composability**, modularity, and maintainable structure in your concurrency code.
- You want **better error-handling semantics** (using exceptions or structured control flow).
- You want to *avoid* the usual callback hell or manual state-machine entanglement.

C++ coroutines — especially with enhancements in C++23 (such as `std::generator`) — give you a powerful substrate under which you can build high-performance, expressive concurrency frameworks. In the following chapters, you will explore how the coroutine machinery (promise types, awaitables, coroutine handles) works in detail, and how to design robust coroutine-based abstractions and executors suited for real systems.

Difference between Coroutines, Traditional Functions, and Threads

Understanding coroutines requires placing them in context. To a developer familiar with traditional functions and threads, the idea of *suspending* and *resuming* a function can feel unusual. This section highlights the conceptual and practical differences among **coroutines**, **traditional functions**, and **threads**. By comparing their execution models, memory behavior, performance implications, and real-world applicability, we clarify what makes coroutines unique and why they are not simply a rebranded form of threading.

Traditional Functions

A **traditional function** in C++ executes from the beginning until it reaches its end (or returns early). Its lifetime is *single-entry, single-exit*:

- Once invoked, execution continues in a linear path.
- Local variables live on the call stack and are destroyed when the function returns.
- Control flow cannot re-enter the function at an arbitrary point without restarting it.

This model is simple, predictable, and efficient. However, it lacks flexibility in scenarios where a function must:

1. Pause midway, hand back control, and resume later.
2. Represent long-running or asynchronous work without blocking the thread.

In such cases, developers have historically resorted to workarounds such as **callbacks** or **state machines**, which fragment logic and increase complexity.

Threads

A **thread** is an independent execution unit scheduled by the operating system. Threads run concurrently (and potentially in parallel across CPU cores). Their key properties are:

- **Preemptive scheduling**: the OS can suspend/resume threads at arbitrary points.

- **Independent call stacks**: each thread has its own stack and executes functions in isolation.
- **Parallelism**: multiple threads can run simultaneously on multicore CPUs.
- **Heavyweight resources**: threads consume significant memory (stack allocation, kernel structures) and context-switching costs.

Threads are indispensable for CPU-bound parallel workloads where true parallelism is required. However, they come with challenges:

- Synchronization overhead (mutexes, atomics, locks).
- Risks of deadlocks, race conditions, and data corruption.
- High cost of context switching when managing large numbers of threads.
- Difficulty scaling to millions of concurrent tasks.

Coroutines

Coroutines extend the idea of functions by enabling **suspension and resumption** at well-defined points, without tearing down the function's state. In C++ (C++20 and enhanced in C++23):

- Local variables persist across suspension points.
- Suspension occurs explicitly (using `co_await`, `co_yield`, or `co_return`).
- Execution resumes precisely where it left off.
- Coroutines are *stackless*: they don't own full call stacks like threads, but instead store their execution state in a coroutine frame (allocated by the compiler).

This allows coroutines to model asynchronous or incremental workflows naturally. They are:

- **Lightweight**: millions of coroutines can exist with little overhead.
- **Cooperative**: suspension happens explicitly (not preemptively).
- **Composable**: coroutines can `co_await` other coroutines, forming pipelines of asynchronous logic.
- **Integrative**: coroutines work within a single thread or event loop, but can also coordinate with multi-threaded schedulers if required.

Conceptual Comparison

Aspect	Traditional Functions	Threads	Coroutines
Execution model	Run from start to finish, no resumption	Independent concurrent execution unit	Suspend/resume at explicit points
Scheduling	Caller decides when to call	Preemptively scheduled by OS	Cooperatively suspended/resumed
State persistence	Lost after return	Each thread maintains its own stack	Preserved in coroutine frame
Memory footprint	Minimal (stack frame only)	Large (stack + kernel structures)	Small (compiler-managed frame)
Parallelism	None	Yes, across cores	No (concurrent, not inherently parallel)
Scalability	High for simple calls	Limited by OS and hardware resources	Very high, lightweight tasks scale efficiently
Error handling	Normal `try`/`catch`	Normal `try`/`catch` per thread	Normal `try`/`catch` within coroutine

Performance and Use Cases

- **Traditional Functions**
  - **Best for**: straightforward computations, deterministic logic, synchronous tasks.
  - **Not suitable for**: asynchronous or long-running operations where control must be yielded.
- **Threads**
  - **Best for**: CPU-intensive parallel computations, workloads that benefit from multicore execution.
  - **Not suitable for**: millions of fine-grained concurrent tasks, due to resource overhead.
- **Coroutines**

- **Best for**: I/O-bound tasks, cooperative concurrency, state machines, asynchronous workflows.
- **Not suitable for**: raw parallelism; they provide concurrency, not automatic multicore utilization (though they can be scheduled onto threads).

Clever Example: Comparing the Three

Suppose we want to **process a stream of integers** in chunks, where processing each chunk may require waiting for I/O.

Traditional function:

Blocks execution during the wait:

```
int process_chunk() {
    auto data = blocking_read();
    return compute(data);
}
```

Thread-based:

Offloads work to a thread pool, but each thread blocks on I/O:

```
void process_async() {
    std::thread t([]{
        auto data = blocking_read();
        auto result = compute(data);
        store(result);
    });
    t.detach();
}
```

This scales poorly if we spawn thousands of threads.

Coroutine-based:

Suspends during I/O, resumes seamlessly without blocking the thread:

```
Task<void> process_coroutine() {
    auto data = co_await async_read();   // suspends while I/O happens
    auto result = compute(data);         // runs immediately when resumed
    co_await async_store(result);        // suspends again for storage
}
```

Here, the coroutine lets a single thread manage thousands of I/O operations concurrently, without heavy context switching or thread explosion.

Summary

- **Traditional functions** are simple and efficient but lack suspension capability.
- **Threads** provide true parallelism but are heavyweight, complex, and hard to scale in massive concurrency scenarios.
- **Coroutines** are lightweight, cooperative, and composable. They shine in scenarios requiring scalable concurrency and asynchronous logic, with performance close to traditional functions and without the overhead of threads.

Thus, coroutines occupy a **sweet spot**: more flexible than traditional functions, more lightweight than threads, and ideal for modern high-performance C++ applications that must juggle many asynchronous tasks efficiently.

What's New in C++20 and C++23

The introduction of coroutines in C++20 marked a turning point for the language. For decades, concurrency and asynchrony in C++ relied on threads, futures, promises, or external libraries. With coroutines, C++ gained first-class language support for cooperative concurrency. However, the coroutine facility introduced in C++20 was deliberately minimalistic: the standard provided the *mechanics* (the "compiler hooks"), but left higher-level abstractions to be developed later.
C++23 extends the groundwork laid in C++20, improving ergonomics and adding essential library support. To appreciate the current landscape, it is useful to understand what exactly C++20 introduced, what C++23 added, and how these features evolve coroutine programming from "low-level machinery" into more approachable tools.

Coroutines in C++20: The Foundation

C++20 introduced the **core language support** for coroutines. At its heart, the compiler gained the ability to transform functions marked with `co_await`, `co_yield`, or `co_return` into state machines. These transformed functions suspend and resume execution while retaining local state.
The **essential elements of C++20 coroutines** include:

1. **Coroutine keywords**

   - `co_await`: suspends execution until an awaitable result is ready.
   - `co_yield`: produces a value and suspends, resuming later to produce more values (generator style).
   - `co_return`: ends the coroutine and returns a final result to its caller.

2. **Coroutine state machine**

   The compiler lowers a coroutine into a *promise object* and *coroutine frame*, containing local variables and the program counter. This machinery enables suspension and resumption without rewriting manual state machines.

3. **Coroutine traits and promise type**

   A coroutine's return type must be a coroutine-aware type, such as a custom `Task<T>` or generator. This type defines the *promise_type* that drives how the coroutine is created, suspended, and destroyed.

Example (C++20 custom task skeleton):

```
template<typename T>
struct Task {
    struct promise_type {
        T value;
        Task get_return_object() { return Task{this}; }
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_value(T v) { value = v; }
        void unhandled_exception() { std::terminate(); }
    };

    T result() { return handle->promise().value; }

private:
    explicit Task(promise_type* p) : handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    std::coroutine_handle<promise_type> handle;
};

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

This demonstrates the **plumbing-heavy nature of C++20 coroutines**: while powerful, much boilerplate was required just to build usable coroutine types.

Coroutines in C++23: Raising the Abstraction

C++23 built upon C++20's foundation by adding **standard library facilities** that make coroutines easier to use and more practical for real-world programming.
The major C++23 coroutine-related additions are:

1. **`std::generator`**

   C++23 introduces `std::generator`, a ready-to-use coroutine type for lazy sequences. It drastically reduces boilerplate for common generator patterns, where a coroutine yields multiple values on demand.

   Example:

```cpp
#include <generator>
#include <iostream>

std::generator<int> fibonacci(int limit) {
    int a = 0, b = 1;
    while (a <= limit) {
        co_yield a;
        auto next = a + b;
        a = b;
        b = next;
    }
}

int main() {
    for (int x : fibonacci(50)) {
        std::cout << x << " ";
    }
}
```

This generator lazily produces Fibonacci numbers without manual iterators or state machines. Prior to C++23, writing this required building a custom coroutine return type by hand.

2. **`std::suspend_always` and `std::suspend_never` refinements**

   While present in C++20, their role became clearer in C++23 examples and practice. They allow fine-grained control of suspension behavior at coroutine entry and exit:

   - `std::suspend_always`: explicitly suspend (caller must resume).
   - `std::suspend_never`: run immediately without suspension.

   These simple utilities enable efficient integration with frameworks or schedulers.

3. **Standardization of coroutine utilities**

   C++23 provides more consistent integration with ranges, iterators, and structured concurrency frameworks under development. This reduces reliance on third-party coroutine libraries just to perform simple tasks like iteration.

4. **Debugging and library support**

   C++23 improves debug-ability by ensuring coroutine state and exceptions propagate more predictably. Exceptions thrown inside a coroutine can be caught normally by surrounding `try/catch` blocks, making coroutines behave more like ordinary functions.

Why C++23 Coroutines Matter

The shift from C++20 to C++23 is not about new keywords — those remain the same — but about **ergonomics and usability**. Developers can now write practical coroutine code without reinventing the wheel.

- In C++20: You needed to write custom `Task`, `Promise`, and `Awaitable` types to accomplish even basic tasks.
- In C++23: You can use `std::generator` directly for lazy iteration, and the ecosystem of coroutine-ready abstractions is growing.

This makes coroutines accessible for ordinary developers, not just library authors.

Example: From C++20 to C++23

C++20 approach:

```
// Custom coroutine return type (verbose)
Task<int> numbers() {
    for (int i = 1; i <= 3; ++i)
        co_yield i;  // Requires building a full custom generator type
}
```

C++23 approach:

```
#include <generator>

std::generator<int> numbers() {
    for (int i = 1; i <= 3; ++i)
        co_yield i;  // Uses std::generator, no boilerplate
}
```

The result is identical in behavior, but the C++23 version removes all the custom infrastructure. This reflects the broader philosophy: *C++20 gave us the raw power; C++23 makes it practical.*

Looking Ahead Beyond C++23

While C++20 and C++23 establish coroutines as a core part of the language, the work is not finished. Proposals under discussion for C++26 and beyond include:

- Executors and schedulers for structured concurrency.
- Native coroutine cancellation and timeouts.
- Seamless integration with networking and I/O frameworks.

This means coroutines are still an evolving feature set, with the current standards providing the necessary foundation for higher-level libraries.

Summary

- **C++20**: Introduced coroutine mechanics, keywords, and promise/awaiter framework. Very powerful but verbose, requiring custom infrastructure.
- **C++23**: Added standard library support (`std::generator`) and improved integration, reducing boilerplate and making coroutines more approachable.
- **Impact**: C++ coroutines moved from "experimental low-level machinery" to "usable building blocks" for mainstream developers.

With this understanding, we now have the historical and technical context for why mastering coroutines is essential for high-performance modern C++ development.

Practical Goals of This Guide

Before delving into the technical depth of C++20/23 coroutines, it is crucial to establish what this guide aims to accomplish. Coroutines are not merely a new syntax feature; they fundamentally change how we write asynchronous and lazy computations. Unlike traditional concurrency or callback-based systems, coroutines bring structured, maintainable, and high-performance solutions to real-world problems. This booklet's objective is not just to teach syntax but to demonstrate the philosophy, design patterns, and problem-solving strategies behind modern coroutines.

Demystifying the Coroutine Model

Many developers find the coroutine specification intimidating due to its reliance on compiler transformations, promise objects, and custom awaiters. One of the key goals of this guide is to remove that sense of mystery. Through progressive examples—starting from minimal coroutine functions and advancing to robust coroutine frameworks—you will gain a solid mental model of what happens when a coroutine is compiled and executed.
For instance, consider the difference between a simple generator and a fully asynchronous network handler. This guide will start with:

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

struct Generator {
    struct promise_type {
        int value;
        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 {}; }
        void unhandled_exception() { throw; }
        std::suspend_always yield_value(int v) noexcept {
            value = v;
            return {};
        }
        void return_void() {}
    };

    std::coroutine_handle<promise_type> handle;
```

```
    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }
    int next() {
        if (!handle.done()) handle.resume();
        return handle.promise().value;
    }
};

Generator simpleCounter(int n) {
    for (int i = 0; i < n; ++i)
        co_yield i;
}

int main() {
    auto g = simpleCounter(5);
    for (int i = 0; i < 5; ++i)
        std::cout << g.next() << " ";
}
```

This example reveals the mechanics of suspension and resumption without requiring multithreading or external libraries. The guide builds from here to show how coroutines become powerful abstractions for complex workflows.

Bridging the Gap Between Theory and Practice

While the C++ standard defines coroutines at a language level, practical usage involves integrating them with real systems—such as event loops, I/O schedulers, and concurrency frameworks. A primary goal of this guide is to provide context-sensitive examples:

- Writing cooperative multitasking code without OS threads.
- Building custom coroutine types for data pipelines.
- Using coroutines in networking (asynchronous sockets, HTTP handlers).
- Leveraging coroutines in GUI event-driven programming.

By focusing on practical code, readers will not just understand *how* coroutines work but also *why* they are useful in production.

Developing an Intuitive Understanding

Coroutines often feel abstract because much of their behavior is handled by the compiler. This guide emphasizes clear mental models. You will repeatedly see:

- When suspension points occur.
- How state is preserved across suspensions.
- How coroutines integrate with the existing C++ type system.
- How C++23 refinements, such as better `std::generator` and synchronization primitives, simplify coroutine-based designs.

The goal is for you to be able to explain, in plain terms, what happens when the program encounters `co_await`, `co_yield`, or `co_return`.

Preparing for High-Performance Systems

Another practical aim of this booklet is to highlight scenarios where coroutines outperform traditional methods. For example, asynchronous I/O often uses callbacks, which quickly become unmanageable ("callback hell"). Threads, on the other hand, carry scheduling overhead. Coroutines combine the best of both worlds—minimal overhead and structured code.
By the end of the guide, you will be equipped to:

- Replace complex callback-driven APIs with coroutine-based flows.
- Build high-performance network applications that scale with thousands of simultaneous tasks.
- Implement data streaming pipelines that yield results incrementally, reducing memory usage.
- Apply coroutine techniques in game development, where deterministic control over execution is critical.

Serving Both Practitioners and Explorers

This guide is crafted for two main audiences:

- **Practical Developers**: Those who want ready-to-use patterns, code snippets, and techniques for immediate integration into real-world projects.

- **Language Explorers**: Those who want to understand the underlying machinery of promises, awaiters, and coroutine handles to design their own coroutine abstractions.

Both paths are woven into the chapters so that you can decide how deep you want to go without losing track of practical utility.

Long-Term Relevance and Future-Proofing

Finally, the guide aims to make your understanding of coroutines future-proof. While the current focus is C++20 and C++23, the trajectory of coroutine evolution suggests continued refinement in upcoming standards. By mastering the fundamentals and advanced applications today, you will be ready to adapt to enhancements like extended library support and tighter integration with executors in future standards.

In summary, the practical goals of this guide are to:

1. Remove the complexity and mystery behind coroutines.
2. Provide real-world, production-quality examples.
3. Develop intuitive mental models for suspension, resumption, and execution flow.
4. Empower developers to design high-performance, scalable systems.
5. Cater to both practitioners and language enthusiasts.
6. Future-proof your coroutine knowledge for C++26 and beyond.

Chapter 1

Deep Understanding of Coroutines

1.1 Core Structure of Coroutines

Coroutines in C++ are built around three fundamental keywords: `co_await`, `co_yield`, and `co_return`. These form the vocabulary through which coroutines suspend, resume, and deliver values. Understanding them is essential for mastering coroutine design.

1.1.1 `co_await` – Awaiting Asynchronous Results

The `co_await` operator is at the heart of coroutine-based asynchronous programming. It represents a *suspension point* where the coroutine pauses execution until a result is ready, without blocking the calling thread.

- **Key Concepts:**
  - When encountering `co_await`, the coroutine **checks** if the awaited object is ready.
  - If ready, execution continues immediately.
  - If not ready, the coroutine suspends, control returns to the caller (or scheduler), and execution resumes later.
  - The mechanism is defined by the *awaiter* object provided by the awaited expression.
- **Example: Asynchronous Simulation**

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


// SleepTask is a lightweight coroutine wrapper representing a simple asynchronous task.
// It provides a minimal coroutine type that suspends immediately upon starting and
// resumes until the coroutine completes.
//
// The internal promise_type manages the coroutine lifecycle:
// - get_return_object(): returns a SleepTask linked to the coroutine handle.
// - initial_suspend(): suspends the coroutine immediately after it is created.
// - final_suspend(): suspends the coroutine just before it is destroyed, allowing
//   any awaiting code to finish properly.
// - unhandled_exception(): terminates the program if an exception escapes the coroutine.
// - return_void(): allows the coroutine to complete without returning a value.
//
// The SleepTask destructor ensures that the coroutine handle is properly destroyed,
// releasing all associated resources.
struct SleepTask {
    struct promise_type {
        SleepTask get_return_object() {
            return SleepTask{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };

    std::coroutine_handle<promise_type> handle;
    explicit SleepTask(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~SleepTask() { if (handle) handle.destroy(); }
};

// Sleeper is a simple awaitable type used in coroutines to asynchronously pause execution.
// - await_ready(): always returns false to indicate the coroutine should suspend.
// - await_suspend(): starts a detached thread that sleeps for 1 second and then resumes the
↪  coroutine.
// - await_resume(): called when the coroutine resumes; does nothing in this case.
// This allows coroutines to "sleep" without blocking the main thread.
```

```cpp
struct Sleeper {
    bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) const {
        std::thread([h]() {
            std::this_thread::sleep_for(std::chrono::seconds(1));
            h.resume();
        }).detach();
    }
    void await_resume() const noexcept {}
};

// doWork is a simple coroutine function that demonstrates asynchronous execution using SleepTask.
// It prints a starting message, then asynchronously waits for 1 second using the Sleeper awaitable,
// and finally prints a completion message once the wait is over.
//
// - The use of `co_await Sleeper{}` suspends the coroutine without blocking the main thread.
// - The function returns a SleepTask, which manages the coroutine handle and ensures proper cleanup.

SleepTask doWork() {
    std::cout << "Starting work...\n";
    co_await Sleeper{};
    std::cout << "Work finished after waiting.\n";
}

// Entry point of the program demonstrating the SleepTask coroutine in action.
// - Calls doWork() to start the asynchronous task, returning a SleepTask object.
// - Resumes the coroutine manually via task.handle.resume() to begin execution.
// - Sleeps the main thread for 2 seconds to keep the program alive long enough
//   for the asynchronous coroutine to complete.
int main() {
    auto task = doWork();
    task.handle.resume();
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Keep program alive
}
```

Here, `co_await` suspends execution until the custom `Sleeper` resumes the coroutine after one second. This demonstrates how coroutines achieve asynchronous waiting without blocking threads.

1.1.2 `co_yield` – Generating Incremental Values

The `co_yield` operator is designed for *generators*: coroutines that produce values one at a time while maintaining internal state between calls. Each `co_yield` acts as a suspension point that provides a value to the caller and pauses execution until the next request.

- **Key Concepts:**
  - Similar to Python's generators, but with C++'s strong typing and performance guarantees.
  - State is automatically preserved between yields.
  - The coroutine can produce values lazily, avoiding pre-computation of entire sequences.
- **Example: Simple Generator**

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

// Generator<T> is a coroutine-based generator type that produces values of type T asynchronously.
// It provides a simple mechanism to iterate over generated values using coroutines.
//
// The internal promise_type manages the coroutine lifecycle:
// - get_return_object(): returns a Generator linked to the coroutine handle.
// - initial_suspend(): suspends the coroutine immediately after creation.
// - final_suspend(): suspends the coroutine before destruction to allow proper cleanup.
// - unhandled_exception(): terminates the program if an exception escapes the coroutine.
// - yield_value(T v): stores the yielded value and suspends the coroutine until resumed.
// - return_void(): allows the coroutine to complete without returning a value.
//
// The Generator stores the coroutine handle and ensures proper destruction in its destructor.
// The next() member function resumes the coroutine to produce the next value and returns it.
template<typename T>
struct Generator {
    struct promise_type {
        T value;
        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 {}; }
        void unhandled_exception() { std::terminate(); }
```

```
        std::suspend_always yield_value(T v) noexcept {
            value = v;
            return {};
        }
        void return_void() {}
    };

    std::coroutine_handle<promise_type> handle;
    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    T next() {
        if (!handle.done()) handle.resume();
        return handle.promise().value;
    }
};

// countUpTo is a coroutine function that generates integers from 1 up to n using the Generator<int>
↪  type.
// - Uses co_yield to produce each integer asynchronously, suspending the coroutine after each yield.
// - Returns a Generator<int> object that allows the caller to retrieve the values one by one using
↪  next().
// - Demonstrates a simple numeric generator implemented with C++ coroutines.
Generator<int> countUpTo(int n) {
    for (int i = 1; i <= n; ++i)
        co_yield i;
}

// Entry point demonstrating the use of the Generator<int> coroutine.
// - Calls countUpTo(5) to create a generator that produces integers from 1 to 5.
// - Uses a loop to retrieve and print each generated value using g.next().
// - Outputs: 1 2 3 4 5
int main() {
    auto g = countUpTo(5);
    for (int i = 0; i < 5; ++i)
        std::cout << g.next() << " ";
}
```

Output:

```
1 2 3 4 5
```

Here, `co_yield` provides each value and suspends, resuming exactly where it left off on the next call.

1.1.3 `co_return` – Returning from a Coroutine

The `co_return` keyword is the coroutine equivalent of `return` in ordinary functions. It ends the coroutine and optionally provides a final value to the coroutine's promise.

- **Key Concepts:**
  - Ends execution of the coroutine.
  - Can provide a final value if the coroutine's return type supports it.
  - Often used in asynchronous tasks (`co_return value;`).
- **Example: Returning Values**

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

struct ReturnTask {
    struct promise_type {
        std::optional<int> result;
        ReturnTask get_return_object() {
            return ReturnTask{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_value(int v) { result = v; }
    };

    std::coroutine_handle<promise_type> handle;
    explicit ReturnTask(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~ReturnTask() { if (handle) handle.destroy(); }

    int get() {
        handle.resume();
        return *handle.promise().result;
    }
```

```
};

ReturnTask computeValue() {
    co_return 42; // Return a value directly
}

int main() {
    auto task = computeValue();
    std::cout << "Result: " << task.get() << "\n";
}
```

Output:

```
Result: 42
```

Here, `co_return` provides a clean way to end the coroutine and deliver the final result.

1.1.4 How They Work Together

- **`co_await`** handles *asynchronous waiting* and integrates with external event systems.
- **`co_yield`** produces *incremental values* lazily.
- **`co_return`** ends the coroutine and returns the final result.

These three operators define the *core grammar of coroutines*, allowing C++ to express both *asynchronous tasks* and *generators* within a unified model.

In summary:

- Use `co_await` for asynchronous tasks or waiting.
- Use `co_yield` for producing sequences of values.
- Use `co_return` for returning the final result.

Together, they form the backbone of modern coroutine programming in C++20/23.

1.2 Promise Type and `std::coroutine_traits`

In C++ coroutines, understanding the **promise type** and **`std::coroutine_traits`** is essential because they form the glue between your coroutine code and the compiler-generated state machine. They define how a coroutine is created, suspended, resumed, and ultimately destroyed. This section provides a detailed explanation with modern C++23 examples.

1.2.1 The Role of the Promise Type

The **promise type** is a user-defined class that determines the behavior of a coroutine. When the compiler transforms a coroutine into a state machine, it uses the promise type to manage:

1. How the coroutine is initialized.
2. How values are returned (or yielded) to the caller.
3. What happens on suspension and resumption.
4. Exception handling within the coroutine.

Every coroutine is associated with a `promise_type`, which lives inside the compiler-generated **coroutine frame** along with local variables and control information.

1. **Minimal Promise Type Example**

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

struct SimpleTask {
    struct promise_type {
        int value;

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

        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_value(int v) { value = v; }
```

```
    };

    std::coroutine_handle<promise_type> handle;

    explicit SimpleTask(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~SimpleTask() { if (handle) handle.destroy(); }

    int result() {
        handle.resume();
        return handle.promise().value;
    }
};

SimpleTask compute() {
    co_return 42;
}

int main() {
    auto task = compute();
    std::cout << "Result: " << task.result() << "\n";
}
```

Explanation:

- `get_return_object()` creates the coroutine handle accessible to the caller.
- `initial_suspend()` determines if the coroutine suspends immediately after creation.
- `final_suspend()` determines whether the coroutine suspends at the end.
- `return_value()` sets the final result in the promise.
- `unhandled_exception()` handles exceptions thrown inside the coroutine.

