Yet Another C++ Coroutine Tutorial

2023-04-24

Introduction

I know, everyone has written one. My grandma recently published hers, after telling me how bad she finds the one her hairdresser wrote. I suspect many people have similar experiences of pouring a dozen of these resources into their heads and still wondering how the HELL these fucking things work and when they finally figure it out, they want to write it down too - “properly” this time. This is exactly what I will do in this blog post as well.

Let me note that some of the sentences in this post are frankly just ridiculous. You have to read them carefully, word by word, and think hard about what they say. I think this topic is truly complicated and there is no other way but understanding that complexity. Hopefully this post can help you with that and lead to somewhere where you can operate coroutines in code that does something useful and interesting. If not and you find certain questions not adequately answered or entirely unadressed, feel

free to write me an email and help me make this tutorial better.

I first tried to write this tutorial with some small classes that fake doing async IO, but it became hard to extend the examples to something meaningful, so instead I will use my own async IO library aiopp. Reading along should be enough and this blog post is intended to be self-contained, but if you want to run some of the code, look around in the aiopp repository (especially `examples/`). In fact most of the snippets in this blog post are parts or simplified versions of code in that repository.

For this tutorial I will assume that you have some idea of what coroutines are, but to summarize briefly, they are functions that can suspend themselves, meaning they stop themselves *without returning*, even in the middle of their body and then can later can be resumed, continuing execution at the point they suspended from earlier.

So when in the past you would write something like this (lots of complications and error handling omited):

```
io.read(fd, buffer.data(), buffer.size(),
    [](std::error_code ec, size_t readBytes) {
        buffer.resize(readBytes);
        handleReadData(buffer);
    });
```

With coroutines you would just do this:

```
std::vector<uint8_t> buffer(1024, 0);
const auto [ec, readBytes] = co_await io.read(fd, buffer.data(), buffer.size());
buffer.resize(readBytes);
handleReadData(buffer);
```

It *looks* like synchronous code, but `co_await` will suspend the execution of the function (coroutine) that contains this code, your program does something else (like waiting for the read to complete) and when the read is finished, the coroutine will be resumed and the read can be handled. You don't really have to worry about whether the operation might outlive a buffer or about sharing ownership of the object that initiated the operation (a coroutine) with the IO operation. Of course you have to worry about it once, when you build the framework and that's bad enough, but after that it will not invite you repeatedly to mess it up, like callback based async code does.

In a conventional function the variables live on the stack and when the function returns, the whole stack frame is thrown away. A stack frame of a function will never outlive the stack frame of the function that called it. With a coroutine of course this is not possible, so the coroutine state is typically allocated on the heap.

My First Coroutine

Let's start with the minimal coroutine example I can come up with, explain it and build it up:

```cpp
#include <coroutine>

class BasicCoroutine {
public:
    struct Promise {
        BasicCoroutine get_return_object() { return BasicCoroutine {}; }

        void unhandled_exception() noexcept { }

        void return_void() noexcept { }

        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
    };
    using promise_type = Promise;
};

BasicCoroutine coro()
{
    co_return;
}

int main()
{
```

```
    coro();
}
```

(this example is self-contained)

Not very small, I know. And it doesn't do anything either - Already love them! I promise I'll build it up to something useful, but let's first analyze this snippet. `coro` is a coroutine simply because it contains the keyword `co_return`. If a function contains `co_await`, `co_return` or `co_yield`, it's a coroutine. The return type, even though it doesn't do much, has to be this complicated, because coroutines are rewritten by the compiler in a certain way. This is what a coroutine is rewritten as:

```
ReturnType someCoroutine(Parameters parameter)
{
    auto* frame = new coroutineFrame(std::forward<Parameters>(parameters));
    auto returnObject = frame->promise.get_return_object();
    co_await frame->promise.initial_suspend();
    try
    {
        <body-statements>
    }
    catch (...)
    {
        frame->promise.unhandled_exception();
```

```
    }
    co_await frame->promise.final_suspend();
    delete frame;
    return returnObject;
}
```

So as we said earlier, the coroutine frame, called the "coroutine state" is allocated on the heap, the function parameters of the coroutine and later all local variables and temporaries are saved inside it. Another object is also allocated inside it, which is called the "*promise*". There are different ways the compiler can select a promise type for a coroutine, but what we will use for now is that it will look for `ReturnType::promise_type` (see also the minimal example). It might be worth mentioning, that there are also `coroutine_traits` which you can use to associate a promise type with any coroutine return type, e.g. standard library types. Most of the logic of the coroutine (apart from it's body) is handled through the promise. When we later suspend the coroutine, the coroutine state will also contain information about the point we suspended from, so we can resume to that point later.

