C++20 Coroutines – Complete Guide

© 2021 RNDr. Šimon Tóth[1]

@HappyCerberus @SimonToth

@Happy.Cerberus @SimonToth

@SimonToth83

C++20 brought us initial support for coroutines. In this article, we will go over several examples of coroutines that build upon each other. Word of warning, though, the support in C++20 is mainly targeted at library implementors. C++23 should be bringing additional support that should cover at least the most common use cases.

This article was originally published at:
`https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d`

Also available in video form: `https://www.youtube.com/watch?v=w-dmOHhBX9o`

For all the examples and bash scripts, visit:
`https://github.com/HappyCerberus/article-cpp20-coroutines`

So, what is a coroutine? A coroutine is any function that contains a `co_return`, `co_yield` or `co_await`.

Fundamentally, the C++20 coroutines are syntactic sugar on top of function objects. The compiler will generate a framework of code around your coroutine. This code relies on user-defined return and promise types. Until we get some standard types in C++23, you will need to write these yourself.

A coroutine that does nothing

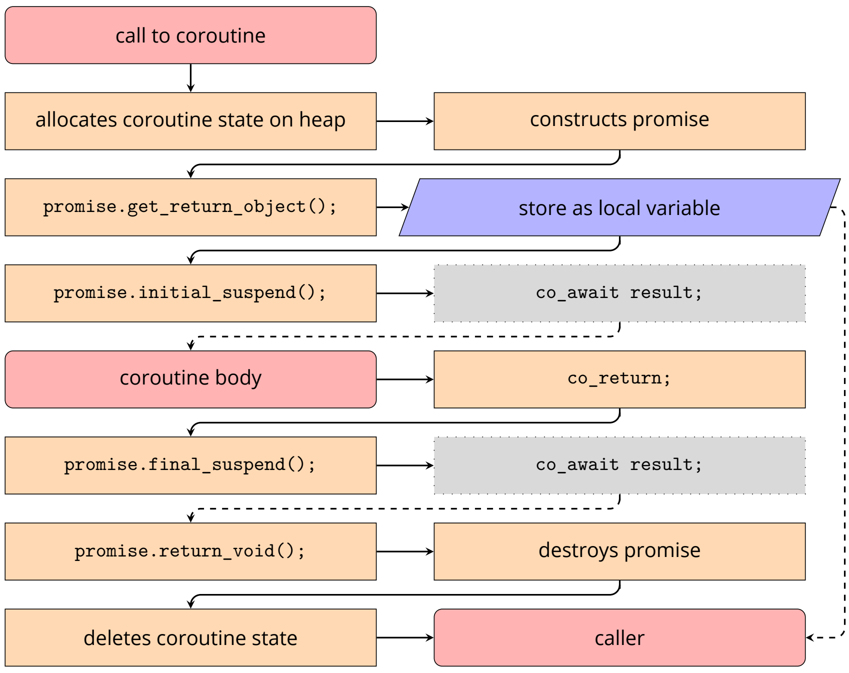
Let's have a look at the simples coroutine you can currently write:

[1] Redistribution and use of this document without modification is permitted. Extended free licenses for educational purposes are available upon request.

simplest/simplest.cc

```cpp
#include <coroutine>
// The caller-level type
struct Task {
    // The coroutine level type
    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() {}
    };
};
Task myCoroutine() {
    co_return; // make it a coroutine
}
int main() {
    Task x = myCoroutine();
}
```

OK, so this might seem like a lot of boilerplate for just calling a coroutine that immediately returns and does nothing. But it is an excellent place to start looking at the framework code generated by the compiler.



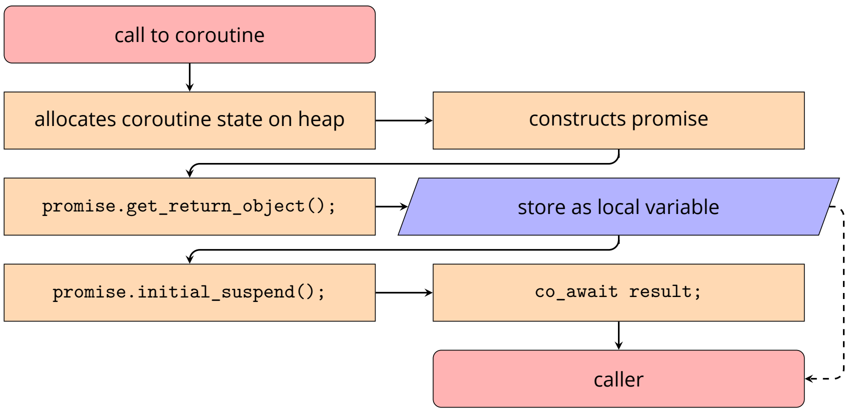
We will discuss `co_await` later, but I have deliberately marked them as dotted in this diagram because calling `co_await` on an instance of `std::suspend_never` will immediately return. You can see this happening if you run the instrumented versions of the examples that you will find in the linked repository[2].