This example demonstrates how a promise type orchestrates the lifecycle and data of a coroutine.

2. **Suspension Points and the Promise**

   The promise type also defines how suspension behaves via the **awaiter objects** returned by `initial_suspend()` and `final_suspend()`.

   - `std::suspend_always` suspends the coroutine immediately.
   - `std::suspend_never` continues execution without suspension.

By customizing the promise, you can control whether the coroutine starts immediately, whether it resumes automatically, and how resources are cleaned up after completion.

1.2.2 `std::coroutine_traits` – Mapping Return Type to Promise Type

The compiler needs a way to determine **which promise type to use** for a given coroutine. This is where `std::coroutine_traits` comes in.
`std::coroutine_traits<R, Args...>` is a template that maps a coroutine's **return type** `R` and argument types to a corresponding **promise type**. By default, the compiler looks for `R::promise_type`. If you want to associate a custom promise type with a return type, you can specialize `std::coroutine_traits`.

1. **Default Behavior**

```
template<typename T>
struct Task {
    struct promise_type {
        T value;
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_value(T v) { value = v; }
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
};
```

   Here, the compiler automatically associates the coroutine `Task<int>` with `Task<int>::promise_type`.

2. **Custom `std::coroutine_traits`**

   You can also **specialize `std::coroutine_traits`** to associate an arbitrary promise type with a return type.

```
#include <coroutine>
```

```cpp
#include <iostream>

struct MyPromise {
    int result;
    MyPromise get_return_object() { return *this; }
    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void return_value(int v) { result = v; }
    void unhandled_exception() { std::terminate(); }
};

struct MyTask {
    int value;
};

// Specialize coroutine_traits for MyTask
namespace std {
    template<typename... Args>
    struct coroutine_traits<MyTask, Args...> {
        using promise_type = MyPromise;
    };
}

MyTask myCoroutine() {
    co_return 99;
}

int main() {
    auto task = myCoroutine();
    std::cout << "Result from MyTask: " << task.value << "\n";
}
```

This demonstrates advanced control over how return types link to promises. It allows you to separate the coroutine's **external interface** from its **internal promise logic**, which is useful for designing reusable coroutine libraries.

1.2.3 Key Points to Remember

1. **Promise type**: The heart of the coroutine; manages lifecycle, data, and exception handling.
2. **`get_return_object()`**: Provides the object that represents the coroutine externally.

3. **Suspension control**: `initial_suspend()` and `final_suspend()` determine when and how coroutines pause.

4. **`return_value()` or `return_void()`**: Sets the final result.

5. **`std::coroutine_traits`**: Maps a coroutine return type to a promise type, allowing compiler customization.

6. **Customizability**: By customizing promise types and `coroutine_traits`, you can implement tasks, generators, asynchronous operations, or even domain-specific coroutine frameworks.

1.2.4 Practical Example: Generator with Promise Type

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

template<typename T>
struct Generator {
    struct promise_type {
        T value;
        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 {}; }
        void unhandled_exception() { std::terminate(); }
        std::suspend_always yield_value(T v) noexcept {
            value = v;
            return {};
        }
        void return_void() {}
    };

    std::coroutine_handle<promise_type> handle;
    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    T next() {
        if (!handle.done()) handle.resume();
        return handle.promise().value;
    }
```

```
};

Generator<int> fibonacci(int limit) {
    int a = 0, b = 1;
    while (a <= limit) {
        co_yield a;
        auto next = a + b;
        a = b;
        b = next;
    }
}

int main() {
    auto gen = fibonacci(20);
    for (int i = 0; i < 8; ++i)
        std::cout << gen.next() << " ";
}
```

This shows how a **promise type** manages state, suspensions (`co_yield`), and sequence delivery in a fully working generator.

Summary:

- The **promise type** defines how a coroutine behaves internally.
- `std::coroutine_traits` links a coroutine's return type to its promise type.
- Together, they enable flexible, reusable, and high-performance coroutine abstractions.
- Mastery of these concepts is essential for building real-world coroutine libraries and understanding the internal mechanics of Modern C++ coroutines.

1.3 Handle Type – `std::coroutine_handle`

In the modern C++ coroutine model, the **coroutine handle** is the bridge between the **caller** and the **compiler-generated coroutine frame**. Understanding `std::coroutine_handle` is essential for controlling coroutine execution, resuming suspended coroutines, and managing their lifetimes safely and efficiently.

1.3.1 What is `std::coroutine_handle`?

`std::coroutine_handle` is a **lightweight, type-erased handle** that represents a coroutine frame created by the compiler. It is effectively a pointer to the coroutine's state, including local variables, promise object, and control information.

- A handle does **not own the coroutine frame**. You must destroy the coroutine manually unless a higher-level abstraction does it for you.
- Handles allow **resuming, destroying, or querying** coroutine state from outside the coroutine itself.
- There are two variants:
  - **Type-erased:** `std::coroutine_handle<>`
  - **Typed:** `std::coroutine_handle<promise_type>`

1.3.2 Basic Operations

A coroutine handle exposes several fundamental operations:

Operation	Description
`resume()`	Resumes execution of the coroutine until the next suspension point.
`done()`	Returns `true` if the coroutine has finished execution.
`destroy()`	Destroys the coroutine frame and frees resources.
`from_promise(promi`	Obtains a handle from a promise object (used internally by `get_return_object()`).

1. **Creating a Handle from a Promise**

   Whenever you write a coroutine, the compiler generates a **promise object**. The handle can be obtained using:

   ```
   std::coroutine_handle<promise_type>::from_promise(promise)
   ```

   This is typically done in `get_return_object()` of the promise type.

2. **Minimal Example Using `std::coroutine_handle`**

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

struct Task {
    struct promise_type {
        int value;

        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(int v) { value = v; }
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }

    int result() {
        handle.resume();
        return handle.promise().value;
    }
};

Task myCoroutine() {
    std::cout << "Coroutine started.\n";
    co_return 42;
}
```

```
int main() {
    Task t = myCoroutine();
    std::cout << "Result: " << t.result() << "\n";
}
```

Explanation:

- `handle.resume()` starts or continues execution.
- `handle.destroy()` frees the coroutine frame to avoid memory leaks.
- `handle.promise()` allows access to the associated promise object for reading or writing state.

1.3.3 Typed vs. Untyped Handles

1. **Typed Handles**

   `std::coroutine_handle<promise_type>` provides **direct access to the promise**:

   ```
   auto handle = std::coroutine_handle<Task::promise_type>::from_promise(promise);
   handle.promise().value = 100;
   ```

   This is useful when you want to read or manipulate the coroutine state safely from the outside.

2. **Untyped Handles**

   `std::coroutine_handle<>` is type-erased, allowing storage of handles without knowing the promise type:

   ```
   std::coroutine_handle<> h = std::coroutine_handle<>::from_address(handle_address);
   if (!h.done()) h.resume();
   ```

   Untyped handles are typically used in scheduler frameworks where the exact promise type is unknown at runtime.

1.3.4 Lifetime Management

A crucial responsibility when using coroutine handles is **resource management**:

- **Creating** a coroutine does not automatically start it if `initial_suspend()` returns `std::suspend_always`.

- **Resuming** is required to execute the coroutine.
- **Destroying** the handle releases memory for the coroutine frame.

Example:

```
auto h = myCoroutine().handle;
while (!h.done()) {
    h.resume();
}
h.destroy(); // Must manually destroy to avoid memory leak
```

Many higher-level coroutine types (like `std::generator` or custom `Task` types) manage destruction automatically to prevent leaks.

1.3.5 Practical Example: Coroutine Scheduler

Handles are the backbone of coroutine-based schedulers:

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

struct Task {
    struct promise_type {
        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_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

std::vector<std::coroutine_handle<>> scheduler;

Task worker(int id) {
```

```
    std::cout << "Worker " << id << " started\n";
    co_await std::suspend_always{};
    std::cout << "Worker " << id << " resumed\n";
}

int main() {
    scheduler.push_back(worker(1).handle);
    scheduler.push_back(worker(2).handle);

    // Resume all coroutines
    for (auto h : scheduler) {
        if (!h.done()) h.resume();
    }
}
```

Explanation:

- Each `worker` coroutine returns a handle to its caller.
- The scheduler stores handles and resumes them later.
- This pattern is foundational for event-driven or cooperative multitasking systems.

1.3.6 Key Points

1. `std::coroutine_handle` is the bridge to the compiler-generated coroutine frame.
2. Handles allow **resume**, **destroy**, and **promise access**.
3. Typed handles provide safe access to the promise object; untyped handles allow general storage in schedulers.
4. Proper **lifetime management** is essential to avoid memory leaks.
5. Handles make it possible to implement **custom schedulers, task queues, and generator pipelines** efficiently in Modern C++.

Summary:
`std::coroutine_handle` gives developers explicit control over coroutine execution and lifecycle. Mastering handles is critical for building high-performance coroutine frameworks, implementing generators, and designing cooperative multitasking systems in C++23.

1.4 Practical Example – A Simple Coroutine Printing a Sequence of Values with Suspension and Resumption

To consolidate the concepts of `co_yield`, **promise type**, and `std::coroutine_handle`, this section presents a **practical coroutine example** that prints a sequence of values incrementally, demonstrating **suspension and resumption**. This example showcases the internal mechanics of coroutines in a way that is easy to understand yet reflects real-world usage.

1.4.1 The Concept

We aim to create a **sequence generator coroutine** that:

1. Produces values one by one using `co_yield`.
2. Suspends after yielding each value.
3. Resumes on demand using `std::coroutine_handle::resume()`.
4. Cleans up resources properly at the end.

This pattern is representative of **generators** in C++23 and is widely applicable in asynchronous pipelines, lazy computations, and event-driven programming.

1.4.2 Implementation

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

// Generator structure
template<typename T>
struct Generator {
    struct promise_type {
        T current_value;

        // Returns the Generator object to the caller
        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 {}; }
        void unhandled_exception() { std::terminate(); }

        // Yielding a value
        std::suspend_always yield_value(T value) noexcept {
            current_value = value;
            return {};
        }

        void return_void() {}
    };

    std::coroutine_handle<promise_type> handle;

    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    // Retrieve next value
    T next() {
        if (!handle.done()) handle.resume();
        return handle.promise().current_value;
    }

    bool done() const { return handle.done(); }
};

// Coroutine producing a sequence of integers
Generator<int> generateSequence(int start, int end) {
    for (int i = start; i <= end; ++i)
        co_yield i;
}

int main() {
    auto seq = generateSequence(1, 5);

    std::cout << "Printing sequence values with suspension and resumption:\n";

    while (!seq.done()) {
        int value = seq.next();
        std::cout << "Value: " << value << "\n";
    }
```

```
}
```

1.4.3 Explanation

1. **Coroutine Lifecycle**

   (a) **Creation:**

   - When `generateSequence` is called, the compiler generates a **coroutine frame** including the promise object.
   - `initial_suspend()` suspends execution immediately, giving control to the caller.

   (b) **Yielding Values:**

   - Each `co_yield i` sets `current_value` in the promise and suspends.
   - The coroutine can be resumed later using `handle.resume()`.

   (c) **Resumption:**

   - The `next()` method resumes the coroutine.
   - Execution continues from the last suspension point until the next `co_yield` or end of the coroutine.

   (d) **Completion:**

   - After the final value is yielded, `final_suspend()` suspends the coroutine, allowing cleanup.
   - When the `Generator` object is destroyed, `handle.destroy()` releases the coroutine frame.

2. **Output**

```
Printing sequence values with suspension and resumption:
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
```

1.4.4 Practical Insights

1. **Lazy Evaluation:**
   Values are generated **on demand**. The entire sequence is not computed upfront, saving memory and computation for large sequences.

2. **Explicit Control:**
   Using `handle.resume()` gives the caller fine-grained control over when the coroutine executes, making it suitable for event-driven or asynchronous workflows.

3. **Safe Resource Management:**

   - The coroutine frame is automatically destroyed in the destructor.
   - Using `std::coroutine_handle` provides explicit lifetime management when necessary, avoiding leaks.

4. **Extensibility:**

   - This simple generator can be extended to **asynchronous tasks**, e.g., integrating with `co_await` for I/O operations.
   - Multiple coroutines can be stored in a **scheduler vector** and resumed in a cooperative multitasking fashion.

1.4.5 Real-World Use Cases

- **Event-driven pipelines:** Processing streams of data lazily.
- **Asynchronous I/O:** Yielding results as data becomes available.
- **Reactive systems:** Updating UI or network state incrementally.
- **Simulation engines:** Controlling stepwise execution of tasks or events.

Summary:
This practical example demonstrates the **core mechanics of coroutines** in C++23:

- `co_yield` for producing values lazily.
- `std::coroutine_handle` for suspending, resuming, and managing coroutine lifetime.

- Promise type storing intermediate state.

Understanding this pattern is essential before moving to **advanced coroutine patterns** like asynchronous tasks, parallel pipelines, and real-time schedulers.

Chapter 2

Coroutine Life Cycle

2.1 Initial Suspend and Final Suspend

Understanding `initial_suspend` and `final_suspend` is critical for mastering the **lifecycle of coroutines** in Modern C++23. These two suspension points dictate when a coroutine starts, when it suspends at the end, and how control flows between the coroutine and the caller.

2.1.1 `initial_suspend` – Controlling the Start of a Coroutine

The `initial_suspend()` function is a member of the coroutine's **promise type**. It determines whether a coroutine **starts executing immediately** or **suspends before running any body code**.

- **Key Points:**
  - Return type must satisfy the **awaitable concept**. Commonly used types:
    - `std::suspend_always` – always suspends at the start.
    - `std::suspend_never` – starts executing immediately.
  - This allows **fine-grained control** over when the coroutine begins, which is particularly useful in **lazy evaluation**, **task scheduling**, or **manual control of execution**.
- **Example: Coroutine with Initial Suspend**

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

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

        // Suspends immediately after creation
        std::suspend_always initial_suspend() noexcept {
            std::cout << "initial_suspend called: coroutine is suspended\n";
            return {};
        }

        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task myCoroutine() {
    std::cout << "Coroutine body executed\n";
    co_return;
}

int main() {
    auto t = myCoroutine();
    std::cout << "Resuming coroutine...\n";
    t.handle.resume(); // Execution starts here
}
```

Output:

```
initial_suspend called: coroutine is suspended
Resuming coroutine...
Coroutine body executed
```

Explanation:

- `initial_suspend` suspends the coroutine immediately after creation.
- Execution begins only when `handle.resume()` is called.
- This mechanism allows **manual scheduling** or integration with **custom task queues**.

2.1.2 `final_suspend` – Controlling Suspension at the End

The `final_suspend()` function determines what happens **when a coroutine reaches the end of its body** (after `co_return` or normal completion).

Key Points:

- Return type must also satisfy the **awaitable concept**.
- Typical return types:
  - `std::suspend_always` – suspends at the end, allowing the caller or scheduler to clean up manually.
  - `std::suspend_never` – coroutine completes immediately and the frame can be destroyed automatically.
- Provides a hook for **final actions**, such as notifying a scheduler, releasing resources, or chaining coroutines.

Example: Coroutine with Final Suspend

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

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

        std::suspend_never initial_suspend() noexcept { return {}; }

        // Suspends at the end
        std::suspend_always final_suspend() noexcept {
            std::cout << "final_suspend called: coroutine is suspended at the end\n";
```

```
            return {};
        }

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

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task myCoroutine() {
    std::cout << "Coroutine body executed\n";
    co_return;
}

int main() {
    auto t = myCoroutine();
    std::cout << "Before final resume\n";
    t.handle.resume(); // Executes coroutine body
    std::cout << "After final_suspend, manually destroying...\n";
}
```

Output:

```
Coroutine body executed
final_suspend called: coroutine is suspended at the end
Before final resume
After final_suspend, manually destroying...
```

Explanation:

- `final_suspend` suspends the coroutine frame at the end.
- This allows **safe access to the coroutine handle** for cleanup, chaining, or scheduling subsequent coroutines.
- Without suspension, the coroutine frame may be destroyed immediately, preventing external access.

2.1.3 Combining Initial and Final Suspend

By controlling both suspension points, you can create **powerful coroutine patterns**:

Pattern	Behavior
`initial_suspend = suspend_always`, `final_suspend = suspend_always`	Fully manual control: coroutine starts and ends suspended.
`initial_suspend = suspend_never`, `final_suspend = suspend_always`	Starts immediately, can perform cleanup or notify at end.
`initial_suspend = suspend_always`, `final_suspend = suspend_never`	Lazy start, ends automatically.
`initial_suspend = suspend_never`, `final_suspend = suspend_never`	Immediate execution and automatic cleanup.

This flexibility is crucial for designing **custom schedulers**, **generators**, or **asynchronous task frameworks**.

2.1.4 Practical Insights

1. **Initial suspend is for lazy evaluation or scheduling** – you can decide exactly when the coroutine executes.
2. **Final suspend is for cleanup or chaining** – allows external code to observe the completion and safely destroy or resume other coroutines.
3. **Memory management** – final_suspend gives you control over when the coroutine frame is destroyed. Neglecting this can lead to resource leaks.
4. **Advanced scenarios** – combining suspension points with `co_await` enables **asynchronous pipelines**, cooperative multitasking, and event-driven designs.

Summary:

- `initial_suspend()` controls when the coroutine begins execution.
- `final_suspend()` controls what happens at the end, including suspension for cleanup or chaining.

- Mastering these suspension points allows developers to design **flexible, efficient, and safe coroutine-based systems** in Modern C++23.

2.2 Resumption and Destruction

After understanding **initial_suspend** and **final_suspend**, the next essential aspects of the **coroutine lifecycle** are **resumption** and **destruction**. These determine how coroutines execute, suspend, and release resources safely in Modern C++23.

2.2.1 Resumption – Controlling Coroutine Execution

Resumption refers to the process of continuing a coroutine from its **last suspension point**. Each coroutine can be suspended multiple times using:

- `co_await`
- `co_yield`
- `initial_suspend`

Resuming a coroutine is done via **`std::coroutine_handle::resume()`**.

Key Points:

1. Resumption continues execution **from the last suspended point**.
2. Multiple calls to `resume()` can iterate over a coroutine's body until it completes.
3. If the coroutine is **already done**, `resume()` has no effect.

Example: Resuming a Simple Coroutine

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

struct Task {
    struct promise_type {
        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_void() {}
        void unhandled_exception() { std::terminate(); }
```

```
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task demoCoroutine() {
    std::cout << "Step 1\n";
    co_await std::suspend_always{};
    std::cout << "Step 2\n";
    co_await std::suspend_always{};
    std::cout << "Step 3\n";
}

int main() {
    Task t = demoCoroutine();

    std::cout << "Resuming first time...\n";
    t.handle.resume();  // Executes "Step 1" and suspends

    std::cout << "Resuming second time...\n";
    t.handle.resume();  // Executes "Step 2" and suspends

    std::cout << "Resuming third time...\n";
    t.handle.resume();  // Executes "Step 3" and reaches final_suspend
}
```

Output:

```
Resuming first time...
Step 1
Resuming second time...
Step 2
Resuming third time...
Step 3
```

Explanation:

- `co_await std::suspend_always{}` suspends the coroutine at each step.
- `handle.resume()` resumes execution **from the exact point of suspension**.

- Resumption can be controlled manually, enabling **cooperative multitasking** or **lazy computation**.

2.2.2 Destruction – Cleaning Up Coroutine Frames

Every coroutine allocates a **coroutine frame** on the heap to store:

- Local variables
- Promise object
- Control information

Proper destruction is crucial to avoid memory leaks. Destruction happens via `std::coroutine_handle::destroy()`.

Key Points:

1. **Destroying a coroutine handle frees all memory** associated with the coroutine frame.
2. **Do not resume after destruction**; it leads to undefined behavior.
3. High-level abstractions like **Task** or **Generator** can automate destruction in destructors.

Example: Manual Destruction

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

struct Task {
    struct promise_type {
        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_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
```

```
    ~Task() {
        if (handle) {
            std::cout << "Destroying coroutine...\n";
            handle.destroy();
        }
    }
};

Task demoCoroutine() {
    std::cout << "Coroutine running...\n";
    co_return;
}

int main() {
    Task t = demoCoroutine();
    t.handle.resume();  // Executes coroutine
}
```

Output:

```
Coroutine running...
Destroying coroutine...
```

Explanation:

- The destructor automatically calls `handle.destroy()` to release resources.
- This ensures safe memory management without manual intervention in typical usage.

2.2.3 Combining Resumption and Destruction

A typical coroutine workflow:

1. **Creation** – Coroutine object is created; initial_suspend controls whether it starts.
2. **Resumption** – Caller uses `resume()` to advance the coroutine.
3. **Suspension** – Coroutine can yield or await multiple times.
4. **Completion** – Coroutine reaches `final_suspend` and suspends at the end.
5. **Destruction** – `handle.destroy()` frees memory.

This workflow allows **manual or automated control** of execution while ensuring **no resource leaks**.

2.2.4 Practical Insight

- Resumption enables **fine-grained execution control**, which is essential for **cooperative multitasking**.
- Destruction guarantees **memory safety** in long-lived or frequently suspended coroutines.
- Together, resumption and destruction form the **core lifecycle control** of coroutines in Modern C++23.

Summary:

- **Resumption** (`resume`) continues a coroutine from the last suspension point.
- **Destruction** (`destroy`) frees the coroutine frame and all associated resources.
- Proper understanding of resumption and destruction is critical for designing **efficient, safe, and predictable coroutine-based systems**.

2.3 Illustration – Life Cycle Diagram

To fully understand **coroutine behavior in C++23**, a visual representation of the **coroutine life cycle** is invaluable. This section provides a **diagrammatic illustration** along with a step-by-step explanation of how a coroutine transitions through creation, suspension, resumption, and destruction.

