Writing custom C++20 coroutine systems

[Simon Tatham, 2023-08-06]

[Coroutines trilogy: C preprocessor | **C++20 native** | general philosophy]

Introduction

In 2023, I went on a three-day training course introducing all the new features of C++20, for people who already spoke earlier versions of C++.

One of those new features is built-in language support for coroutines. I was particularly looking forward to that section of the course, because I'm an enormous fan of coroutines in general as a coding style. So much so, in fact, that I went to the effort of inventing a technique for implementing them in C, by abuse of the C preprocessor – and I actually used it in production code, too. PuTTY makes heavy use of that technique in its implementation of the SSH protocol, and in a few other places too. And another program of mine, the exact real calculation tool `spigot`, uses the same technique in C++.

So I was very keen to find out everything about the new C++ native coroutine support – I *at least* wanted to know whether it would be a good idea to migrate `spigot` away from my preprocessor hack to the new system (since that's already written in C++), and I also wanted to know all the other things I might be able to use it for.

Unfortunately, C++20's coroutine system is large, complicated, and takes a lot of explaining. So in a three-day course that also has to cover all the *other* major and minor new features in C++20, there turned out not to be much time to go into all its details. (Not to mention that you also have to start by explaining what coroutines even *are*, because the rest of the people on the course had not all been huge fans of them for decades!)

So, after the course, I went away and studied on my own, and wrote the introduction to C++ coroutines that I'd have *liked* to see. By the time I'd finished, there was enough of it that it could probably make a three-day course in its own right!

Having gone to the effort of learning it all and writing it up, it seems a shame to keep it to myself. So this article is my personal learning notes, polished up into a form that I hope will be useful to some other people too.

I'll show a lot of code snippets in this article, for illustration purposes. Most of them won't work on their own: they'll normally have pieces missing, so as not to lose the important detail among the irrelevant ones. Sometimes I'll even make them deliberately wrong, for clarity (such as putting a method definition inside a class, which would really have to be defined out of line further down the file to avoid a forward reference). Most sections also come with a downloadable full working version of the same code, which will actually compile and run.

Overview of the C++ coroutine system

C++20's coroutine system has a lot of flexibility that Python's generators (for example) don't. But the flip side of that flexibility is that you have to do a lot of work if you want to use it. It's not so much an actual coroutine system; it's more of a construction kit you can *build* a coroutine system out of.

The starting point is quite similar between the two. You don't have to declare a function as being a coroutine *outside* the function body. (In particular, a caller doesn't necessarily know whether there's a coroutine involved or not – it only knows that when it calls the function, it gets back an object.)

In Python, putting at least one `yield` statement inside the function body causes magic to happen, and the effect of the magic is that when you call the function, none of the code inside it runs (yet), and instead you get back a special 'generator' object in which the code of the function is suspended, and will run a little at a time whenever you do an operation on the generator object. There's only one kind of generator; the operations it supports are fixed (`next()` and `send()`), and they always do the same things.

In C++, it's the same, except that there are three different magic keywords: `co_yield`, `co_await` and `co_return`. Any of those in the function body will make a function into a coroutine. But you get a lot more control over what happens next.

For example, if your coroutine is mostly supposed to *receive* data, you'd quite like it to run immediately to the point where it's ready to receive its first item. That way, when someone calls the coroutine to create a new instance of it, that instance comes back already in a state where you can give it a value. In Python you can't do this – the generator always starts off suspended at the very beginning – and so the usual approach is to call `next()` once on the returned object to 'prime the pump'. But in C++, you can configure a particular class of coroutines to behave that way if you like.

In Python, calling `yield` inside the coroutine always suspends the code and returns control to the caller of `next()`. So if you want the yielded value to be passed to a different coroutine (say, to transfer objects from a producing coroutine to a consuming one), you have to do that by hand at the call site. In C++, you can do it that way if you want to, but you can *also* arrange for coroutines to transfer control back and forth between themselves automatically, so that control isn't returned to the caller until a value is actually available of the type the *caller* wanted. Or you can have `co_yield` not suspend the coroutine at all, and carry on running.

In Python, the same `yield` statement is used to deliver output values *and* accept input values (if the coroutine expects any at all). In C++, there are separate `co_yield` and `co_await` statements that you can use for the two purposes: `co_yield` to deliver an output value, and `co_await` to wait for an input value, or (if you prefer) just to wait for some event to have taken place. So if your coroutine is reading objects from *here* and writing them to *there*, each statement makes it clear which kind of thing it's doing.

In Python, there's no real concept of a 'return value' from the coroutine, separate from the stream of yielded values. In C++, there is: you can use the `co_return` statement to terminate the whole coroutine, and you can give it a value to return, which can have a different type and semantics from whatever you've been yielding.

As a complicated example, you could imagine writing a coroutine that implements a network protocol, by repeatedly being called back from an event loop that does the low-level network I/O to send and receive individual protocol messages or packets. You might set it up so that it can call `co_await` to get the next input packet (which might not need to suspend the coroutine at all if a received packet was already in the queue); it can call `co_yield` to deliver an output packet; and when it's finished, it can call `co_return` to signal the outcome of the entire conversation, such as whether the transaction it tried to perform over the network was successful.

However, **none of this is already done for you**. In order to use coroutines at all in C++20, you must write a lot of setup code which fills in answers to all of the following questions:

- Should the coroutine start running immediately on creation, or start off suspended at the entry point?
- What data types are acceptable to `co_await`, `co_yield` and `co_return`? What does each one do? What happens to the values passed to them? What data, if any, does the coroutine get back *from* a `co_await` or `co_yield` once it resumes?
- Which `co_await` and `co_yield` operations actually suspend this coroutine, and which ones are able to carry on running immediately?
- When the coroutine is suspended, does it immediately return control to the caller who last resumed it, or does it switch to running a different suspended coroutine without giving the caller a choice in the matter?

In Python, the answers to most of these questions are fixed by the language design. In C++, the answer to all of them is: *it's entirely up to you*. You can choose all these answers however best suits your particular program. But on the other hand, you *have* to do all the work to fill in all of those answers!

At least, in C++20 you had no choice but to do all the work yourself. But C++23 has introduced a much more ready-made coroutine system called `std::generator`, which answers all of these questions in a way similar to Python generators. So if all you want is a thing in C++ that behaves like a Python generator, and if you're prepared to wait for C++23 to be widely available, you don't have to engage with any of these details at all. It's all been done for you.

But if you want to use all this extra flexibility, or if you don't want to wait for C++23, or both, then you have some work to do. This article will show you how to do it.

Summary of the data types

A C++ coroutine system involves quite a lot of separate data types. Before we get down to the details of what methods each one has to provide, in this section I identify them all and explain how they inter-relate.

There's one data type provided by the C++ implementation itself, which is a **coroutine handle**. This is the object that contains all the magic. It identifies a particular instance of your coroutine code, together with all its internal variables and its state of execution (e.g. whether it's currently running, and if not, where it will resume from the next time it's restarted). It provides a 'resume' method that you can call to actually start your coroutine running, or restart it after it was suspended.

All the remaining data types are provided by you, the implementor of a particular coroutine setup. So you can define them in a way that specifies the way you want this particular coroutine setup to behave.

The most obvious type is the one that the coroutine function is declared as returning. This is the only type that the *user* of your coroutine setup will see (whether they're defining coroutines via your setup, calling them, or both). So, for this article, I'm going to call it the **user-facing type**, and an instance of it is a **user-facing object**.

The user-facing type is the thing that the caller of a coroutine will actually interact with. So the methods you define on it will be dependent on *how* you expect the user to interact with it. For example, you might want to make it behave like an iterable (by giving it `begin()` and `end()` methods, and an associated iterator type with an increment operator), so that you can use it in a range-`for` loop. (C++23's ready-made `std::generator` does all that.) Or you might want to give it methods that allow you to treat it like an `istream`, or an `ostream`, or maybe none of those is relevant to you at all and you just want to give it a set of custom methods that tie it into your program's event loop. It's entirely up to you.

There are actually *no* specific requirements imposed by C++ on what the user-facing type should look like. It doesn't have to include any specific named methods or fields or member types. In fact it doesn't even have to be a *class* type, if you don't want it to be – you could perfectly well make it some trivial thing like `int`, if you wanted to, and have it just be an index into a giant table of all your currently active coroutines. It's more usual to make it a user-defined class type, but it's not *necessary*.

Behind the scenes, in the implementation, the primary data type defining your particular coroutine setup is called the **promise type**[1]. This is inferred from the user-facing type (and, optionally, the rest of the coroutine's arguments). It must be a class, and it must provide various specially named methods to control policy

decisions such as whether to start off suspended, whether to transfer control to another coroutine on termination, and what to do if the body of the coroutine executes `co_yield`, `co_await` or `co_return`. There is exactly one of these per instance of a coroutine (the system creates one at the moment the coroutine is initially called), which also makes it a good place to store data relating to that coroutine instance. For example, in a stacked coroutine setup, you might include pointer fields linking together the promise objects in a stack. In a coroutine setup that delivers output values via `co_yield`, a convenient place to put the yielded value is in the promise object, and then the user-facing object can recover it from there to return to the caller.

Finally, there's a family of types called **awaiters** (or sometimes *awaitables*). These set the policy for each individual event that tries to suspend the coroutine. Specifically, every time a coroutine executes `co_yield`, `co_await` or `co_return`, the promise type will construct a fresh awaiter for that particular event, and methods of that awaiter will be called to specify what will happen (such as whether the coroutine will suspend itself at all; if so, whether it will transfer control to another coroutine or back to the caller; and when the coroutine resumes, what value is returned to the coroutine as the value of the `co_yield` or `co_await` expression). Awaiter types can be specific to your particular setup, or they can be general enough to share between multiple types of coroutine. In particular, the standard library provides some default awaiter types that do the simplest possible things, called `std::suspend_always` and `std::suspend_never`.

How to write a coroutine promise class

Getting started: make it at least compile

Before I get into the details of what all the methods do, let's start by exhibiting the smallest from-scratch C++ coroutine system that will at least *compile*, and run some code from inside the coroutine. Then when I list the things you have to do to get this far, you can relate each one to the example you've already seen.