Then the return object of the coroutine is created through the promise. `get_return_object` has to return something that is convertible to the return type of the coroutine.

The promise member functions `initial_suspend` and `final_suspend` return *awaitables* and provide an opportunity to control the behavior of the coroutine at the start and

the end. *Awaitables* are just things you can `co_await`. Depending on the return type of your coroutine, the coroutine itself might be awaitable, but other things can be awaitable as well. I would say that most things you `co_await` are not coroutines (e.g. `io.recv()` above is not). I will explain awaitables in detail shortly but for now know that `co_await std::suspend_never` is pretty much a noop and terminates immediately, without suspending the coroutine - it seems the name has been chosen well! A common choice for `initial_suspend` is also `std::suspend_always`, which will result in your coroutine suspending immediately after creating the return object. This would correspond to a "cold-start" or "lazy" coroutine, which you will have to resume at least once first before it starts executing its body. The coroutine from our minimal example is analogously called a "hot-start" or "eager" coroutine.

Additionally upon reaching `co_return <expr>; promise.return_void()` or `promise.return_value(<expr>)` is called, depending on whether `<expr>` is void or not and which function is defined for the *promise* type. Note that `return_value` also returns `void` and it is used to store the value that was `co_return`ed inside the promise. If the coroutine returns void, `return_void` is also called at the end of the body, even if the coroutine does not contain `co_return;` explicitly.

Our next goal is to `co_await` something!

Awaitables

Let's start with half a UDP echo server first, that we will then coroutinize (sic):

```
#include "aiopp/ioqueue.hpp"
#include "aiopp/socket.hpp"

#include "spdlogger.hpp"

using namespace aiopp;

class Server {
public:
    Server(IoQueue& io, Fd&& socket)
        : io_(io)
        , socket_(std::move(socket))
    {
    }

    void start() { receive(); }

private:
    void receive()
    {
        receiveBuffer_.resize(1024, '\0');
        io_.recvfrom(socket_, receiveBuffer_.data(), receiveBuffer_.size(), 0,
            &clientAddr_,
```

```
            [this](std::error_code ec, int receivedBytes) {
                if (ec) {
                    spdlog::error("Error in recvfrom: {}", ec.message());
                    receive();
                    return;
                }

                receiveBuffer_.resize(receivedBytes);
                spdlog::info("received: '{}'", receiveBuffer_);
                receive();
            });
    }

    IoQueue& io_;
    Fd socket_;
    std::string receiveBuffer_;
    ::sockaddr_in clientAddr_;
};

int main()
{
    setLogger(std::make_unique<SpdLogger>());

    auto socket = createSocket(SocketType::Udp,
        IpAddress::parse("0.0.0.0").value(), 4242);
```

```
    if (socket == -1) {
        return 1;
    }

    IoQueue io;
    Server server(io, std::move(socket));
    server.start();
    io.run();
    return 0;
}
```

The full echo server is here: echo-udp.cpp

I hope you can guess well enough what all the aiopp stuff does. Note that I simplified the ownership of some of the objects. Everything is in `Server` and as long as that lives longer than `io.run()` runs, we're fine.

To build a coroutine version of this application, we need to create an *awaitable* for `recvfrom` operations. An *awaitable* is an object for which `operator co_await` is defined. There is also another way via `promise_type::await_transform`, but we won't discuss that in this blog post.

Before I explain more, I will show you an *awaitable* for our `recvfrom` operation:

```
struct IoResult {
```

```
    std::error_code ec;
    int result;
};

class RecvFromAwaitable {
public:
    RecvFromAwaitable(IoQueue& io, int socket, void* buf, size_t len,
        ::sockaddr_in* srcAddr)
        : io_(io)
        , socket_(socket)
        , buf_(buf)
        , len_(len)
        , srcAddr_(srcAddr)
    {
    }

    auto operator co_await()
    {
        struct Awaiter {
            RecvFromAwaitable& awaitable;
            IoResult result = {};

            bool await_ready() const noexcept { return false; }

            void await_suspend(std::coroutine_handle<> handle) noexcept
```

```
            {
                awaitable.io_.recvfrom(awaitable.socket_, awaitable.buf_,
                    awaitable.len_, 0, awaitable.srcAddr_,
                    [this, handle](std::error_code ec, int receivedBytes) {
                        result = IoResult { ec, receivedBytes };
                        handle.resume();
                    });
            }

            IoResult await_resume() const noexcept { return result; }
        };
        return Awaiter { *this };
    }

private:
    IoQueue& io_;
    int socket_;
    void* buf_;
    size_t len_;
    ::sockaddr_in* srcAddr_;
};
```

`operator co_await` must return an *awaiter* object, which should provide three methods: `await_ready`, `await_suspend` and `await_resume`.