Before we move on, let's explore this example a bit more. What would happen if we changed the `initial_suspend()` return type to `std::suspend_always`?

[2]https://github.com/HappyCerberus/article-cpp20-coroutines

simplest/simplest-leaking.cc

```cpp
#include <coroutine>

// The caller-level type
struct Task {
    // The coroutine level type
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() { }
        void unhandled_exception() { }
    };
};

Task myCoroutine() {
    co_return; // make it a coroutine
}

int main() {
    auto c = myCoroutine();
}
```


call to coroutine
allocates coroutine state on heap
constructs promise
promise.get_return_object();
store as local variable
promise.initial_suspend();
co_await result;
caller

We have created a problem with this change. The coroutine is now stranded (leaked).

Calling `co_await` on the instance of `std::suspend_always{}` results in the suspension of the coroutine. Which in turn returns control to the caller. However, the caller (function main) has no means to resume the coroutine. So let's fix that.

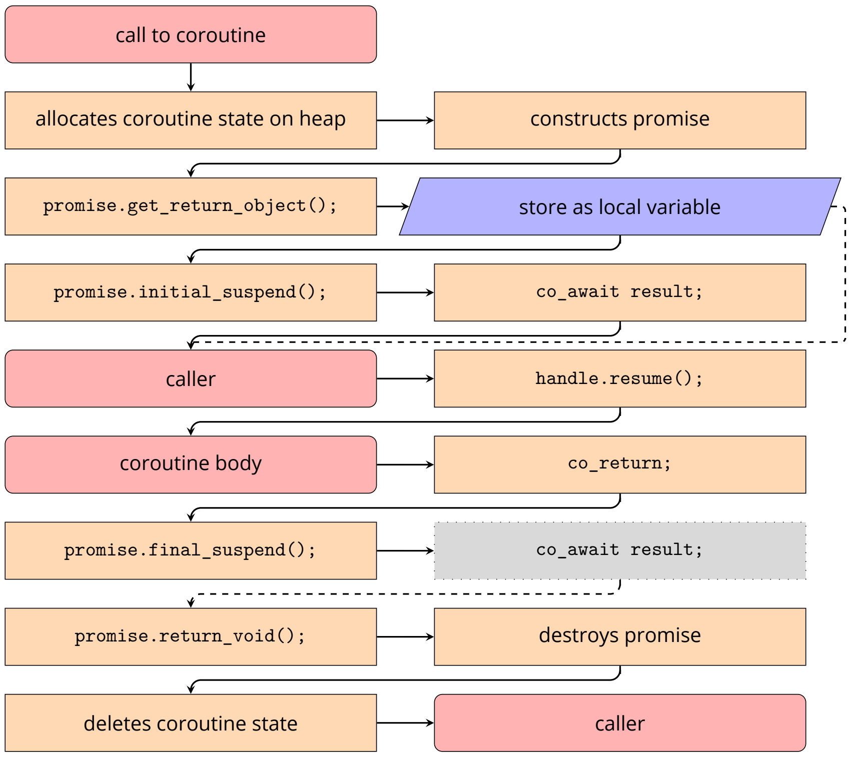
simplest/simplest-fixed.cc

```
#include <coroutine>

// The caller-level type
struct Task {
    // The coroutine level type
    struct promise_type {
        using Handle = std::coroutine_handle<promise_type>;

        Task get_return_object() {
            return Task{Handle::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() { }
        void unhandled_exception() { }
    };

    explicit Task(promise_type::Handle coro) : coro_(coro) {}
    ~Task() {
        if (coro_ && !coro_.done())
            coro_.destroy();
    }

    void destroy() { coro_.destroy(); }
    void resume() { coro_.resume(); }

private:
    promise_type::Handle coro_;
};

Task myCoroutine() {
    co_return; // make it a coroutine
}

int main() {
    auto c = myCoroutine();
    c.resume();
    // c.destroy();
}
```