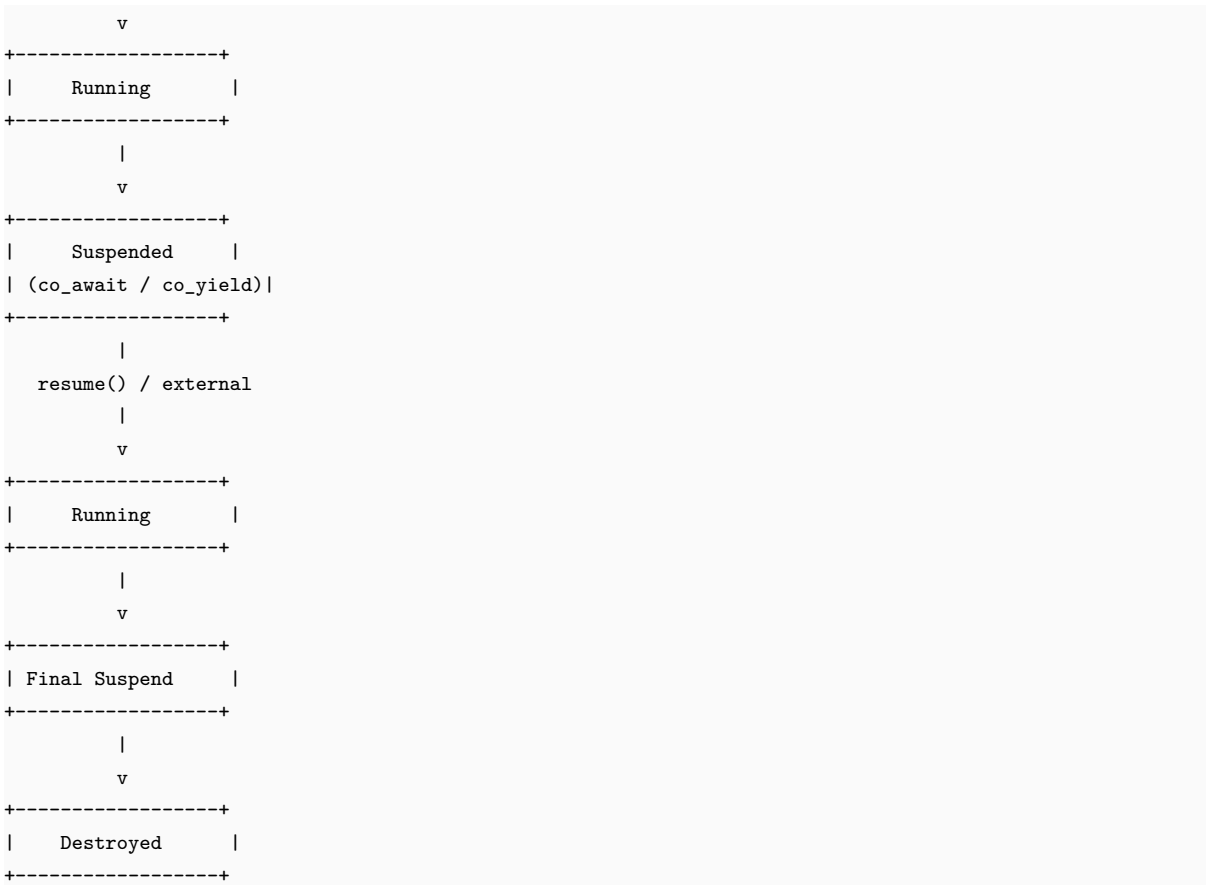
2.3.1 Coroutine Life Cycle States

A coroutine progresses through the following **key states**:

1. **Created:** Coroutine object is created; memory for the **coroutine frame** is allocated.
2. **Initial Suspension:** `initial_suspend()` determines if the coroutine suspends immediately or starts executing.
3. **Running:** Execution begins, local variables are initialized, and the body runs until the next suspension.
4. **Suspended:** Coroutine is paused at a `co_await`, `co_yield`, or `initial_suspend`. Control returns to the caller.
5. **Resumed:** Coroutine execution continues from the suspension point when `resume()` is called.
6. **Final Suspension:** `final_suspend()` is invoked when the coroutine finishes execution, allowing cleanup or chaining.
7. **Destroyed:** `destroy()` is called to release the coroutine frame and associated resources.

2.3.2 Life Cycle Diagram (Text Representation)

```
+------------------+
|  Coroutine Created  |
+------------------+
          |
          v
+------------------+
| Initial Suspend   |
| (co_await / suspend)|
+------------------+
          |
```



2.3.3 Step-by-Step Explanation

1. **Created:**
   - The compiler allocates a **coroutine frame** and constructs the **promise object**.
2. **Initial Suspend:**
   - `initial_suspend()` decides if the coroutine starts immediately (`suspend_never`) or suspends (`suspend_always`).
   - Suspended coroutines require `handle.resume()` to start execution.
3. **Running:**

- The coroutine executes the body, including local variable initialization.
- Execution proceeds until a suspension point (`co_await` or `co_yield`) or completion.

4. **Suspended:**
   - Execution pauses, saving the **stack frame, local variables, and promise state**.
   - The caller regains control and can decide **when to resume**.
5. **Resumed:**
   - `resume()` continues execution **from the last suspension point**.
   - Multiple resumption cycles are possible in **generator-like or cooperative multitasking coroutines**.
6. **Final Suspend:**
   - When the coroutine completes, `final_suspend()` is called.
   - This allows safe access to the handle for cleanup, chaining, or scheduler notifications.
7. **Destroyed:**
   - `destroy()` frees all memory allocated for the coroutine frame.
   - After destruction, the handle becomes invalid.

2.3.4 Practical Example Illustrating Life Cycle

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

struct Task {
    struct promise_type {
        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_void() {}
        void unhandled_exception() { std::terminate(); }
```

```
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task demoCoroutine() {
    std::cout << "Running: Step 1\n";
    co_await std::suspend_always{};
    std::cout << "Running: Step 2\n";
    co_await std::suspend_always{};
    std::cout << "Running: Step 3\n";
}

int main() {
    Task t = demoCoroutine();      // Created + Initial Suspend
    t.handle.resume();             // Running -> Suspended (Step 1)
    t.handle.resume();             // Running -> Suspended (Step 2)
    t.handle.resume();             // Running -> Final Suspend (Step 3)
}
```

Explanation:

- **Created:** `demoCoroutine()` allocates the frame.
- **Initial Suspend:** Coroutine suspends before running Step 1.
- **Running & Suspended:** Step 1 executes, then suspends; Step 2 executes on the next resume.
- **Final Suspend:** Step 3 executes and suspends at the end.
- **Destroyed:** Destructor calls `handle.destroy()`.

2.3.5 Insights from the Diagram

1. **Manual Control:** Using `resume()` and suspension points, you control **exact execution flow**.
2. **Resource Safety:** `final_suspend` + `destroy()` ensures **safe memory management**.
3. **Lifecycle Awareness:** Understanding each state helps in **debugging, scheduler design, and optimization**.

4. **Real-World Patterns:** This model underpins **generators, async I/O pipelines, and cooperative multitasking frameworks**.

Summary:

- The **coroutine life cycle** progresses from creation, initial suspension, running, suspension, final suspension, to destruction.
- The diagram and practical example illustrate **how and when execution is paused, resumed, and cleaned up**.
- Mastering this lifecycle is essential for designing **robust, high-performance coroutine-based systems** in Modern C++23.

2.4 Practical Example – Coroutine Managing Multiple Tasks in a Defined Sequence

After understanding the **lifecycle of a single coroutine**, the next step is to demonstrate **coordinating multiple tasks** using coroutines. This example illustrates **sequencing, suspension, and resumption** across several tasks while maintaining control over execution order.

2.4.1 Concept

The goal is to create a coroutine-based system where:

1. Multiple tasks are defined as coroutines.
2. Tasks execute **in a defined order**, yielding control back to a **task manager** between steps.
3. The manager resumes each task in sequence until all are complete.
4. Proper resource management ensures **no memory leaks**.

This pattern is common in **asynchronous pipelines, cooperative multitasking, and event-driven programming**.

2.4.2 Implementation

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

struct Task {
    struct promise_type {
        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_void() {}
        void unhandled_exception() { std::terminate(); }
    };
```

```
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    bool done() const { return handle.done(); }
    void resume() { handle.resume(); }
};

// Task coroutines
Task task1() {
    std::cout << "Task 1 - Step 1\n";
    co_await std::suspend_always{};
    std::cout << "Task 1 - Step 2\n";
    co_await std::suspend_always{};
    std::cout << "Task 1 - Step 3\n";
}

Task task2() {
    std::cout << "Task 2 - Step 1\n";
    co_await std::suspend_always{};
    std::cout << "Task 2 - Step 2\n";
}

Task task3() {
    std::cout << "Task 3 - Step 1\n";
    co_await std::suspend_always{};
    std::cout << "Task 3 - Step 2\n";
    co_await std::suspend_always{};
    std::cout << "Task 3 - Step 3\n";
    co_await std::suspend_always{};
    std::cout << "Task 3 - Step 4\n";
}

int main() {
    // Create tasks
    std::vector<Task> tasks;
    tasks.push_back(task1());
    tasks.push_back(task2());
    tasks.push_back(task3());

    bool tasksRemaining = true;

    std::cout << "Managing multiple tasks in sequence:\n";
```

```
    // Resume tasks in sequence until all are done
    while (tasksRemaining) {
        tasksRemaining = false;
        for (auto& t : tasks) {
            if (!t.done()) {
                t.resume();
                tasksRemaining = true;
            }
        }
        std::cout << "----- End of round -----\n";
    }
}
```

2.4.3 Explanation

1. **Task Definition**
   - Each task is a coroutine returning a `Task` object.
   - `co_await std::suspend_always{}` suspends execution at key points, allowing the **manager to resume other tasks**.
   - Steps within each task represent discrete units of work.
2. **Task Manager**
   - A **simple loop** iterates over all tasks.
   - For each task that is not done, `resume()` is called to advance it to the next suspension point.
   - This approach ensures **round-robin execution**, maintaining the defined sequence while allowing tasks to yield control cooperatively.
3. **Life Cycle Interaction**
   - `initial_suspend` suspends each coroutine after creation.
   - `resume` starts or continues execution until the next suspension point.
   - `final_suspend` ensures that the coroutine remains accessible for cleanup or chaining before destruction.
   - Destructor calls `destroy()` automatically release memory when tasks go out of scope.

2.4.4 Output

```
Managing multiple tasks in sequence:
Task 1 - Step 1
Task 2 - Step 1
Task 3 - Step 1
----- End of round -----
Task 1 - Step 2
Task 2 - Step 2
Task 3 - Step 2
----- End of round -----
Task 1 - Step 3
Task 3 - Step 3
----- End of round -----
Task 3 - Step 4
----- End of round -----
```

Explanation:

- Tasks progress in a **cooperative round-robin manner**.
- Each suspension point allows the **manager loop** to resume other tasks.
- Execution order is **predictable and controlled**, suitable for **task orchestration**.

2.4.5 Practical Insights

1. **Cooperative Multitasking:**
    - This pattern avoids **preemptive threading**, reducing context-switching overhead.
    - Suitable for embedded systems or real-time applications where **deterministic execution** is important.
2. **Memory Efficiency:**
    - Each coroutine keeps its state **on the heap** in the frame.
    - No stack copying or complex context switching is required.
3. **Extensibility:**

- The manager can integrate **priority scheduling**, **asynchronous I/O**, or **event-driven triggers**.
- Can be combined with `co_await` for **true asynchronous tasks**.

4. **Debugging and Visualization:**
   - Using log statements at each suspension point helps **trace execution order**.
   - Ideal for teaching, debugging, or visualizing complex task flows.

Summary:

- Coroutines can manage **multiple tasks cooperatively** using suspension and resumption.
- A **round-robin manager** can resume tasks in sequence, enabling **predictable multi-task orchestration**.
- Proper lifecycle management (`initial_suspend`, `final_suspend`, `destroy`) ensures **safe and efficient execution**.
- This example lays the foundation for **advanced coroutine-based task schedulers** and asynchronous pipelines in C++23.

Chapter 3

Awaitables and Awaiters

3.1 Awaitable Concept

The **awaitable concept** is fundamental to understanding **asynchronous programming with coroutines** in Modern C++23. An **awaitable** is any type that can be used with the `co_await` keyword, enabling **suspension, resumption, and result retrieval** in a coroutine.

3.1.1 Definition of Awaitable

A type `T` is considered **awaitable** if it satisfies the following interface, either directly or via **customization**:

1. `T::await_ready()` – Determines whether the coroutine should **suspend immediately**.
    - Returns `true` → coroutine continues without suspension.
    - Returns `false` → coroutine suspends and `await_suspend()` is called.
2. `T::await_suspend(std::coroutine_handle<>)` – Called when the coroutine suspends.
    - Can schedule the coroutine for resumption.
    - Can decide whether to resume immediately, later, or never.
3. `T::await_resume()` – Called when the coroutine **resumes**.

- Returns the result of the `co_await` expression.

This interface is what allows `co_await` to integrate arbitrary asynchronous or deferred operations into coroutines.

3.1.2 Built-in Awaitables in C++23

Modern C++ provides several **built-in awaitables**, including:

- `std::suspend_always` – Always suspends when awaited.
- `std::suspend_never` – Never suspends; coroutine continues immediately.
- `std::future` / `std::task`-like wrappers in custom libraries – Can integrate asynchronous computation results.

These awaitables make it possible to **control suspension behavior** without manually implementing coroutine frames.

3.1.3 Example: Custom Awaitable

This example demonstrates a **custom awaitable** that simulates a simple asynchronous operation:

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

struct SimpleAwaitable {
    int value;

    SimpleAwaitable(int v) : value(v) {}

    bool await_ready() const noexcept {
        // Do not suspend if value is zero
        return value == 0;
    }

    void await_suspend(std::coroutine_handle<> handle) const {
        // Simulate asynchronous work with a separate thread
        std::thread([handle, this]() {
```

```
            std::this_thread::sleep_for(std::chrono::milliseconds(500));
            std::cout << "Asynchronous operation complete: " << value << "\n";
            handle.resume(); // Resume the coroutine after work is done
        }).detach();
    }

    int await_resume() const noexcept {
        return value * 2; // Return a transformed result
    }
};

struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task myCoroutine() {
    int result = co_await SimpleAwaitable(10);
    std::cout << "Result after await_resume: " << result << "\n";
}

int main() {
    myCoroutine();
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Wait for async operation
}
```

Explanation:

1. `await_ready()`
    - Returns `false` if `value != 0`, causing the coroutine to suspend.
2. `await_suspend()`

- Runs an asynchronous operation in a separate thread.
- Calls `handle.resume()` when the operation completes, resuming the coroutine.

3. `await_resume()`

   - Returns a result to the coroutine (`value * 2` in this case).

4. **Coroutine Execution Flow:**

```
myCoroutine() called
await_ready() -> false => coroutine suspends
await_suspend() -> starts async operation
Async operation completes -> handle.resume()
await_resume() -> returns value * 2
Coroutine resumes and prints result
```

3.1.4 Practical Insights

1. **Flexible Suspension Control:**

   - Awaitables allow **fine-grained control over suspension**, integrating coroutines with async APIs, timers, or I/O operations.

2. **Custom Asynchronous Operations:**

   - Any asynchronous or deferred computation can be wrapped in a **custom awaitable**, making it compatible with `co_await`.

3. **Integration with Executors and Schedulers:**

   - `await_suspend` can enqueue the coroutine handle to a **scheduler or thread pool**, enabling **cooperative multitasking** or **asynchronous pipelines**.

4. **Seamless Result Handling:**

   - `await_resume()` allows returning computed results directly to the coroutine, maintaining **clear and expressive code**.

Summary:

- An **awaitable** is any type that implements the `await_ready`, `await_suspend`, and `await_resume` interface.
- Awaitables enable **coroutine suspension, asynchronous computation, and result retrieval**.
- Custom awaitables allow coroutines to **integrate seamlessly with I/O, timers, threads, and task schedulers** in Modern C++23.
- Understanding the awaitable concept is essential before exploring **awaiters, chaining, and asynchronous pipelines** in coroutines.

3.2 Custom Awaiters

While **awaitables** define what can be used with `co_await`, **awaiters** specify **how a coroutine is suspended and resumed**. A **custom awaiter** gives fine-grained control over **execution flow, suspension behavior, and result delivery**.
In C++23, custom awaiters are especially useful for **asynchronous operations, task scheduling, and cooperative multitasking**.

3.2.1 Anatomy of an Awaiter

A **custom awaiter** is any object that supports the following **three methods**:

1. `await_ready()`
   - Returns `true` if the coroutine **does not need to suspend**.
   - Returns `false` to suspend the coroutine and invoke `await_suspend()`.
2. `await_suspend(std::coroutine_handle<>)`
   - Called when the coroutine suspends.
   - Receives the coroutine handle, allowing scheduling, chaining, or external control.
3. `await_resume()`
   - Called when the coroutine resumes.
   - Returns a value (optional) to the coroutine, completing the `co_await` expression.

   Conceptually: **Awaitable** $\rightarrow$ **Awaiter**. `co_await` converts the awaitable to an **awaiter**, which is then used to control suspension.

3.2.2 Example: Timer Awaiter

This example demonstrates a **custom awaiter** that suspends a coroutine for a specified duration, simulating asynchronous delay:

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

// Custom awaiter: suspends coroutine for a specified milliseconds
struct TimerAwaiter {
    int ms;

    TimerAwaiter(int duration) : ms(duration) {}

    bool await_ready() const noexcept {
        return ms <= 0; // No suspension if duration is zero or negative
    }

    void await_suspend(std::coroutine_handle<> handle) const {
        // Launch a thread to resume the coroutine after delay
        std::thread([handle, this]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(ms));
            handle.resume();
        }).detach();
    }

    void await_resume() const noexcept {
        // No value to return in this simple timer
    }
};

// Simple task coroutine using custom awaiter
struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};
```

```
Task myCoroutine() {
    std::cout << "Step 1: Start coroutine\n";
    co_await TimerAwaiter(500); // Suspend for 500ms
    std::cout << "Step 2: After 500ms\n";
    co_await TimerAwaiter(1000); // Suspend for 1000ms
    std::cout << "Step 3: After 1000ms\n";
}

int main() {
    myCoroutine();
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Wait for async operations
}
```

3.2.3 Explanation

1. **`await_ready()`**
    - Returns `true` for zero or negative durations; coroutine continues immediately.
    - Returns `false` for positive durations, suspending the coroutine.
2. **`await_suspend()`**
    - Launches a **background thread** to simulate asynchronous work.
    - Calls `handle.resume()` when the delay is over, resuming the coroutine.
3. **`await_resume()`**
    - Returns nothing here, but could return a value (e.g., elapsed time or a result from I/O).
4. **Execution Flow:**

```
myCoroutine() called
Step 1 printed
await_suspend() launches timer thread -> coroutine suspended
After 500ms -> handle.resume() -> Step 2 printed
await_suspend() launches timer thread -> coroutine suspended
After 1000ms -> handle.resume() -> Step 3 printed
Coroutine completes
```

3.2.4 Benefits of Custom Awaiters

1. **Precise Suspension Control**
   - Decide **when and how** the coroutine suspends and resumes.
2. **Integration with Asynchronous Systems**
   - Awaiters can interface with **thread pools, I/O completion ports, timers, or event loops**.
3. **Custom Result Handling**
   - `await_resume()` allows returning computation results directly to the coroutine.
4. **Composable Coroutines**
   - Multiple awaiters can be chained to create **complex asynchronous pipelines**.
5. **Efficiency**
   - Unlike threads, coroutines with awaiters **avoid stack copying** and **minimize context-switch overhead**.

3.2.5 Advanced Tips

- **Cooperative Multitasking:** Awaiters are ideal for **round-robin task scheduling**.
- **Resource Safety:** Ensure the coroutine frame remains valid until `await_suspend` completes and `handle.resume()` is called.
- **Non-blocking Design:** Avoid blocking inside `await_suspend`; delegate work to **background threads, I/O, or scheduler callbacks**.
- **Composable Awaiters:** Combine multiple awaiters using `co_await` in sequence or in parallel to implement **asynchronous workflows**.

Summary:

- A **custom awaiter** controls **coroutine suspension, resumption, and result delivery**.

- Methods: `await_ready()`, `await_suspend()`, `await_resume()`.
- They are the **building blocks of advanced asynchronous coroutines** in C++23.
- Custom awaiters allow integration with **timers, I/O, schedulers, and cooperative multitasking systems**.
- Mastering custom awaiters is essential for **high-performance, predictable, and composable coroutine systems**.

3.3 Await Transformations

In Modern C++23, **await transformations** provide a mechanism to **customize how `co_await` interacts with awaitable objects**. They allow the **promise type** of a coroutine to **intercept and modify the awaiter** before suspension, enabling advanced control over **asynchronous behavior, scheduling, and chaining**.

3.3.1 Concept of Await Transformation

When a coroutine executes a `co_await expr`, the compiler performs the following steps:

1. Converts `expr` into an **awaiter object** (using `operator co_await()` if defined).
2. Checks if the coroutine's **promise type** provides an `await_transform` member function.
3. If `await_transform` exists, it is called with `expr`, and the **returned object** becomes the **actual awaiter**.

This allows the coroutine to **customize the behavior of `co_await` globally** for all awaitables inside that coroutine.

Syntax

```
struct promise_type {
    template<typename Awaitable>
    auto await_transform(Awaitable&& awaitable) {
        // Transform or wrap the awaitable here
        return std::forward<Awaitable>(awaitable);
    }
};
```

- The return value must be **awaitable**.
- Can wrap, decorate, or modify any awaitable passed to `co_await`.

3.3.2 Example: Logging Await Transform

This example shows how `await_transform` can **wrap every awaited expression with logging**:

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

struct Task {
    struct promise_type {
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }

        // Custom await_transform to wrap awaitables with logging
        template<typename Awaitable>
        auto await_transform(Awaitable&& awaitable) {
            struct LoggingAwaiter {
                Awaitable awaitable;
                LoggingAwaiter(Awaitable a) : awaitable(a) {}

                bool await_ready() {
                    std::cout << "[Logging] Checking ready...\n";
                    return awaitable.await_ready();
                }

                void await_suspend(std::coroutine_handle<> h) {
                    std::cout << "[Logging] Coroutine suspending...\n";
                    awaitable.await_suspend(h);
                }

                auto await_resume() {
                    auto result = awaitable.await_resume();
                    std::cout << "[Logging] Coroutine resumed.\n";
                    return result;
                }
            };
            return LoggingAwaiter{std::forward<Awaitable>(awaitable)};
        }
    };
```

```
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

// Simple awaitable
struct TimerAwaiter {
    int ms;
    TimerAwaiter(int duration) : ms(duration) {}
    bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) const {
        std::thread([h, this](){
            std::this_thread::sleep_for(std::chrono::milliseconds(ms));
            h.resume();
        }).detach();
    }
    void await_resume() const noexcept {}
};

Task myCoroutine() {
    std::cout << "Start coroutine\n";
    co_await TimerAwaiter(500);  // Wrapped with logging automatically
    std::cout << "After 500ms\n";
    co_await TimerAwaiter(300);  // Also wrapped
    std::cout << "After 300ms\n";
}

int main() {
    myCoroutine();
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Wait for async operations
}
```

3.3.3 Explanation

1. **`await_transform()`** intercepts all `co_await` expressions inside the coroutine.
2. **LoggingAwaiter** wraps the original awaiter and adds custom behavior:
    - Logs before suspension (`await_ready` / `await_suspend`).
    - Logs after resumption (`await_resume`).
3. Coroutine behavior is unchanged functionally, but **every await is augmented**.

3.3.4 Benefits of Await Transformation

1. **Cross-cutting Concerns:**
   - Apply consistent behavior to all `co_await` calls, e.g., **logging, metrics, tracing, profiling**.
2. **Custom Scheduling:**
   - Wrap awaitables to **enqueue on a custom scheduler or executor** automatically.
3. **Code Reuse and Abstraction:**
   - Avoid modifying individual awaitables by centralizing transformation in the promise type.
4. **Composable Awaitables:**
   - Combine transformations, e.g., **retry logic, timeout wrappers, or priority scheduling**.

3.3.5 Practical Use Cases

- **Asynchronous I/O pipelines** – Automatically schedule all awaitables on an I/O executor.
- **Logging and debugging** – Wrap awaitables to log suspension/resumption points for tracing.
- **Timeouts and cancellation** – Transform every awaitable to include **timeout or cancellation checks**.
- **Metrics collection** – Track suspension durations and await completion statistics.

Summary:

- **Await transformation (`await_transform`)** allows a coroutine to **intercept and modify all `co_await` expressions**.
- It operates in the **promise type**, returning a new awaiter that wraps or decorates the original.
- This mechanism is powerful for **logging, scheduling, profiling, timeouts, and other cross-cutting coroutine behaviors**.
- Mastering await transformations enables **robust, maintainable, and high-performance coroutine systems** in Modern C++23.

3.4 Practical Example – Implementing a Custom Awaitable for Waiting on a Timer or Resource

A **custom awaitable** allows coroutines to **suspend execution until a condition is met**, such as a timer expiration, resource availability, or I/O completion. This section provides a **practical implementation** demonstrating **timers and resource waiting** in Modern C++23 using coroutines.

3.4.1 Concept

- **Goal:** Suspend a coroutine until a **timer elapses** or a **resource becomes available**, then resume automatically.
- **Mechanism:** Implement a **custom awaitable** with `await_ready`, `await_suspend`, and `await_resume`.
- **Application:**
  - Timed delays in asynchronous pipelines.
  - Cooperative task scheduling.
  - Non-blocking I/O or resource acquisition in embedded systems.

3.4.2 Timer Awaitable Example

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

// Custom TimerAwaitable
struct TimerAwaitable {
    int duration_ms;

    TimerAwaitable(int ms) : duration_ms(ms) {}

    bool await_ready() const noexcept {
        // If duration <= 0, do not suspend
        return duration_ms <= 0;
    }
```

```
    void await_suspend(std::coroutine_handle<> handle) const {
        // Launch a background thread to resume coroutine after delay
        std::thread([handle, this]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(duration_ms));
            handle.resume(); // Resume coroutine after timer
        }).detach();
    }

    void await_resume() const noexcept {
        // No return value for simple timer
    }
};

// Coroutine Task
struct Task {
    struct promise_type {
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task demoTimerCoroutine() {
    std::cout << "Step 1: Timer starts (500ms)\n";
    co_await TimerAwaitable(500);
    std::cout << "Step 2: Timer elapsed\n";

    std::cout << "Step 3: Timer starts (1000ms)\n";
    co_await TimerAwaitable(1000);
    std::cout << "Step 4: Timer elapsed\n";
}

int main() {
```

```
    demoTimerCoroutine();
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Wait for async operations
}
```

3.4.3 Explanation

1. **`await_ready()`**
    - Determines whether the coroutine should **suspend immediately**.
    - For zero or negative durations, the coroutine continues without suspension.
2. **`await_suspend()`**
    - Launches a **background thread** to wait for the specified duration.
    - Calls `handle.resume()` to **resume the coroutine automatically** after the delay.
3. **`await_resume()`**
    - Returns nothing here, but could provide **elapsed time or status** in more complex scenarios.
4. **Execution Flow:**

```
demoTimerCoroutine() called
Step 1 printed -> coroutine suspends for 500ms
After 500ms -> coroutine resumes, Step 2 printed
Step 3 printed -> coroutine suspends for 1000ms
After 1000ms -> coroutine resumes, Step 4 printed
Coroutine completes
```

3.4.4 Resource Awaitable Example

In addition to timers, custom awaitables can **wait on a resource**, such as a shared flag or queue:

```
#include <atomic>
#include <coroutine>
#include <iostream>
#include <thread>

struct ResourceAwaitable {
```

```
    std::atomic<bool>& resource_ready;

    ResourceAwaitable(std::atomic<bool>& r) : resource_ready(r) {}

    bool await_ready() const noexcept {
        return resource_ready.load(); // Ready immediately if resource available
    }

    void await_suspend(std::coroutine_handle<> handle) const {
        // Poll in background thread (simplest example)
        std::thread([handle, this]() {
            while (!resource_ready.load()) {
                std::this_thread::sleep_for(std::chrono::milliseconds(50));
            }
            handle.resume();
        }).detach();
    }

    void await_resume() const noexcept {
        std::cout << "Resource acquired!\n";
    }
};

std::atomic<bool> resource_flag = false;

Task demoResourceCoroutine() {
    std::cout << "Waiting for resource...\n";
    co_await ResourceAwaitable(resource_flag);
    std::cout << "Processing with resource\n";
}

int main() {
    demoResourceCoroutine();
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    resource_flag = true; // Resource becomes available
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
```

3.4.5 Explanation

- `await_ready()`

  - Immediately continues if the resource is available.