When an *awaitable* is `co_await`ed, the `operator co_await` is executed and the *awaiter* is obtained from it. Then `awaiter.await_ready()` is called. If it returns `true`, the coroutine does not suspend and `await_resume` is called immediately, producing the return value of the expression. This is how `std::suspend_never` works - It simply returns `true` from `await_ready()`! If `await_ready` is `false`, the coroutine is suspended, meaning all state necessary to resume it later is stored in the coroutine state and `await_suspend` is called, being passed a `std::coroutine_handle` to the coroutine being suspended. Among other things and most importantly this handle can be used to resume the coroutine later. When a suspended coroutine is resumed, `await_resume` is called, returning the result to be produced by the `co_await` expression.

I keep a reference to the *awaitable* in the *awaiter* which should be fine, because both objects live until the end of the `co_await` expression and the *awaiter* will be destroyed before the *awaitable*. By the way you could also create an awaiter directly and `co_await` that (without the *awaitable* in between), but that is a less conventional approach and I would not have an opportunity to demonstrate `operator co_await` to you. And of course it would not work with this *awaiter*, because it needs a reference to an *awaitable*.

We can use our new awaitable like this:

```
BasicCoroutine serve(IoQueue& io, const Fd& socket)
{
    while (true) {
```

```
        std::string receiveBuffer(1024, '\0');
        ::sockaddr_in clientAddr;
        const auto [recvEc, receivedBytes] = co_await RecvFromAwaitable(
            io, socket, receiveBuffer.data(), receiveBuffer.size(), 0, &clientAddr);
        if (recvEc) {
            spdlog::error("Error in recvfrom: {}", recvEc.message());
            continue;
        }
        receiveBuffer.resize(receivedBytes);
        spdlog::info("received: '{}'", receiveBuffer);
    }
}

int main()
{
    auto socket = createSocket(SocketType::Udp,
        IpAddress::parse("0.0.0.0").value(), 4242);
    if (socket == -1) {
        return 1;
    }

    IoQueue io;
    serve(io, socket);
    io.run();
    return 0;
```

```
}
```

The full echo server is here: echo-udp-coro.cpp

Isn't that much prettier? There is much less noise and stupid lambdas and code that doesn't even tell you what the program actually does. It's much easier to understand too, ignoring the (high) complexity hidden in the `RecvFromAwaitable` and `BasicCoroutine` (more of that to come btw.).

Let's recap:

- `main()` is called and calls `serve`
  - The coroutine state is allocated, including the *promise* object
  - The return object of the coroutine is created via `promise.get_return_object()`
  - `co_await promise.initial_suspend()` is executed, which in our case is `co_await std::suspend_never{}`, i.e. it terminates immediately - the coroutine body starts executing
  - A `RecvFromAwaitable` is constructed packaging the parameters of our async IO operation
  - The *awaiter* is obtained from the `RecvFromAwaitable`
  - `awaiter.await_ready()` returns false, so the coroutine is suspended
  - `await_suspend()` is called, which initiates the async IO operation (recvfrom), storing the handle to the suspended coroutine in the lambda callback.

- Since `serve` suspended, it returned control to the caller and we are back in `main`
- `io.run()` is called and we wait to receive a message
  - `io.recvfrom` completes and the callback is called
  - We store the result inside the *awaiter* and call `handle.resume()` to resume the coroutine (`serve`)
    - `await_resume` is called inside `serve` to produce the result of the `co_await` expression.
    - The return value is processed and we print the received message
    - We `co_await RecvFromAwaitable` again (see above)

TCP Echo Server

Please pretend that I built various other classes like `RecvFromAwaitable` for other methods of `IoQueue`. In reality I wrote some ugly template class that packages arguments into a tuple and then applies them to the lambda, inclusion of which in here would distract too much from the topic of this blog post. Eventually we have a few functions that return *awaitables* for other `IoQueue` methods that could be copy-pasted from the `RecvFromAwaitable` above (but aren't). In the following I will pretend we have such functions for `accept`, `recv` and `send`.