We need to make the `coroutine_handle` available to caller. To do that, we pass it through the `get_return_object()` call where we create it from the promise instance. Then the caller can either call `resume()` or `destroy()` on the suspended coroutine. Note that calling these methods on a coroutine that isn't suspended is undefined behaviour. It also makes sense to make the caller-level type move-only to avoid confusion over the handle ownership.

simplest/simplest-fixed.cc

```cpp
Task(const Task&) = delete;
Task& operator=(const Task&) = delete;
Task(Task&& t) noexcept : coro_(t.coro_) { t.coro_ = {}; }
Task& operator=(Task&& t) noexcept {
    if (this == &t) return *this;
    if (coro_) coro_.destroy();
    coro_ = t.coro_;
    t.coro_ = {};
    return *this;
}
```



A suspended coroutine exists as pure data. We can therefore handle it as such, for example, pass it between threads. We will rely on this property later when working with `co_await`.

So far, our demonstration coroutine did nothing. So let's move on to the first useful example, which is a generator.

A generator coroutine

Generators rely on the `co_yield` keyword. The `co_yield expr;` expression is shorthand for `co_await promise.yield_value(expr);`. The promise type controls what it means to yield a value and whether / how the coroutine suspends. Here we use the `std::suspend_always` since we want to stop the coroutine until the next `get_next()` call.

generator/generator.cc

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

// The caller-level type
struct Generator {
    // The coroutine level type
    struct promise_type {
        using Handle = std::coroutine_handle<promise_type>;

        Generator get_return_object() {
            return Generator{Handle::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int value) {
            current_value = value;
            return {};
        }
        void unhandled_exception() { }
        int current_value;
    };

    explicit Generator(promise_type::Handle coro) : coro_(coro) {}

    int get_next() {
        coro_.resume();
        return coro_.promise().current_value;
    }

private:
    promise_type::Handle coro_;
};

Generator myCoroutine() {
    int x = 0;
    while (true) {
        co_yield x++;
    }
}

int main() {
    auto c = myCoroutine();
    int x = 0;
    while ((x = c.get_next()) < 10) {
        std::cout << x << "\n";
    }
}
```

Because this coroutine contains an endless loop, it will never be cleaned up naturally. However, since the coroutine only runs inside the `get_next()` call, we can safely call `destroy()` to clean it up in the Generators destructor.

Now, the usability of this coroutine is a bit awkward due to the `get_next()` function. But we can easily add a bit more boilerplate and turn this into a range, like this example on cppreference: https://en.cppreference.com/w/cpp/coroutine/coroutine_handle#Example.

Awaitable

We have already used two awaitables, `std::suspend_never` and `std::suspend_always`. Let's look at how awaitables work and how you can write your own to interface with asynchronous libraries.

Similarly to the promise type, the compiler will generate code around the awaitable type. If the promise type provides a `await_transform(expr);` method, the `co_await expr;` call will be transformed into `co_await promise.await_transform(expr);`. Therefore, the promise type can control which awaitable types are allowed to appear inside the coroutine body and potentially return different awaitables based on the expression (note that expr here doesn't have to be awaitable).

```cpp
struct promise_type {
    // only allow std::suspend_always inside of the coroutine
    std::suspend_always await_transform(std::suspend_always s) {
        return std::suspend_always{};
    }
};
```

The awaitable type is required to provide three methods:

awaitable/awaitable.cc

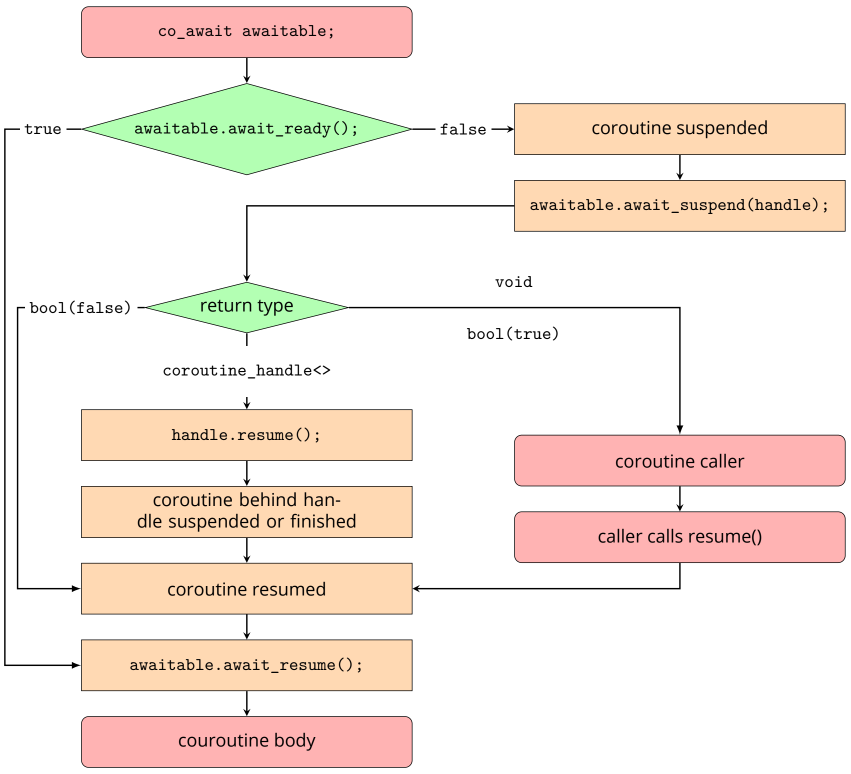
```cpp
struct awaitable {
    bool await_ready();