So, without further ado, here's my 'hello, world' of coroutines:

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

class UserFacing {
  public:
    class promise_type {
      public:
        UserFacing get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        void return_void() {}
        void unhandled_exception() {}
        std::suspend_always final_suspend() noexcept { return {}; }
    };
};

UserFacing demo_coroutine() {
    std::cout << "hello, world" << std::endl;
    co_return;
}

int main() {
    UserFacing demo_instance = demo_coroutine();
}
```

This fulfills the minimal set of requirements in order to be able to even *define* a coroutine with a given user-facing type.

Those requirements are:

Associate the user-facing type with a promise type. When the compiler sees that you've defined a coroutine with the return type `UserFacing`, the first thing it needs to do is to figure out what the associated promise type is.

The *default* way to do this (but not the only way) is to arrange that `UserFacing::promise_type` is a type name. In this really tiny example I've done that by defining the `promise_type` class entirely *inside* `UserFacing`. A

second approach would be to define it separately, and include a declaration inside `UserFacing` along the lines of

```
class UserFacing {
    // ...
  public:
    using promise_type = TheActualPromiseType;
    // ...
};
```

There's a third approach, which doesn't require you to put *any* definition inside the user-facing type at all. This involves providing your own specialisation of the `std::coroutine_traits` template (overriding the default version, which is what looks for `UserFacing::promise_type`). We'll see an example of that in a later section, in which the return type of the coroutine will be a type we didn't define ourselves, namely a `std::unique_ptr`.

Construct an instance of the user-facing type to return to the user. When your coroutine is instantiated, the C++ implementation will take care of creating an instance of the promise type, and allocating storage for it. But it's up to you to make the actual user-facing object that the caller will receive.

That's done by the `get_return_object()` method in the promise type. In this example, `UserFacing` is an object with no data members, so `get_return_object()` constructs one in a completely trivial way.

In a less trivial example, you'd want to give the user-facing object more information than that. In particular, the user-facing object will almost certainly need access to the coroutine handle and the promise object, in order to communicate with the still-suspended coroutine. This is the smallest possible example, so we haven't done that yet, but we'll see how to do it in the next section.

Specify whether the coroutine should start suspended, or run immediately. This is done by the `initial_suspend()` method, which must return a type that functions as an awaiter. We'll see the full details of awaiters in a later section, but for this initial example, we're just using one of the standard awaiter types provided by the C++ library: `std::suspend_never`. This causes the coroutine *not* to suspend itself on startup, which means that before control returns to `main()`, the coroutine code will actually start running, and print our "hello, world" message.

If we had instead made `initial_suspend()` return the other standard type, `std::suspend_always`, then the newly created coroutine would be suspended at the beginning of its function body, *before* the code that prints "hello, world". So that message wouldn't be printed until the next part of `main()` did something to *resume* the coroutine. But we haven't yet shown how to do that, so for the moment, our example coroutine starts off not suspended, because that's the only way it will print our hello message at all.

(`std::suspend_always` and `std::suspend_never` don't contain any interesting data, only methods that return fixed values. So you don't need to provide any information when you construct one, which is why we can just say `return {}` in the method body.)

Specify what to do when the coroutine returns normally. In this example, the coroutine doesn't return a value at all (the `co_return` statement has no parameter). So I've filled in a method called `return_void()` in the promise type, which will be called when `co_return` is executed without a value, or when code falls off the end of the coroutine.

If I'd wanted the coroutine to have a non-void return value, then I would have instead implemented a method called `return_value()` taking one argument, which would have been called when the coroutine executed `co_return` *with* a return-value expression.

Note that you must implement *exactly one* of those two methods in a promise type! It's an error not to implement either, *and* it's an error to implement both. So you're not allowed to write a coroutine that returns a value from some `co_return` statements and no value from others.

(And if you write a coroutine that's supposed to return a value, and instead fall off the end of the function without executing `co_return` at all, that's undefined behaviour – the same as it would be if you left the normal `return` out of a non-void ordinary function.)

Specify what happens if an exception propagates out of the coroutine. If the code inside the coroutine throws an exception and nothing catches it, then the promise object's `unhandled_exception()` method will be called. It

can retrieve the exception and store it, and then contrive to do something useful with it.

We'll see how to do that in a later section, and show a non-trivial example of something it's useful for. For this minimal example, I've just made `unhandled_exception()` do nothing at all, which means that the exception will be silently dropped, and the coroutine will be left in the same state as if it had terminated cleanly.

Specify what happens when the coroutine terminates. This is done by the `final_suspend()` method, which looks very like `initial_suspend()` except that this one has to be declared `noexcept` (because an exception at this point would be too inconvenient to deal with).

`final_suspend()` is called when the coroutine terminates by *any* means – whether by returning cleanly *or* by throwing an exception.

Here, `final_suspend()` returns `std::suspend_always`, which means that when the coroutine finishes (either via `co_return` or by falling off the end of the function body), it will enter a suspended state and return control to the caller.

You *must not* return a value from `final_suspend()` that causes the terminated coroutine to try to continue running! That will lead to a crash, or some other form of undefined behaviour. The only useful thing you might do in this method *other* than returning straight to the caller is to transfer control to a different suspended coroutine. I'll show an example of that in a later section.

Full source code for this section: `co_demo.minimal.cpp`.

Suspending the coroutine with `co_await`

So far, our demo coroutine is completely pointless, because you can't suspend and resume it, which is the whole point of coroutines. So that's the next thing to do: in this section we'll create a way to suspend it.

C++ provides two keywords you can use to suspend a coroutine: `co_await` and `co_yield`. `co_await` is the more basic one: `co_yield` is defined in terms of it. So we'll tackle `co_await` first.

The idea of having two keywords is that 'await' is used to indicate a thing you want to wait *for*, whereas 'yield' is used to specify a value you want to deliver to someone else. You don't *have* to use the two keywords for those purposes – you're writing the code that makes them do things, so you can make them do what you like! – but those are the intended purposes.

The general idea is that, whatever operand you provide to the `co_await` keyword, the compiler has to convert it into an awaiter object that will tell it how to manage the suspension.

The very *simplest* way to do this is to provide an operand to `co_await` which is *already* a valid awaiter object. And we already know how to construct one kind of awaiter, by using the ready-made library types `std::suspend_always` and `std::suspend_never`. So we can suspend our existing demo coroutine with as little code as possible by simply making one of those and awaiting it:

```
UserFacing demo_coroutine() {
    std::cout << "we're about to suspend this coroutine" << std::endl;
    co_await std::suspend_always{};
    std::cout << "this won't be printed until after we resume" << std::endl;
    co_return;
}
```

That's the simplest way to suspend a coroutine, but more usually, if your coroutine is going to wait for something, you'll want to make arrangements to have it resumed when that thing happens. So you need to run some handler code of your own.

There are two places you can insert handler code:

- If your promise type has a method called `await_transform()` which accepts an argument of the same type you passed to `co_await`, then that method is called, and the argument is replaced with its return

value. (You can provide methods of this name for multiple argument types, of course, via normal function overloading.)
- If a function is in scope called `operator co_await` accepting an argument of the appropriate type, then that function is called, and the argument is replaced with its return value.[2]

(If *both* of those things exist, then they're called in sequence, so that `co_await foo` might end up constructing an awaiter by calling `operator co_await(promise.await_transform(foo))`.)

So you can pass a type to `co_await` which represents the actual thing you want to wait for, and behind the scenes, your promise type's `await_transform()` can do the work of returning an appropriate awaiter.

In a real-world example, you'd probably make `await_transform()` do some actual work. For example, if your program was based around some kind of event loop that was monitoring I/O channels like network connections, then it would have some data structure that told it what to do about each possible I/O event, perhaps including what coroutine(s) to resume when an event happened. So if a coroutine wanted to wait for a particular I/O event before doing any more work, you'd write `co_await` *`event`* (probably involving a type you'd made up to represent descriptions of I/O events in a way that was convenient to write inside a coroutine), and then the promise's corresponding `await_transform()` function would do the work of inserting the calling coroutine into the event loop's data structures to arrange for that to actually *happen*.

But we don't have a complicated event loop in this example, so we'll keep it simple. We'll just make up a dummy event type containing no data, and an `await_transform()` that accepts one.

```
struct Event {
    // you could put a description of a specific event in here
};

class UserFacing {
    // ...
    class promise_type {
      public:
        // ...
        std::suspend_always await_transform(Event) {
            // you could write code here that adjusted the main
            // program's data structures to ensure the coroutine would
            // be resumed at the right time
            return {};
        }
    };
};
```

Then you can give `co_await` one of your event descriptions:

```
UserFacing demo_coroutine() {
    std::cout << "we're about to suspend this coroutine" << std::endl;
    co_await Event{};
    std::cout << "this won't be printed until after we resume" << std::endl;
    co_return;
}
```

In this example, my `await_transform()` function returns the type `std::suspend_always`. You could also make it return a custom awaiter type, which would give you more control. In particular, in some setups, it might turn out that the event you wanted to wait for had *already* happened – in which case a custom awaiter could decide not to suspend the coroutine after all, and just carry straight on.

A custom awaiter can also control the *return value* of the `co_await` expression. For example, suppose the event you were waiting for was some kind of network transaction delivering an output value, or delivering a status indicating whether it had been successful or not. Then you'd like to write code such as

```
UserFacing demo_coroutine() {
    // You might set up co_await to return actual data
    ip_address addr = co_await async_dns_lookup("hostname.example.com");
```

```
    // Or a boolean indicating success or failure
    if (co_await attempt_some_network_operation(addr)) {
        std::cout << "success!" << std::endl;
    } else {
        std::cout << "failed, fall back to some other approach" << std::endl;
    }
}
```

But we'll see how to write custom awaiters in a later section. This section is long enough already!

Full source code for this section: co_demo.await.cpp.

Resuming the coroutine

Now we've got a *nearly* useful coroutine system. We can make a coroutine, start running code in it, and suspend it. But this still isn't useful unless we can *resume* the coroutine after it's been suspended!

The way to resume a suspended coroutine is to call the `resume()` method on its *coroutine handle*. So the first thing we're going to have to do is to get hold of the handle in the first place.

Coroutine handles are a templated type, whose parameter is the promise type. So if your promise type is called `P` (for example), then the handle for a coroutine with that promise type is called `std::coroutine_handle<P>`.

(There's also a generic type `std::coroutine_handle<>`, which is a shorthand for `std::coroutine_handle<void>`. This is the coroutine-handle analogue of the `void *` generic pointer: it's a type that can store *any* coroutine handle, no matter what its promise type.)

A coroutine handle is created by the implementation at the same time as the promise object, and it's easy to convert either one into the other:

- To get the handle from a promise, call the static method `from_promise()` on the coroutine handle type, passing it a reference to your promise object.
- To get the promise from a handle, call the method `promise()` on the handle, which returns a reference to the associated promise object.

When we're constructing a new coroutine instance, the system calls our promise type's `get_return_object()` method. That already has a reference to the promise object, namely `*this`. So we can use that to find the coroutine handle.

What do we do with it once we've got it? We pass it to the constructor of the user-facing object, because that's the object the end user will manipulate in order to resume the coroutine, so it's going to have to know where to find the coroutine handle.

`std::coroutine_handle<P>` is quite verbose and I don't want to type it all the time, so we'll start by making a convenient alias for it, `handle_type`. So we end up doing this:

```
class UserFacing {
  public:
    class promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    class promise_type {
      public:
        UserFacing get_return_object() {
            auto handle = handle_type::from_promise(*this);
            return UserFacing{handle};
        }
        // ...
    };