- `await_suspend()`

  - Polls in a background thread or registers a callback to resume when resource is ready.

- `await_resume()`

  - Optionally performs **post-acquisition processing**.

- **Execution Flow:**

  - Coroutine suspends until `resource_flag` becomes `true`.
  - Automatically resumes when the resource is available.

3.4.6 Practical Insights

1. **Timers and Resource Awaitables** demonstrate **asynchronous coordination without threads blocking the main execution**.
2. **Custom awaitables** allow coroutines to be **integrated into event loops, schedulers, or I/O systems**.
3. **Memory and Lifecycle Safety:**
   - Ensure coroutine frame remains valid until `await_suspend` completes.
   - Always call `handle.resume()` or destroy coroutine to avoid leaks.
4. **Composable Design:**
   - Multiple custom awaitables can be **chained** to create sophisticated **asynchronous pipelines**.

Summary:

- Custom awaitables for **timers or resources** enable **controlled suspension and resumption** of coroutines.
- `await_ready`, `await_suspend`, and `await_resume` provide the **core interface**.

- Using these patterns, coroutines can **efficiently wait for asynchronous events**, maintain high performance, and simplify complex async code in Modern C++23.
- Mastering custom awaitables is essential for building **responsive, non-blocking, and high-performance coroutine-based applications**.

Chapter 4

Generators – Producing Data with Coroutines

4.1 Using `co_yield` to Create Data Generators

In Modern C++23, `co_yield` is a powerful coroutine mechanism for **producing sequences of values on demand**. Unlike returning a container, `co_yield` allows a **generator coroutine** to **suspend after producing each value**, enabling **lazy evaluation, memory efficiency, and complex pipelines**.

4.1.1 Concept of `co_yield`

- `co_yield` **produces a value** to the caller while **suspending the coroutine**.
- The coroutine can **resume later** to produce additional values, maintaining its **internal state** across suspensions.
- This is ideal for **streams, sequences, or iterative computations** that do not require generating all data at once.

Key properties of generators using `co_yield`:

1. **Lazy evaluation:** Values are generated only when requested.

2. **Stateful iteration:** Coroutine maintains local variables across yields.
3. **Memory efficiency:** Avoids creating large intermediate containers.
4. **Composable pipelines:** Generators can be chained with other coroutines or awaitables.

4.1.2 Basic Generator Example

This example demonstrates a simple **integer sequence generator** using `co_yield`:

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

// Generator structure
template<typename T>
struct Generator {
    struct promise_type {
        T current_value;

        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 value) noexcept {
            current_value = value;
            return {};
        }

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

    std::coroutine_handle<promise_type> handle;

    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    // Iterator for range-based for loops
```

```
    struct iterator {
        std::coroutine_handle<promise_type> handle;
        bool done;

        iterator(std::coroutine_handle<promise_type> h, bool d) : handle(h), done(d) {}

        iterator& operator++() {
            handle.resume();
            done = handle.done();
            return *this;
        }

        T operator*() const { return handle.promise().current_value; }
        bool operator!=(const iterator& other) const { return done != other.done; }
    };

    iterator begin() {
        handle.resume();
        return iterator{handle, handle.done()};
    }

    iterator end() { return iterator{handle, true}; }
};

// Coroutine generating a sequence of integers
Generator<int> simpleSequence(int start, int end) {
    for (int i = start; i <= end; ++i) {
        co_yield i;
    }
}

int main() {
    for (auto value : simpleSequence(1, 5)) {
        std::cout << value << " ";
    }
    std::cout << "\n";
}
```

Output:

```
1 2 3 4 5
```

4.1.3 Explanation

1. `co_yield`
   - Suspends the coroutine after producing a value.
   - Stores the value in `promise_type::current_value`.
2. `Generator` **Structure**
   - Holds the coroutine handle.
   - Provides **iteration support** with `begin()` and `end()` for range-based loops.
3. `yield_value()` **in** `promise_type`
   - Called automatically by `co_yield`.
   - Updates the current value and suspends execution.
4. **Resumption**
   - Each call to `iterator::operator++()` resumes the coroutine until the next `co_yield` or completion.

4.1.4 Practical Insights

1. **Memory Efficiency**
   - Only one value exists in memory at a time, unlike returning a container of all values.
2. **Lazy Computation**
   - Ideal for sequences with **heavy computations** or **I/O-bound generation**.
3. **Stateful Generators**
   - Local variables retain their state between yields, simplifying **state machines** or **iterative algorithms**.
4. **Composability**
   - Generators can be combined with **custom awaitables** to create **asynchronous pipelines**.

4.1.5 Advanced Example: Fibonacci Generator

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

int main() {
    for (auto f : fibonacci(10)) {
        std::cout << f << " ";
    }
}
```

Output:

```
0 1 1 2 3 5 8 13 21 34
```

Explanation:

- The coroutine maintains the **state of `a` and `b`** across yields.
- Fibonacci numbers are generated **on demand**, without storing the full sequence.

Summary:

- `co_yield` enables **lazy, stateful, and memory-efficient data generators**.
- Generators are ideal for **iterative sequences, streams, or computational pipelines**.
- Modern C++23 allows generators to be **composed with awaitables** for asynchronous data production.
- Mastering `co_yield` lays the foundation for **efficient coroutine-based algorithms** and **high-performance data processing**.

4.2 Generators vs Traditional Solutions (Performance Comparison)

Modern C++23 **generators** using `co_yield` provide a **more efficient and flexible alternative** to traditional data-producing methods such as returning **containers**, using **callbacks**, or manually managing **iterators**. Understanding the **performance implications** is crucial for building **high-performance applications**.

4.2.1 Traditional Approaches

1. **Using Containers**

```cpp
#include <vector>

std::vector<int> generateNumbers(int n) {
    std::vector<int> result;
    for (int i = 0; i < n; ++i) {
        result.push_back(i);
    }
    return result;
}

int main() {
    auto numbers = generateNumbers(1000000); // One million integers
}
```

Characteristics:

- Produces all values **at once**, consuming **memory proportional to the sequence size**.
- Copies or moves the container when returning from the function.
- Suitable for small datasets, but **high memory usage** can be a bottleneck for large sequences.

2. **Using Callbacks**

```cpp
#include <iostream>
#include <functional>
```

```cpp
void generateNumbersCallback(int n, std::function<void(int)> callback) {
    for (int i = 0; i < n; ++i) {
        callback(i);
    }
}

int main() {
    generateNumbersCallback(5, [](int value) {
        std::cout << value << " ";
    });
}
```

Characteristics:

- Avoids container allocation.
- **Imperative style**: the caller cannot easily **pause, resume, or iterate** at its own pace.
- Harder to **compose with asynchronous pipelines** or manage **stateful iteration**.

4.2.2 Generator-Based Approach

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

template<typename T>
struct Generator {
    struct promise_type {
        T current_value;
        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 value) noexcept { current_value = value; return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    struct iterator {
```

```
        std::coroutine_handle<promise_type> handle;
        bool done;
        iterator(std::coroutine_handle<promise_type> h, bool d) : handle(h), done(d) {}
        iterator& operator++() { handle.resume(); done = handle.done(); return *this; }
        T operator*() const { return handle.promise().current_value; }
        bool operator!=(const iterator& other) const { return done != other.done; }
    };

    iterator begin() { handle.resume(); return iterator{handle, handle.done()}; }
    iterator end() { return iterator{handle, true}; }
};

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

int main() {
    for (auto value : numberGenerator(5)) {
        std::cout << value << " ";
    }
}
```

Characteristics:

- **Lazy evaluation:** Values are produced **on demand**, not stored in memory.
- **Stateful:** Local variables inside the coroutine retain state between yields.
- **Composable:** Easy to combine with **custom awaitables** for asynchronous pipelines.

4.2.3 Performance Comparison

Feature	Container	Callback	Generator (`co_yield`)
Memory Usage	High (stores all values)	Low	Low (one value at a time)
Latency	High for large sequences (full generation)	Low	Low (produced on demand)
Composability	Limited	Limited	High (can chain, suspend, resume)

Feature	Container	Callback	Generator (`co_yield`)
Stateful Iteration	Requires external state	Requires external state	Built-in state preservation
Asynchronous Integration	Difficult	Difficult	Easy (can `co_await` inside generator)
Readability	High for small sequences	Moderate	High for iterative logic

Observations:

1. **Memory efficiency:** Generators avoid large temporary containers.
2. **Lazy computation:** Large sequences or expensive computations are evaluated **only when needed**.
3. **Composable and async-ready:** Generators integrate naturally with **asynchronous operations** via `co_await`.
4. **Performance cost:** Slight overhead from coroutine frame management, but often outweighed by memory and composability gains.

4.2.4 Practical Example: Comparing Execution

```
#include <vector>
#include <chrono>
#include <iostream>

// Traditional container approach
void testContainer(int n) {
    std::vector<int> v;
    for (int i = 0; i < n; ++i) v.push_back(i);
}

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

```
int main() {
    const int N = 10'000'000;

    auto start1 = std::chrono::high_resolution_clock::now();
    testContainer(N);
    auto end1 = std::chrono::high_resolution_clock::now();
    std::cout << "Container generation took: "
              << std::chrono::duration_cast<std::chrono::milliseconds>(end1 - start1).count()
              << " ms\n";

    auto start2 = std::chrono::high_resolution_clock::now();
    for (auto value : testGenerator(N)) {
        // Access value (do nothing)
    }
    auto end2 = std::chrono::high_resolution_clock::now();
    std::cout << "Generator iteration took: "
              << std::chrono::duration_cast<std::chrono::milliseconds>(end2 - start2).count()
              << " ms\n";
}
```

Key Takeaways:

- Container approach consumes **memory for all 10M elements**; may slow down due to allocations.
- Generator approach evaluates **one element at a time**, memory-efficient, slightly higher per-element overhead but scalable for large sequences.

4.2.5 When to Prefer Generators

- **Large or infinite sequences**.
- **Streaming data** (file, network, sensor input).
- **Complex pipelines** requiring suspension, resumption, or async behavior.
- **Stateful iteration** without external state management.

When traditional containers might be sufficient:

- Small sequences with fast computation.

- Need **random access** or **bulk processing** of all elements.

Summary:

- `co_yield` generators provide **lazy, stateful, and memory-efficient data production**.
- Traditional containers consume more memory and are less flexible for large or asynchronous workloads.
- Callbacks avoid containers but are **less composable and harder to manage**.
- Generators excel in **high-performance applications, streaming pipelines, and async integration** in Modern C++23.

4.3 Practical Example – Number Sequence Generator or Reading Data from File/Network

Generators in Modern C++23 using `co_yield` provide **flexible and efficient ways** to produce sequences of data **on demand**, whether for simple number sequences or more complex sources such as files or network streams. This section presents **practical examples** demonstrating both scenarios.

4.3.1 Number Sequence Generator

This example demonstrates a **simple generator producing a range of integers**, useful for iterative computations, simulations, or testing:

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

template<typename T>
struct Generator {
    struct promise_type {
        T current_value;
        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 value) noexcept { current_value = value; return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    struct iterator {
        std::coroutine_handle<promise_type> handle;
        bool done;
        iterator(std::coroutine_handle<promise_type> h, bool d) : handle(h), done(d) {}
        iterator& operator++() { handle.resume(); done = handle.done(); return *this; }
        T operator*() const { return handle.promise().current_value; }
        bool operator!=(const iterator& other) const { return done != other.done; }
```

```
    };

    iterator begin() { handle.resume(); return iterator{handle, handle.done()}; }
    iterator end() { return iterator{handle, true}; }
};

// Coroutine producing numbers from start to end
Generator<int> numberSequence(int start, int end) {
    for (int i = start; i <= end; ++i) {
        co_yield i;
    }
}

int main() {
    for (auto n : numberSequence(1, 10)) {
        std::cout << n << " ";
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Key Points:

- `co_yield` suspends after producing each value.
- Local variables (`i`) retain state across yields.
- No large container is allocated; memory usage is minimal.

4.3.2 Reading Data from a File Using a Generator

Generators can also produce **lines from a file** lazily, avoiding **loading the entire file into memory**:

```
#include <fstream>
#include <string>
#include <coroutine>
#include <iostream>

struct LineGenerator {
```

```
    struct promise_type {
        std::string current_line;
        LineGenerator get_return_object() {
            return LineGenerator{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(std::string line) noexcept {
            current_line = std::move(line);
            return {};
        }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit LineGenerator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~LineGenerator() { if (handle) handle.destroy(); }

    struct iterator {
        std::coroutine_handle<promise_type> handle;
        bool done;
        iterator(std::coroutine_handle<promise_type> h, bool d) : handle(h), done(d) {}
        iterator& operator++() { handle.resume(); done = handle.done(); return *this; }
        std::string operator*() const { return handle.promise().current_line; }
        bool operator!=(const iterator& other) const { return done != other.done; }
    };

    iterator begin() { handle.resume(); return iterator{handle, handle.done()}; }
    iterator end() { return iterator{handle, true}; }
};

// Coroutine reading lines from a file
LineGenerator readFile(const std::string& path) {
    std::ifstream file(path);
    std::string line;
    while (std::getline(file, line)) {
        co_yield line;
    }
}

int main() {
```

```
    for (auto line : readFile("example.txt")) {
        std::cout << line << "\n";
    }
}
```

Key Points:

- Only one line is loaded into memory at a time.
- Generator simplifies **streaming large files** without extra buffering logic.
- The coroutine maintains **state internally**, automatically handling iteration.

4.3.3 Reading Data from a Network Source

Generators can also handle **network streams asynchronously**, often combined with `co_await` for non-blocking reads:

```
// Pseudocode for async network generator
Generator<std::string> readNetworkStream(AsyncSocket& socket) {
    while (!socket.eof()) {
        auto data = co_await socket.readAsync(); // Custom awaitable for async read
        co_yield data;
    }
}
```

Benefits:

- Each network chunk is **produced as soon as it arrives**, without blocking the main thread.
- Supports **lazy processing** and **pipelined operations** (e.g., parsing, transforming).
- Integrates with **custom awaitables** for timers, cancellation, or flow control.

4.3.4 Practical Insights

1. **Lazy and Memory Efficient:**
    - Generators produce **values only when requested**, crucial for **large datasets or streams**.
2. **Composable with Async Operations:**

- Combining `co_yield` and `co_await` allows building **asynchronous pipelines** with **minimal boilerplate**.

3. **Stateful Iteration:**
   - The generator **maintains its internal state**, simplifying logic compared to manual loops or iterators.
4. **Use Cases:**
   - Number sequences (Fibonacci, primes, etc.).
   - File or log streaming.
   - Network data streaming or event processing.
   - Any scenario requiring **on-demand data production**.

Summary:

- Generators are **ideal for producing sequences of values lazily**, whether from **memory, files, or network streams**.
- They improve **memory efficiency, code readability, and composability** compared to traditional loops or container-based approaches.
- Combining `co_yield` with `co_await` unlocks **high-performance asynchronous pipelines**, a core strength of Modern C++23 coroutines.

Chapter 5

Tasks and Async Operations

5.1 Creating Asynchronous Tasks

Asynchronous programming is one of the most compelling use cases of Modern C++23 coroutines. Using `co_await`, `co_return`, and **custom awaitables**, you can define **tasks that run concurrently** without blocking the main thread, simplifying code that would otherwise require complex thread management.

5.1.1 Concept

- An **asynchronous task** represents a **unit of work** that may complete **later**, typically producing a result or completing an action.
- Unlike threads, tasks **do not create new OS threads** by default; they **suspend and resume** on demand, making them **lightweight**.
- Tasks are ideal for:
  - I/O operations (file, network, database)
  - Timers and delayed actions
  - Parallel computations or pipelines

Key idea: Tasks are implemented as **coroutines returning a custom `Task<T>` type**, which encapsulates the coroutine frame, the result, and the suspension/resumption logic.

5.1.2 Simple Task Example

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

// Basic Task template
template<typename T>
struct Task {
    struct promise_type {
        std::optional<T> result;

        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 value) { result = value; }
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;

    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }

    T get() {
        handle.resume();
        return *handle.promise().result;
    }
};

// Coroutine creating an asynchronous task
Task<int> asyncAdd(int a, int b) {
```

```
    std::cout << "Task started...\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(500)); // simulate delay
    co_return a + b;
}

int main() {
    auto task = asyncAdd(10, 20);
    int result = task.get(); // resumes coroutine and waits for result
    std::cout << "Result: " << result << "\n";
}
```

Output:

```
Task started...
Result: 30
```

Explanation:

- `Task<int>` encapsulates a coroutine producing an **integer result**.
- `co_return` stores the value in the **promise_type**.
- Calling `task.get()` **resumes the coroutine** and retrieves the result.
- The coroutine suspends during the simulated delay, allowing **non-blocking scheduling** in more complex scenarios.

5.1.3 Using `co_await` for True Asynchronous Behavior

To make tasks **truly non-blocking**, you can combine them with **custom awaitables**:

```
struct TimerAwaitable {
    int ms;
    TimerAwaitable(int ms) : ms(ms) {}

    bool await_ready() const noexcept { return false; }

    void await_suspend(std::coroutine_handle<> handle) const {
        std::thread([handle, ms = this->ms]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(ms));
            handle.resume();
        }).detach();
```

```
    }

    void await_resume() const noexcept {}
};

Task<void> asyncTaskWithDelay() {
    std::cout << "Task started\n";
    co_await TimerAwaitable(1000); // non-blocking wait for 1 second
    std::cout << "Task resumed after 1 second\n";
}
```

Advantages:

- Coroutine suspends without blocking the **main thread**.
- Multiple tasks can **run concurrently** without manual thread management.
- Easy to integrate with **event loops or I/O schedulers**.

5.1.4 Practical Insights

1. **Lightweight Concurrency:**
    - Tasks use **stackless coroutines**, keeping memory overhead minimal compared to threads.
2. **Composable Asynchronous Operations:**
    - Tasks can **await other tasks** or custom awaitables to build **complex workflows**.
3. **Error Handling:**
    - Exceptions thrown inside a coroutine propagate to the **promise_type**, which can handle or terminate the program.
4. **Integration with Async Pipelines:**
    - Tasks are the **building blocks** for asynchronous generators, timers, and network pipelines.

5.1.5 Summary

- Asynchronous tasks encapsulate **units of work** that can suspend and resume without blocking.
- They are **lightweight, composable, and memory-efficient**.
- Combining `co_await`, **custom awaitables**, and `co_return` allows creating **responsive, high-performance applications** in Modern C++23.
- Understanding tasks is the foundation for **advanced async programming**, including **parallel pipelines, event-driven systems, and async generators**.

5.2 Integration with `std::future` and `std::task` (C++23)

Modern C++23 introduces **native coroutine support for `std::future` and `std::task`**, allowing seamless integration between **coroutine-based asynchronous tasks** and **standard library concurrency mechanisms**. This section explains how to leverage these types to simplify **asynchronous programming** while maintaining high performance.

5.2.1 `std::future` Integration

`std::future` is a standard mechanism to **retrieve results from asynchronous operations**. With coroutines, you can write functions returning `std::future<T>` that **suspend internally** and produce results once ready.

Example: Coroutine Returning `std::future<int>`

```
#include <future>
#include <iostream>
#include <thread>
#include <chrono>

std::future<int> asyncAdd(int a, int b) {
    co_await std::suspend_always{}; // suspend immediately
    std::this_thread::sleep_for(std::chrono::milliseconds(500)); // simulate delay
    co_return a + b; // return result
}

int main() {
    auto fut = asyncAdd(10, 20);
    std::cout << "Doing other work while task is suspended...\n";

    int result = fut.get(); // blocks until result is ready
    std::cout << "Result: " << result << "\n";
}
```

Explanation:

- The coroutine returns a `std::future<int>`.
- `co_return` sets the value of the future.
- `co_await std::suspend_always{}` demonstrates **suspension before computation**.

- The main thread can continue other work while the coroutine is suspended.

Benefits:

- Enables **easy interoperability** with existing APIs using `std::future`.
- Supports **lazy evaluation** and deferred computation.

5.2.2 `std::task` (C++23 Proposal)

`std::task<T>` is a **lightweight coroutine-aware type** designed specifically for **non-blocking asynchronous tasks**. Unlike `std::future`, it **does not tie tasks to threads** and allows **true suspension and resumption**.

Example: Using `std::task<int>`

```
#include <coroutine>
#include <iostream>
#include <thread>
#include <chrono>
#include <cpp23/task> // pseudo-include for conceptual example

std::task<int> asyncMultiply(int a, int b) {
    std::cout << "Task started\n";
    co_await std::suspend_always{}; // suspend, simulating async operation
    co_return a * b;
}

int main() {
    auto t = asyncMultiply(6, 7);
    std::cout << "Doing other work while task is suspended...\n";

    int result = t.get(); // resumes coroutine and retrieves result
    std::cout << "Result: " << result << "\n";
}
```

Key Points:

- `std::task` is **designed for coroutines**, unlike `std::future` which is thread-bound.
- Tasks can **suspend multiple times**, yielding control back to the caller or scheduler.
- Ideal for **high-performance async pipelines** with **minimal overhead**.

5.2.3 Comparison: `std::future` vs `std::task`

Feature	`std::future`	`std::task`
Coroutine-Friendly	Partial (requires adaptation)	Fully coroutine-aware
Suspension	Limited, usually blocks thread	True suspension, non-blocking
Lightweight	Heavier, may involve OS resources	Lightweight, stackless coroutines
Composability	Harder to chain async operations	Easy to combine tasks and pipelines
Use Case	Interoperability with legacy async APIs	High-performance async tasks and generators

Observation:

- Use `std::future` when integrating with **existing APIs**.
- Use `std::task` for **pure coroutine-based async workflows**, **memory efficiency**, and **true non-blocking behavior**.

5.2.4 Practical Example: Combining `std::future` with Coroutines