    // one of:
    void await_suspend(std::coroutine_handle<>) {}
    bool await_suspend(std::coroutine_handle<>) {}
    std::coroutine_handle<> await_suspend(std::coroutine_handle<>) {}

    void await_resume() {}
};
```

Let's have a look at a simple (albeit pointless) example.

awaitable/custom-awaitable.cc

```cpp
struct Sleeper {
    constexpr bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) const {
        auto t = std::jthread([h,l = length] {
            std::this_thread::sleep_for(l);
            h.resume();
        });
    }
    constexpr void await_resume() const noexcept {}
    const std::chrono::duration<int, std::milli> length;
};
```

The coroutine suspends before entering the `await_suspend` method. There is no data race since we create a new thread inside this method (after the suspension point). Also note, that while we are following the "control is returned to the caller" branch in the previous diagram,



we resume the coroutine in the newly spawned thread (not in the caller).

We can then use it within a coroutine:

awaitable/custom-awaitable.cc

```cpp
Task myCoroutine() {
    using namespace std::chrono_literals;
    auto before = std::chrono::steady_clock::now();
    co_await Sleeper{200ms};
    auto after = std::chrono::steady_clock::now();
    std::cout << "Slept for " << (after-before) / 1ms << " ms\n";
}
```

When (not) to use coroutines

Lastly, I want to talk a bit about when you might want to use coroutines and their upsides and downsides. Upfront, generators are useful, and you should use them even now if you have good use cases.

Single-threaded environment

If you are constrained to a single thread, coroutines are a solution for asynchronous processing that you otherwise don't have access to. It should be possible to implement a Javascript-style event-loop using the current coroutine support.

Extremely threaded environments

The other use case lies on the opposite side of the spectrum. Coroutines might offer memory and file descriptor savings if your product requires numerous light threads.

Using an asynchronous library with awaitable support

This one is a bit obvious. If you write user code and your libraries give you pre-built awaitables and support types for coroutines, writing coroutines might be the cleaner approach. You will avoid cluttering your code with mutexes since all synchronisation with coroutines happens behind `co_await` calls.

Capacity controlled environments

So what about the rest? One case where I see issues with coroutines is capacity controlled environments. With synchronous processing, you can control the capacity of your service by controlling the number of threads. When reaching an overload, you can then reject requests when they arrive. Unfortunately, preventing overload gets a lot more complicated with coroutines, where you could quickly end up with an overload situation in the middle of handling a request.

Timeouts

Another tricky feature I haven't figured out how to do with coroutines yet is handling timeouts. For example, the following snippet was in most systems I worked with in the past:

```cpp
Throttler tr;
// mu is absl::Mutex (std::mutex+std:condition_variable combination)
auto has_slot = [&tr]() { return !tr.Full(); }
if (mu.LockWhenWithTimeout(Condition(&has_slot), 200 /*ms*/)) {
  // sucesfull slot grab
  do_work();
} else {
  // timeout branch
  bail_out();
}
mu.Unlock();
```

I have trouble coming up with a coroutine equivalent, specifically safely implementing the timeout branch.

Thank you for reading

Thank you for reading this article. Did you enjoy it? If yes, pass it along to coleagues or classmates that might benefit from reading it.