Take a look at this simple TCP echo server:

```
BasicCoroutine echo(IoQueue& io, Fd socket)
{
    while (true) {
        std::string recvBuffer(1024, '\0');
        const auto [rec, receivedBytes]
            = co_await recv(io, socket, recvBuffer.data(), recvBuffer.size());
        if (rec) {
            spdlog::error("Error in receive: {}", rec.message());
            break;
        }

        if (receivedBytes == 0) { // Connection closed
            break;
        }

        recvBuffer.resize(receivedBytes);

        const auto [sec, sentBytes]
            = co_await send(io, socket, recvBuffer.data(), recvBuffer.size());
        if (sec) {
            spdlog::error("Error in send: {}", sec.message());
            break;
        }
```

```
        if (sentBytes == 0) { // Connection closed
            break;
        }
    }
}

BasicCoroutine serve(IoQueue& io, Fd&& listenSocket)
{
    while (true) {
        const auto [ec, fd] = co_await accept(io, listenSocket, nullptr, nullptr);
        if (ec) {
            spdlog::error("Error in accept: {}", ec.message());
            continue;
        }
        echo(io, Fd { fd });
    }
}

int main()
{
    auto socket = createTcpListenSocket(IpAddress::parse("0.0.0.0").value(), 4242);
    if (socket == -1) {
        return 1;
    }
```

```
    IoQueue io;
    serve(io, std::move(socket));
    io.run();
    return 0;
}
```

The final echo server is here: echo-tcp-coro.cpp

With what we did so far, this should already work! You can interact with it by running `nc 127.0.0.1 4242`.

Also we did something cool here, which might not be obvious right away. We `co_await accept` in a loop and call `echo` afterwards, which will handle a single connection. This makes it possible to handle multiple clients at once, without extra effort. When `accept` completes, i.e. we got a new connection and `serve` is resumed, we call echo, which will immediately `co_await recv` and return control back to us, so we can accept again, so that both `accept` and `recv` are issued at the same time. Pretty sweet!

Here it becomes important that we chose to use `std::suspend_never` for `final_suspend`! You will often (later) see task types (like `BasicCoroutine`) that keep a handle to a coroutine and clean it up in the destructor, like `if (handle_) handle_.destroy()`. In this case you would have to change `final_suspend` to return `std::suspend_always` (or something else that suspends).

With `std::suspend_never`, as we learned earlier (see above to remind yourself), the coroutine will not suspend after executing the coroutine body and the coroutine state will be destroyed immediately. This will invalidate all coroutine handles. So if we called `handle_.destroy()` in `~BasicCoroutine`, the coroutine would be long dead and our coroutine handle would be dangling, which would result in a use-after-free bug (or double-free if you will). Of course it is not just illegal (straight to jail) to destroy a coroutine through a dangling handle, but to do anything with it, including checking if the coroutine is done!

Analogously if you use `std::suspend_always` for `final_suspend`, the coroutine state will not be destroyed automatically and you **have to** destroy the coroutine through its handle if you don't want to leak it. If we went for this approach, our `BasicCoroutine` destructor would have destroyed the coroutine and since `echo(io, Fd { fd })` returns a temporary `BasicCoroutine`, the `BasicCoroutine` would destroy the coroutine as soon as we suspend `echo` from the first `co_await recv` and return control to `serve`. In summary we want our coroutine to outlive the `BasicCoroutine` objects, because we don't use them and they get destroyed early.

There is still a tiny problem with this. Since TCP is a stream protocol, send might not send all the data we have given it, i.e. `sentBytes` might be less than `recvBuffer.size()` and we should actually send in a loop until the whole buffer has been sent or an error occured (or the connection was closed). Ideally we would just introduce a new function `sendAll` which will do what I just explained, like so:

```
BasicCoroutine sendAll(IoQueue& io, const Fd& socket, const std::string& buffer)
{
    size_t offset = 0;
    while (offset < buffer.size()) {
        const auto [ec, sentBytes]
            = co_await send(io, socket, buffer.data() + offset,
                buffer.size() - offset);
        if (ec) {
            spdlog::error("Error in send: {}", ec.message());
            co_return;
        }

        if (sentBytes == 0) {
            co_return;
        }

        offset += sentBytes;
    }
}
```