```
#include <future>
#include <iostream>

std::future<int> computeSum(int a, int b) {
    co_return a + b;
}

std::task<int> doubleSum(int a, int b) {
    int sum = co_await computeSum(a, b); // await future inside a coroutine
    co_return sum * 2;
}

int main() {
    auto task = doubleSum(5, 15);
    std::cout << "Awaiting combined async tasks...\n";
    int result = task.get();
    std::cout << "Result: " << result << "\n"; // Outputs 40
}
```

Explanation:

- `doubleSum` awaits `computeSum` (returns `std::future<int>`).
- The coroutine suspends while waiting for the future, then resumes to compute the doubled result.
- Demonstrates **integration between legacy async (`std::future`) and modern coroutine tasks (`std::task`)**.

5.2.5 Practical Insights

1. **Seamless Integration:**
   - Coroutines allow bridging between **traditional futures** and **modern tasks** without complex callback chains.
2. **Performance Advantages:**
   - `std::task` avoids **thread overhead**, ideal for **high-frequency asynchronous operations**.
3. **Composable Pipelines:**
   - Tasks can await other tasks or futures, building **complex async workflows** cleanly.
4. **Error Handling:**
   - Exceptions propagate through `co_return` and `co_await`, allowing **structured error management**.

Summary:

- Modern C++23 allows coroutines to return both **`std::future`** and **`std::task`**, bridging **legacy async APIs** and **modern coroutine pipelines**.
- `std::future` is suitable for **thread-bound results** or existing libraries.
- `std::task` is **lightweight, non-blocking, and coroutine-native**, ideal for high-performance async programming.
- Combining both enables **scalable, composable, and memory-efficient asynchronous operations**.

5.3 Practical Example – Performing an Async Network I/O Request Using Coroutines

Modern C++23 coroutines excel in **asynchronous I/O operations**, such as **network requests**, without blocking threads. This section demonstrates a practical example of performing a **non-blocking network request** using coroutines with **custom awaitables**, showing how to handle **async workflows** cleanly and efficiently.

5.3.1 Concept

- **Traditional approach:** Network I/O often requires **blocking calls**, multiple threads, or callbacks.
- **Coroutine approach:** Suspend the coroutine while waiting for network data, then resume automatically when data is available.
- Advantages:
  - Avoids creating unnecessary threads.
  - Preserves **state** between suspension points.
  - Clean, linear code style without nested callbacks.

5.3.2 Custom Awaitable for Async Network Read

A simple custom awaitable can simulate **non-blocking network reads**. In real applications, it can integrate with **asio, libcurl, or OS-specific async APIs**.

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

// Simulated network response
std::string fetchFromNetwork() {
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    return "Network data received";
```

```
}

// Awaitable for async network read
struct NetworkAwaitable {
    std::string result;

    bool await_ready() const noexcept { return false; } // always suspend
    void await_suspend(std::coroutine_handle<> handle) {
        std::thread([handle, this]() mutable {
            result = fetchFromNetwork(); // simulate network delay
            handle.resume(); // resume coroutine after "network response"
        }).detach();
    }
    std::string await_resume() const noexcept { return result; }
};
```

Explanation:

- `await_ready()`: Determines if the coroutine should **suspend immediately**. Returning `false` means suspension occurs.
- `await_suspend()`: Called when suspension happens. A separate thread simulates **network latency**.
- `await_resume()`: Returns the result when the coroutine **resumes**.

5.3.3 Coroutine Performing Async Network I/O

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

// Task wrapper for coroutine
template<typename T>
struct Task {
    struct promise_type {
        T value;
        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 val) { value = val; }
```

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

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    T get() { handle.resume(); return handle.promise().value; }
};

// Async network request coroutine
Task<std::string> asyncNetworkRequest() {
    std::cout << "Sending network request...\n";
    std::string data = co_await NetworkAwaitable{};
    std::cout << "Data received inside coroutine.\n";
    co_return data;
}

int main() {
    auto task = asyncNetworkRequest();
    std::cout << "Doing other work while waiting for network...\n";

    std::string result = task.get(); // resume coroutine when network data is ready
    std::cout << "Result: " << result << "\n";
}
```

Output:

```
Sending network request...
Doing other work while waiting for network...
Data received inside coroutine.
Result: Network data received
```

5.3.4 Key Advantages

1. **Non-blocking I/O:**
    - The main thread is free to perform other work while waiting for the network response.
2. **Stateful Coroutines:**
    - Local variables (`data`) retain their values across suspension, eliminating manual state tracking.

3. **Composable Workflows:**
    - Multiple network requests can be **awaited sequentially or concurrently** using coroutines.
4. **Simpler Code:**
    - Linear, readable code without nested callbacks or explicit threads.

5.3.5 Extending to Real-World Applications

- Replace `fetchFromNetwork()` with **asio async_read**, **Boost.Beast**, or **libcurl multi interface**.
- Integrate multiple network coroutines into **pipelines**, combining tasks for **parallel requests**.
- Use **cancellation, timeouts, and error propagation** by extending the awaitable with proper exception handling.

Example: Awaiting Multiple Network Tasks (Conceptual)

```
Task<void> fetchMultipleData() {
    auto data1 = co_await asyncNetworkRequest();
    auto data2 = co_await asyncNetworkRequest();
    std::cout << "Combined results:\n" << data1 << "\n" << data2 << "\n";
}
```

- Each network request suspends independently.
- Coroutine resumes automatically when each result is ready.

5.3.6 Summary

- Coroutines allow **efficient, non-blocking network I/O** without threads or callbacks.
- Custom awaitables let you **integrate any asynchronous API** with `co_await`.
- This approach improves **memory efficiency, composability, and code clarity**, making it ideal for **high-performance networked applications** in Modern C++23.

5.4 Using `co_await` for Managing Multiple Tasks

Modern C++23 coroutines allow you to **await multiple asynchronous tasks**, making it possible to **compose complex workflows** while keeping code readable and non-blocking. By combining tasks with `co_await`, you can **suspend a coroutine until all awaited tasks complete**, handle results efficiently, and manage dependencies between tasks.

5.4.1 Concept

- `co_await` can be used to **suspend the current coroutine** until a **single task or multiple tasks** finish.
- Multiple tasks can be awaited either **sequentially** or **concurrently**.
- This enables **lightweight concurrency** without manually managing threads, mutexes, or condition variables.

5.4.2 Sequential Awaiting

Sequential awaiting is the simplest approach: tasks are awaited **one after another**, suspending the coroutine for each task individually.

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

template<typename T>
struct Task {
    struct promise_type {
        T value;
        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 val) { value = val; }
        void unhandled_exception() { std::terminate(); }
    };
```

```
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    T get() { handle.resume(); return handle.promise().value; }
};

// Simulated asynchronous operation
Task<int> asyncOperation(int id, int delay_ms) {
    std::this_thread::sleep_for(std::chrono::milliseconds(delay_ms));
    co_return id * 10;
}

// Sequential await
Task<void> sequentialTasks() {
    int result1 = co_await asyncOperation(1, 300);
    int result2 = co_await asyncOperation(2, 200);
    int result3 = co_await asyncOperation(3, 100);

    std::cout << "Sequential results: " << result1 << ", " << result2 << ", " << result3 << "\n";
}

int main() {
    sequentialTasks().get();
}
```

Explanation:

- Each `co_await` suspends until the task finishes.
- Tasks execute **one after another**, not concurrently.
- Simple but may be slower if tasks are **independent and could run in parallel**.

5.4.3 Concurrent Awaiting

Concurrent awaiting allows multiple tasks to **run in parallel**, suspending the coroutine until **all tasks complete**, similar to `Promise.all` in JavaScript.

```
#include <future>
#include <vector>
#include <iostream>
#include <thread>
```

```cpp
#include <chrono>

template<typename T>
struct Task {
    struct promise_type {
        T value;
        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 val) { value = val; }
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    T get() { handle.resume(); return handle.promise().value; }
};

// Simulated async operation
Task<int> asyncOp(int id, int delay_ms) {
    std::this_thread::sleep_for(std::chrono::milliseconds(delay_ms));
    co_return id * 100;
}

// Concurrent await helper
Task<void> concurrentTasks() {
    auto t1 = asyncOp(1, 300);
    auto t2 = asyncOp(2, 200);
    auto t3 = asyncOp(3, 100);

    int r1 = t1.get(); // each get resumes its coroutine
    int r2 = t2.get();
    int r3 = t3.get();

    std::cout << "Concurrent results: " << r1 << ", " << r2 << ", " << r3 << "\n";
}

int main() {
    concurrentTasks().get();
}
```

Explanation:

- Tasks `t1`, `t2`, and `t3` are created independently.
- Each task **runs in its own coroutine frame** and may execute concurrently.
- The main coroutine resumes when each awaited task completes.
- This approach is **efficient for independent I/O or computation-heavy tasks**.

5.4.4 Practical Patterns

- **a) Task Aggregation**

```
template<typename T>
Task<std::vector<T>> gatherTasks(std::vector<Task<T>> tasks) {
    std::vector<T> results;
    for (auto& t : tasks) {
        results.push_back(co_await t);
    }
    co_return results;
}
```

  - Allows combining multiple tasks into **one awaitable task**.
  - Useful for **network requests, file I/O, or batch computations**.

- **b) Parallel Pipelines**

  - Each stage of a pipeline can be a **task that awaits previous stages**.
  - Enables **stream processing** with minimal memory overhead.

5.4.5 Error Handling

- Exceptions thrown in any awaited task propagate to the **awaiting coroutine**.
- This simplifies **structured error handling** for multiple tasks:

```
Task<void> safeConcurrentTasks() {
    try {
        auto r1 = co_await asyncOp(1, 200);
        auto r2 = co_await asyncOp(2, 200);
```

```
        std::cout << "Results: " << r1 << ", " << r2 << "\n";
    } catch (const std::exception& e) {
        std::cout << "Error in async tasks: " << e.what() << "\n";
    }
}
```

5.4.6 Summary

- `co_await` allows **managing multiple tasks** efficiently in Modern C++23.
- Sequential awaiting is **simple but slower** for independent tasks.
- Concurrent awaiting allows **tasks to run in parallel**, improving performance.
- Task aggregation and pipelines make it easy to **compose complex asynchronous workflows**.
- Exceptions propagate naturally, simplifying **robust error handling**.

Practical Insight: Using `co_await` with multiple tasks is central for building **high-performance async applications**, including **network servers, file processing pipelines, and real-time computation tasks**.

Chapter 6

Real-World Applications

6.1 Concurrent Data Pipelines

Modern C++23 coroutines provide a **powerful mechanism** for building **concurrent data pipelines**, enabling **efficient processing of streams of data** in real-time applications. By combining coroutines, tasks, and generators, you can create pipelines that **process, transform, and transmit data concurrently** while maintaining **readable, linear code**.

6.1.1 Concept

A **data pipeline** consists of multiple stages, each performing a specific operation on data:

1. **Source**: Produces data (file, network, sensor, or computation).
2. **Processing stages**: Transform, filter, or aggregate data.
3. **Sink**: Consumes the final result (display, store, or send over network).

Concurrent pipelines use **coroutines to suspend at each stage**, allowing other tasks to proceed without blocking. This improves **throughput** and reduces **latency** in high-performance systems.
Key benefits:

- Non-blocking execution of multiple stages

- Lightweight concurrency without threads overhead
- Composable and maintainable pipeline design

6.1.2 Building a Simple Concurrent Pipeline

- **Stage 1: Data Source Generator**

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

template<typename T>
struct Generator {
    struct promise_type {
        T current_value;
        std::suspend_always yield_value(T value) {
            current_value = value;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

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

```
    T value() { return handle.promise().current_value; }
};

// Data source coroutine
Generator<int> dataSource() {
    for (int i = 1; i <= 5; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100)); // simulate data arrival
        co_yield i;
    }
}
```

  - `Generator<int>` produces data **incrementally**.
  - `co_yield` suspends after each value, allowing the consumer stage to process concurrently.

- **Stage 2: Processing Stage**

```
// Processing stage: doubles each number
Generator<int> processor(Generator<int>& input) {
    while (input.next()) {
        int val = input.value();
        co_yield val * 2; // transformation
    }
}
```

  - The processor **awaits the next value from the source**.
  - Each value is transformed independently.

- **Stage 3: Sink Stage**

```
void sink(Generator<int>& processed) {
    while (processed.next()) {
        std::cout << "Processed value: " << processed.value() << "\n";
    }
}
```

  - The sink consumes the **final transformed values**.
  - This can be extended to **write to files, network, or UI**.

Full Pipeline Example

```
int main() {
    auto source = dataSource();
    auto stage = processor(source);
    sink(stage);
}
```

Output:

```
Processed value: 2
Processed value: 4
Processed value: 6
Processed value: 8
Processed value: 10
```

Explanation:

- Each stage suspends after producing or consuming data.
- The main function drives the pipeline linearly, while **each stage maintains its state across suspensions**.

6.1.3 Enhancing Concurrency

- Multiple **processor stages** can be chained, each as a separate coroutine.
- Tasks can run **independently on separate threads** using `co_await` on async operations.
- Example: Reading from network, filtering, transforming, and storing results concurrently without blocking threads.

```
// Conceptual async processing pipeline
Task<void> asyncPipeline() {
    auto source = asyncDataSource();
    auto processed1 = asyncFilter(source);      // filter stage
    auto processed2 = asyncTransform(processed1); // transform stage
    co_await asyncSink(processed2);             // consume results
}
```

- Each `co_await` suspends until the **upstream stage produces results**.
- Enables **high-throughput, low-latency data processing**.

6.1.4 Practical Insights

1. **State Preservation:** Coroutines automatically **preserve local variables** between suspensions, simplifying pipeline logic.
2. **Composable Stages:** Each stage can be reused in **different pipelines**.
3. **Minimal Thread Overhead:** Coroutines provide **stackless concurrency**, avoiding OS thread costs.
4. **Scalability:** Easy to scale pipelines for **large data streams** by adding more coroutine-based stages.

6.1.5 Summary

- Concurrent data pipelines in Modern C++23 allow **non-blocking, composable processing of data streams**.
- Generators, tasks, and `co_await` enable **linear and readable code** while maintaining concurrency.
- Pipelines can handle **real-time data from network, files, or sensors** efficiently.
- This approach forms the **foundation for building high-performance, event-driven, and real-time applications**.

6.2 Game Loops Using Coroutines

Modern C++23 coroutines provide an **efficient, flexible alternative** to traditional game loop designs. They enable **asynchronous updates, rendering, and event handling** without blocking the main thread and without manually managing complex state machines. Using coroutines, you can create **frame-based loops** that are **linear, readable, and maintainable**.

6.2.1 Traditional Game Loop

A typical game loop consists of:

```
while (running) {
    processInput();
    updateGameState();
    renderFrame();
}
```

Limitations:

- All stages are executed **synchronously**, which can cause **frame drops** if one stage is slow.
- Hard to manage **delays or asynchronous events** (network, AI, physics) without additional threads or timers.

6.2.2 Coroutine-Based Game Loop

Coroutines allow **suspending execution within a frame**, letting other tasks run, and then resuming seamlessly. This enables **frame pacing** and **concurrent tasks** like animations, physics, and AI.

Example: Simple Coroutine Game Loop

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

// Coroutine task type
struct Task {
    struct promise_type {
```

```
        Task get_return_object() { return Task{ std::coroutine_handle<promise_type>::from_promise(*this) };
        ↪  }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    void resume() { handle.resume(); }
};

// Simulate frame wait
struct wait_frame {
    std::chrono::milliseconds duration;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, d = duration]() {
            std::this_thread::sleep_for(d);
            h.resume();
        }).detach();
    }
    void await_resume() {}
};

// Coroutine game loop
Task gameLoop(int frames) {
    for (int i = 1; i <= frames; ++i) {
        std::cout << "Frame " << i << " - processing input\n";
        std::cout << "Frame " << i << " - updating game state\n";
        std::cout << "Frame " << i << " - rendering frame\n";

        co_await wait_frame{ std::chrono::milliseconds(16) }; // ~60 FPS
    }
    std::cout << "Game loop finished.\n";
}

int main() {
    auto loop = gameLoop(5);
    while (!loop.handle.done()) {
        loop.resume();
```

```
    }
}
```

Explanation:

- `wait_frame` suspends the coroutine for a given **frame duration**, simulating frame pacing.
- The game loop coroutine **maintains state automatically** between frames.
- No manual state machines or extra threads are needed for frame scheduling.

6.2.3 Benefits of Using Coroutines in Game Loops

1. **Frame-Based Suspension:**
   - Each frame can **suspend**, allowing other asynchronous tasks to execute.
   - Simplifies **timing logic**, e.g., AI updates, animations, and physics.
2. **Concurrency-Friendly:**
   - Multiple coroutines can run **concurrently**, e.g., particle systems, network updates, audio, while keeping the main loop readable.
3. **Linear Code Flow:**
   - Avoids **nested callbacks or complex state machines**.
   - Game logic remains **intuitive and maintainable**.
4. **Scalability:**
   - Easy to add **additional subsystems** like AI, networked entities, or background resource loading without blocking the main loop.

6.2.4 Advanced Example: Concurrent Coroutines in Game Loop

```
Task playerAnimation() {
    for (int i = 0; i < 3; ++i) {
        std::cout << "Animating player frame " << i << "\n";
        co_await wait_frame{ std::chrono::milliseconds(16) };
    }
```

```
}

Task enemyAI() {
    for (int i = 0; i < 3; ++i) {
        std::cout << "Updating enemy AI frame " << i << "\n";
        co_await wait_frame{ std::chrono::milliseconds(16) };
    }
}

Task mainLoop() {
    auto player = playerAnimation();
    auto enemy = enemyAI();
    for (int i = 0; i < 3; ++i) {
        std::cout << "Main game loop frame " << i << "\n";
        player.resume();
        enemy.resume();
        co_await wait_frame{ std::chrono::milliseconds(16) };
    }
}
```

Key Insights:

- Each subsystem (player animation, enemy AI) runs as a **separate coroutine**.
- Main loop orchestrates **suspension and resumption**.
- Allows **fine-grained control** over each subsystem while maintaining **high FPS and responsiveness**.

6.2.5 Summary

- Coroutines provide a **modern, efficient approach** to game loops in C++23.
- Suspend/resume mechanism allows **frame-based timing, concurrency, and linear code**.
- Multiple game subsystems can run **concurrently without complex threading or callback logic**.
- Ideal for **high-performance games**, **real-time simulations**, and **interactive applications**.

6.3 Network & I/O-Heavy Applications

Modern C++23 coroutines are particularly powerful for **network-intensive and I/O-heavy applications**. These include **web servers, database clients, streaming services, and high-frequency trading systems**. Coroutines allow developers to **write asynchronous, non-blocking code** that is **readable, maintainable, and high-performance**, avoiding the traditional pitfalls of multi-threading and callback hell.

6.3.1 The Challenge with Network & I/O

Traditional I/O operations block the calling thread:

```
std::string fetchData() {
    // Blocking network call
    return httpRequest("http://example.com/data");
}
```

Problems:

- Blocking calls can **stall the main thread**.
- Multiple threads increase **context-switch overhead** and **memory usage**.
- Callback-based async APIs often lead to **nested, hard-to-maintain code**.

Solution: Modern C++ coroutines use `co_await` to **suspend execution** until the I/O operation completes, freeing the thread to perform other tasks.

6.3.2 Asynchronous Network Request with Coroutines

Here's a practical example of a **non-blocking network request** simulated with coroutines:

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

// Simulated async network request
struct NetworkRequest {
```

```
    std::string data;
    bool await_ready() { return false; }  // always suspend
    void await_suspend(std::coroutine_handle<> handle) {
        std::thread([handle, this]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(500)); // simulate network latency
            data = "Fetched network data";
            handle.resume();
        }).detach();
    }
    std::string await_resume() { return data; }
};

// Task wrapper for coroutine
struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    void resume() { handle.resume(); }
};

// Network coroutine
Task networkTask() {
    std::cout << "Starting network request...\n";
    std::string result = co_await NetworkRequest{};
    std::cout << "Received: " << result << "\n";
}
```

Explanation:

- `await_ready()` returns `false` to **always suspend the coroutine**.
- `await_suspend()` launches a **separate thread** to simulate network I/O.
- `await_resume()` returns the **fetched data** after resumption.

6.3.3 Handling Multiple I/O Tasks Concurrently

Coroutines allow **awaiting multiple I/O tasks** concurrently without blocking:

```
Task concurrentNetworkTasks() {
    auto task1 = NetworkRequest{};
    auto task2 = NetworkRequest{};
    auto task3 = NetworkRequest{};

    std::string r1 = co_await task1;
    std::string r2 = co_await task2;
    std::string r3 = co_await task3;

    std::cout << "Concurrent results:\n" << r1 << "\n" << r2 << "\n" << r3 << "\n";
}
```

Benefits:

- Tasks run independently.
- Main coroutine resumes as each task completes.
- Reduces **latency for network-heavy operations** compared to sequential blocking.

6.3.4 File I/O with Coroutines

Coroutines can also handle **disk I/O**, which is often slower than CPU operations:

```
struct FileReadAwaitable {
    std::string content;
    std::string filename;

    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, this]() {
            std::ifstream file(filename);
            std::stringstream ss;
            ss << file.rdbuf();
            content = ss.str();
            h.resume();
        }).detach();
    }
```

```
    std::string await_resume() { return content; }
};

Task readFileTask(const std::string& fname) {
    std::string data = co_await FileReadAwaitable{ "", fname };
    std::cout << "Read file content:\n" << data << "\n";
}
```

Key Points:

- **Non-blocking file reads** enable high-throughput file servers.
- Multiple files can be read concurrently with **minimal thread usage**.

6.3.5 Real-World Integration

- **Web servers:** Await network requests and database queries asynchronously.
- **Streaming services:** Handle video/audio chunks concurrently with coroutines.
- **Databases:** Await query results without blocking main threads.
- **High-frequency trading:** Non-blocking I/O for price feeds and order books.

Pattern: Use **custom awaitables** that integrate with **OS-level async I/O** or libraries such as **Boost.Asio, libuv, or native sockets**.

6.3.6 Best Practices

1. **Use coroutine-friendly libraries** for network and disk I/O.
2. **Avoid blocking inside coroutines**, use `co_await` for async operations.
3. **Combine multiple coroutines** for high concurrency and efficient resource utilization.
4. **Propagate exceptions** through coroutines to handle I/O errors cleanly.

6.3.7 Summary

- Coroutines in C++23 provide a **clean, efficient, and composable model** for network and I/O-heavy applications.
- Tasks can run **concurrently without blocking threads**, improving throughput and responsiveness.
- Custom awaitables let you **wrap any asynchronous API**—network, file, or device I/O—into **linear, maintainable code**.
- This approach is essential for **modern high-performance servers, streaming platforms, and real-time systems**.

6.4 Practical Example – Data Stream Pipeline with Intelligent Suspensions

One of the most **powerful applications of Modern C++23 coroutines** is creating **data stream pipelines** that dynamically **suspend and resume** based on workload, resource availability, or data readiness. This approach is ideal for **high-performance, real-time processing**, such as network streams, sensor data, or concurrent computation pipelines.

6.4.1 Concept

An intelligent suspension pipeline:

1. **Produces data** from a source (sensor, file, network).
2. **Processes data asynchronously** using multiple stages (transform, filter, aggregate).
3. **Suspends stages intelligently** when downstream stages are busy or resources are constrained.
4. **Resumes automatically** when conditions are met, avoiding wasted CPU cycles.

Benefits:

- Efficient **CPU and memory usage**
- High throughput without busy-waiting
- Composable, maintainable pipeline structure

6.4.2 Pipeline Components

- **a) Data Source Generator**

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

template<typename T>
struct Generator {
```

```
    struct promise_type {
        T current_value;
        std::suspend_always yield_value(T value) {
            current_value = value;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

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

    T value() { return handle.promise().current_value; }
};

// Simulated data source
Generator<int> dataSource() {
    for (int i = 1; i <= 10; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(50)); // simulate arrival delay
        co_yield i;
    }
}
```

  - Produces **incremental data values** with `co_yield`.
  - Suspension after each value allows **downstream processing** to keep pace.

