My tutorial and take on C++20 coroutines

David Mazières

February, 2021

Introduction

Over the last 25 years, I've written a lot of event-driven code in C++. A typical example of event-driven code is registering a callback that gets invoked every time a socket has data to be read. Once you have read an entire message, possibly after many invocations, you parse the message and invoke another callback from a higher layer of abstraction, and so forth. This kind of code is painful to write because you have to break your code up into a bunch of different functions that, because they are different functions, don't share local variables.

As an example, here's a subset of the methods on the `smtpd` class of Mail Avenger, my SMTP server written in C++03:

```
void cmd_rcpt (str cmd, str arg);
void cmd_rcpt_0 (str cmd, str arg, int, in_addr *, int);
void cmd_rcpt_2 (str addr, int err);
void cmd_rcpt_3 (str addr, str errmsg);
void cmd_rcpt_4 (str addr, str errmsg, int local);
void cmd_rcpt_5 (str addr, str errmsg, str err);
void cmd_rcpt_6 (str addr, str err);
```

Step 1, `cmd_rcpt`, seems like a reasonable function, called in response to a client issuing an SMTP "RCPT" command. Processing the RCPT command depends on certain information being cached about the client. If the information if not cached, it launches an asynchronous task to probe the client and returns. The asynchronous task, when it completes, goes "back" to step 0, `cmd_rcpt_0`, which just calls `cmd_rcpt` again, but needs to be a different function because the client-probing code expects a callback to which it can provide additional arguments. Various other things then may need to happen asynchronously, and every possible return point from an asynchronous call needs to be its own method. Pretty gross.

C++11 made the situation considerably better by introducing lambda expressions. Now you only need one `cmd_rcpt` method on the class, and can use nested lambda expressions for the remaining ones. Better yet, lambdas can capture local variables from enclosing functions. Nonetheless, you still need to break your code into many functions. It's clumsy to skip multiple steps or support situations where the order of issuing asynchronous events may change at runtime. Finally, you often end up fighting the right-hand margin of your text editor as your nested lambda expressions get further and further indented.

I was super excited to see that C++20 supports coroutines, which should hugely improve the process of writing event-driven code. Now that someone has finally published a book (or at least a draft of a book) on C++20, I eagerly got a copy a few days ago and read it. While I found the book did a reasonable job on *concepts* (the language feature) and other C++20 improvements, I sadly found the explanation of coroutines utterly incomprehensible. Same for almost every other explanation I found on the web. Hence, I had to dig through the specification and cppreference.org to figure out what was really going on.

This blog post represents my attempt to explain coroutines—basically the tutorial I wish I'd had 48 hours ago when I just wanted to figure this stuff out.

Tutorial

Roughly speaking, coroutines are functions that can invoke each other but do not share a stack, so can flexibly suspend their execution at any point to enter a different coroutine. In the true spirit of C++, C++20 coroutines are implemented as a nice little nugget buried underneath heaps of garbage that you have to wade through to access the nice part. Frankly, I was disappointed by the design, because other recent language changes were more tastefully done, but alas not coroutines. Further obfuscating coroutines is the fact that the C++ standard library doesn't actually supply the heap of garbage you need to access coroutines, so you actually have to roll your own garbage and then wade through it. Anyway, I'll try to save any further editorializing for the end of this blog post...

One other complication to be aware of is that C++ coroutines are often explained and even specified using the terms *future* and *promise*. These terms have nothing to do with the types `std::future` and `std::promise` available in the C++ `<future>` header. Specifically, `std::promise` is *not* a valid type for a coroutine promise object. Nothing in my blog post outside this paragraph has anything to do with `std::future` or `std::promise`.

With that out of the way, the nice little nugget C++20 gives us is a new operator called `co_await`. Roughly speaking, the expression "`co_await a;`" does the following:

1. Ensures all local variables in the current function—which must be a *coroutine*—are saved to a heap-allocated object.
2. Creates a callable object that, when invoked, will resume execution of the coroutine at the point immediately following evaluation of the `co_await` expression.
3. Calls (or more accurately *jumps to*) a method of `co_await`'s target object `a`, passing that method the callable object from step 2.

Note that the method in step 3, when it returns, does not return control to the coroutine. The coroutine only resumes execution if and when the callable from step 2 is invoked. If you've used a language supporting call with current continuation, or played with the Haskell `Cont` monad, the callable in step 2 is a bit like a continuation.

Compiling code using coroutines

Since C++20 is not yet fully supported by compilers, you'll need to make sure your compiler implements coroutines to play with them. I'm using GCC 10.2, which seems to support coroutines so long as you compile with the following flags:

```
g++ -fcoroutines -std=c++20
```

Clang's support is less far along. You need to install llvm libc++ and compile with:

```
clang++ -std=c++20 -stdlib=libc++ -fcoroutines-ts
```

Unfortunately, with clang you also need to include the coroutine header as `<experimental/coroutine>` rather than `<coroutine>`. Moreover, a number of types are named `std::experimental::xxx` instead of `std::xxx`. Hence, as of this writing, the examples below won't compile out-of-the box with clang, but ideally should with a future release.

If you want to play around, all the demos in this blog post are available in a single file corodemo.cc.

Coroutine handles

As previously mentioned, the new `co_await` operator ensures the current state of a function is bundled up somewhere on the heap and creates a callable object whose invocation continues execution of the current function. The callable object is of type `std::coroutine_handle<>`.

