Understanding C++ Coroutine Implementation

Alexander Obregon · Follow

11 min read · Aug 17, 2024

Image Source

Introduction

Coroutines are an advanced feature in C++ that allow for more efficient asynchronous programming. In this beginner-friendly article, we will take a look at the fundamental concepts behind the implementation of coroutines in C++, understand their syntax, examine their use cases, and see how they can be employed to create more efficient asynchronous programs.

C++ Coroutines Basics

Coroutines were introduced in C++20, bringing a powerful new tool to the language for managing asynchronous operations. Traditionally, asynchronous programming in C++ required the use of threads or callbacks, which could quickly become complex and difficult to manage. Coroutines simplify this process by allowing

functions to be suspended and resumed, making it easier to write code that handles tasks like file I/O, networking, or even concurrent operations.

Basically, a coroutine is a function that can yield control back to the caller without losing its state. This means that a coroutine can pause its execution at a certain point, return a value (or wait for some asynchronous event), and then continue executing from where it left off.

Coroutines are not a new concept in programming, having been used in languages like Python and JavaScript for years. However, their introduction into C++ represents an evolution in how developers can handle asynchronous tasks in this performance-critical language.

Syntax and Mechanics of Coroutines in C++

Understanding the syntax and mechanics of coroutines in C++ is important for effectively utilizing them in your programs. Let's take a deeper look into the components that make up coroutines in C++, how they work, and how you can use them to simplify your code.

Coroutine Functions and Suspension Points

A coroutine function is a special type of function in C++ that can be suspended and resumed at specific points during its execution. This is achieved using three key keywords: `co_await`, `co_yield`, and `co_return`.

- **`co_await`**: This keyword is used to suspend the execution of the coroutine until a particular condition is met or an asynchronous operation is completed.
- **`co_yield`**: This keyword allows the coroutine to produce a value and suspend its execution. It can be resumed later, continuing from the point after `co_yield`.
- **`co_return`**: This keyword is used to return a value from the coroutine and finalize its execution.

Here's a basic example of a coroutine that uses `co_await`:

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

struct MyCoroutine {
    struct promise_type {
```

```
        MyCoroutine 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(); }
    };
};