    // ...
};
```

Of course, this is no use unless our user-facing type *has* a constructor that can accept a coroutine handle. Also, it will need a data field to store it in, and at least one method that *does* something with it:

```
class UserFacing {
    // ...

  private:
    handle_type handle;

    UserFacing(handle_type handle) : handle(handle) {}

  public:
    void resume() {
        handle.resume();
    }
};
```

Here, I've done the simplest possible thing, by giving the user-facing type a `resume()` method that simply calls `resume()` on the coroutine handle.

Note that I've made the constructor private. I think that's good practice for the usual reasons you make methods private: the only legitimate call to the constructor is the one in `get_return_object()`, and so its exact API is an internal interface between the promise type and the user-facing type. So you want to be able to change it on a whim, without the risk of having to modify other uses of it elsewhere.

In this case, the constructor call is in the `promise_type` class, which is defined inside `UserFacing`, and therefore automatically has access to its private members. If we had defined the promise type separately, we'd have had to declare it explicitly as a `friend` class, or else just make this constructor public.

And now we can actually resume our coroutine, by calling the new method on the user-facing object:

```
UserFacing demo_coroutine() {
    std::cout << "we're about to suspend this coroutine" << std::endl;
    co_await Event{};
    std::cout << "we've successfully resumed the coroutine" << std::endl;
}

int main() {
    UserFacing demo_instance = demo_coroutine();
    std::cout << "we're back in main()" << std::endl;
    demo_instance.resume();
}
```

which will produce the following output:

```
we're about to suspend this coroutine
we're back in main()
we've successfully resumed the coroutine
```

At last – it's taken three sections of this document, but we now have the ability to suspend and resume our coroutine!

(Note that I've also removed the `co_return` from the end of `demo_coroutine()`. The only reason we needed it in the previous examples was because there had to be *some* kind of `co_`*`foo`* keyword in the function body to make it count as a coroutine at all. But now we have a `co_await`, that's doing the same job, and we don't need a pointless `co_return` any more – we can let control fall off the end of the routine in the usual way.)

Full source code for this section: `co_demo.resume.cpp`.

Disposing of the coroutine state

Now that we've stored the coroutine handle in the user-facing object, it's a good moment to deal with the boring part: memory management.

We didn't write code ourselves to allocate the promise object. The C++ implementation did that itself, behind the scenes. So we need to worry about how it will be deallocated again. Until we do that, our coroutine system will have a built-in memory leak.

This doesn't happen automatically. You have to arrange it yourself, by calling the `destroy()` method on the coroutine handle.

The obvious place to do this is in the destructor of the user-facing object. But in that case you also need to protect against double-freeing, in case a caller *copies* the user-facing object.

So the simplest thing is to delete the copy constructor and copy assignment operator, so that people can't accidentally copy the user-facing object, and will have to move it instead. Also, set up a move constructor and move assignment operator which null out the coroutine handle in the moved-from object, so that if people *do* move the user-facing object, they don't end up double-freeing the coroutine handle:

```
class UserFacing {
    // ...

  private:
    handle_type handle;

    UserFacing(handle_type handle) : handle(handle) {}

    UserFacing(const UserFacing &) = delete;
    UserFacing &operator=(const UserFacing &) = delete;

  public:
    UserFacing(UserFacing &&rhs) : handle(rhs.handle) {
        rhs.handle = nullptr;
    }
    UserFacing &operator=(UserFacing &&rhs) {
        if (handle)
            handle.destroy();
        handle = rhs.handle;
        rhs.handle = nullptr;
        return *this;
    }

    ~UserFacing() {
        if (handle)
            handle.destroy();
    }
};
```

If you *do* want to be able to copy the user-facing object, then you'll have to do something more elaborate to ensure that the coroutine is only cleaned up when the last reference to it vanishes. Probably the easiest way would be to set up something involving a `shared_ptr` with a custom deallocator.

Full source code for this section: co_demo.destroy.cpp.

Delivering output values with `co_yield`

Now we've got the basics, we can start adding refinements. The first of those is `co_yield`.

As an example, we're going to arrange for our demo coroutine to deliver a stream of integers to its caller. We'd like to end up being able to write something like this:

```
UserFacing demo_coroutine() {
    co_yield 100;
    for (int i = 1; i <= 3; i++)
        co_yield i;
    co_yield 200;
}
```

and give our user-facing type a `next_value()` method which will return the integers 100, 1, 2, 3, 200 in sequence if you call it repeatedly.

In order for `co_yield` to be legal in the body of the coroutine, the promise type must provide a method called `yield_value`, taking a value of the type you're yielding. In this example, we want to yield integers, so we must define `yield_value(int)`.

The return value of `yield_value` will be treated as if you'd passed it to `co_await`. So it must be either an awaiter type itself, or else something that can be turned into an awaiter by `await_transform` or `operator co_await` or both.

In this case, the simplest thing is to make it an awaiter itself, and again, use the trivial standard awaiter type `std::suspend_always`. That way, every execution of `co_yield` will pass control back to the caller, so that it can consume the value we just yielded.

But `yield_value` also has to *do* something with its argument! Nothing in the C++ implementation will take care of communicating the yielded value to the caller of the coroutine. We have to write code to do that ourselves.

The simplest thing is to store the value inside the promise object, giving it an extra field to hold it:

```
class UserFacing {
    // ...
    class promise_type {
        // ...
      public:
        int yielded_value;

        std::suspend_always yield_value(int value) {
            yielded_value = value;
            return {};
        }
    };
};
```

Now, when the coroutine executes `co_yield 100`, a method call will be generated on the promise object, namely `yield_value(100)`. The implementation shown above will store 100 into the `yielded_value` member variable, and then return an awaiter that suspends the coroutine.

Suspending the coroutine means that control will return to whoever called `handle.resume()`. In the previous section, that was called by a method of the user-facing type. So now we should modify that method so that it returns an `int` instead of `void`, and retrieves the value stored in the promise object:

```
class UserFacing {
    // ...

  public:
    int next_value() {
        handle.resume();
        return handle.promise().yielded_value;
    }
};
```

With this setup, the first five calls to `next_value()` on the user-facing object will return the values yielded by the example coroutine shown above: 100, 1, 2, 3, and 200. So far, so good!

But what will the *next* call return, if there is one?

On the sixth call to `next_value()`, the coroutine will be resumed immediately after the `co_yield 200` statement. That's the last statement in the coroutine body, so control flow will fall off the end, causing the coroutine to finish, and suspend itself at its termination point via the `std::suspend_always` awaiter returned from `final_suspend()`.

But nothing was *yielded*. So nothing called the `yield_value` method. So nothing wrote to the `yielded_value` member variable. So it will still have the same value it had before.

In other words, the sixth call to `next_value()` will return 200 *again*, simply because that was what was lying around in `yielded_value` from the previous call.

To fix this, the simplest thing is to write some dummy value into `yielded_value` *before* resuming the coroutine. Then, if it turned out not to yield anything at all, the dummy value will still be there after the `resume()`.

We could designate some specific `int` value as meaning 'nothing was yielded'. Maybe 0, or −1, or `INT_MIN`, or some other thing we thought we were unlikely to need for real. But that's not a C++20 way of doing things. A nicer approach is to change the type of `yielded_value` from `int` to `std::optional<int>`, so that there's a way to represent 'no integer here at all' that can be distinguished from *any* possible `int` value.

Then the replacement `next_value()` looks like this:

```
class UserFacing {
    // ...

  public:
    std::optional<int> next_value() {
        auto &promise = handle.promise();
        promise.yielded_value = std::nullopt;
        handle.resume();
        return promise.yielded_value;
    }
};
```

Now the return type of `next_integer()` is also `std::optional<int>`, which means that the caller will also get to find out whether an actual integer was yielded or not.

Full source code for this section: `co_demo.yield.cpp` (the simpler version without `std::optional`), and `co_demo.yield_optional.cpp` (the full version that uses `std::optional` so it can can signal end of stream).

Detecting that a coroutine has finished

In the previous section, we arranged that if you call `next_value()` on the user-facing object *six* times, you first receive five `std::optional<int>` containing the numbers 100, 1, 2, 3, 200 (as yielded by the code inside the coroutine), and then an empty one indicating the end of the sequence.

What happens if the caller calls `next_value()` *again*, a seventh time?

At this point, the coroutine will have reached its termination point; `final_suspend()` will have been called on the promise object. Resuming the coroutine after this is an outright error, which will (at least in gcc) lead to a crash.

If your caller code is 100% reliable about recognising the `std::nullopt` sentinel value as the end of the sequence, this might not matter, because it might avoid ever calling the `next_value()` method again. But if it's not quite that organised, you'd like to arrange that `next_value()` is safe to call an extra time. Perhaps it should return `std::nullopt` on *all* calls after the coroutine runs out of things to yield.

To arrange this, we need to find out whether the coroutine has finished, and not try to resume it if it has. Fortunately, that's easy, because coroutine handles also come with a `done()` method, returning `bool`.

```
class UserFacing {
    // ...

  public:
    std::optional<int> next_value() {
        auto &promise = handle.promise();
        promise.yielded_value = std::nullopt;
        if (!handle.done())
```

```
            handle.resume();
        return promise.yielded_value;
    }
};
```

That's better: now you can't crash the coroutine machinery *even* if you call methods of the user-facing object's API in the wrong order.

Full source code for this section: `co_demo.done.cpp`.

Delivering a final result with `co_return`

Now I'll show how to set things up so that the `co_return` statement can be used to deliver an extra piece of data.

For many types of coroutine, you probably don't need this at all. But I suggested one example in a previous section where you might want to: if the coroutine is performing a network transaction, by being called back from an event loop every time network data arrives, then it might use `co_yield` and `co_await` to interact with the network service (sending and receiving data respectively), and then use `co_return` to communicate with whatever part of *this program* requested the transaction in the first place, telling it whether the transaction was successful, or what the result of a query was.

If you want to use `co_return` with a value, then you have to define a `return_value` method in your promise type, which accepts a value of whatever type you want to `co_return`:

```
class UserFacing {
    // ...
  public:
    class promise_type {
      public:
        std::optional<std::string> returned_value;

        void return_value(std::string value) {
            returned_value = value;
        }
    };