Unfortunately we cannot simply call this function from `echo`. If we did, the `send`s from `sendAll` and `recv`s from `echo` would get interleaved just like the `accept`s from `serve` and the `recv`s from `echo`. And since `sendAll` is holding a reference to the receive buffer, this interleaving might cause the receive buffer to get changed by the `echo` function

under our feet! Even if we do a simple fix of copying the entire buffer and passing it into `sendAll` by value (dirty and lazy), another `recv` could in theory complete before the `sendAll` and initiate *another* `sendAll`, potentially destroying the stream property of the TCP connection by interleaving data that should not be interleaved. To be correct, we should pass control to `sendAll` until it finishes sending data and only then regain control in `echo` - or in other words: we want to `co_await sendAll()`!

Awaitable Coroutines

Let's just try to `co_await` a `BasicCoroutine`:

```
BasicCoroutine echo(IoQueue& io, Fd socket)
{
    while (true) {
        std::string recvBuffer(1024, '\0');
        spdlog::info("recv");
        const auto [rec, receivedBytes]
            = co_await recv(io, socket, recvBuffer.data(), recvBuffer.size());
        if (rec) {
            spdlog::error("Error in receive: {}", rec.message());
            break;
        }

        if (receivedBytes == 0) { // Connection closed
```

```
            break;
        }

        recvBuffer.resize(receivedBytes);
        co_await sendAll(io, socket, recvBuffer);
    }
}
```

Unfortunately we get the following error: `error: no member named 'await_ready' in 'BasicCoroutine'`. This is because, like I said earlier, a coroutine doesn't have to be an *awaitable* and ours isn't! Fixing this means, that we have to provide an `operator co_await` for our coroutine object.

Instead of extending `BasicCoroutine`, we introduce a new `Task` to be returned by `sendAll` (more on why later). The name of this class is borrowed from many other libraries and aligns with a proposal for a type to be added to the C++ standard. Even programming languages like C# and Python use this name to represent an asynchronous operation that can be awaited, so we will use it too.

Let's start with a copy of `BasicCoroutine`, but add a `operator co_await`, so we can `co_await` it:

```
class Task {
public:
```

```
    struct Promise {
        Task get_return_object() { return Task { }; }

        void unhandled_exception() noexcept { }

        void return_void() noexcept { }

        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
    };
    using promise_type = Promise;

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

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

What do we want our `Task` to do really? We want to suspend the calling coroutine

(`echo` in this case), when we start `sendAll` and only resume the calling coroutine, when `sendAll` has finished. The best way to hook into "when a coroutine has finished" is the *awaiter* returned by `final_suspend`, because as we learned earlier that awaiter will be suspended, when the coroutine body has completed. That means we probably want to introduce a custom awaiter, which we will call `FinalAwaiter`:

```
class Task {
public:
    struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }
        void await_suspend(std::coroutine_handle<> handle) noexcept { }
        void await_resume() const noexcept { }
    };

    struct Promise {
        Task get_return_object() { return Task { }; }

        void unhandled_exception() noexcept { }

        void return_void() noexcept { }

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

```
    using promise_type = Promise;

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

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

Now we have all the pieces and we just need to draw the rest of the fucking owl.

If we want to resume the calling coroutine later, we need to save a handle to it and we need to save it somewhere we can get to from `FinalAwaiter`, which will be the *promise*. To get to the *promise* from the `Awaiter`, we need a handle to the coroutine that belongs to the `Task`. Let's start by passing that from `Promise::get_return_object` to `Task()` and keeping a copy in `Awaiter` as well:

```
class Task {
public:
```

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

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

    void unhandled_exception() noexcept { }

    void return_void() noexcept { }

    std::suspend_never initial_suspend() noexcept { return {}; }
    FinalAwaiter final_suspend() noexcept { return {}; }
};
using promise_type = Promise;

Task() = default;

struct Awaiter {
    std::coroutine_handle<Promise> handle;
```

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

    auto operator co_await() noexcept
    {
        return Awaiter { handle_ };
    }

private:
    explicit Task(std::coroutine_handle<Promise> handle)
        : handle_(handle)
    {
    }

    std::coroutine_handle<Promise> handle_;
};
```

Note that we specified the promise type explicitly for the newly introduced coroutine handles. This is necessary to make the `.promise()` method available.

Remember that a handle to the calling coroutine (the one that `co_awaits` the `Task`) is passed to `Awaiter::await_suspend`. Let's add a member variable to `Promise` and save a

handle to the calling corouting in there. We will call this member variable `continuation`:

```
    struct Promise {
        std::coroutine_handle<> continuation;

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

        void unhandled_exception() noexcept { }

        void return_void() noexcept { }

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

    struct Awaiter {
        std::coroutine_handle<Promise> handle;

        bool await_ready() const noexcept { return false; }

        void await_suspend(std::coroutine_handle<> calling) noexcept
```

```
        {
            handle.promise().continuation = calling;
        }

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

Now all that's left to make it work, is to get the `continuation` handle in `FinalAwaiter::await_suspend`, which is called when the called coroutine (`sendAll`) completes, and `.resume()` it:

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

        template <typename P>
        void await_suspend(std::coroutine_handle<P> handle) noexcept
        {
            handle.promise().continuation.resume();
        }

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

We are not quite done, but this should work already. Let's iron out the kinks before we proceed.

First of all, since we now have a `FinalAwaiter` which suspends, the coroutine state is not destroyed automatically anymore. To prevent a memory leak, we need to add a destructor to `Task`:

```
            ~Task()
            {
                if (handle_) {
                    handle_.destroy();
                }
            }
```

This might also be a good time to mention that a coroutine is "done", when it "is suspended at its final suspended point", so when the *awaiter* returned by `final_suspend` has been `co_await`ed. The difference, again, between `await_ready` returning `true` or `false` is whether the coroutine state is destroyed automatically, which is why we need to destroy it manually this time.

Also I find it a bit awkward, that when we start `sendAll`, it starts doing its thing and suspends at `co_await send`, control is passed back to us (`echo`) and then we `co_await` the `Task` returned by it (`sendAll`). If `co_await send` would not have suspended `sendAll`, it might have completed immediately, `co_await`ed our `FinalAwaiter` and we would

attempt to resume the calling coroutine, before it has even been stored in the promise, because it was never suspended and `await_suspend` has not been called yet! This is why this time, we want to return `std::suspend_always` from `initial_suspend`. This also means that when we first `co_await` a `Task`, we should resume it once, so it starts executing:

```
// In Awaiter (returned from operator co_await)
void await_suspend(std::coroutine_handle<> continuation) noexcept
{
    handle.promise().continuation = continuation;
    handle.resume();
}
```

This also invites us to do something called "symmetric transfer" in which we can suspend one coroutine and resume another without consuming additional stack space. Read this post by Lewis Baker to learn more: Understanding Symmetric Transfer. To use it, we simply have to return a coroutine handle from `await_suspend` instead of calling `resume()` on it:

```
// In Awaiter (returned from operator co_await)
auto await_suspend(std::coroutine_handle<> calling) noexcept
{
    handle.promise().continuation = calling;
    return handle;
}
```

We can also use this in `FinalAwaiter::await_suspend`:

```
        template <typename P>
        auto await_suspend(std::coroutine_handle<P> handle) noexcept
        {
            return handle.promise().continuation;
        }
```

The cppcoro library has a check for whether a compiler supports symmetric transfer, but judging from my trials on godbolt it seems that symmetric transfer works for all compilers that don't complain about their respective C++20 flags (GCC 11 and later, MSVC 19.29 and later), so I will just use it.

Now that `initial_suspend` returns `std::suspend_always` our coroutine does not start executing until we `co_await` it, meaning that it would make sense to make our `Task` type `[[nodiscard]]`, otherwise creating the coroutine would simply do nothing and that is likely a mistake we want to help catch:

```
class [[nodiscard]] Task {
    // ...
};
```

The last tweak we want to make to `Task` is to `Awaiter::await_ready`. It returns `false`, which works for our code, but in case the coroutine is already done when we `co_await`

it, we don't want to suspend, but rather return the value right away:

```
            bool await_ready() const noexcept
            {
                return !handle || handle.done();
            }
```

Putting it all together, we get this:

```
class [[nodiscard]] Task {
public:
    struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }

        template <typename P>
        auto await_suspend(std::coroutine_handle<P> handle) noexcept
        {
            return handle.promise().continuation;
        }

        void await_resume() const noexcept { }
    };

    struct Promise {
        std::coroutine_handle<> continuation;
```

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

    void unhandled_exception() noexcept { }

    void return_void() noexcept { }

    std::suspend_always initial_suspend() noexcept { return {}; }
    FinalAwaiter final_suspend() noexcept { return {}; }
};
using promise_type = Promise;

Task() = default;