MyCoroutine myCoroutineFunction() {
    std::cout << "Start of coroutine\n";
    co_await std::suspend_always{};
    std::cout << "Resume of coroutine\n";
}
```

In this example, the coroutine starts execution, reaches the `co_await` keyword, and suspends itself. The `std::suspend_always` object tells the coroutine to suspend every time it encounters this point. When resumed, it continues from where it left off.

Detailed Breakdown of the Coroutine Lifecycle

The lifecycle of a coroutine in C++ involves several distinct phases, each of which plays a crucial role in the function's ability to suspend and resume execution. Understanding this lifecycle is key to mastering coroutines.

Initialization Phase:

- When a coroutine is first called, it doesn't start executing immediately. Instead, its initialization phase sets up the coroutine environment. The `initial_suspend` method in the promise type determines whether the coroutine should start right away or wait for an explicit signal to begin.

Execution Phase:

- Once the coroutine is initialized, it begins execution until it hits a suspension point, such as `co_await`, `co_yield`, or `co_return`. During this phase, the coroutine performs its intended operations up to the point where it needs to pause.
- The state of the coroutine, including local variables and the current point of execution, is saved. This allows the coroutine to be resumed later from the exact same state.

Suspension Phase:

- In this phase, the coroutine is suspended, meaning it temporarily stops execution and yields control back to the caller or another coroutine. The suspension can occur due to an explicit `co_await` or `co_yield`, or even at the start or end of the coroutine depending on the `initial_suspend` and `final_suspend` methods.
- While suspended, the coroutine's state is maintained in memory, ready to be resumed at a later time.

Resumption Phase:

- The coroutine can be resumed at any point after being suspended. Resuming a coroutine restores its saved state, allowing it to continue executing as if it had never paused. The resumption phase is initiated by the coroutine's caller or by another coroutine that controls its execution.

Completion Phase:

- A coroutine eventually completes its execution, either by reaching its natural end or by executing a `co_return` statement. The completion phase involves cleaning up any resources used by the coroutine, and the `final_suspend` method is called to make sure proper suspension before the coroutine fully exits.

The Role of Promise Type in Coroutines

The promise type is a critical component in the coroutine machinery. It acts as a bridge between the coroutine and the code that interacts with it. The promise type is responsible for several key operations that determine how the coroutine behaves.

get_return_object:

- This method is called when the coroutine is first created. It typically returns a handle to the coroutine itself, which can be used by the caller to control and monitor the coroutine's execution. The handle might allow resuming, destroying, or retrieving the result from the coroutine.

initial_suspend and final_suspend:

- The `initial_suspend` method decides whether the coroutine should start running immediately or if it should wait until an explicit resume call is made.

Similarly, the `final_suspend` method determines if the coroutine should perform any final suspension steps before completing.

- `initial_suspend` can return either `std::suspend_always`, which means the coroutine will suspend before doing anything, or `std::suspend_never`, which means it will start executing immediately.
- `final_suspend` often returns `std::suspend_always` to make sure that the coroutine cleans up its resources properly before exiting.

return_void and return_value:

- Depending on the coroutine's design, it may or may not return a value. The `return_void` method is used when the coroutine doesn't return anything, while `return_value` is used when the coroutine returns a specific value.

unhandled_exception:

- If an exception occurs inside a coroutine and isn't caught, the `unhandled_exception` method is invoked. This method usually terminates the program or performs some error handling, depending on the specific implementation.

Here's an example that demonstrates the promise type with a coroutine returning a value:

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

struct ReturnValue {
    struct promise_type {
        int value;
        ReturnValue get_return_object() { return ReturnValue{this}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_value(int v) { value = v; }
        void unhandled_exception() { std::terminate(); }
    };

    promise_type* promise;
    ReturnValue(promise_type* p) : promise(p) {}
    int get() { return promise->value; }
};
```

```
ReturnValue computeValue() {
    co_return 42;
}

int main() {
    ReturnValue result = computeValue();
    std::cout << "Computed Value: " << result.get() << std::endl;
}
```

In this example, the coroutine `computeValue` returns an integer value using `co_return`. The promise type manages the returned value, and the `get_return_object` method returns a `ReturnValue` object that the caller can use to retrieve the result.

Custom Awaitable Types

In C++, coroutines are not limited to working with built-in types like `std::suspend_always` or `std::suspend_never`. You can create custom awaitable types that define specific suspension and resumption behavior.

An awaitable type is any type that implements the following methods:

- **await_ready**: Determines if the coroutine should suspend or continue without suspension.
- **await_suspend**: Defines what happens when the coroutine is suspended. This can involve storing the coroutine handle, initiating an asynchronous operation, or interacting with other coroutines.
- **await_resume**: Defines what happens when the coroutine is resumed, often returning a value or performing some final action before execution continues.

Here's an example of a custom awaitable type:

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

struct Timer {
    std::chrono::milliseconds duration;
    Timer(std::chrono::milliseconds d) : duration(d) {}
```

```
    bool await_ready() const { return false; }
    void await_suspend(std::coroutine_handle<> h) const {
        std::thread([h, this]() {
            std::this_thread::sleep_for(duration);
            h.resume();
        }).detach();
    }
    void await_resume() const {}
};

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(); }
    };

    void run() {
        std::cout << "Starting timer...\n";
        co_await Timer{std::chrono::seconds(2)};
        std::cout << "Timer finished.\n";
    }
};