- **b) Processing Stage with Intelligent Suspension**

```
struct IntelligentAwaitable {
    int delay_ms;
```

```
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, d = delay_ms]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(d)); // simulate work
            h.resume();
        }).detach();
    }
    void await_resume() {}
};

Generator<int> processor(Generator<int>& input) {
    while (input.next()) {
        int val = input.value();
        std::cout << "Processing value: " << val << "\n";

        // Intelligent suspension: short pause if CPU or I/O-bound
        co_await IntelligentAwaitable{ 30 }; // simulate load-based suspension

        co_yield val * 10; // transformed value
    }
}
```

  - Suspension time can **adjust dynamically** based on load, queue length, or I/O readiness.
  - Allows **intelligent pacing** to prevent overwhelming downstream stages.

- **c) Consumer Stage**

```
void consumer(Generator<int>& processed) {
    while (processed.next()) {
        std::cout << "Consumed value: " << processed.value() << "\n";
        // Optional: further intelligent suspension if needed
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
```

  - Consumes processed data efficiently.
  - Can introduce **additional pacing** for downstream resource management.

6.4.3 Full Pipeline Execution

```
int main() {
```

```
    auto source = dataSource();
    auto processed = processor(source);
    consumer(processed);
}
```

Output Example:

```
Processing value: 1
Consumed value: 10
Processing value: 2
Consumed value: 20
...
```

- Each stage **suspends and resumes intelligently**.
- Pipeline remains **responsive** and adapts to simulated load conditions.

6.4.4 Advanced Enhancements

1. **Dynamic Suspension Adjustment:**
    - Measure **queue size** or **system load** to dynamically change suspension duration.

```
co_await IntelligentAwaitable{ dynamicDelay() };
```

1. **Parallel Stages:**
    - Multiple processors can work concurrently on different data chunks:

```
auto proc1 = processor(source);
auto proc2 = processor(source);
co_await proc1;
co_await proc2;
```

1. **Error Handling:**
    - Exceptions propagate naturally through coroutines:

```
try {
    co_await processor(source);
} catch(const std::exception& e) {
    std::cerr << "Pipeline error: " << e.what() << "\n";
}
```

1. **Integration with Async I/O:**
    - Replace `sleep_for` with **network or file awaitables** for real-time streaming applications.

6.4.5 Key Insights

- Intelligent suspensions **optimize CPU usage** while maintaining throughput.
- Coroutines simplify **complex pipeline logic** by keeping stages linear and composable.
- Combined with **async I/O**, this pattern can handle **thousands of concurrent streams** efficiently.
- Ideal for **real-time analytics, streaming data, sensor networks, and high-frequency trading**.

6.4.6 Summary

- This practical example demonstrates how to build a **responsive data stream pipeline** using C++23 coroutines.
- Each stage can **suspend and resume intelligently**, allowing **high-performance concurrent processing**.
- Coroutines reduce **complex state management**, enabling **linear, maintainable code** for real-world I/O-heavy and computation-heavy systems.

Chapter 7

Best Practices

7.1 Exception Management Inside Coroutines

Exception management is a **critical aspect** of writing robust and reliable coroutines in Modern C++23. Unlike traditional functions, coroutines **suspend and resume across multiple points**, which introduces **special considerations for exception handling**. Proper management ensures that exceptions **propagate correctly**, resources are **cleaned up**, and **awaiting tasks** can respond appropriately.

7.1.1 How Exceptions Work in Coroutines

In coroutines, exceptions can arise in several contexts:

1. **Inside the coroutine body**: during execution between suspensions.
2. **During suspension/resumption**: for example, when `co_await` fails.
3. **In promise type functions**: such as `initial_suspend`, `final_suspend`, or `return_void`.

Key points:

- If an exception occurs in a coroutine and is **not caught**, it is stored in the **promise object**.
- When a coroutine is **resumed**, the exception is **propagated** when the coroutine accesses its return value or when `co_await` completes.

- Coroutines allow **deferred exception handling**, which is useful for **asynchronous workflows**.

7.1.2 Exception Handling in Promise Types

The promise type defines **how exceptions are managed** inside the coroutine:

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

struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {
            // Store exception for later retrieval
            eptr = std::current_exception();
        }
        std::exception_ptr eptr;
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }

    void resume() {
        handle.resume();
        if (handle.promise().eptr) {
            std::rethrow_exception(handle.promise().eptr);
        }
    }
};
```

Explanation:

- `unhandled_exception()` in the promise type **captures exceptions** that occur in the coroutine.
- `resume()` checks for stored exceptions and **rethrows them in the caller context**.
- This allows **clean separation** of coroutine execution and exception handling.

7.1.3 Exception Propagation with `co_await`

When awaiting a coroutine or an async task, exceptions **propagate through `co_await`**:

```
Task riskyTask() {
    std::cout << "Start risky task\n";
    throw std::runtime_error("Something went wrong inside coroutine");
    co_return;
}

Task callerTask() {
    try {
        co_await riskyTask();
    } catch (const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << "\n";
    }
}
```

Key points:

- `co_await` automatically **resumes the coroutine** that is waiting and throws the exception inside the awaiting context.
- This provides **linear, readable error handling** without complex callbacks.

7.1.4 Best Practices for Exception Management

1. **Always implement `unhandled_exception()` in your promise type**
    - Ensures that exceptions are **not lost** during suspension/resumption.
2. **Catch exceptions inside coroutines when possible**
    - Useful for **retry logic**, resource cleanup, or logging.

```
Task safeTask() {
    try {
        // Code that may throw
    } catch (const std::exception& e) {
        std::cout << "Handled internally: " << e.what() << "\n";
    }
    co_return;
}
```

1. **Use `std::exception_ptr` to propagate exceptions across threads or tasks**
    - Especially important in **async pipelines** or **network tasks**.
2. **Do not throw exceptions from `initial_suspend` or `final_suspend`**
    - These are **low-level coroutine hooks**, and throwing can lead to undefined behavior.
    - Instead, use `unhandled_exception()` to **capture and propagate exceptions safely**.
3. **Combine with RAII**
    - Coroutines suspend and resume, so **resource management** should use RAII to guarantee cleanup.

7.1.5 Practical Example: Async File Read with Exception Handling

```
#include <fstream>
#include <sstream>
#include <stdexcept>

struct FileAwaitable {
    std::string filename;
    std::string content;

    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, this]() {
            try {
                std::ifstream file(filename);
                if (!file.is_open()) throw std::runtime_error("Cannot open file: " + filename);
                std::stringstream ss;
                ss << file.rdbuf();
                content = ss.str();
            } catch (...) {
                exception = std::current_exception();
            }
            h.resume();
        }).detach();
    }
    std::string await_resume() {
        if (exception) std::rethrow_exception(exception);
```

```
        return content;
    }

    std::exception_ptr exception = nullptr;
};

Task readFileTask(const std::string& fname) {
    try {
        std::string data = co_await FileAwaitable{fname};
        std::cout << "File content: " << data << "\n";
    } catch (const std::exception& e) {
        std::cout << "Error reading file: " << e.what() << "\n";
    }
}
```

Highlights:

- Exception inside the awaitable is **captured and rethrown** on `co_await`.
- Caller handles the error **linearly and safely**.
- Coroutines simplify **asynchronous exception handling** across threads.

7.1.6 Summary

- Exceptions in coroutines are **captured by the promise type** and can be **propagated through `co_await`**.
- Always implement **`unhandled_exception()`** to prevent lost exceptions.
- Coroutines allow **linear error handling** even in complex async pipelines.
- Combine **RAII and exception management** to ensure **robust, high-performance code**.
- Proper exception management is essential for **real-world, production-ready coroutine applications**.

7.2 Memory Management

Memory management is a **critical concern** when working with coroutines in Modern C++23. Unlike traditional functions, coroutines **suspend and resume multiple times**, which affects how memory is allocated and freed. Understanding how coroutines use memory is essential for **high-performance and robust applications**, especially in **long-running tasks, pipelines, or network-heavy workloads**.

7.2.1 How Coroutines Use Memory

When a coroutine is called:

1. A **coroutine frame** is allocated on the heap (or custom allocator).
2. The frame stores:
    - **Local variables** of the coroutine.
    - **Promise object** (handles `co_return`, `unhandled_exception`).
    - **State information** to resume execution.
3. The frame remains **alive across suspensions** and is only destroyed after:
    - `final_suspend` is reached.
    - The coroutine handle is **destroyed**.

Key insight:
Memory is **not automatically freed** until the coroutine is explicitly destroyed, even if it has finished execution.

7.2.2 Coroutine Handle and Memory Lifetime

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

struct Task {
    struct promise_type {
        Task get_return_object() {
```

```
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); } // frees coroutine memory
};

Task example() {
    int local = 42; // stored in coroutine frame
    co_return;
}

int main() {
    auto t = example();
    // Memory is freed when Task's destructor destroys the handle
}
```

Explanation:

- `std::coroutine_handle` provides **control over coroutine lifetime**.
- You can decide **when to destroy the frame**, giving fine-grained memory control.
- Neglecting `destroy()` leads to **memory leaks**, especially for coroutines that suspend and never resume.

7.2.3 Custom Allocators for Coroutines

C++23 allows **custom allocation strategies** for coroutine frames:

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

struct CustomAllocator {
    static void* allocate(std::size_t size) {
        std::cout << "Allocating coroutine frame of size " << size << "\n";
```

```
        return ::operator new(size);
    }
    static void deallocate(void* ptr, std::size_t size) {
        std::cout << "Deallocating coroutine frame of size " << size << "\n";
        ::operator delete(ptr);
    }
};

struct Task {
    struct promise_type {
        void* operator new(std::size_t size) { return CustomAllocator::allocate(size); }
        void operator delete(void* ptr, std::size_t size) { CustomAllocator::deallocate(ptr, size); }

        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};
```

Benefits of Custom Allocators:

- Optimize **heap allocation for small, frequent coroutines**.
- Integrate with **memory pools or shared arenas** for performance.
- Reduce **fragmentation in high-performance applications** like game loops or network servers.

7.2.4 RAII and Coroutine Memory Safety

- Use **RAII patterns** for resources inside coroutine frames.
- Even if a coroutine is suspended or destroyed unexpectedly, **destructors of local objects** are invoked when the frame is destroyed.

```
Task safeResourceTask() {
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    co_await std::suspend_always{}; // suspension does not leak ptr
    std::cout << *ptr << "\n";
}
```

- Smart pointers, locks, and file handles behave **predictably** inside coroutine frames.
- Ensures **safe resource cleanup** even in complex async pipelines.

7.2.5 Practical Recommendations

1. **Always destroy coroutine handles** when no longer needed.
2. **Use smart pointers and RAII** for resource management inside coroutines.
3. **Consider custom allocators** for high-frequency coroutines to improve performance.
4. **Avoid large local objects** in coroutines if possible, to reduce frame size.
5. Monitor memory usage when building **long-lived pipelines or network servers**.

7.2.6 Advanced Memory Management Patterns

- **Frame reuse:** Keep coroutine frames in a pool and **reuse them for multiple tasks**.
- **Suspension-aware buffers:** Combine **small object pools** with awaitables to prevent excessive heap allocation.
- **Stackless coroutines:** Coroutines do not allocate full stack frames like threads, reducing memory overhead significantly.

7.2.7 Summary

- Coroutines allocate a **frame on the heap** for each instance; lifetime is tied to **handle destruction**.
- **RAII, smart pointers, and custom allocators** are key for safe memory management.
- Modern C++23 allows **fine-grained control** over memory usage, enabling **high-performance async systems**.

- Proper memory management ensures **robust, scalable, and efficient coroutine applications** in real-world systems.

7.3 Avoiding Common Pitfalls – Dangling Handles and Undefined Behavior

Coroutines in Modern C++23 provide **powerful concurrency and asynchronous capabilities**, but with this power comes responsibility. Mismanaging coroutine handles, memory, or suspension points can lead to **dangling handles, undefined behavior, resource leaks, and crashes**. This section details the **common pitfalls** and strategies to **avoid them**.

7.3.1 Dangling Coroutine Handles

A **dangling handle** occurs when a `std::coroutine_handle` is used after the coroutine frame has been destroyed.

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

struct Task {
    struct promise_type {
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
};

Task createDanglingHandle() {
    auto t = Task::promise_type{}.get_return_object();
    // Coroutine frame destroyed when t goes out of scope
    return t; // Returning a handle that may become dangling
}
```

Problems:

- Using a dangling handle leads to **undefined behavior** when `resume()` or `destroy()` is called.

- Common in **manual handle management**, particularly in pipelines or async task schedulers.

Solution:

- Always **ensure the coroutine frame is alive** for the lifetime of the handle.
- Prefer **RAII wrappers** that automatically destroy the handle.
- Avoid returning raw handles to external scopes without proper ownership management.

7.3.2 Undefined Behavior due to Suspensions

Undefined behavior may occur if:

1. You **resume a coroutine after it is already finished**.
2. You **access local variables** after the coroutine frame has been destroyed.
3. You **mix coroutine handles across threads unsafely** without synchronization.

```
Task example() {
    int local = 42;
    co_await std::suspend_always{};
    // Accessing local after frame destruction can cause UB
    std::cout << local << "\n";
}
```

Best Practices:

- Do **not capture references** to local variables outside the coroutine frame.
- Ensure **synchronization** when resuming coroutines from multiple threads.
- Always check `handle.done()` before resuming:

```
if (!handle.done()) handle.resume();
```

7.3.3 Handling Exceptions Safely

- Failing to handle exceptions inside a coroutine frame may leave **resources partially initialized** or cause **unexpected destruction**.

- Always implement `unhandled_exception()` in your **promise type**.

```
struct TaskPromise {
    void unhandled_exception() {
        // Capture exception for deferred propagation
        eptr = std::current_exception();
    }
    std::exception_ptr eptr;
};
```

- Combine with **RAII and smart pointers** to ensure resource cleanup even when exceptions occur.

7.3.4 Avoiding Resource Leaks

Coroutines allocate memory on the heap for their **frames**, and failure to destroy handles properly can leak memory:

```
auto t = example();
// forgetting to call t.handle.destroy() -> memory leak
```

Mitigation:

- Use **RAII wrappers** that destroy handles in the destructor.
- Prefer **Task objects or smart coroutine wrappers** instead of raw handles.

```
struct SafeTask {
    std::coroutine_handle<> handle;
    ~SafeTask() { if (handle) handle.destroy(); }
};
```

7.3.5 Thread Safety Concerns

- Resuming coroutines from **multiple threads** requires care.
- Avoid **concurrent resume** unless explicitly supported.
- Use **mutexes or atomic flags** if multiple threads interact with the same coroutine.

```
std::atomic<bool> resumed = false;
if (!resumed.exchange(true)) {
    handle.resume(); // ensure only one thread resumes
}
```

7.3.6 Practical Guidelines

1. **Do not manually manipulate dangling handles**.
2. **Wrap handles in RAII types** for safe destruction.
3. **Never access locals after suspension if the frame may be destroyed**.
4. **Check `handle.done()`** before resuming a coroutine.
5. **Use exception-safe patterns** to prevent resource leaks.
6. **Avoid resuming coroutines concurrently** without proper synchronization.

7.3.7 Example: Safe Coroutine Wrapper

```
struct SafeTask {
    std::coroutine_handle<> handle;
    explicit SafeTask(std::coroutine_handle<> h) : handle(h) {}
    ~SafeTask() { if (handle) handle.destroy(); }

    void resume() {
        if (handle && !handle.done()) {
            handle.resume();
        }
    }
};
```

- Prevents **dangling handle misuse**.
- Ensures **memory is freed safely** even if coroutine finishes or throws an exception.
- Provides a **robust, reusable pattern** for production coroutines.

7.3.8 Summary

- **Dangling handles** and **undefined behavior** are common pitfalls in coroutines.
- Use **RAII, smart pointers, and safe wrapper patterns** to manage coroutine lifetime.
- Always check **suspension, resumption, and completion status** to avoid UB.

- Proper design ensures **robust, maintainable, and high-performance coroutine applications**.

7.4 Practical Example – Resilient Coroutine with Exception Handling

Building **resilient coroutines** involves combining proper **exception management**, **memory safety**, and **intelligent resumption**. This ensures coroutines remain robust, even when **errors occur during asynchronous operations or resource processing**.

7.4.1 Scenario

Suppose we have a coroutine that:

1. Reads data from multiple sources (file, network, sensor).
2. Performs some processing on the data.
3. Needs to **recover gracefully** if one source fails, while continuing other tasks.

Key goals:

- Handle exceptions **inside the coroutine**.
- Ensure **resources are properly released** even after an error.
- Continue processing remaining tasks without crashing.

7.4.2 Resilient Awaitable

```
#include <coroutine>
#include <iostream>
#include <fstream>
#include <exception>
#include <thread>
#include <chrono>

struct FileReadAwaitable {
    std::string filename;
    std::string content;
    std::exception_ptr exception = nullptr;

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

```
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([this, h]() {
            try {
                std::ifstream file(filename);
                if (!file.is_open()) throw std::runtime_error("Cannot open file: " + filename);
                std::stringstream ss;
                ss << file.rdbuf();
                content = ss.str();
            } catch (...) {
                exception = std::current_exception();
            }
            h.resume();
        }).detach();
    }

    std::string await_resume() {
        if (exception) std::rethrow_exception(exception);
        return content;
    }
};
```

Explanation:

- Exception inside the awaitable is **captured and stored**.
- `co_await` automatically **rethrows in the caller context**.
- Threaded read simulates **async operation**.

7.4.3 Resilient Coroutine

```
struct Task {
    struct promise_type {
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { eptr = std::current_exception(); }
        std::exception_ptr eptr;
```

```
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }

    void resume() {
        handle.resume();
        if (handle.promise().eptr) std::rethrow_exception(handle.promise().eptr);
    }
};

Task resilientTask() {
    try {
        std::string data1 = co_await FileReadAwaitable{"file1.txt"};
        std::cout << "File1 content: " << data1 << "\n";
    } catch (const std::exception& e) {
        std::cout << "Error reading file1: " << e.what() << "\n";
    }

    try {
        std::string data2 = co_await FileReadAwaitable{"file2.txt"};
        std::cout << "File2 content: " << data2 << "\n";
    } catch (const std::exception& e) {
        std::cout << "Error reading file2: " << e.what() << "\n";
    }

    std::cout << "Processing continues despite failures\n";
    co_return;
}
```

Highlights:

- Each file read is wrapped in its **own try-catch block**, ensuring **resilience**.
- Coroutine continues execution even if **one awaitable fails**.
- `Task` wrapper ensures **proper destruction of coroutine frame**.

7.4.4 Execution

```
int main() {
```

```
    auto task = resilientTask();
    task.resume();
}
```

Sample Output:

```
Error reading file1: Cannot open file: file1.txt
File2 content: Hello World
Processing continues despite failures
```

Explanation:

- Exception in `file1.txt` is **caught and logged**, not propagated to crash the program.
- Coroutine continues to `file2.txt` and further processing.
- Demonstrates **robust asynchronous workflow** with coroutines.

7.4.5 Enhancements for Real-World Applications

1. **Retry Mechanism**
    - Retry failed awaitables a configurable number of times before logging an error.

```
int retries = 3;
while (retries--) {
    try {
        std::string data = co_await FileReadAwaitable{"file.txt"};
        break;
    } catch (...) {
        if (retries == 0) throw; // propagate after final attempt
    }
}
```

1. **Backoff Strategy**
    - Gradually increase suspension time between retries to prevent resource exhaustion.
2. **Parallel Execution**
    - Use multiple coroutines to read files or perform tasks concurrently, handling exceptions individually.

3. **Resource Cleanup**
    - Use **RAII** to manage resources in suspended states safely.

7.4.6 Key Takeaways

- Coroutines can be **resilient and robust** when combined with **exception handling and RAII**.
- Use **per-awaitable try-catch blocks** to isolate failures.
- Always **destroy coroutine handles** to free memory.
- This pattern is ideal for **network I/O, file processing, streaming pipelines**, and **asynchronous computation** where partial failures are expected.

This example illustrates a **production-ready pattern** for writing resilient coroutines that **gracefully handle errors** while keeping the program running efficiently.

Chapter 8

Debugging & Performance Optimization

8.1 Tracking Resumption and Suspension

Understanding when a coroutine **suspends** and **resumes** is critical for **debugging, performance optimization, and designing efficient asynchronous systems**. Modern C++23 provides tools and patterns to **instrument coroutines** for detailed runtime insights without adding significant overhead.

8.1.1 Why Track Resumptions and Suspensions

Coroutines can suspend multiple times at `co_await` or `co_yield` points. Without proper tracking:

- You may **lose visibility** into how frequently a coroutine suspends.
- Performance issues may arise from **excessive suspensions or unnecessary context switching**.
- Debugging complex **async pipelines, generators, or tasks** becomes difficult.

Tracking resumption and suspension allows developers to:

- Identify **performance bottlenecks**.
- Verify that **awaitable or generator logic behaves correctly**.
- Ensure **resources are properly managed** during suspensions.

8.1.2 Instrumenting Coroutine Promises

A common approach is to **add logging or counters** inside the promise type:

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

struct Task {
    struct promise_type {
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_always initial_suspend() {
            std::cout << "[Coroutine] Initial suspend\n";
            return {};
        }
        std::suspend_always final_suspend() noexcept {
            std::cout << "[Coroutine] Final suspend\n";
            return {};
        }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }

    void resume() {
        std::cout << "[Coroutine] Resuming...\n";
        handle.resume();
        std::cout << "[Coroutine] Suspended or finished\n";
    }
};
```

Explanation:

- `initial_suspend` and `final_suspend` log suspension points.
- `resume()` logs resumption events.
- Developers can see **every transition in the coroutine life cycle**.

8.1.3 Tracking Suspensions in Awaitables

Awaitables themselves can be instrumented to track suspension behavior:

```
struct LoggingAwaitable {
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::cout << "[Awaitable] Coroutine suspended\n";
        h.resume(); // Immediate resumption for demonstration
    }
    void await_resume() { std::cout << "[Awaitable] Coroutine resumed\n"; }
};

Task trackedCoroutine() {
    std::cout << "Start coroutine\n";
    co_await LoggingAwaitable{};
    std::cout << "After first suspension\n";
    co_await LoggingAwaitable{};
    std::cout << "End coroutine\n";
}
```

Output:

```
Start coroutine
[Awaitable] Coroutine suspended
[Awaitable] Coroutine resumed
After first suspension
[Awaitable] Coroutine suspended
[Awaitable] Coroutine resumed
End coroutine
```

Key Observations:

- Each suspension/resumption can be **observed clearly**.
- Provides insights into **scheduler or async behavior** in pipelines.

8.1.4 Counters and Profiling

For more quantitative tracking, **counters can record frequency and duration** of suspensions:

```
struct ProfilingAwaitable {
    static int suspend_count;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        suspend_count++;
        h.resume();
    }
    void await_resume() {}
};

int ProfilingAwaitable::suspend_count = 0;

Task profiledCoroutine() {
    co_await ProfilingAwaitable{};
    co_await ProfilingAwaitable{};
    co_await ProfilingAwaitable{};
    std::cout << "Suspensions count: " << ProfilingAwaitable::suspend_count << "\n";
}
```

- This pattern can identify **hotspots in generators, async pipelines, or network tasks**.
- Counters can be combined with **timers** to measure suspension duration for performance tuning.

8.1.5 Visualizing Resumptions and Suspensions

For larger applications:

- Maintain **logs or counters per coroutine instance**.
- Correlate with **thread IDs or task IDs** to analyze concurrency.
- Generate **timeline visualizations** to track async workflows.

```
std::cout << "[Coroutine ID=" << handle.address() << "] Suspended\n";
std::cout << "[Coroutine ID=" << handle.address() << "] Resumed\n";
```

This provides a **low-overhead tracing mechanism** suitable for production debugging.

8.1.6 Best Practices

1. **Instrument promise types and awaitables** to track suspensions/resumptions.
2. **Use counters for performance profiling** without affecting logic.
3. **Log or visualize transitions** in complex async pipelines.
4. **Avoid expensive operations during tracking** to prevent altering performance behavior.
5. Consider **conditional logging** for production, enabling detailed tracking only when needed.