    // ...
    std::optional<std::string> final_result() {
        return handle.promise().returned_value;
    }
};
```

As usual, it's up to you what you *do* with the returned value. In this case, I've just stored it in another member variable of the promise object, and provided a method on the user-facing type that can return it on request. So you could instantiate a coroutine of this type, keep calling `next_value()` until the coroutine has finished yielding things, and then call `final_result()` to get whatever status or summary information the return value was reporting.

The most important thing, if you do this: **you must also remove the `return_void` method of the promise type**. It's the rules: the C++ standard says that your promise type *must* implement *exactly one* of `return_void` and `return_value`.[3]

3

Full source code for this section: `co_demo.return.cpp`.

Handling exceptions thrown out of the coroutine

We've now filled in most of the methods in the original minimal example. But one we haven't touched yet is the `unhandled_exception()` method.

You can throw and catch exceptions inside a coroutine as normal (although, for some reason, there's a rule that you can't `co_yield` or `co_await` inside a catch block). But if you throw an exception that *isn't* caught within the coroutine, then what happens?

For a start, that will automatically terminate the coroutine, just like throwing an exception right out of a normal function. You won't be able to resume the coroutine afterwards: there would be nowhere to resume *from*.

But just before that happens, the `unhandled_exception()` method will be called on the promise object. This doesn't take any actual arguments, but it can retrieve the in-flight exception by calling `std::current_exception()`, which will return a value of type `std::exception_ptr`. Then you can propagate that exception to somewhere else, by calling `std::rethrow_exception()` to re-throw it.

In many cases, the obvious thing to do is to propagate an exception in the coroutine to the call site where it was last resumed. For example, in the 'generator' style of coroutine we're using as our current running example, which is yielding a sequence of values, you'd probably like it if an exception thrown inside the coroutine propagated out to the place where someone had called `next_value()` on the promise object.

(If nothing else, that's how exceptions work in Python generators, so if you want something that behaves as much as possible like one of those, this is how you get it.)

So you might do something like this:

```
class UserFacing {
    // ...
  public:
    class promise_type {
        // ...

      public:
        std::exception_ptr exception = nullptr;
        void unhandled_exception() {
            exception = std::current_exception();
        }
    };

    std::optional<int> next_value() {
        auto &promise = handle.promise();
        promise.yielded_value = std::nullopt;
        promise.exception = nullptr;
        if (!handle.done())
            handle.resume();
        if (promise.exception)
            std::rethrow_exception(promise.exception);
        return promise.yielded_value;
    }
};
```

(Note that we set `promise.exception` to a null pointer before resuming the coroutine, for the same reason that we clear `promise.yielded_value`: if we didn't, and the client made a further call to `next_value()`, it would throw the same exception again, simply because it was still lying around in that field.)

If you *don't* do anything in `unhandled_exception()`, then the exception will simply be thrown away at the point where it exits the coroutine. It's as if the coroutine body was contained in an implicit `try`/`catch` statement, where the catch clause calls your `unhandled_exception()` method and then assumes that that's all it needs to do.

This isn't the *only* thing you might want to do with an exception thrown out of a coroutine. If you set up a system in which one coroutine can call another as a subroutine (and have it yield on its behalf), then you'd probably want an exception thrown in the sub-coroutine to propagate out into the calling coroutine, just as if they were a caller/callee pair of ordinary functions. In that situation, you'd still want the `unhandled_exception()` method to store the exception, but you'd rethrow it in a different situation. We'll see an example of that in a later section.

Full source code for this section: `co_demo.exception.cpp`.

Writing custom awaiters

Every time your coroutine is suspended, or even *potentially* suspended, an 'awaiter' object is constructed, and used to control whether the suspension happens and what its effects are.

So far, we've been getting along with only the ready-made awaiter types provided by the standard library: `std::suspend_always` and `std::suspend_never`. Now it's finally time to tackle writing our own.

An awaiter type doesn't have to be a derived class of any specific base. It's just any type that implements the following three methods (in which some of the types are not fully specified):

```
class Awaiter {
  public:
    bool await_ready();
    SuspendReturnType await_suspend(std::coroutine_handle<OurPromiseType> handle);
    ResumeReturnType await_resume();
};
```

The first of these methods, `await_ready()`, controls whether the coroutine gets suspended at all. If `await_ready()` returns `true`, the coroutine just continues running. If it returns `false`, the coroutine is suspended.

(Why that way round? You probably executed `co_await` because you needed to wait *for something* – for example, maybe this coroutine requires the result of some other operation in order to make further progress, so it has to wait until that result is available. So the idea is that `await_ready()` tests whether the *thing you're waiting for* is already ready. If so, it returns `true`, and you know you don't need to wait for it after all.)

If `await_ready()` returns `false`, then `await_suspend()` is called. This receives the coroutine handle (which means it can also access the promise object, by calling `handle.promise()`). It has a choice of return types:

- It can return `void`, in which case the coroutine is suspended and control is returned to whoever last resumed it.
- It can return `bool`, in which case the coroutine is only suspended if the return value is `true`. A return value of `false`[4] means that the coroutine doesn't suspend after all.
- It can return *a handle to another coroutine*. In this case, the coroutine is suspended, but control is not immediately returned to the code that last resumed it. Instead, the coroutine whose handle you returned will resume running. Of course that coroutine could *also* transfer to another coroutine in turn when it suspends. Control will only be returned to the resumer when a coroutine suspends *without* nominating another coroutine to switch to.

What if you'd like `await_suspend()` to only *conditionally* transfer control to another coroutine? Then you have to declare it as returning a coroutine handle (or it can't nominate another coroutine at all), but it needs a value to return meaning 'don't resume anything else, just return to the caller'.

For this purpose, the standard library provides a 'no-op coroutine' which always suspends itself without doing anything. So if you've declared `await_suspend()` as returning a coroutine handle, and in a particular case you want to just return to the caller, you can return `std::noop_coroutine()` to do that.

OK, what if you'd like `await_suspend()` to also be able to *conditionally* not suspend at all?

In that case, you can return the coroutine handle you were passed as an argument. Then the same coroutine will be immediately resumed, which is the same thing as not suspending it in the first place.

So the version of `await_suspend()` that returns a coroutine handle is the most general of them all: it can choose not to suspend at all (by returning its argument), to suspend and return to the caller (by returning `std::noop_coroutine()`), *or* to transfer to a different coroutine. The `void` and `bool` return types are just simpler ways to get a subset of those behaviours.[5]

Finally, `await_resume()` is called when the coroutine is ready to carry on running (whether because it was suspended and later resumed, or because it wasn't suspended in the first place). The return value of `await_resume()` is passed in to the coroutine itself, as the value of the `co_await` or `co_yield` expression.

For example, if you decided to set up an awaiter that would let you use `co_await` to wait for a network transaction to complete, you could also arrange that the result of the transaction was passed to the coroutine in

this way. That would let you write code along the lines of

```
UserFacing my_coroutine() {
    // ...

    std::optional<SomeData> result = co_await SomeNetworkTransaction();
    if (result) {
        // do something with the output
    } else {
        // error handling
    }

    // ...
}
```

just as if you were doing the same thing with ordinary function calls.

That's everything that an awaiter can do. For convenience, here's a summary of all the situations where you can provide a custom awaiter:

- When the coroutine is initially created. This lets you choose whether it starts suspended, or to immediately begin running until it reaches its first yield or await point.
- On any call to `co_await` or `co_yield`. This lets you configure those operations to not suspend at all, or to hand control over to a different coroutine; to communicate with the rest of the program to arrange when the coroutine will next be resumed; and to return a useful value to the coroutine after the await or yield is complete.
- When the coroutine terminates (either by return or by exception). In this situation, you must not try to make it continue running, but you can either suspend it normally at its end point, or transfer control to a different coroutine as its final action.

Full source code for this section, demonstrating lots of simple custom awaiters: `co_demo.awaiters.cpp`.

Identifying the promise type using `std::coroutine_traits`

In all the example code so far, I've shown the promise type being defined as a class called `promise_type` inside the user-facing type. But in the section where I set that up, I said that wasn't the only way to do it.

What *really* happens, when you write a coroutine, is that the C++ implementation makes an instance of the templated type `std::coroutine_traits`, based on the type signature of the coroutine function. Then it asks *that* what the promise type is.

The *default* implementation of that template, provided by the C++ standard library, finds the return type `T` of the function, and then expects `T::promise_type` to be a type that it can use as the promise. But if that's not where you want to keep the promise type, you can override that behaviour, by defining your own template specialisation of `std::coroutine_traits`.

Why might you want to do that, instead of the default approach of defining `T::promise_type`?

One reason is if you *can't* put a type name of your choice inside the user-facing type. This might happen if it's a type that's not under your control at all. For example, it might be a type defined in the standard library, such as `std::unique_ptr`. Or it might be something even simpler, like a plain pointer type, or even an `int`. In a later section I'll show one example of why you might want to use a type you don't control.

Another reason is that the `std::coroutine_traits` template doesn't just get to examine the *return* type of the coroutine. It gets to see all the argument types as well. So if your promise type needs to depend on those, then you'll need to write a template specialisation to specify it.

Here are a few concrete examples, to show the syntax.

Firstly, the simplest possible example:

```
template <>
struct std::coroutine_traits<UserFacing> {
    using promise_type = SomePromiseType;
};
```

This tells the compiler: if a coroutine is declared as returning the type `UserFacing` (whatever that is) *and taking no arguments at all*, then its promise type should be `SomePromiseType` (which we imagine you've defined already).

Next, add some specific argument types:

```
template <>
struct std::coroutine_traits<UserFacing, bool, char *> {
    using promise_type = SomePromiseType;
};
```

This specialisation will match only a coroutine which returns `UserFacing` and takes exactly two arguments, of types `bool` and `char *`.

But often you don't really care about the argument types. Here's how to ignore them:

```
template <typename... ArgTypes>
struct std::coroutine_traits<UserFacing, ArgTypes...> {
    using promise_type = SomePromiseType;
};
```

This specialisation will match *any* coroutine which returns `UserFacing`, regardless of its number and type of arguments. So if you want the promise type to depend *only* on the coroutine's return type, but you still don't want to (or can't) do it by defining `UserFacing::promise_type`, then this is how you can do it most generally.

Giving the promise type access to the coroutine parameters

In all the examples so far, I haven't written any constructor in the promise class. As a result, C++ automatically invents a default constructor which takes no arguments.

But that's not the only way to do it. If a suitable constructor is available, the promise class will be constructed in a way that gives it the coroutine's parameters as arguments. For example, you could do this:

```
class UserFacing {
  public:
    class promise_type {
      public:
        promise_type(int x, std::string_view sv) { /* ... */ }
        // ...
    };
};

UserFacing demo_coroutine(int x, std::string y) {
    // ...
    co_return;
}
```

and then when you make an instance of the coroutine by calling, say, `demo_coroutine(1, "foo")`, the promise type's constructor will be called with the same arguments `1, "foo"` that the coroutine itself will receive.

The argument types don't have to be exactly identical. It only has to be possible to convert the arguments to fit, in the same way that an ordinary function call would work. For example, here, I've made the constructor take a `std::string_view` where the actual function takes a `std::string`, and this still works fine – the compiler does the automatic conversion from `string` to `string_view` when it calls the promise constructor.

Doing it that way avoids an extra copy operation: if I'd made the promise constructor take an ordinary `std::string`, then the compiler would have had to copy-construct the input string an extra time in order to pass

an independent copy of it to the promise constructor. Unless you really *need* that, it's better to avoid the extra work.

(Of course, I could also have declared the promise constructor as taking a `const std::string &`, the way it would usually be done in earlier versions of C++. But `string_view` is the shiny new more flexible way of doing this kind of thing.)

Why might you want to do this?

One reason is so that some of the coroutine's parameters can be *entirely* intended to control the promise class, and ignored by the coroutine itself. For example, suppose you needed the promise class to contain a pointer to something in the main-loop infrastructure. The easiest way to get that pointer into the promise class in the first place would be at construction time. So you might do something like this:

```
class UserFacing {
  public:
    class promise_type {
        MainLoopThingy *mlt;

      public:
        template<typename... ArgTypes>
        promise_type(MainLoopThingy *mlt, ArgTypes &&...) : mlt(mlt) {
            // maybe also tie this promise object into the main loop right here
        }

        // ...
    };
};

UserFacing demo_coroutine(MainLoopThingy *, int x, std::string y) {
    // this code ignores the MainLoopThingy and uses just the other parameters
    co_return;
}
```

In this example, I've set up the promise type's constructor as a variadic template, so that it doesn't care what arguments it gets *other* than the initial `MainLoopThingy` pointer that it wants to save. So coroutines using this promise class don't all have to have exactly the same function prototype: they *only* have to have a `MainLoopThingy *` as their first argument.

(If you're going to do things this way, you might *also* want to use the `std::coroutine_traits` system shown in the previous section to select the promise type, so that you can define coroutines with different pointer types in the first parameter and automatically select different promise types for them. But that's only necessary if they also need to have the *same* user-facing return type.)

Another special case of this is if your coroutine is a method of a class. (That's perfectly allowed, and I'll talk more about it in a later section.) In that situation, the promise type's constructor (if you declare one with arguments at all) will receive a reference to the class as its first parameter, followed by the method arguments. For example:

```
class MyClass;

class UserFacing {
  public:
    class promise_type {
        MyClass *c;
      public:
        template<typename... ArgTypes>
        promise_type(MyClass &c, ArgTypes &&...) : c(&c) {}

        // ...
    };
};

class MyClass {
```

```
  public:
    UserFacing coroutine_method(int x, std::string y) {
        // ...
        co_return;
    }
};
```

Here, when `coroutine_method` is called on an instance of `MyClass`, the promise type constructor will receive a reference to the class instance (the same as if the method itself had referred to `*this`), followed by the `int` and `std::string` arguments given to the method call.

In the promise type, I've shown the same kind of templated constructor as before, so that the method's argument list can be whatever it likes. But the constructor is expecting to receive a `MyClass &` as its first parameter, and will store a pointer to the class. This enables the promise type to work closely with its containing class.

However, beware! If the instance of `MyClass` is copied, then the promise class can only hold a pointer to one of the copies. And if it's moved, the promise class will need its pointer to be updated. So if you're going to do this at all, you should either make `MyClass` completely immovable (by deleting its copy and move constructors *and* its copy and move assignment operators), or else make it a move-only type and add code in the move constructor and move assignment operator to update the pointer in its associated promise class.

(That's why I showed the promise type as containing a *pointer* to `MyClass`, rather than a reference. A pointer can be overwritten later, if the owning class moves.)

Examples of non-trivial coroutine systems

Now I've listed all the various bits and pieces that C++ lets you put into your promise and awaiter types. So in principle you have all the knowledge you need to go off and use it in interesting ways.

But it's probably helpful to show a couple of examples of how to put the pieces together into an interesting shape.

Shuttling along a producer/adapter/consumer chain

One of the earliest examples of coroutine use – perhaps *the* earliest, from the presentation of them in 'The Art of Computer Programming' – is to have one coroutine producing a stream of objects, and another consuming them. The coroutine setup gives each of those pieces of code the illusion that it's calling the other one as a subroutine, and allows it to organise its control flow however it thinks best, even if that involves multiple loop and `if` statements with 'calls' to the other routine in several different places.

An obvious extension of this is to lengthen the chain to more than two routines, so that a coroutine in the middle is receiving a stream of values from the coroutine to its 'left' and passing on filtered or modified values to the one on its 'right'. This means that a coroutine in the middle needs two different ways to ask to suspend itself: one when it's ready to receive a new value, and one when it has a value to deliver.

In C++, we have two syntaxes that are good for just that kind of thing: a coroutine can use `co_await` to ask for a value from its supplier, and `co_yield` to provide a value to its consumer. Also, if we write some custom awaiters, we can arrange for each coroutine to automatically transfer control back and forth along the chain, only returning to the caller when there's a final output value to be delivered.

(That's not the only way to do it. An alternative would be to write a sort of 'executor' loop in `main()` which had a list of suspended coroutines, and whenever one suspended itself, it would deliver to the executor some indication of which one to resume next. In some situations you might prefer that – especially if you had to have some infrastructure of that type for other reasons too, like migrating things between threads, or polling on I/O sources. But in this example I want to demonstrate the use of custom awaiters to implement a purely computational multi-coroutine setup *without* a separate executor.)

To illustrate this, let's set up an actual example: I'll use coroutines to implement the well known 'FizzBuzz' coding exercise, in which you iterate over consecutive integers from 1 upwards, printing "Fizz" for a multiple of 3, "Buzz" for a multiple of 5, "FizzBuzz" if a number is both at once, or printing the number itself if it's neither.

In order for our coroutines to pass control directly to each other, they'll have to be linked together in some way, so they can find each other. We'll do this by passing each coroutine's user-facing object as an argument to the function that constructs the next one.

In other words, we want to end up writing coroutines in this kind of style:

```
UserFacing generate_numbers(int limit) {
    for (int i = 1; i <= limit; i++) {
        Value v;
        v.number = i;
        co_yield v;
    }
}

UserFacing check_multiple(UserFacing source, int divisor, std::string fizz) {
    while (std::optional<Value> vopt = co_await source) {
        Value &v = *vopt;

        if (v.number % divisor == 0)
            v.fizzes.push_back(fizz);

        co_yield v;
    }
}
```

and then `main()` would construct and use a chain of them like this[6]:

6

```
int main() {
    UserFacing c = generate_numbers(200);
    c = check_multiple(std::move(c), 3, "Fizz");
    c = check_multiple(std::move(c), 5, "Buzz");
    while (std::optional<Value> vopt = c.next_value()) {
        // print output in the appropriate format
    }
}
```

So we start by constructing the ultimate source coroutine, whose job is just to generate a sequence of integers, and wrap them up into a 'Value' structure (where later coroutines can accumulate the fizzes and buzzes that will be printed for that number). Then we call the second coroutine function `check_multiple()` to append a step that checks for multiples of 3 and marks them as 'Fizz', and then add a second instance of that for multiples of 5 and 'Buzz'.

Each user-facing object is passed as an argument to the coroutine that will consume its output – via `std::move`, because our user-facing objects are deliberately uncopyable to avoid double-freeing of coroutine handles.

Each coroutine requests input from its supplier by means of a `co_await` expression using the user-facing object as an argument, which means that our promise type is going to have to implement an `await_transform(UserFacing &)` function that returns an appropriate awaiter. And each coroutine delivers output via `co_yield`, which means that the promise type will also need a `yield_value(Value)` function which returns a different awaiter.

Note that these coroutines know where they're awaiting values *from*, but they don't know where they're yielding values *to*. In particular, the two instances of `check_multiple()` we constructed will be doing completely different things in response to `co_yield`: the 'Fizz' coroutine will be passing its yielded values to the 'Buzz' coroutine, but the 'Buzz' coroutine will execute exactly the same code and *its* output values will have to go back to `main()` and be returned from the call to `c.next_value()`.

Also, each coroutine yields values in the form of our 'Value' structure, but by the time those values reach the consumer (whether it's another coroutine or `main()`, they've become a `std::optional<Value>`. This allows us to signal the end of the stream by returning `std::nullopt`.

OK, that's how we want to *use* the system. Now we have to implement it!

To make a coroutine system that can be used in the style shown above, we start from something very like the example code we've been developing so far: our promise type has a `yielded_value` field which the user-facing object's `next_value()` method can retrieve an item from. The new wrinkle is that a consumer coroutine *C* has to be able to `co_await` another one *P*, in which case:

- *C* has to transfer control to *P* on suspend, via a custom awaiter invoked by the `co_await P` expression
- *C* has to tell *P* its own identity, so that *P* knows where to transfer control back to
- *P* has to transfer control back to *C* via another custom awaiter invoked by `co_yield`
    - but if a coroutine executes `co_yield` when it was *not* first awaited by another coroutine, then the same awaiter must instead suspend normally, returning a value to the ultimate caller in `main()`
- when *C*'s awaiter resumes *C*, it must retrieve *P*'s `yielded_value` field so that that can become the return value from `co_await`.

We do this by having two awaiter types: `InputAwaiter` to handle requests for data via `co_await`, and `OutputAwaiter` to handle providing data via `co_yield`.

Here's the `InputAwaiter`, and the `await_transform()` function that makes one when needed:

```
class UserFacing {
    // ...

    class InputAwaiter {
        promise_type *promise;
      public:
        InputAwaiter(promise_type *promise) : promise(promise) {}

        bool await_ready() { return false; }

        std::coroutine_handle<> await_suspend(std::coroutine_handle<>) {
            promise->yielded_value = std::nullopt;
            return handle_type::from_promise(*promise);
        }

        std::optional<Value> await_resume() {
            return promise->yielded_value;
        }
    };

    // ...

    class promise_type {
        promise_type *consumer;

        // ...
        InputAwaiter await_transform(UserFacing &uf) {
            promise_type &producer = uf.handle.promise();
            producer.consumer = this;
            return InputAwaiter{&producer};
        }
    };
};
```

In the above code, `await_ready()` does the default thing of always actually suspending the coroutine. `await_suspend()` has to transfer control to the coroutine whose output we're awaiting, which it does by using a pointer to that coroutine's promise object, which was provided to the `InputAwaiter` constructor.

To get the value yielded by the producer coroutine back to our own coroutine, `await_resume()` recovers it from the `yielded_value` field of the other promise object, in exactly the same way that `UserFacing::next_value()` did in our earlier example. Also, to detect the case where the producer coroutine terminated *without* having yielded anything, `await_suspend()` clears the `yielded_value` field before transferring control to the coroutine – again, just the same way that `UserFacing::next_value()` does it. So the logic of recovering a yielded value is the same here (where the value is being passed to the next coroutine in the chain) and in `next_value()` (where it's being returned to `main()`).

The other important point is that our promise object needs a new field, so that a promise object knows what its 'consumer' is – that is, the coroutine that it's yielding values to, if any. This is initialised by `await_transform()`: when a consumer coroutine is about to await a producer, it fills in the producer's `consumer` field to point back to itself. That way the `OutputAwaiter` will know where to transfer control back to.

Speaking of which: here's `OutputAwaiter`, and the `yield_value()` function that returns one.