int main() {
    Task task;
    task.run();
    std::this_thread::sleep_for(std::chrono::seconds(3)); // Wait for coroutine
}
```

In this example, `Timer` is a custom awaitable type that suspends the coroutine for a specified duration. The coroutine resumes automatically after the specified time, allowing you to integrate time-based delays or other asynchronous tasks into your coroutine workflow.

Use Cases for C++ Coroutines with Examples

Coroutines provide several use cases where they offer considerable advantages over traditional programming techniques, particularly in handling asynchronous operations, managing concurrency, and creating more maintainable and readable code. Below, we explore some of the key use cases for C++ coroutines, focusing on how they can be effectively applied in real-world scenarios.

Asynchronous I/O Operations

One of the most powerful use cases for coroutines is managing asynchronous I/O operations. In traditional C++ programming, asynchronous I/O often involves callbacks, which can quickly lead to “callback hell” — a scenario where nested callbacks become difficult to manage and read. Coroutines solve this problem by allowing you to write asynchronous code in a sequential manner, making it more intuitive and easier to maintain.

Consider a scenario where you need to perform multiple asynchronous file operations, such as reading and processing data from multiple files concurrently. With coroutines, you can do this without blocking the main thread, allowing your program to remain responsive.

Here’s an example of how coroutines can be used to perform asynchronous file reading:

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

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

    promise_type* promise;
    FileReader(promise_type* p) : promise(p) {}

    struct Awaiter {
        std::ifstream file;
        std::string line;

        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<> h) {
            while (std::getline(file, line)) {
                std::cout << "Processing: " << line << std::endl;
                h.resume();
            }
        }
        void await_resume() {}
    };
```

```
    Awaiter readAsync(const std::string& filename) {
        Awaiter awaiter;
        awaiter.file.open(filename);
        return awaiter;
    }
};

FileReader processFiles() {
    co_await FileReader{}.readAsync("file1.txt");
    co_await FileReader{}.readAsync("file2.txt");
}

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

In this example, the `FileReader` coroutine reads and processes data from multiple files asynchronously. The coroutine suspends while waiting for each line to be read, allowing other tasks to proceed concurrently. This non-blocking approach is particularly useful in applications like servers or data processing pipelines where I/O operations are common.

Concurrency and Task Coordination

Coroutines are also highly effective in scenarios where you need to manage concurrency or coordinate multiple tasks. Traditional thread-based concurrency can be complex to manage, especially when dealing with synchronization and shared resources. Coroutines offer a simpler model for concurrency by allowing multiple tasks to be interleaved without requiring explicit thread management.

For example, in a game loop or a simulation, you might need to manage multiple entities that perform actions asynchronously, such as moving, updating, or interacting with each other. Coroutines allow you to write this logic in a straightforward manner, without the need for complex thread management or locking mechanisms.

Here's an example that demonstrates using coroutines for task coordination in a simple simulation:

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

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

    promise_type* promise;
    Task(promise_type* p) : promise(p) {}

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

        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<> h) {
            std::thread([h, this]() {
                std::this_thread::sleep_for(delay);
                h.resume();
            }).detach();
        }
        void await_resume() {}
    };

    Awaiter moveEntity(int id, int distance) {
        std::cout << "Entity " << id << " moving " << distance << " units\n";
        return Awaiter{std::chrono::milliseconds(500 * distance)};
    }

    Awaiter updateEntity(int id) {
        std::cout << "Entity " << id << " updating\n";
        return Awaiter{std::chrono::milliseconds(100)};
    }
};

Task runSimulation() {
    co_await Task{}.moveEntity(1, 5);
    co_await Task{}.updateEntity(1);
    co_await Task{}.moveEntity(2, 3);
    co_await Task{}.updateEntity(2);
    co_await Task{}.moveEntity(1, 2);
}

int main() {
    runSimulation();
    std::this_thread::sleep_for(std::chrono::seconds(5)); // Wait for simulatic
}
```

In this example, the `runSimulation` coroutine manages the movement and updating of two entities in a simulation. Each task (movement or update) is suspended for a specified delay, allowing other tasks to proceed in the meantime. This approach simplifies the management of concurrent tasks, making the code easier to read and maintain.

Building Event-Driven Systems

Event-driven systems, such as user interfaces or real-time processing systems, often rely on a continuous loop that handles events like user input, network messages, or sensor data. Coroutines are particularly well-suited for building such systems because they allow you to write event-driven code in a linear, easy-to-follow manner.