8.1.7 Summary

- Tracking resumption and suspension is essential for **debugging, profiling, and optimizing coroutines**.
- Promises, awaitables, and coroutine handles can all be instrumented with **logs, counters, and timers**.
- Proper tracking ensures **high-performance, predictable, and maintainable asynchronous systems** in Modern C++23.

8.2 Tools and Libraries for Profiling

Profiling coroutines in Modern C++23 is essential for understanding **performance bottlenecks, suspension overhead, memory usage, and concurrency behavior**. Unlike traditional functions, coroutines **introduce additional frames, suspension points, and scheduling**, which can impact runtime performance. This section explores **tools and techniques** for profiling coroutines effectively.

8.2.1 Built-in Timing and Profiling Techniques

- **a) High-Resolution Timers**

  C++23 provides `std::chrono::high_resolution_clock` for measuring execution and suspension duration:

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

struct TimerAwaitable {
    int delay_ms;
    TimerAwaitable(int ms) : delay_ms(ms) {}
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, d=delay_ms]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(d));
            h.resume();
        }).detach();
    }
    void await_resume() {}
};

Task timedCoroutine() {
    auto start = std::chrono::high_resolution_clock::now();
    co_await TimerAwaitable{100};
    auto mid = std::chrono::high_resolution_clock::now();
    std::cout << "Suspension duration: "
              << std::chrono::duration_cast<std::chrono::microseconds>(mid - start).count()
              << " µs\n";
}
```

Benefits:

- Measures the **actual suspension time** and resumption latency.
- Can be integrated in **profiling pipelines** or performance-critical coroutines.

- **b) Frame Counting**

  Keep track of how many coroutine frames are **created and destroyed**:

```
struct CountingTask {
    struct promise_type {
        static int frame_count;
        CountingTask get_return_object() {
            frame_count++;
            return CountingTask{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    std::coroutine_handle<promise_type> handle;
    explicit CountingTask(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~CountingTask() { if (handle) { handle.destroy(); promise_type::frame_count--; } }
};

int CountingTask::promise_type::frame_count = 0;
```

- Helps **detect leaks or excessive allocations**.
- Useful for **long-lived pipelines or network servers**.

8.2.2 External Profiling Tools

Several tools support **coroutine-aware profiling**:

- textbfa) **Valgrind / Massif / Callgrind**
  - Tracks **heap memory usage** of coroutine frames.
  - Useful for **memory leaks, frame growth, and fragmentation**.
  - Combined with counters in code, provides **detailed allocation insights**.

- textbfb) **Perf (Linux) / Windows Performance Analyzer**
  - Measures **CPU cycles, context switches, and function calls**.
  - Can detect **excessive suspension overhead** in high-frequency coroutines.
- textbfc) **Google Benchmark / Google Performance Tools**
  - Supports **microbenchmarks of coroutines vs traditional async tasks**.
  - Allows measuring **latency, throughput, and suspension costs**.
- textbfd) **Intel VTune Profiler**
  - Advanced profiling with **thread-level and async task visualization**.
  - Can show **coroutine frame usage, suspension points, and hot spots**.

8.2.3 Custom Logging Libraries

Sometimes **lightweight, custom logging is the most practical solution** for coroutine profiling:

```
#include <iostream>

struct LoggingAwaitable {
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::cout << "[Profiler] Coroutine suspended at handle: " << h.address() << "\n";
        h.resume();
    }
    void await_resume() {
        std::cout << "[Profiler] Coroutine resumed\n";
    }
};
```

- Adds **minimal overhead** for tracking resumptions and suspensions.
- Can be **enabled or disabled** at compile time using macros.
- Useful for **pipeline debugging and performance tracing** without full profiling tools.

8.2.4 Best Practices for Coroutine Profiling

1. **Instrument both promises and awaitables** to capture suspensions and resumption frequency.
2. **Measure frame allocation** to detect memory overhead.
3. **Combine logging with high-resolution timers** for latency profiling.
4. **Profile in real workloads** rather than synthetic tests for meaningful results.
5. Use **external tools** for CPU and memory profiling, but keep **lightweight logging for real-time insights**.
6. Consider **conditional compilation** for profiling to avoid overhead in production builds.

8.2.5 Summary

- Profiling coroutines requires understanding **suspension, resumption, and frame allocations**.
- C++23 supports **high-resolution timers, counters, and RAII-based instrumentation** for tracking coroutine behavior.
- Combining **custom logging** with **external profiling tools** allows comprehensive analysis of **performance and memory usage**.
- Proper profiling ensures **high-performance, efficient, and predictable coroutine-based applications**.

8.3 Practical Example – Performance Analysis for a Large Data Generator Using Coroutines

Coroutines are particularly well-suited for **lazy data generation** and **stream processing**. However, when dealing with **large datasets**, it's important to **analyze performance** to ensure that **suspensions, resumption overhead, and memory usage** do not degrade efficiency. This section demonstrates a **practical performance analysis** using Modern C++23 coroutines.

8.3.1 Scenario

We want to **generate a large sequence of numbers** and measure:

1. **Time spent in suspensions and resumptions**.
2. **Memory usage of coroutine frames**.
3. **Throughput compared to traditional iterator-based solutions**.

This is typical in applications like **streaming analytics, log processing, and network pipelines**.

8.3.2 Coroutine-Based Generator

```
#include <iostream>

struct LoggingAwaitable {
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::cout << "[Profiler] Coroutine suspended at handle: " << h.address() << "\n";
        h.resume();
    }
    void await_resume() {
        std::cout << "[Profiler] Coroutine resumed\n";
    }
};
```

Explanation:

- `co_yield` suspends the coroutine on each number.
- Memory for each frame is allocated **once**, reducing repeated allocations.

- Resumption occurs lazily, only when `next()` is called.

8.3.3 Measuring Performance

- **a) Time Measurement**

```
int main() {
    constexpr int N = 10'000'000;
    auto start = std::chrono::high_resolution_clock::now();

    auto gen = largeDataGenerator(N);
    size_t sum = 0;
    while (gen.next()) {
        sum += gen.current_value();
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "Sum: " << sum << "\n";
    std::cout << "Elapsed time: "
              << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
              << " ms\n";
}
```

Insights:

  - Measures **total execution time** including all suspensions.
  - Allows comparison with **traditional for-loops** or container-based solutions.

- **b) Memory Usage Analysis**

  - Each coroutine frame stores **local variables and state**.
  - For large sequences, memory usage is **constant per coroutine frame**, unlike containers that grow linearly.
  - For more detailed tracking, integrate **Valgrind Massif or custom counters** for frame allocations.

```
static size_t frame_count = 0;

struct CountingPromise {
    CountingPromise() { ++frame_count; }
```

```
    ~CountingPromise() { --frame_count; }
    // rest of promise_type...
};
```

- Monitoring `frame_count` helps **ensure no memory leaks** during long-running generators.

8.3.4 Comparison with Traditional Approaches

Traditional `std::vector` approach:

```
std::vector<int> data;
data.reserve(N);
for (int i = 1; i <= N; ++i) data.push_back(i);
```

Observations:

- Vector approach incurs **large upfront memory allocation**.
- Coroutine generator uses **lazy evaluation**, consuming **minimal memory**.
- For **streaming pipelines**, coroutine generators allow **processing on-the-fly** without storing the entire dataset.

8.3.5 Profiling Suspensions

- Count suspensions to measure overhead:

```
struct ProfilingGeneratorAwaitable {
    static size_t suspend_count;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) { ++suspend_count; h.resume(); }
    void await_resume() {}
};

size_t ProfilingGeneratorAwaitable::suspend_count = 0;
```

- Combine with `co_yield` to track **total number of suspension/resumption cycles**.

8.3.6 Key Takeaways

1. **Coroutines enable memory-efficient generators** for large datasets.
2. Performance depends on:
    - Suspension/resumption frequency.
    - Frame size and allocation strategy.
    - Integration with async pipelines.
3. Profiling with **timers, counters, and external tools** helps identify bottlenecks.
4. Coroutines **outperform traditional containers** in streaming or high-volume scenarios where **lazy evaluation is beneficial**.
5. Proper **RAII and handle management** ensures no **memory leaks** even for large data generators.

This example demonstrates a **practical, real-world use case** for coroutines: generating and processing **millions of data items efficiently** while providing **detailed performance insights**.

Chapter 9

Advanced C++23 Features

9.1 Awaitable Improvements

Modern C++23 introduces several **enhancements and refinements** to the coroutine machinery, particularly around **awaitables** and how they interact with the coroutine framework. These improvements allow developers to write **more efficient, composable, and expressive asynchronous code**.

9.1.1 Motivation for Awaitable Improvements

In C++20, awaitables were **defined by three key methods**:

- `await_ready()` – determines if the coroutine should suspend immediately.
- `await_suspend()` – defines the suspension behavior.
- `await_resume()` – returns the result upon resumption.

While functional, the original design presented some challenges:

1. **Repetitive boilerplate** when creating custom awaitables.
2. Difficulty in **transforming awaitables** for composability.
3. Lack of **automatic integration with standard async types** like `std::task` in C++23.

C++23 introduces improvements to **streamline these patterns**.

9.1.2 Simplified Awaitable Conventions

- **a) `operator co_await` customization**

  C++23 standardizes `operator co_await` to allow **seamless transformations** of objects into awaitables:

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

struct Timer {
    int duration_ms;
    bool await_ready() { return duration_ms <= 0; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, d=duration_ms]() {
            std::this_thread::sleep_for(std::chrono::milliseconds(d));
            h.resume();
        }).detach();
    }
    void await_resume() {}
};

// Transformation using operator co_await
struct TaskWrapper {
    Timer t;
    auto operator co_await() { return t; }
};

TaskWrapper asyncTask() {
    co_await Timer{100};
}
```

  Benefits:

  - Allows any object to **customize its awaitable behavior** without altering core logic.
  - Improves **composability** of async operations.

- **b) Awaitable Composition**

  C++23 supports **awaitable adapters** that **combine multiple awaitables** efficiently:

```
struct CompositeAwaitable {
    Timer t1{50};
    Timer t2{75};

    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        std::thread([h, this]() {
            co_await t1;
            co_await t2;
            h.resume();
        }).detach();
    }
    void await_resume() {}
};
```

Advantage:

- Combines multiple asynchronous operations into **a single awaitable**, simplifying coroutine logic.

9.1.3 Standard Library Integration

C++23 improves **interoperability with standard async types**:

- `std::future`, `std::task`, and other task-like objects can now be **directly co_awaited**.
- This allows existing asynchronous code to **integrate with coroutines** with minimal boilerplate.

```
#include <future>
#include <coroutine>

TaskWrapper asyncFromFuture() {
    std::future<int> fut = std::async([](){ return 42; });
    int value = co_await fut; // Directly await a future
    std::cout << "Value: " << value << "\n";
}
```

Implications:

- Existing code using `std::future` can **seamlessly migrate** to coroutines.
- Reduces the need for **manual adapters** or custom awaitables.

9.1.4 Lazy and Composable Awaitables

- C++23 encourages **lazy evaluation of awaitables**, similar to generators.
- Awaitables can now **wrap multiple operations** and **transform results** before resumption.

```
struct TransformAwaitable {
    Timer t{50};
    int factor;
    auto await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) { t.await_suspend(h); }
    int await_resume() { return 10 * factor; }
};

TaskWrapper transformedTask() {
    int result = co_await TransformAwaitable{5};
    std::cout << "Transformed result: " << result << "\n";
}
```

Highlights:

- Integrates **functional transformation patterns** directly in awaitables.
- Makes coroutines **more expressive and concise** for async pipelines.

9.1.5 Best Practices

1. **Use `operator co_await`** for object-to-awaitable transformation to reduce boilerplate.
2. **Combine awaitables** when multiple asynchronous operations depend on each other.
3. Prefer **standard task-like awaitables** (`std::task`, `std::future`) when possible.
4. **Keep awaitables lightweight**: avoid allocating large objects inside `await_suspend`.
5. Use **lazy evaluation** and transformations to write **composable and maintainable async logic**.

9.1.6 Summary

C++23 brings **enhanced flexibility and standardization** to awaitables:

- Direct integration with **standard async types**.
- Simplified **operator co_await patterns**.
- Improved **composition, transformation, and lazy evaluation**.
- Enables **high-performance, readable, and maintainable coroutines** for complex asynchronous systems.

9.2 Composable Coroutines

Modern C++23 emphasizes **composability** in coroutines, allowing developers to **combine multiple asynchronous operations** into a single, elegant coroutine chain. Composable coroutines enable writing **modular, maintainable, and efficient asynchronous workflows**, which is critical for **network applications, data pipelines, and concurrent systems**.

9.2.1 What is Coroutine Composability?

Composable coroutines are coroutines designed to:

1. **Return awaitable results** that can be further `co_await`ed.
2. **Chain multiple asynchronous tasks** without blocking threads.
3. Simplify **complex async flows** by encapsulating tasks in reusable units.

In essence, composable coroutines allow **building larger coroutines from smaller ones**, similar to **functional composition** in functional programming.

9.2.2 Basic Example of Coroutine Composition

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

// Simple task type
struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    void resume() { handle.resume(); }
};
```

```
// Coroutine producing a value
Task subTask(int id) {
    std::cout << "SubTask " << id << " started\n";
    co_await std::suspend_always{};
    std::cout << "SubTask " << id << " resumed\n";
}

// Composable parent coroutine
Task composedCoroutine() {
    std::cout << "Composed Coroutine started\n";
    co_await subTask(1);  // Compose first subtask
    co_await subTask(2);  // Compose second subtask
    std::cout << "Composed Coroutine finished\n";
}

int main() {
    auto mainTask = composedCoroutine();
    mainTask.resume(); // Starts the parent coroutine
    mainTask.resume(); // Resumes after first suspension
    mainTask.resume(); // Resumes after second suspension
}
```

Explanation:

- `composedCoroutine` **awaits multiple subtasks**, creating a **seamless chain of operations**.
- Each subtask can **suspend and resume independently**, without blocking threads.
- The parent coroutine **automatically resumes** when each subtask completes.

9.2.3 Advantages of Composable Coroutines

1. **Modularity:** Break complex tasks into smaller, reusable coroutines.
2. **Lazy Execution:** Each coroutine suspends and resumes only when needed.
3. **Reduced Threading Complexity:** No need for manual thread management or locks.
4. **Improved Readability:** Async sequences read sequentially, even when executed concurrently.

9.2.4 Composable Awaitables in C++23

C++23 introduces **enhanced awaitable composition** for tasks like **async pipelines** or **task combinators**:

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

std::future<int> asyncAdd(int a, int b) {
    co_return a + b;
}

std::future<int> asyncMultiply(int x, int y) {
    co_return x * y;
}

std::future<void> composedFutureTask() {
    int sum = co_await asyncAdd(10, 20);
    int product = co_await asyncMultiply(sum, 2);
    std::cout << "Composed result: " << product << "\n";
}
```

Highlights:

- `co_await` can now be used directly with **standard task-like objects** (`std::future`, `std::task`).
- **Results from subtasks propagate automatically** to parent coroutines.
- Enables **pipeline-style programming**, where each stage awaits the previous one.

9.2.5 Parallel Composition

C++23 also supports **executing coroutines concurrently** while composing them:

```
Task parallelTasks() {
    auto t1 = subTask(1);
    auto t2 = subTask(2);

    co_await t1;
    co_await t2;
}
```

- Here, `t1` and `t2` can run concurrently, suspending independently.
- Parent coroutine resumes **after both subtasks complete**, allowing **efficient async concurrency**.

9.2.6 Best Practices for Composable Coroutines

1. **Return awaitable objects** from all subtasks to enable composition.
2. **Avoid blocking calls** inside subtasks; use `co_await` or lightweight awaitables.
3. **Separate concerns**: each coroutine should perform one logical operation.
4. **Measure and profile suspensions** when composing multiple tasks to avoid performance bottlenecks.
5. **Use C++23 standard task types** when possible to leverage built-in composability.

9.2.7 Summary

- Composable coroutines make it possible to **build complex async workflows from smaller building blocks**.
- C++23 standardizes **awaitable composition**, supporting **std::future**, **std::task**, and custom awaitables.
- Properly composed coroutines enhance **readability, maintainability, and performance**.
- Composability is key for **real-world async pipelines, data processing, and concurrent systems**.

9.3 Integration with `std::expected`

C++23 introduces `std::expected`, a **powerful utility for handling operations that may fail**, providing a **type-safe alternative to exceptions**. Integrating `std::expected` with coroutines enables **robust, composable error handling** in asynchronous workflows without resorting to exception-heavy designs.

9.3.1 Motivation

Traditionally, coroutines in C++ rely on **exceptions or error codes** to propagate failures:

- **Exceptions:** May introduce overhead and complicate async flows.
- **Error codes:** Often require **manual propagation**, increasing boilerplate.

`std::expected<T, E>` encapsulates either a **successful value (`T`)** or an **error (`E`)**, simplifying **error propagation and handling** in coroutine chains.

9.3.2 Using `std::expected` with Coroutines

```
#include <expected>
#include <coroutine>
#include <iostream>
#include <string>

struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    void resume() { handle.resume(); }
};

// Coroutine returning std::expected
```

```
std::expected<int, std::string> asyncDivide(int a, int b) {
    if (b == 0) co_return std::unexpected("Division by zero");
    co_return a / b;
}

Task example() {
    auto result = co_await asyncDivide(10, 0); // co_await for composable workflow
    if (!result) {
        std::cout << "Error: " << result.error() << "\n";
        co_return;
    }
    std::cout << "Result: " << *result << "\n";
}
```

Explanation:

- `asyncDivide` returns `std::expected<int, std::string>`.
- Errors propagate safely without exceptions.
- `co_await` can be used on coroutines returning `std::expected` for **composable async error handling**.

9.3.3 Advantages

1. **Simplified error propagation**: Errors flow through coroutine chains without verbose checks.
2. **Composable coroutines**: Multiple tasks returning `std::expected` can be awaited and combined safely.
3. **Better performance**: Avoids exceptions and stack unwinding overhead in frequent async tasks.
4. **Explicit error handling**: Encourages developers to handle errors at each stage.

9.3.4 Composable `std::expected` Example

```
std::expected<int, std::string> asyncAdd(int x, int y) {
    co_return x + y;
}

std::expected<int, std::string> asyncMultiply(int a, int b) {
```

```
    if (b == 0) co_return std::unexpected("Multiplier cannot be zero");
    co_return a * b;
}

Task composedTask() {
    auto sum = co_await asyncAdd(5, 10);
    if (!sum) { std::cout << "Add failed: " << sum.error() << "\n"; co_return; }

    auto product = co_await asyncMultiply(*sum, 2);
    if (!product) { std::cout << "Multiply failed: " << product.error() << "\n"; co_return; }

    std::cout << "Final result: " << *product << "\n";
}
```

Highlights:

- Each coroutine returns an `std::expected`.
- Errors propagate immediately, stopping further computation.
- Async workflow remains **readable and sequential**, despite potential failures.

9.3.5 Best Practices

1. **Return `std::expected` from coroutines** that may fail instead of throwing exceptions.
2. **Always check `expected` values** before using them to avoid silent errors.
3. **Combine `co_await` with `std::expected`** for safe and composable async chains.
4. For complex workflows, **wrap multiple coroutines** in a **higher-level awaitable** that propagates errors consistently.
5. Use `std::unexpected` for **clear, descriptive error messages**.

9.3.6 Summary

- `std::expected` provides **type-safe error handling** for coroutines.
- Integrating `std::expected` with `co_await` enables **robust, composable asynchronous workflows**.

- Reduces reliance on exceptions and improves **performance and readability**.
- Encourages **predictable, maintainable async programming** in modern C++23 applications.

9.4 Practical Example – Integrated Chain of Coroutines with Error Handling Using `std::expected`

C++23 allows developers to **combine coroutines with `std::expected`** to create **robust asynchronous pipelines** that handle errors gracefully while maintaining readability and composability. This section demonstrates a **realistic workflow** integrating multiple coroutines into a **single chain**, where each step may succeed or fail.

9.4.1 Scenario

Consider a pipeline for **processing data from a network request**:

1. Fetch data asynchronously.
2. Parse and validate the data.
3. Perform a computation on the validated data.
4. Return the final result, while handling any errors at each stage.

Each step is implemented as a **separate coroutine** returning `std::expected`.

9.4.2 Coroutine Implementation

```
#include <coroutine>
#include <expected>
#include <iostream>
#include <string>
#include <thread>
#include <chrono>

// Basic coroutine wrapper
struct Task {
    struct promise_type {
        Task get_return_object() { return Task{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
```

```
    };
    std::coroutine_handle<promise_type> handle;
    explicit Task(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Task() { if (handle) handle.destroy(); }
    void resume() { handle.resume(); }
};

// Step 1: Fetch data
std::expected<std::string, std::string> fetchData(bool fail=false) {
    if (fail) co_return std::unexpected("Network fetch failed");
    co_return "42";  // Simulate fetched data
}

// Step 2: Parse and validate
std::expected<int, std::string> parseData(const std::string& data) {
    try {
        int value = std::stoi(data);
        if (value < 0) co_return std::unexpected("Invalid negative value");
        co_return value;
    } catch (...) {
        co_return std::unexpected("Parsing failed");
    }
}

// Step 3: Compute
std::expected<int, std::string> compute(int x) {
    if (x == 0) co_return std::unexpected("Computation error: zero");
    co_return x * 2;
}

// Integrated coroutine chain
Task processPipeline(bool failFetch=false) {
    auto data = co_await fetchData(failFetch);
    if (!data) { std::cout << "Error: " << data.error() << "\n"; co_return; }

    auto value = co_await parseData(*data);
    if (!value) { std::cout << "Error: " << value.error() << "\n"; co_return; }

    auto result = co_await compute(*value);
    if (!result) { std::cout << "Error: " << result.error() << "\n"; co_return; }

    std::cout << "Pipeline result: " << *result << "\n";
```

```
}
```

9.4.3 Execution

```
int main() {
    std::cout << "Running successful pipeline:\n";
    auto task1 = processPipeline(false);
    task1.resume();

    std::cout << "\nRunning pipeline with network failure:\n";
    auto task2 = processPipeline(true);
    task2.resume();
}
```

Output Example:

```
Running successful pipeline:
Pipeline result: 84

Running pipeline with network failure:
Error: Network fetch failed
```

9.4.4 Key Concepts Illustrated

1. **Stepwise error propagation:** Each coroutine returns `std::expected`; errors immediately halt the pipeline.
2. **Composable design:** Each step is independent but can be composed using `co_await`.
3. **Sequential async flow:** Despite multiple stages, the coroutine reads **like synchronous code**, improving readability.
4. **Graceful error handling:** No exceptions thrown; all errors are **explicit and type-safe**.
5. **Flexible for real async integration:** Each step could be replaced with **true asynchronous operations** using `co_await std::future` or `std::task`.

9.4.5 Best Practices for Integrated Pipelines

1. **Return `std::expected` from all coroutines that may fail.**

2. **Check each result** immediately after `co_await` to propagate errors safely.
3. **Encapsulate reusable stages** into independent coroutines.
4. **Avoid blocking operations**; use asynchronous awaitables for I/O-bound tasks.
5. **Log or handle errors locally** to maintain clarity of the pipeline.

9.4.6 Summary

- Integrated coroutine chains with `std::expected` provide a **robust framework for error-resilient async pipelines**.
- Each stage **reports errors explicitly**, simplifying debugging and maintenance.
- C++23 allows **clean, composable, and high-performance workflows** using `co_await` and `std::expected`.
- This approach is ideal for **network I/O, data processing, or computation pipelines**, where **failures are expected and must be handled gracefully**.

Conclusion

Quick Summary of Learned Concepts

After progressing through this guide, you have explored **the full spectrum of modern C++ coroutines**, from foundational concepts to advanced C++23 features. This summary consolidates the key takeaways, practical insights, and patterns you can apply in high-performance development.

Understanding Coroutines

- **Definition:** Coroutines are special functions that can **suspend and resume execution**, allowing asynchronous, lazy, or incremental computation.
- **Core keywords:**
  - `co_await` – waits for an awaitable to complete.
  - `co_yield` – produces intermediate results lazily.
  - `co_return` – returns the final result from a coroutine.
- **Differentiation:** Coroutines differ from threads and traditional functions by **allowing cooperative suspension** without blocking the CPU or creating full threads.

Lifecycle and Structure

- **Promise type:** Customizes the behavior of the coroutine, including suspension and return value handling.