```
class UserFacing {
    // ...

    class OutputAwaiter {
        promise_type *promise;
      public:
        OutputAwaiter(promise_type *promise) : promise(promise) {}

        bool await_ready() { return false; }

        std::coroutine_handle<> await_suspend(std::coroutine_handle<>) {
            if (promise)
                return handle_type::from_promise(*promise);
            else
                return std::noop_coroutine();
        }

        void await_resume() {}
    };

    // ...

    class promise_type {
        // ...
        OutputAwaiter yield_value(Value value) {
            yielded_value = value;
            return OutputAwaiter{consumer};
        }
    };
};
```

This is slightly simpler than `InputAwaiter`. The `yield_value()` method has to fill in the promise object's `yielded_value` field – but it doesn't need to know whether the consumer will be another coroutine or `main()`, because it works the same way in both cases.

But the `await_suspend()` method *does* need to know, because it has to decide whether to transfer control back to its consumer coroutine, or whether to suspend completely and return to `main()`. As discussed in a previous section, it does the latter by returning `std::noop_coroutine()`.

Full source code for this example: `co_shuttle.cpp`. (As I warned in the introduction, the real code will have to move some methods out of line that are shown inside the class definitions above.)

Stacked generators

It's irresistible to compare C++ coroutines with Python generators. I've mentioned the comparison several times already in this article. The ready-made `std::generator` in C++23 defines a coroutine setup that's almost exactly like a Python generator (in that it can only `co_yield` a stream of values, can't `co_await` input or `co_return` a final result, and the user-facing object is an iterable that can be used in combination with range-`for` or other iterator-related systems like views).

One Python feature we haven't talked about yet is the '`yield from`' construction, in which a generator function can specify another source of output values, which will be delivered to the recipient as if yielded by the generator itself, and the generator's code will resume running after the subsidiary iterable is exhausted. Put another way, the first generator is able to call another generator as a subroutine, and let it yield things on its behalf.

You can fake it, of course, by having the primary generator iterate over the secondary one, and manually pass each of its outputs to `co_yield`:

```
std::generator<Value> inner();

std::generator<Value> outer() {
    co_yield "hello";

    // 'yield from inner()'
    for (Value v: inner())
        co_yield v;

    co_yield "goodbye";
}
```

But as you nest ‘subroutine calls’ deeper and deeper in this style, you lose efficiency, because every time a value is yielded to the recipient, each of the coroutines in the call stack is suspended one after another; the yielded value is probably copied (or at least moved) from promise object to promise object the same number of times; and when everything resumes again, the call stack has to be built back up one frame at a time. So a yield and resume from *n* levels deep takes $O(n)$ time. It would be better to have built-in, first-class support for the outer coroutine to allow an inner one to yield (or await) on its behalf, and stay completely out of the way until the inner one has finished. That way a yield and resume would take $O(1)$ time no matter how deep you go.

This is an even more useful feature for more general coroutine systems, which do more complicated things than Python generators, such as bidirectional I/O. If you were writing conventional procedural I/O code with lots of calls to input and output routines, it often makes sense to call a subroutine of your own which can do some of that I/O on your behalf, so that a piece of your control flow can be reused in multiple places. There’s no reason you wouldn’t want to do that in a coroutine-based analogue of the same setup.

In conventional procedural I/O, your subroutine would also be able to return a status value indicating whether its part of the overall task had been successful or not, or what value it had found out. An example might be `getaddrinfo()` in the standard sockets API, which performs a DNS lookup, and ends up returning a record describing the results. And, similarly, in coroutine-structured code that would be just as useful – since the whole point is that you don’t want to change the layout of the code just because it changed to being coroutine-based.

In this section I’ll show how to achieve this using a custom C++20 promise class[7]. 7

The general idea is that you’ll use `co_await` as the equivalent of Python ‘`yield from`’: you pass it a newly constructed coroutine, and you won’t be resumed until the coroutine has *completely finished*. Any values yielded by the subroutine will go to the same place your own yields would go; if the subroutine `co_returns` a value, then that will be the result of the `co_await` expression. So you could write something like this, for example:

```
StackableGenerator<std::string, int> inner() {
    co_yield "hello";
    co_yield "world";
    co_return 2;
}

StackableGenerator<std::string> outer() {
    co_yield "start";
    int result = co_await inner();
    co_yield to_string(result);
    co_yield "finish";
}
```

This example defines a stack of two generators, both yielding `std::string`. The inner one also returns an `int` (here, indicating how many things it yielded). So in this example, the sequence of strings yielded would be:

- “start” from the outer coroutine
- “hello” and then “world” from the inner coroutine
- “2” from the outer coroutine, derived from the integer return value passed back from the inner one (I assume for convenience that we have a helper function that translates an integer to a `std::string`)
- “finish” from the outer coroutine.

In this example, my user-facing type `StackableGenerator` is a template, with two parameter types. The first one is the type the coroutine will be yielding, and is mandatory. The second one indicates what type the coroutine will be `co_returning`: it's optional, and defaults to `void` if you leave it off.

A consequence of this is that *return* types can vary up and down a stack of calls – in the same way that a nesting of ordinary function calls need not all return the same type as each other, they only have to each return the same type that their caller was expecting. But the *yield* type must match: if I'm supposed to be yielding strings, and I call a subroutine to yield things on my behalf, then those will need to be strings too, or my consumer will be very confused.

Our promise class will need to be templated on both types, because it has to deal with both the yield and return type, so it needs to know what they are. But it also has to be possible to link together promise classes with different return types into a list – which means we'll need the promise class to be derived from an abstract base class that has only *one* template parameter, the yield type. Something like this:

```
template <typename Yield>
class PromiseBase {
    // manage a linked list of promises with the same yield type
};

template <typename Yield, typename Return>
class Promise : public PromiseBase<Yield> {
    // all the usual functionality of a promise type
};
```

Actually, I make it even more complicated than that, because of the rule that promise classes must have *exactly one* of the `return_void()` and `return_value()` methods. It's convenient to make a middle layer in this class hierarchy to hold all the common code that *does* need to know the return type (so that it can fully identify the user-facing type) but *doesn't* depend on whether the return type is `void` or not, and then have the actual `Promise` class just add one of the two return methods, by means of defaulting to adding `return_value()` but having a template specialisation for the `void` return type:

```
template <typename Yield>
class PromiseBase {
    // manage a linked list of promises with the same yield type
};

template <typename Yield, typename Return>
class PromiseMid : public PromiseBase<Yield> {
    // most of the standard promise methods, except return_void/return_value
};

template <typename Yield, typename Return>
class Promise : public PromiseMid<Yield, Return> {
  public:
    void return_value(Return r) { /* deal with returning a value */ }
};

template <typename Yield>
class Promise<Yield, void> : public PromiseMid<Yield, void> {
  public:
    void return_void() {}
};
```

The purpose of the `PromiseBase` base class is to form a linked list corresponding to the 'call stack' of coroutines that are each waiting for the next one to finish completely. Here are some more details of that system.

```
template <typename Yield> class PromiseBase {
  public:
    PromiseBase *outer, *parent, *current;
    std::optional<Yield> last_yield;

    PromiseBase() {
        outer = current = this;
```

```
        parent = nullptr;
    }

    void link_to_callee(PromiseBase *callee) {
        callee->outer = outer;
        callee->parent = outer->current;
        outer->current = callee;
    }

    void unlink_from_caller() {
        outer->current = parent;
    }

    // ...
};
```

For every coroutine in a call stack, the field `outer` points to the `PromiseBase` object corresponding to the outermost call on the call stack – the one that was invoked by something that *wasn't* a 'sub-coroutine call' made by `co_await`. Meanwhile, the `parent` field links the routines together into a linked list, with each callee pointing to its caller. (So `outer` is distinguished by its `parent` field being a null pointer.)

The other two fields shown here, `current` and `last_yield`, are only used in the outermost promise object. `current` points to the coroutine that's currently running, i.e. the *innermost* call in the whole call stack; it's easiest to have only `outer->current` hold this value, so that it only ever needs updating in one place, and when resuming the stack of coroutines, it's easy to find which one to resume. Similarly, whenever anything in the whole stack yields a value, it's written into the outermost promise object rather than the one corresponding to the coroutine that did the yielding, so that the user-facing object returning a yielded value to the consumer can retrieve it from the same place regardless.

The method `link_to_callee` appends a newly created coroutine to the end of the chain, when a new subroutine call is being made; it will be called by `await_suspend()`, handling the `co_await` operation that implements the call. We see that it sets the callee's `outer` field to match the caller's; until then, `outer` just pointed at `this`, because the default constructor sets up every `PromiseBase` to believe it's the outermost call in a stack unless it finds out otherwise. Then, once the two objects agree on `outer`, the `outer->current` field is updated to mark the newly called function as the currently running one in the stack.

Conversely, `unlink_from_caller` is called on a callee which is about to terminate, and it resets `outer->current` to point to the routine it's about to return to. That will be called from `final_suspend()` in the terminating coroutine.

For calling and returning from subroutines, we'll need a pair of custom awaiter classes. I'll start by showing `SubroutineAwaiter`, the one that deals with making a call. This will be defined in the form of a base class and two tiny derived classes, in order to handle both `void` and non-void return types. The base class does nearly all the work, and each derived class fills in an appropriate `await_resume()` method, with or without a `return` statement that retrieves the return type:

```
template <typename Yield, typename Return> class SubroutineAwaiterBase {
    UserFacing<Yield, Return> callee;

  protected:
    Promise<Yield, Return> *callee_promise;

  public:
    SubroutineAwaiterBase(UserFacing<Yield, Return> &&callee) : callee(std::move(callee)) {}

    bool await_ready() { return false; }

    template <typename CallerReturn>
    std::coroutine_handle<> await_suspend(Handle<Yield, CallerReturn> caller)
    {
        callee_promise = &callee.handle.promise();
        caller.promise().link_to_callee(callee_promise);
        return callee.handle;
```

```
    }

    void check_exception() {
        if (this->callee_promise->exception)
            std::rethrow_exception(this->callee_promise->exception);
    }
};

template <typename Yield, typename Return>
class SubroutineAwaiter : public SubroutineAwaiterBase<Yield, Return> {
  public:
    Return await_resume() {
        this->check_exception();
        return this->callee_promise->ret;
    }
};

template <typename Yield>
class SubroutineAwaiter<Yield, void> : public SubroutineAwaiterBase<Yield, void> {
  public:
    void await_resume() {
        this->check_exception();
    }
};
```

Some important features of this code:

The template parameter `Return` refers to the return type of the *callee*: it's used to decide what the types of the class members `callee` and `callee_promise` are, and it also controls which of the two derived classes is used, and what type is returned from `await_resume()` if it's returning a value.

However, `await_suspend()` is passed a handle to the *caller* – the coroutine that's going to be suspended in favour of the callee. So that must be a template in its own right, so that it doesn't require the caller and callee to have the same *return* type. They must still have the same *yield* type, but their return types are independent.

`await_suspend()` also returns the coroutine handle of the callee, so that the callee can immediately begin running, without returning control to the consumer of the stack's yields.

The `check_exception()` method in the base class, called in both versions of `await_resume()`, arranges for exceptions thrown within a stack of coroutines to propagate up the stack, so that if the callee throws an exception, the caller can catch and handle it. The exception object is written into the throwing coroutine's promise object in the usual way (just as shown in the earlier exception handling section), but instead of always passing it back to `main()`, this system rethrows it in the calling coroutine if there is one. The user-facing object will only rethrow it in a non-coroutine context if it wasn't caught *anywhere* in the stack, and ended up terminating the *outermost* coroutine.

Finally, note that the `UserFacing` object passed to this awaiter is `std::moved` *into* the awaiter, so that it's consumed completely by the subroutine call. In particular, this means that the awaiter's destructor (which is simple enough for C++ to fill in automatically) will destroy the user-facing object for the callee, which will destroy the coroutine handle and clean up its internal state and promise object.

Here's the second custom awaiter, which is returned from the `final_suspend()` method in the promise object:

```
template <typename Yield, typename Return>
class FinalSuspendAwaiter {
    PromiseMid<Yield, Return> *promise;

  public:
    FinalSuspendAwaiter(PromiseMid<Yield, Return> *promise)
        : promise(promise) {}

    bool await_ready() noexcept { return false; }

    std::coroutine_handle<> await_suspend(Handle<Yield, Return>) noexcept {
```

```
        return promise->parent ? promise->parent->handle()
            : std::noop_coroutine();
    }

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

This awaiter is simpler, because it doesn't have to return a value at all. The only thing it does that's interesting is to automatically pass control to the parent coroutine of the one that's just finished – but only if there *is* a parent. In the case where the outermost coroutine of a call stack has finished, we return `std::noop_coroutine()`, because in *that* case, control is returned to the consumer, for good.

Finally, here's the method in the user-facing object that does the resuming, just to show how that has to vary in this setup:

```
template <typename Yield, typename Return> class UserFacing {
    // ...

  public:
    std::optional<Yield> next_value() {
        auto &outer = handle.promise();
        assert(outer.outer == &outer);
        if (!outer.current)
            return std::nullopt;

        auto curr = outer.current->handle();
        outer.last_yield = std::nullopt;
        outer.exception = nullptr;
        assert(!curr.done());
        curr.resume();

        if (outer.exception)
            std::rethrow_exception(outer.exception);

        return outer.last_yield;
    }
};
```

This is much like previous examples, except that we have to deal with two different promise objects: the outermost one, where yielded values are written, and the innermost one, which contains the coroutine handle we're actually going to resume. We expect `outer` to be the promise object owned by this `UserFacing` object, because `next_value()` should only be called from *outside* this coroutine stack, by a caller who didn't know or care if subroutine calls happened inside it. And the innermost promise is pointed to by `outer.current`, as shown in the linked-list snippet above.

So we use `outer.current` to find the innermost object, and then look in that to find the coroutine handle we resume. We expect that it should never be in a terminated state, because whenever a coroutine on the call stack terminates, it should already have cleaned itself up and popped itself from the linked list before we get back to here. Testing for the whole stack having already terminated is done by checking whether `outer.current` has become a null pointer (which would be done by the final call to `unlink_from_caller()`).

But once the coroutine stack is suspended, the yielded value comes from `outer`, because that's where everything on the stack will leave values; and the only exceptions we're interested in rethrowing to our caller also come from `outer`, because any exceptions thrown deeper in the stack are uninteresting unless they get all the way out here without already having been handled.

Full source code for this example: co_stack.cpp.

Stunts with the coroutine return type and location

Sharing a return type between coroutines and ordinary functions

I'm a big fan of coroutines in general, but even I have to admit that not *every* part of a program is best structured as a coroutine.

Sometimes, you want to have a family of objects with a common interface, but only *some* of them are implemented as coroutines, and others are ordinary C++ objects which implement the same interface by conventional means.

One way to arrange that is to make use of the fact that the caller of a coroutine doesn't need to *know* it's a coroutine. Suppose you have two functions defined in a header file that return the same type:

```
SomeReturnType this_is_a_coroutine(Argument);
SomeReturnType this_is_an_ordinary_function(Argument);
```

From the point of view of the caller of these functions, they have exactly the same API. You call one, and you get back an object of the specified return type.

It's the *definition* of the function that decides whether it's a coroutine or not. Suppose one of those functions has a `co_await`, `co_yield` or `co_return` keyword in its body, and the other doesn't. Then the function without a `co_`*`foo`* will simply be an ordinary function with the specified type signature, and its function body will be responsible for making an object of the appropriate return type and returning it in the usual way:

```
SomeReturnType this_is_an_ordinary_function(Argument) {
    int value = some_intermediate_computation();
    SomeReturnType to_return { value };
    return to_return;
}
```

But if the other function *does* contain at least one `co_`*`foo`*, then the C++ coroutine mechanism springs into action: it will figure out the promise type that goes with this function signature, and automatically generate code that constructs an instance of the promise type and calls its `get_return_object()` function to make an object to return to the initial caller.

But from the caller's point of view, it called both functions in the same way, and in each case, it got back an identical-looking object. It's just that those objects will have some kind of a method (say, `get_value()`), which in one case is implemented normally, and in the other case works by resuming a coroutine hidden behind the scenes somewhere.

The obvious way to have an object type that can behave in more than one way is to make it an abstract base class, with an interface of virtual methods, and then have derived classes that implement those methods in different ways. So we'll start with one of those:

```
class AbstractBaseClass {
  public:
    virtual ~AbstractBaseClass() = default;
    virtual std::string get_value() = 0;
};
```

A *non*-coroutine derived class that implements this interface is a perfectly ordinary piece of C++:

```
class ConventionalDerivedClass : public AbstractBaseClass {
  public:
    ConventionalDerivedClass() = default;
    std::string get_value() override {
        return "hello from ConventionalDerivedClass::get_value";
    }
};
```

But then we might also make another derived class that wraps a coroutine handle, and implements the `get_value()` method by resuming it and communicating with the promise type, along the lines of this:

```
class CoroutineDerivedClass : public AbstractBaseClass {
  private:
    friend class Promise;
```

```
    Handle handle;

    CoroutineDerivedClass(Handle handle) : handle(handle) {}

  public:
    std::string get_value() override {
        handle.promise().yielded_value = "";
        handle.resume();
        return handle.promise().yielded_value;
    }

    ~CoroutineDerivedClass() {
        handle.destroy();
    }
};
```

I haven't shown the implementation of the promise type here, because it looks just like several examples we've already gone over in detail in previous sections. (I've kept things simple for this example by having `get_value()` return a plain `std::string` instead of a `std::optional<`*`something`*`>`.)

The only question is: what is the actual return type of the pair of coroutine and non-coroutine functions? It can't be `AbstractBaseClass` itself, because the two derived classes will be different sizes. It's going to have to be a pointer to a dynamically allocated object: either the raw pointer type `AbstractBaseClass *`, or more likely a smart pointer type such as `std::unique_ptr<AbstractBaseClass>`.

Either way, we can't make *that* return type contain a thing called `promise_type`. If it's a raw pointer, it can't contain named fields at all; if it's a smart pointer from the standard library, then the standard library is in control of what named fields it has, and user code can't add to that. So we're going to have to use the `std::coroutine_traits` technique for identifying the promise type, as shown in an earlier section:

```
template<typename... ArgTypes>
struct std::coroutine_traits<std::unique_ptr<AbstractBaseClass>, ArgTypes...> {
    using promise_type = Promise;
};
```

That arranges that if we define a function returning `std::unique_ptr<AbstractBaseClass>`, and its body contains any `co_`*`foo`* keyword that makes it a coroutine, then its promise type is going to be `Promise`. So as long as `Promise::get_return_object()` cooperates with this definition by having a return type compatible with `std::unique_ptr<AbstractBaseClass>`, this will all work[8]:

[8]

```
class Promise {
    // ...
  public:
    std::unique_ptr<CoroutineDerivedClass> get_return_object() {
        return std::unique_ptr<CoroutineDerivedClass>(
            new CoroutineDerivedClass(Handle::from_promise(*this)));
    }
};
```

And now we can make a pair of functions that have the same return type, but totally different semantics:

```
std::unique_ptr<AbstractBaseClass> demo_coroutine() {
    co_yield "hello from coroutine, part 1";
    co_yield "hello from coroutine, part 2";
}

std::unique_ptr<AbstractBaseClass> demo_non_coroutine() {
    return std::make_unique<ConventionalDerivedClass>();
}
```

in which the body of `demo_non_coroutine()` is run immediately when it's called, and constructs an object to return, whereas the coroutine-style function body of `demo_coroutine()` is suspended, and the only code that runs during the initial call is in the promise object. But the caller of these functions gets back two opaque

implementations of the same abstract base class, and can call `get_value()` on either one equally well, without having to know or care how each instance of the object implements that method.

This would all have worked just the same if I'd chosen to use the raw pointer type `AbstractBaseClass *` as my return type, instead of the fancy `std::unique_ptr`. You can still use that in the `std::coroutine_traits` specialisation. The only difference is that the caller who receives an `AbstractBaseClass *` from either function is responsible for manually deleting it.

Full source code for this example: `co_abstract.cpp`. (Again, the full source code fills in details I glossed over for clarity, like having to define methods out of line.)

Hiding a coroutine inside a class implementation

One of the useful properties of coroutines is that there's a data object that identifies a particular computation in progress, and that data object is directly accessible to the rest of the program, which can talk to it in other ways than just letting it run.

For example, C++ coroutines make it very easy to cleanly *abandon* a computation you don't need any more, by destroying the user-facing wrapper object, which destroys the coroutine handle, which frees the coroutine's promise object and internal state, which runs the destructors for all the coroutine's local variables. Assuming all those destructors do their jobs, all the memory will be freed, any other resources (like open files) will be released, and nothing will crash. (Compare that to the difficulty of cleanly terminating a *thread* that you've suddenly decided you don't need any more, without anything going wrong.)

Another thing you might want to do is to 'peek into' the coroutine's state while it's suspended. For example, if the coroutine represents an ongoing computation in a GUI program, then it might be useful to the GUI to look inside it every so often to update a progress bar. But this is rather awkward in the free-function style, because the internal variables of a coroutine can't be accessed in any useful way from outside it.

You can work around that by making the coroutine also be a method of a class. Then it can access the class member variables as well as its own local variables – so if there's any variable you want to be able to inspect while the coroutine is suspended, you can make it a class member variable.[9]

9

In other words, we want to arrange that one of our class methods is a coroutine; the class constructor calls that method to get the coroutine handle; but then the class *itself* takes over the duties that would normally be done by the user-facing object returned from the coroutine. Then one of the class's methods is implemented by resuming the coroutine, so that it can do a sequence of operations on each call in a stateful manner; but other methods can be interleaved with that one, to access the same class variables.