~Task()
{
    if (handle_) {
        handle_.destroy();
    }
}

struct Awaiter {
```

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

        bool await_ready() const noexcept { return !handle || handle.done(); }

        auto await_suspend(std::coroutine_handle<> calling) noexcept
        {
            handle.promise().continuation = calling;
            return handle;
        }

        void await_resume() const noexcept { }
    };

    auto operator co_await() noexcept
    {
        return Awaiter { handle_ };
    }

private:
    explicit Task(std::coroutine_handle<Promise> handle)
        : handle_(handle)
    {
    }

    std::coroutine_handle<Promise> handle_;
```

```
};
```

The final `Task` class is here: task.hpp

And here's the application that uses it:

```
Task sendAll(IoQueue& io, const Fd& socket, const std::string& buffer)
{
    size_t offset = 0;
    while (offset < buffer.size()) {
        const auto [ec, sentBytes]
            = co_await send(io, socket, buffer.data() + offset,
                buffer.size() - offset);
        if (ec) {
            spdlog::error("Error in send: {}", ec.message());
            co_return;
        }

        if (sentBytes == 0) { // Connection closed
            co_return;
        }

        offset += sentBytes;
    }
}
```

```
BasicCoroutine echo(IoQueue& io, Fd socket)
{
    while (true) {
        std::string recvBuffer(1024, '\0');
        const auto [ec, receivedBytes]
            = co_await recv(io, socket, recvBuffer.data(), recvBuffer.size());
        if (ec) {
            spdlog::error("Error in receive: {}", ec.message());
            break;
        }

        if (receivedBytes == 0) { // Connection closed
            break;
        }

        recvBuffer.resize(receivedBytes);
        co_await sendAll(io, socket, recvBuffer);
    }
}

BasicCoroutine serve(IoQueue& io, Fd&& listenSocket)
{
    while (true) {
        const auto [ec, fd] = co_await accept(io, listenSocket, nullptr, nullptr);
```

```
        if (ec) {
            spdlog::error("Error in accept: {}", ec.message());
            continue;
        }
        echo(io, Fd { fd });
    }
}

int main()
{
    auto socket = createTcpListenSocket(IpAddress::parse("0.0.0.0").value(), 4242);
    if (socket == -1) {
        return 1;
    }

    IoQueue io;
    serve(io, std::move(socket));
    io.run();
    return 0;
}
```

The final echo server is here: echo-tcp-coro.cpp

Returning Values From Coroutines

The final thing that bothers me about this echo server is that when `sentBytes` is `0` (the connection has been closed between receiving and sending), we attempt another `recv`, which will immediately fail. Ideally we would return `std::pair<std::error_code, bool>` from `sendAll` which contains a potential error and an EOF ("end-of-file") flag, so we can return from `echo` early, if needed:

```
        const auto [sendEc, eof] = co_await sendAll(io, socket, recvBuffer);
        if (sendEc || eof) {
            break;
        }
```

and:

```
Task<std::pair<std::error_code, bool>> sendAll(
    IoQueue& io, const Fd& socket, const std::string& buffer)
{
    size_t offset = 0;
    while (offset < buffer.size()) {
        const auto [ec, sentBytes]
            = co_await send(io, socket, buffer.data() + offset, buffer.size() - offset);
        if (ec) {
            spdlog::error("Error in send: {}", ec.message());
            co_return std::make_pair(ec, false);
        }
```

```
        if (sentBytes == 0) { // Connection closed
            co_return std::make_pair(ec, true);
        }

        offset += sentBytes;
    }
    co_return std::make_pair(std::error_code {}, false);
}
```

I mentioned earlier that we can define `Promise::return_value` instead of `Promise::return_void` to do that. And like the continuation earlier, we will store the result in the `Promise` as well. Additionally we have to make `Awaiter::await_resume` return our result from the promise, to make the `co_await` expression return a result.

To support arbitrary return types, I have made `Task` a template class and to make sure it still works with void, I specialized the `Promise` type for `Result=void`. I used `requires` expressions to define different `Awaiter::await_resume` functions for different Result template arguments.