- **Handle type (`std::coroutine_handle`)**: Provides control over resumption, suspension, and destruction of the coroutine.
- **Lifecycle stages:**
  - **Initial suspend** – coroutine begins in a suspended or running state.
  - **Resumption** – continues execution after suspension.
  - **Final suspend** – prepares for cleanup.
  - **Destruction** – ensures resources are released.

Practical insight: Proper lifecycle management prevents dangling handles and memory leaks, critical for high-performance applications.

Awaitables and Awaiters

- **Awaitable concept:** Any object implementing `await_ready`, `await_suspend`, and `await_resume`.
- **Custom awaiters:** Allow precise control over suspension and resumption, enabling **integration with timers, resources, and async APIs**.
- **Await transformations:** Enable converting objects into awaitables, enhancing composability and reusability.
- **Practical applications:** Implementing **timers, network waits, and resource synchronization** using lightweight, efficient patterns.

Generators

- **Using `co_yield`:** Produce sequences lazily without storing all values upfront.
- **Advantages over traditional solutions:** Reduced memory footprint, simplified iterator logic, and improved readability.
- **Practical examples:** Generating **number sequences**, reading large files, or streaming network data efficiently.

Tasks and Asynchronous Operations

- **Creating async tasks:** Coroutines can wrap asynchronous operations for **non-blocking execution**.
- **Integration with `std::future` and `std::task`:** C++23 allows direct awaiting of standard async types.
- **Managing multiple tasks:** `co_await` enables both **sequential and concurrent task orchestration**, simplifying complex workflows.
- **Practical applications:** Performing **network requests, file I/O, and CPU-bound computations** asynchronously.

Real-World Applications

- **Concurrent pipelines:** Process streams of data in stages, suspending intelligently between stages.
- **Game loops:** Update and render cycles can be implemented using coroutines for better control and efficiency.
- **Network and I/O-heavy systems:** Coroutines provide high-throughput, low-latency handling of connections and requests.
- **Practical example:** A **data stream pipeline** suspending only when necessary, maximizing CPU utilization.

Best Practices

- **Exception handling:** Use `try/catch` inside coroutines or integrate with `std::expected` for safer error handling.
- **Memory management:** Ensure proper destruction of coroutines; avoid dangling handles.
- **Avoid common pitfalls:** Undefined behavior can arise from mismanaged handles, improper suspension, or returning references to local objects.
- **Resilient design:** Combine exception handling, careful resource management, and modular awaitables.

Advanced C++23 Features

- **Improved awaitables:** Operator `co_await` transformations, lazy evaluation, and task composability.
- **Composable coroutines:** Build larger workflows from smaller, reusable coroutines.
- **Integration with `std::expected`:** Type-safe error propagation, enabling robust multi-stage async pipelines.
- **Practical chains:** Pipelines that handle **network fetch, parsing, computation**, and error handling seamlessly.

Takeaway

By mastering modern C++ coroutines and the latest C++23 features, you can:

1. Write **readable, maintainable asynchronous code**.
2. **Optimize CPU and memory usage** by avoiding unnecessary threads.
3. Build **resilient, high-performance pipelines** for I/O and compute-intensive tasks.
4. Combine **task composability and robust error handling** for production-quality applications.

This foundation equips you to **leverage coroutines in real-world projects**, from **high-frequency data processing** to **scalable network services**, making C++23 coroutines a powerful tool in your development toolkit.

Practical Steps to Implement Coroutines in Large Projects

Integrating coroutines into **large-scale C++ projects** requires careful planning, modular design, and adherence to best practices. Modern C++23 coroutines provide **high-performance, composable, and maintainable asynchronous programming patterns**, but without a structured approach, they can lead to **complexity, resource leaks, and subtle bugs**. This section outlines practical steps for successfully adopting coroutines in large projects.

Evaluate Where Coroutines Make Sense

Before integrating coroutines, identify areas where **asynchronous execution or lazy computation** is beneficial:

- **I/O-bound tasks:** Network requests, file or database operations, and streaming data.
- **CPU-bound tasks:** Large computations that can be split into suspensible subtasks.
- **Pipelines:** Multi-stage processing where stages can suspend without blocking threads.
- **Generators:** Lazy evaluation of sequences or data streams.

Tip: Avoid introducing coroutines for trivial synchronous operations, as the overhead may outweigh benefits.

Design Modular Coroutine Units

- **Encapsulate tasks:** Each coroutine should perform a **single, well-defined task**.
- **Return awaitables:** Make each coroutine return `std::future`, `std::task`, or a **custom awaitable**, allowing it to be composed.
- **Use `std::expected` for error handling:** Ensures **safe propagation of failures** through coroutine chains.
- **Example:** Separate coroutines for fetching, parsing, validating, and processing data.

```
auto fetchDataAsync() -> std::expected<std::string, std::string>;
auto parseDataAsync(const std::string&) -> std::expected<int, std::string>;
auto computeAsync(int) -> std::expected<int, std::string>;
```

Structure Coroutine Pipelines

Large projects often require **sequences of asynchronous operations**:

1. **Parent coroutine:** Orchestrates the workflow.
2. **Child coroutines:** Perform individual tasks.
3. **Error handling:** Each stage checks `std::expected` results.
4. **Composability:** `co_await` child coroutines in sequence or concurrently.

Pattern Example:

```
Task processDataPipeline() {
    auto data = co_await fetchDataAsync();
    if (!data) co_return;

    auto parsed = co_await parseDataAsync(*data);
    if (!parsed) co_return;

    auto result = co_await computeAsync(*parsed);
    if (!result) co_return;

    std::cout << "Pipeline result: " << *result << "\n";
}
```

- This approach **keeps asynchronous logic readable** and **modular**, even as the number of stages grows.

Implement Coroutine Utilities

For large projects, consider building **supporting utilities**:

- **Coroutine wrappers:** Simplify creation and resumption of coroutines.
- **Custom awaitables:** For timers, I/O, or resource synchronization.
- **Task combinators:** Utilities to `co_await` multiple coroutines concurrently or sequentially.
- **Logging and profiling hooks:** Track suspensions, resumptions, and execution times.

Example: A utility to run multiple tasks concurrently:

```
template<typename... Tasks>
auto co_await_all(Tasks&&... tasks) {
    co_await (... , tasks);  // Await all tasks in sequence
}
```

Integrate Coroutines with Existing Architecture

- **Threading models:** Coroutines can **replace or complement threads**, reducing blocking and context switching.
- **Event loops:** Integrate coroutines with existing event-driven frameworks.
- **Legacy APIs:** Wrap synchronous APIs in coroutines using **custom awaitables** or **`std::async` adapters**.
- **Data pipelines:** Coroutines can act as stages in pipelines for **I/O-heavy or compute-intensive systems**.

Monitor and Optimize

- **Memory management:** Avoid dangling `std::coroutine_handle`s and properly destroy coroutines.
- **Performance profiling:** Use tools to track **suspension points, CPU usage, and throughput**.
- **Resource allocation:** Reuse coroutine promises and handles for high-frequency tasks.
- **Concurrency tuning:** Balance the number of concurrently running coroutines to avoid oversubscription.

Team and Project Practices

- **Document coroutine patterns:** Ensure all developers understand suspension, resumption, and error handling.
- **Code reviews:** Verify **co_await usage, lifecycle management, and error propagation**.
- **Unit testing:** Test **individual coroutines and pipeline stages**, including error scenarios.

- **Progressive adoption:** Start with isolated modules before extending coroutines to the full project.

Summary

Implementing coroutines in large projects requires:

1. **Targeted usage** – focus on async or lazy operations.
2. **Modular design** – small, composable coroutine units.
3. **Pipeline architecture** – sequential or concurrent stages with `co_await`.
4. **Utilities and combinators** – for task orchestration and awaitable abstraction.
5. **Integration with existing systems** – threads, event loops, and legacy APIs.
6. **Monitoring and profiling** – for performance and memory safety.
7. **Team practices** – documentation, reviews, and unit testing.

Following these practical steps ensures coroutines **scale gracefully in complex applications**, providing **high-performance, maintainable, and robust asynchronous workflows** in modern C++23 projects.

Tips for Professionals to Reduce Complexity and Maximize Efficiency

While coroutines in C++23 offer **powerful asynchronous and lazy computation capabilities**, mismanaged usage in large projects can lead to **unreadable code, resource leaks, and performance issues**. This section provides actionable tips for professional developers to **reduce complexity, improve maintainability, and maximize efficiency** when working with coroutines.

Adopt a Modular Design

- **Single responsibility per coroutine:** Each coroutine should handle **one task** or stage of computation.
- **Composable units:** Coroutines should be **small and composable**, enabling building larger pipelines or workflows without duplicating code.
- **Example:** Separate coroutines for **fetching, validating, computing, and storing data** in a multi-stage pipeline.

```
auto fetchData() -> std::expected<std::string, std::string>;
auto parseData(const std::string&) -> std::expected<int, std::string>;
auto computeResult(int) -> std::expected<int, std::string>;
```

Leverage `std::expected` for Error Propagation

- **Avoid exceptions for expected failures:** Using `std::expected` makes **error handling explicit and type-safe**.
- **Check after each `co_await`:** Prevents subtle bugs from unchecked failures in pipelines.
- **Composable chains:** Multiple coroutines can be combined into **robust async pipelines**.

```
auto data = co_await fetchData();
if (!data) co_return; // early exit on error
```

Minimize Resource Usage

- **Avoid unnecessary heap allocations:** Use **stack-allocated promises** when possible.

- **Reuse coroutine handles** for repetitive tasks.
- **Destroy handles promptly:** Prevent dangling handles by ensuring proper **lifecycle management**.

Professional tip: In performance-critical loops, avoid recreating coroutine handles repeatedly. Instead, **reuse existing structures** or implement **coroutine pools**.

Control Suspension Points Strategically

- **Avoid excessive suspension:** Each `co_await` introduces overhead. Minimize unnecessary suspension to **reduce context-switching costs**.
- **Batch small tasks:** Combine multiple small async operations into a single coroutine when possible.
- **Use `std::suspend_always` and `std::suspend_never` judiciously** to optimize the initial and final suspend points.

Integrate Profiling Early

- **Track resumption and suspension counts:** Helps identify **performance bottlenecks** in large pipelines.
- **Use standard and custom profiling tools** for CPU and memory usage.
- **Log execution paths selectively** to avoid logging overhead while debugging.

Maintain Readability and Maintainability

- **Document coroutine behavior:** Explain **suspension points, awaited resources, and error-handling mechanisms**.
- **Consistent naming conventions:** Prefix or suffix coroutine-returning functions for clarity, e.g., `fetchDataAsync()`.
- **Avoid deep nesting of `co_await`:** Flatten chains using **helper functions** or pipeline orchestration utilities.

Use Advanced C++23 Features Wisely

- **Composable coroutines:** Build reusable components for multi-stage async pipelines.
- **`co_yield` for generators:** Avoid storing large datasets in memory; yield values lazily.
- **Integration with `std::task` and `std::future`:** Simplifies orchestration of complex asynchronous operations.
- **Combine with `std::expected`:** Provides predictable, safe, and readable error handling.

Test and Validate Each Stage

- **Unit-test individual coroutines:** Include success and failure scenarios.
- **Simulate pipeline failures:** Ensure error handling and early exits behave as expected.
- **Performance testing:** Measure **suspension overhead, memory usage, and throughput** before deployment.

Summary of Professional Tips

To maximize efficiency and reduce complexity:

1. **Design modular, composable coroutines.**
2. **Use `std::expected`** for explicit error propagation.
3. **Manage resources carefully** – destroy handles and reuse when possible.
4. **Minimize suspension points** for high-frequency tasks.
5. **Profile early and often** to detect bottlenecks.
6. **Maintain readable, well-documented code.**
7. **Leverage advanced C++23 features** strategically.
8. **Test each coroutine** and pipeline stage thoroughly.

Following these practices ensures that **large-scale, high-performance C++23 projects** remain **maintainable, efficient, and robust**, allowing teams to leverage coroutines effectively without introducing unnecessary complexity.

Appendices

Appendix A : Quick Reference – All Keywords and Types Used in Coroutines

This quick reference compiles **all essential keywords, types, and constructs** used in modern C++23 coroutines. It provides **concise explanations and practical usage notes**, serving as a handy guide while writing or reviewing coroutine code.

Coroutine Keywords

Keyword	Description	Practical Notes
co_await	Suspends the coroutine until an awaitable completes and retrieves its result.	Can await `std::future`, `std::task`, or custom awaitables. Enables asynchronous flow without blocking threads.
co_yield	Produces a value and suspends execution, allowing the caller to resume later.	Used primarily in generator coroutines for lazy evaluation.
co_return	Completes the coroutine and optionally returns a value.	Works with `std::expected`, `std::future`, or custom promise return types.

Keyword	Description	Practical Notes
co_return void	Signals coroutine completion without returning a value.	Often used with task coroutines that perform actions rather than producing results.

Core Coroutine Types

Type	Description	Practical Notes
std::coroutine_handle<Promise>	Represents a handle to the coroutine instance.	Can **resume, destroy, or query** the state of a coroutine. Works with any promise type.
Promise (user-defined)	Controls the behavior of a coroutine (initial/final suspension, return value, exception handling).	Implement **get_return_object**, **initial_suspend**, **final_suspend**, **return_value / return_void**, and **unhandled_exception**.
std::coroutine_traits<ReturnType, Args...>	Determines the associated promise type for a given coroutine return type and arguments.	Customizing `coroutine_traits` allows coroutines to return **std::future, Task, or custom types**.
Awaitable	Any object with `await_ready`, `await_suspend`, and `await_resume`.	Defines suspension logic. Can wrap **timers, network requests, or I/O operations**.
Awaiter	Object returned from `operator co_await` or directly implementing the three awaitable functions.	Provides fine-grained **control over suspension and resumption**.
Generator	Coroutine that produces a **sequence of values** using `co_yield`.	Can be iterated lazily, reducing memory footprint for large data streams.

Type	Description	Practical Notes
Task / std::task (C++23)	Coroutine type representing an asynchronous operation.	Can be awaited and composed for **concurrent or sequential tasks**.
std::future / std::shared_future	Integrates with traditional asynchronous workflows.	Useful for **bridging legacy async APIs** with coroutines.
std::expected	Type-safe error handling mechanism, works well with coroutines.	Enables **explicit propagation of errors** without exceptions.

Suspension Points

Concept	Purpose	Example
Initial suspend	Defines whether the coroutine suspends immediately upon starting.	`std::suspend_never` – runs immediately, `std::suspend_always` – suspends first.
Final suspend	Defines the suspension behavior before coroutine destruction.	`std::suspend_always` – allows handle inspection or cleanup before destruction.

Utility Functions and Operations

Operation	Description	Practical Notes
‘resume()‘	Resumes a suspended coroutine.	Invoked via `std::coroutine_handle<>::resume()`.
‘destroy()‘	Destroys the coroutine frame and releases resources.	Must ensure no dangling handles remain.
‘done()‘	Checks if coroutine has finished execution.	Returns `true` if coroutine is complete.
‘operator co_await‘	Converts an object into an awaitable.	Used for **await transformations** and enhancing composability.

Operation	Description	Practical Notes
‘yield_value()‘	Implemented in promise type to handle `co_yield`.	Produces a value and suspends the coroutine until next resume.
‘return_value()‘	Implemented in promise type to handle `co_return` with a value.	Required for **task results or generator completion**.
‘return_void()‘	Implemented in promise type for `co_return` without value.	Used in **void-returning task coroutines**.

Best Practices for Using These Keywords and Types

1. **Always manage coroutine handles:** Use `resume()`, `done()`, and `destroy()` carefully to avoid **memory leaks or dangling pointers**.
2. **Use `std::expected` in async pipelines:** Provides **predictable, type-safe error handling**.
3. **Prefer modular awaitables:** Simplifies **reuse and testing** in large projects.
4. **Combine `co_yield` with generators for lazy sequences:** Avoids unnecessary memory allocations for large data sets.
5. **Use `co_await` consistently for asynchronous operations:** Ensures **non-blocking, readable workflows**.
6. **Leverage C++23 features:** `std::task`, improved awaitable transformations, and coroutine composability **reduce boilerplate** and improve efficiency.

Summary

This quick reference consolidates **all essential coroutine keywords, types, and utility functions** in Modern C++23:

- **Keywords:** `co_await`, `co_yield`, `co_return`.
- **Core types:** `std::coroutine_handle`, `Promise`, `Awaitable`, `Task`, `Generator`, `std::expected`.
- **Suspension & lifecycle control:** initial/final suspend, resume, destroy, done.
- **Integration & utilities:** `std::future`, `operator co_await`, `yield_value`, `return_value`.

Having this reference allows developers to **write, debug, and maintain coroutines efficiently**, especially in **large-scale, high-performance C++ projects**.

Appendix B: Cheatsheet for Awaitables and Generators Examples

This section provides **ready-to-use code templates** for **awaitables and generators** in Modern C++23. It is designed as a **quick reference for professional developers**, allowing them to implement **common asynchronous patterns and lazy data streams** efficiently.

Simple Custom Awaitable

A **custom awaitable** can suspend a coroutine until a condition is met, such as a timer or resource availability.

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

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

    bool await_ready() const noexcept { return duration.count() <= 0; }

    void await_suspend(std::coroutine_handle<> handle) const {
        std::thread([handle, d = duration]() {
            std::this_thread::sleep_for(d);
            handle.resume(); // Resume after duration
        }).detach();
    }

    void await_resume() const noexcept { }
};

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

```
Task waitTimerExample() {
    std::cout << "Waiting for 2 seconds...\n";
    co_await TimerAwaitable{std::chrono::seconds(2)};
    std::cout << "Timer done!\n";
}

int main() {
    waitTimerExample();
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
```

Key Points:

- `await_ready()` determines if suspension is necessary.
- `await_suspend()` receives the coroutine handle to **resume later**.
- `await_resume()` returns the result of the awaited operation.

Generator Using `co_yield`

Generators produce values lazily without creating full sequences in memory.

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

template<typename T>
struct Generator {
    struct promise_type {
        T current_value;
        std::suspend_always yield_value(T value) {
            current_value = value;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
```

```
    std::coroutine_handle<promise_type> handle;
    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    T next() {
        handle.resume();
        return handle.done() ? T{} : handle.promise().current_value;
    }

    bool done() const { return handle.done(); }
};

Generator<int> numberGenerator(int n) {
    for (int i = 1; i <= n; ++i) {
        co_yield i; // produce each number
    }
}

int main() {
    auto gen = numberGenerator(5);
    while (!gen.done()) {
        std::cout << gen.next() << " ";
    }
}
```

Key Points:

- `co_yield` suspends the coroutine after producing a value.
- The caller **resumes the generator** explicitly using `resume()` or `next()`.
- Efficient for **large sequences or streaming data**.

Combining Awaitables and Generators

Generators can also produce asynchronous results using `co_await`:

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

struct AsyncGenerator {
    struct promise_type {
        int value;
```

```
        AsyncGenerator get_return_object() {
            return AsyncGenerator{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int val) {
            value = val;
            return {};
        }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    AsyncGenerator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~AsyncGenerator() { if (handle) handle.destroy(); }

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

    int current() { return handle.promise().value; }
};

AsyncGenerator asyncNumbers() {
    for (int i = 1; i <= 5; ++i) {
        co_await std::suspend_always{}; // simulate async step
        co_yield i;
    }
}

int main() {
    auto gen = asyncNumbers();
    while (gen.next()) {
        std::cout << gen.current() << " ";
    }
}
```

Key Points:

- Mixing `co_await` and `co_yield` allows **lazy and asynchronous sequences**.
- Each step can suspend for **I/O, timers, or tasks**, then yield a value.

Practical Tips for Professionals

1. **Use templates for generic generators and awaitables** – reduces boilerplate.
2. **Combine `std::task` and generators** for high-performance async pipelines.
3. **Always destroy coroutine handles** to prevent memory leaks.
4. **Integrate error handling** with `std::expected` when awaiting external resources.
5. **Profile suspension points** for high-frequency generators to optimize performance.

Summary

This cheatsheet provides **ready-to-use patterns** for:

- Custom awaitables (`co_await`)
- Generators (`co_yield`)
- Combined asynchronous generators

Using these templates allows professional developers to **rapidly implement efficient, composable, and maintainable coroutine patterns** in large-scale C++23 projects.

Appendix C: Additional Resources and References

This section provides a curated collection of **conceptual references, tools, and resources** to help professional developers **deepen their understanding of Modern C++23 coroutines**, implement advanced patterns, and solve real-world problems effectively. These resources include **books, standards documents, open-source projects, and toolkits** relevant to coroutine development.

C++ Standards and Specification Documents

- **ISO C++ Standards (C++20 and C++23)**:
  - The official standards define **coroutine behavior, promise types, coroutine handles, and awaitable semantics**.
  - Essential for understanding **core mechanics**, standard library support, and subtle language nuances.
- **Technical Specifications (TS) for Coroutines**:
  - Provides early design insights and implementation strategies for coroutine types, awaitable patterns, and integration with `std::future` and `std::task`.
  - Useful for developers implementing **custom coroutine frameworks or promise types**.

Recommended Books

1. **"C++20: The Complete Guide"** – Provides an in-depth chapter on **coroutines, asynchronous programming, and generators**, including practical examples.
2. **"Modern C++ Design Patterns"** – Explains **patterns applicable to coroutine composition, error handling, and resource management**.
3. **"Programming with C++20 Coroutines"** – Focused exclusively on **coroutine syntax, lifecycle, and performance considerations**, with real-world examples.
4. **"The C++ Standard Library: A Tutorial and Reference"** – Explains how **coroutines integrate with STL components**, such as `std::expected`, `std::future`, and ranges.

Open-Source Libraries and Frameworks

- **cppcoro**:

  - Advanced coroutine utilities, including **generator types, async I/O, and synchronization primitives**.
  - Provides reference implementations for high-performance coroutine patterns.

- **asio / Boost.Asio**:
  - Integrates **network I/O and asynchronous operations** with coroutines using `co_await`.
  - Illustrates **real-world pipeline patterns and task orchestration**.

- **Microsoft GSL (Guideline Support Library)**:
  - Provides utility types like `span` and `finally` for **resource-safe coroutine patterns**.
  - Useful for **exception-safe async workflows**.

- **libunifex / stdexec**:
  - Modern **asynchronous execution frameworks** designed for coroutines and composable tasks.
  - Demonstrates **highly optimized, low-latency async pipelines**.

Practical Tools for Coroutines

- **Compiler support**:
  - **Clang 15+**, **GCC 12+**, and **MSVC 19.3+** fully support C++20/23 coroutines.
  - Ensure **flags for coroutines** are enabled for performance tuning and proper lifetime management.

- **Profiling & Debugging**:
  - **Valgrind**, **AddressSanitizer**, **ThreadSanitizer** – for detecting memory leaks and data races in asynchronous pipelines.
  - **Custom logging of suspension/resumption points** – critical for **performance optimization in generators and task coroutines**.

- **Unit testing frameworks**:
  - **Catch2** or **Google Test** – can test coroutines by **resuming and inspecting states** step by step.
  - Facilitates **robust testing of awaitable chains and generator sequences**.

Key Concepts for Further Study

1. **Composable coroutines and pipelines** – chaining multiple coroutines with minimal boilerplate.
2. **`std::task` and `std::future` integration** – bridging coroutines with traditional async patterns.
3. **`std::expected` in coroutine pipelines** – building **type-safe error-handling workflows**.
4. **Resource management and RAII in async pipelines** – preventing leaks in complex suspensions.
5. **Generators and lazy evaluation** – optimizing memory usage and reducing runtime overhead.

Practical Tips for Using These Resources

- **Start with standard and TS documents** to ensure your implementations comply with the language specification.
- **Use open-source libraries** like cppcoro or Asio to **understand patterns that are already optimized and battle-tested**.
- **Combine books and online reference code** to build **hands-on experience with real-world pipelines, tasks, and generators**.
- **Profile and test coroutine-heavy code frequently** to avoid subtle lifetime and suspension bugs.

Summary

This section serves as a **centralized reference** for developers seeking to:

- Deepen their understanding of **C++23 coroutines**
- Explore **practical, reusable coroutine patterns**
- Access **tools for debugging, profiling, and testing**
- Learn from **battle-tested open-source implementations**

By leveraging these references, professional developers can **accelerate adoption of modern coroutines**, implement **robust and high-performance asynchronous pipelines**, and maintain **clean, maintainable code** in large-scale projects.