One quite simple way to arrange this is to make the return type of the coroutine simply *be* the coroutine handle:

```
class Promise;

template<class... ArgTypes>
struct std::coroutine_traits<std::coroutine_handle<Promise>, ArgTypes...> {
    using promise_type = Promise;
};

class Promise {
    // ...

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

Again, we use a template specialisation of `std::coroutine_traits` to specify that if a coroutine returns a `std::coroutine_handle<Promise>`, then its promise type is simply `Promise` itself[10].

10

And then we can write a class that has a coroutine as a private method, and transfer the coroutine handle straight out into the owning class, without having to mess about with a separate user-facing object:

```
class CoroutineHolder {
    int param;
    SomeType mutable_state;
    std::coroutine_handle<Promise> handle;

    std::coroutine_handle<Promise> coroutine() {
        for (int i = 0; i < param; i++) {
            co_await something_or_other;
            adjust(mutable_state);
            co_yield something_else;
        }
    }

  public:
    CoroutineHolder(int param) : param(param), handle(coroutine()) {}
    ~CoroutineHolder() { handle.destroy(); }

    void do_stateful_thing() {
        if (!handle.done())
            handle.resume();
    }

    int query_state() { return mutable_state.some_field; }
};
```

In this setup the coroutine can read class member variables like `param` which were set up by the class constructor, and therefore it doesn't need to be called with any extra arguments of its own. And it can *write* to class member variables (I've shown an example here called `mutable_state`), which a client of the class can query via methods *other* than the one that resumes the coroutine.

Of course, you still have to do some work in the promise object to make `co_await` and `co_yield` do appropriately useful things.

(One way to do this might be to give the promise object a pointer to the class, using the technique shown in an earlier section, and then have it *delegate* the `await_transform` and `yield_value` methods to the class itself, via the standard C++ technique of `std::forward`. That would let you use the same promise object with different classes that wanted to make their own choices about how to await and yield things. But I won't show an example of that; I think it's beyond the scope of this article.)

Full source code for this example: `co_class.cpp`.

Making a lambda function into a coroutine

As well as free functions and class methods, another kind of function that can be turned into a coroutine is a lambda function. This works just as well as any other coroutine, as long as you specify its return type explicitly:

```
int main() {
    auto lambda_coroutine = []() -> UserFacing {
        co_yield 100;
        for (int i = 1; i <= 3; i++)
            co_yield i;
        co_yield 200;
    };
    UserFacing lambda_coroutine_instance = lambda_coroutine();

    // now do something with that user-facing object
}
```

This has all the same advantages as ordinary lambda functions, in that you can put your coroutine code right next to the code that will consume its output, and capture local variables that are in scope at the time.

If you do this inside another coroutine, it opens up some fun possibilities for new types of parallel control flow structure. For example, you could make a sort of 'parallel `while`', by instantiating multiple coroutines in the

form of lambdas, and then inventing a type of `co_await` that inserts all of them into the program's main event loop and then waits until all of them have finished before continuing the containing coroutine. Or perhaps in another context you might want to wait until *one* of them finishes, and then destroy the rest. The possibilities are endless!

Full source code for this example: `co_lambda.cpp`.

Loose change: details I haven't gone into

This article is already pretty long, and I *still* haven't had time to say all the things there are to say about C++ coroutines. Here's a quick list of things I *haven't* mentioned already.

When a coroutine is created, the internal state (containing the coroutine's local variables) is dynamically allocated. In an embedded context, you might need to control *how*, or where, that memory is allocated. I understand that you can do this by providing an `operator new` and `operator delete` as methods of the promise type. Also, if the allocation fails, you can also arrange fallback handling by providing a promise method called `get_return_object_on_allocation_failure()`, supplementing the existing `get_return_object()`. But I haven't tried those out, and would have to look up the details if I ever needed to do it.

In my examples of exception handling above, we propagate an exception out of the coroutine back to the caller by having the `unhandled_exception()` promise method store it, and then the user-facing object retrieves it and rethrows it after the coroutine handle's `resume()` method returns. According to the C++ standard, it looks as if another way to achieve the same thing (and with less coding!) might be to re-throw the exception *in* the `unhandled_exception()` method. But all the example code I've found elsewhere does it my way, passing it back to the user-facing object and letting that rethrow it. I don't know why the other technique isn't used. Probably it's less flexible.

I normally think of coroutines as an alternative to threads. But, apparently, one of the reasons why C++ has this very flexible coroutine system is so that coroutines can be used *in conjunction* with threads: you might resume a coroutine on a different thread from the one it was last resumed on, so that coroutines might sometimes yield to each other within a single thread but also sometimes be running in parallel in different threads. (I understand that this is part of why the 'promise type' is so named, and why some of the standard's examples refer to the user-facing type as a 'future'.) I haven't discussed that at all here, because I'm still generally a single-threaded kind of programmer. But there's probably a whole extra level of complexity you can get involved with if you want to have 1000 coroutines representing specific tasks currently in progress, schedule them across 16 hardware threads to make best use of your machine's CPUs, transfer data between them in a way that may or may not require inter-thread synchronisation, and avoid deadlocks at every turn …

Finally, I've only really talked about the internals of the user-facing type that interact with the coroutine, and not about the API that the same type provides to the user. For all my examples of 'generator'-type coroutines (ones that yield a stream of values), I simply invented a trivial API consisting of a `next_value()` method that a client can call to return whatever value the coroutine yields next. But there's a lot more to say about writing *nice* user-facing APIs. For example, a generator coroutine could also usefully provide an iterator system, so that you can use it in a range-`for` statement, or in conjunction with the `<views>` system. (Indeed, the C++23 `std::generator` already does this.) I haven't even begun to talk about the way to do *that*, because it's not really about coroutines at all – making a class that can be used in a range-`for`, or one that behaves like a `view` or an `istream` or an `ostream` or whatever, is a completely separate aspect of C++, independent of whether the source (or sink) of the data is a coroutine. There's probably scope for another whole article on that subject.

Of course, there may be more still to say, that I don't even know about!

Conclusion

Phew, that was a lot of words! No wonder it didn't fit in that training course I went on.

I said in the introduction that one of my aims in learning about all this was to find out whether it would be good to convert my existing C++ program `spigot` so that it uses C++ language coroutines in place of my preprocessor-based strategy. Now I'm done, I think the answer is: it would certainly be good to do that one way or another, but I have several options for exactly *how* to do it, and I'll need to decide which!

(The natural way to use my preprocessor-based coroutine system in C++ is to put the coroutine macros in one method of a class, so that the class's member variables store all the coroutine state that persists between yields, and every time that particular method is called, it resumes from the last place it left off. In that respect, the closest thing to a drop-in replacement is the technique I described above for hiding a coroutine inside a class implementation. That gets you almost exactly the same code structure, without preprocessor hacks, and with the new ability to have the coroutine declare its local variables more like a normal function. But it also doesn't get you any *extra* usefulness – it's very possible that a more profound redesign of `spigot` would be a better idea.)

I also wanted to find out what else this facility might be useful for. I've got a lot of ideas about that, but I've no idea which ones will be useful yet. I look forward to seeing what the rest of the world comes up with!

Footnotes

With any luck, you should be able to read the footnotes of this article in place, by clicking on the superscript footnote number or the corresponding numbered tab on the right side of the page.

But just in case the CSS didn't do the right thing, here's the text of all the footnotes again:

1. Personally I would have called that the 'policy type', because its main role is to specify the policy for your coroutines. I understand that the C++ standard called it a 'promise' because it can also play that role in the promise/future model of asynchronous computation. But it's a policy type *always*, and a literal promise type *sometimes*, so I'd have said the name 'policy type' would be more central to its purpose. But 'promise' is what the standard called it, and that name appears in identifiers, so it's what we have to go with.

2. I don't really know what the `operator co_await` approach is for. `operator co_await` has to be a free function – that is, not a member of a class – so it can't access the data in your promise object. So it seems generally less flexible. In all my examples, I'll stick to using `await_transform()`.

3. I don't understand why you can't have both the return methods. It *is* legal to implement more than one instance of `return_value` taking different argument types, so that the coroutine can choose to return more than one different kind of thing (perhaps on different exit paths), because C++ function overloading will pick the right method for the type in each `co_return` statement. I don't see why it *shouldn't* be just as legal to have a set of more than one valid return type *including* `void`. But it isn't.

4. Beware that the boolean return values from `await_ready()` and `await_suspend()` have opposite semantics! `await_ready()` must return `true` to make the coroutine continue running, whereas `await_suspend()` must return `false`.

5. I don't understand why `await_ready()` needs to exist at all. `await_suspend()` can *also* cause the coroutine not to suspend itself, and has more information to base the decision on, since it gets a copy of the coroutine handle. I have no idea why there needs to be a separate 'ready' function.

6. You might look at this example and ask 'Why bother?', because if these coroutines were not linked by custom awaiters, each one could just as easily retrieve values from its supplier by calling its `next_value()` method instead of this fancy `co_await` system – the same way you'd do a job of this shape with Python generators, or with the C++23 ready-made `std::generator`. That is simpler, but it would stack all the coroutines' state on the physical call stack, which might become a problem if there were 1000 coroutines instead of 3 in the chain. Also, with a few more refinements, you could arrange for this system to suspend itself at any point in the chain and resume later – for example, if the coroutines in the middle *also* had to pause for some other purpose like network I/O.

7. Looking at the C++23 standard, it *looks* as if the ready-made `std::generator` also provides a stack feature analogous to Python '`yield from`'. Instead of using the `co_await` syntax as I've done here, they apparently do it by passing some kind of different type to `co_yield`, although I'm not exactly sure what – the definition in the standard is incomprehensible, and at the time of writing this, I couldn't find any compiler implementing C++23 generators to test with. In any case, their system doesn't also permit a separate return value, so I think the setup I show here is more suitable for more general uses of coroutines, e.g. in bidirectional I/O scenarios. I'm only showing generators as a simple example.

8. In this example, I've avoided using `std::make_unique`, which is usually the convenient way to dynamically allocate an object and immediately wrap its address up into a `std::unique_ptr`. Instead, I've done it the pedestrian way, by using the `new` operator and then passing the pointer to `std::unique_ptr`'s constructor. That's because `std::make_unique` requires the object's constructor to be public, and in the example code above, I made the constructor of `CoroutineDerivedClass` private, so that only its friend class `Promise` can call it. So `get_return_object()` can *directly* call that constructor, because it's a method of a friend class – but it can't authorise `std::make_unique` to call the same constructor on its behalf.

9. This isn't the only use case for making a class member function into a coroutine. Another simpler reason might be to return a stream of data generated from inside the class. For example, a class wrapping a database of some kind might have a query method that returns a `std::generator` that delivers the results of the query one row at a time, constructing them lazily on demand.

10. If only `std::coroutine_handle` had thought to define a `promise_type` type name, we wouldn't *even* have to do that! Oh well.