This is the whole thing:

```
template <typename Result = void>
class [[nodiscard]] Task {
```

```
public:
    struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }

        template <typename P>
        auto await_suspend(std::coroutine_handle<P> handle) noexcept
        {
            return handle.promise().continuation;
        }

        void await_resume() const noexcept { }
    };

    struct Promise {
        std::coroutine_handle<> continuation;
        Result result;

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

        void unhandled_exception() noexcept { }

        void return_value(Result&& res) noexcept { result = std::move(res); }
```

```
    std::suspend_always initial_suspend() noexcept { return {}; }
    FinalAwaiter final_suspend() noexcept { return {}; }
};
using promise_type = Promise;

Task() = default;

~Task()
{
    if (handle_) {
        handle_.destroy();
    }
}

struct Awaiter {
    std::coroutine_handle<Promise> handle;

    bool await_ready() const noexcept { return !handle || handle.done(); }

    auto await_suspend(std::coroutine_handle<> calling) noexcept
    {
        handle.promise().continuation = calling;
        return handle;
    }
```

```
        template <typename T = Result>
        requires(std::is_same_v<T, void>)
        void await_resume() noexcept { }

        template <typename T = Result>
        requires(!std::is_same_v<T, void>)
        T await_resume() noexcept { return std::move(handle.promise().result); }
    };

    auto operator co_await() noexcept { return Awaiter { handle_ }; }

private:
    explicit Task(std::coroutine_handle<Promise> handle)
        : handle_(handle)
    {
    }

    std::coroutine_handle<Promise> handle_;
};

template <>
struct Task<void>::Promise {
    std::coroutine_handle<> continuation;
```

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

    void unhandled_exception() noexcept { }

    void return_void() noexcept { }

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

The `Task` class is here: task.hpp

You may ask if we should have or could have extended our `BasicCoroutine` to handle return values as well, but we would have to change so much about it, that the name would not be justified anymore and we would almost have `Task` anyways but without `operator co_await`. We would have to make `final_suspend` return `std::suspend_always` as well, so the coroutine frame lives long enough for us to get the result from the `Promise` through the `BasicCoroutine` object. And then we couldn't use it anymore to "branch off" with `echo` because it returns a temporary and now that the `BasicCoroutine` object has ownership of the coroutine, it would be destroyed in `~BasicCoroutine`. So we still have reasons to keep a type without a non-void return value, like `BasicCoroutine`,

around.

Also now that we have a `Task` class that owns the coroutine, it is possible that we destroy it (by destroying the `Task`) when it still has queued IO operations that essentially keep a reference to our *awaiter* (in the lambda capture). To avoid a use-after-free, it is necessary to cancel IO operations if the *awaiter* being referenced in the `IoQueue` dies. I did not include that in this tutorial yet, because I haven't figured out how to do that nicely myself. Maybe I'll edit it into this blog post in the future or maybe you figure it out and let me know if you find something that works well for you.

Where To Go Next

Lastly I have to mention that if you want to do anything multi-threaded or if you want to handle exceptions, there is extra work you have to do and you have to be a lot more careful everywhere. I'm not going to do that here.

Although I might have (sort of) figured out how to use coroutines, I don't really have experience with writing larger programs (beyond simple echo servers). It's possible that some of the solutions I have presented here might just be very annoying or problematic if used in larger code bases. I hope I get a chance and the inspiration to write about this in a later blog post.

Tips

My number one, huge, seriously golden tip: **use AddressSanitizer**. It's very easy to build coroutine programs that use-after-free or leak memory without you realizing. I wholeheartedly suggest to turn on ASan and not turn it off again, as long as you are messing around with coroutines. At some point every code you write compiles, lots of it works and the only one telling you it's completely wrong is ASan.

To help me understand which functions are called when and when objects are destroyed, I added logging to all objects on creation/destruction/copy with something like this: DebugLogging. I also added logs to every magic method - all the `await_*`s and all the `Promise` methods. It might seem silly dumping so many logs, but it really helps a lot to get an understanding of the control flow and I definitely needed it to get any understanding of coroutines in C++.

Sources

- Asymmetric Transfer - Lewis Baker's blog. It's very detailed and it felt a bit overwhelming to me at the beginning, but it leaves few open questions and is a great read if you already sort of know how coroutines work.
- cppcoro. The de-facto helper library for coroutines in C++. The link points to a fork, which seems to be more maintained than the original repository by Lewis Baker, who seems to really know their coroutines.

- Coroutines on cppreference. Everyone knows this website is good, but this particular site is more helpful than I would have thought.
- C++20 Coroutines and io_uring. I tried working with coroutines a few times and just got confused without ever really getting anywhere. This helped me get started more than any other resource and I modeled my post after it. I just tried to go a bit further in some directions and explain some things that did not get clear to me reading that blog post.