For instance, in a GUI application, you might have various event handlers that respond to user actions like clicks, key presses, or mouse movements. With coroutines, you can structure these event handlers as asynchronous functions that wait for specific events, process them, and then return control to the main event loop.

Here's an example of using coroutines in an event-driven system:

```
#include <iostream>
#include <coroutine>
#include <functional>
#include <queue>

struct Event {
    int id;
    std::string data;
};

struct EventQueue {
    std::queue<Event> events;

    void push(const Event& event) {
        events.push(event);
    }

    Event pop() {
        if (!events.empty()) {
            Event event = events.front();
            events.pop();
            return event;
        }
```

```
            return Event{-1, ""}; // No events
        }
    };

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

        promise_type* promise;
        EventHandler(promise_type* p) : promise(p) {}

        struct Awaiter {
            EventQueue& queue;

            bool await_ready() { return !queue.events.empty(); }
            void await_suspend(std::coroutine_handle<> h) {
                while (queue.events.empty()) {
                    // Simulate waiting for an event
                    std::this_thread::sleep_for(std::chrono::milliseconds(100));
                }
                h.resume();
            }
            Event await_resume() {
                return queue.pop();
            }
        };

        Awaiter waitForEvent(EventQueue& queue) {
            return Awaiter{queue};
        }
    };

    EventHandler handleEvents(EventQueue& queue) {
        while (true) {
            Event event = co_await EventHandler{}.waitForEvent(queue);
            std::cout << "Handling event " << event.id << ": " << event.data << std
        }
    }

    int main() {
        EventQueue queue;
        std::thread eventProducer([&queue]() {
            for (int i = 0; i < 5; ++i) {
                queue.push({i, "EventData" + std::to_string(i)});
                std::this_thread::sleep_for(std::chrono::milliseconds(300));
            }
        });
```

```
    handleEvents(queue);
    eventProducer.join();
}
```

In this example, the `EventHandler` coroutine waits for events in a queue and processes them as they arrive. The coroutine suspends itself if no events are available, resuming once an event is pushed into the queue. This pattern is particularly useful in scenarios where events arrive asynchronously, such as in GUI applications or real-time data processing systems.

Simplifying State Machines

State machines are a common pattern in software engineering, used to manage the state transitions of an object or system. However, implementing state machines can be challenging, especially when the states involve asynchronous operations. Coroutines simplify the creation and management of state machines by allowing each state to be represented as a coroutine that can suspend and resume based on the state transitions.

For example, consider a state machine for a network connection that can be in one of several states: `Connecting`, `Connected`, `Disconnecting`, or `Disconnected`. Each state might involve waiting for a network event, such as a successful connection or a disconnection request. Using coroutines, you can model each state as a coroutine that handles its specific operations and then transitions to the next state.

Here's an example of a coroutine-based state machine:

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

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

    promise_type* promise;
    ConnectionState(promise_type* p) : promise(p) {}
```

```
    struct Awaiter {
        std::string state;

        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<> h) {
            std::cout << "Current state: " << state << std::endl;
            h.resume();
        }
        void await_resume() {}
    };

    Awaiter handleState(const std::string& newState) {
        return Awaiter{newState};
    }
};

ConnectionState manageConnection() {
    co_await ConnectionState{}.handleState("Connecting");
    co_await ConnectionState{}.handleState("Connected");
    co_await ConnectionState{}.handleState("Disconnecting");
    co_await ConnectionState{}.handleState("Disconnected");
}

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

In this example, the `manageConnection` coroutine models the state transitions of a network connection. Each state is handled by the coroutine, which suspends and resumes as it transitions between states. This approach makes it easier to manage complex state machines, especially when they involve asynchronous operations.

Conclusion

C++ coroutines offer a powerful and flexible approach to managing asynchronous programming, enabling developers to write more intuitive and maintainable code. By simplifying tasks like asynchronous I/O, concurrency, event handling, and state management, coroutines improve the efficiency and readability of your programs. As you explore and implement coroutines in your projects, you'll find that they provide a strong solution to many of the challenges associated with modern C++ development.

- *C++20 Coroutines*