A coroutine handle behaves a lot like a C pointer. It can be easily copied, but it doesn't have a destructor to free the memory associated with coroutine state. To avoid leaking memory, you must generally destroy coroutine state by calling the `coroutine_handle::destroy` method (though in certain cases a coroutine can destroy itself on completion). Also like a C pointer, once a coroutine handle has been destroyed, coroutine handles referencing that same coroutine will point to garbage and exhibit undefined behavior when invoked. On the plus side, a coroutine handle is valid for the entire execution of a coroutine, even as control flows in and out of the coroutine many times.

Now let's look more specifically at what `co_await` does. When you evaluate the expression `co_await a`, the compiler creates a coroutine handle and passes it to the method `a.await_suspend(coroutine_handle)`.[1] The type of `a` must support certain methods, and is sometimes referred to as an "awaitable" object or an "awaiter."

Now let's look at a complete program that uses `co_await`. For now, ignore the `ReturnObject` type—it's just part of the garbage we have to get through to access `co_await`.

```
#include <concepts>
#include <coroutine>
#include <exception>
```

[1]Note that if `a` has an `operator co_await`, or if the so-called promise type (discussed later) has an `await_transform` method, it's possible that you end up executing `o.await_suspend` for some awaitable object `o` different from `a`.

```cpp
#include <iostream>

struct ReturnObject {
  struct promise_type {
    ReturnObject get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void unhandled_exception() {}
  };
};

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

ReturnObject
counter(std::coroutine_handle<> *continuation_out)
{
  Awaiter a{continuation_out};
  for (unsigned i = 0;; ++i) {
    co_await a;
    std::cout << "counter: " << i << std::endl;
  }
}

void
main1()
{
  std::coroutine_handle<> h;
  counter(&h);
  for (int i = 0; i < 3; ++i) {
    std::cout << "In main1 function\n";
    h();
  }
  h.destroy();
}
```

Output:

```
In main1 function
counter: 0
In main1 function
counter: 1
```

```
In main1 function
counter: 2
```

Here counter is a function that counts forever, incrementing and printing an unsigned integer. Even though the calculation is stupid, what's exciting about the example is that the variable `i` maintains its value even as control switches repeatedly between `counter` and the function `main1` that invoked it.

In this example, we call `counter` with a `std::coroutine_handle<>*`, which we stick into our `Awaiter` type. In its `await_suspend` method, this type stores the coroutine handle produced by `co_await` into `main1`'s coroutine handle. Each time `main1` invokes the coroutine handle, it triggers one more iteration of the loop in `counter`, which then suspends execution again at the `co_await` statement.

For simplicity, we store the coroutine handle every time `await_suspend` is called, but the handle does not change across invocations. (Recall the handle is like a pointer to the coroutine state, so while value of `i` may change in this state, the pointer itself remains the same.) We could just as easily have written:

```
void
Awaiter::await_suspend(std::coroutine_handle<> h)
{
  if (hp_) {
    *hp_ = h;
    hp_ = nullptr;
  }
}
```

You will note that there are two other methods on `Awaiter`, because these are required by the language. `await_ready` is an optimization. If it returns `true`, then `co_await` does not suspend the function. Of course, you could achieve the same effect in `await_suspend`, by resuming (or not suspending) the current coroutine, but before calling `await_suspend`, the compiler must bundle all state into the heap object referenced by the coroutine handle, which is potentially expensive. Finally, the method `await_resume` here returns `void`, but if instead it returned a value, this value would be the value of the `co_await` expression.

The `<coroutine>` header provides two pre-defined awaiters, `std::suspend_always` and `std::suspend_never`. As their names imply, `suspend_always::await_ready` always returns false, while `suspend_never::await_ready` always returns true. The other methods on these classes are empty and do nothing.

The coroutine return object

In the previous example, we ignored the return type of `counter`. However, the language restricts the allowable return types of coroutines. Specifically, the return type of a coroutine—call it `R`—must be an object type with a nested type `R::promise_type`.[2] Among

[2]Note, you can override the `promise_type` by specializing `std::coroutine_traits`.

other requirements, R::promise_type must include a method R get_return_object() that returns an instance of the outer type R. The result of get_return_object() is the return value of the coroutine function, in this case counter(). Note that in many discussions of coroutines, the return type R is referred to as a future, but for clarity I'll just call it the return object type.

Instead of passing a coroutine_handle<>* into counter, it would be nicer if we could just return the handle from counter(). We can do that if we put the coroutine handle inside the return object. Since promise_type::get_return_object computes the return object, we simply need that method to stick the coroutine handle into the return object. How can we get a coroutine handle from within get_return_object? As it happens, the coroutine state referenced by a coroutine_handle contains an instance of promise_type at a known offset, and so std::coroutine_handle allows us to compute a coroutine handle from the promise object.

Thus far, we've glossed over the template argument to coroutine handles, which are actually declared like this:

```
template<class Promise = void> struct coroutine_handle;
```

A std::coroutine_handle<T> for any type T can be implicitly converted to a std::coroutine_handle<void>. Either type can be invoked to resume the coroutine with the same effect. However, the non-void types allow you to convert back and forth between a coroutine handle and the promise_type sitting in the coroutine state. Specifically, within the promise type, we can get the coroutine handle using the static method coroutine_handle::from_pomise:

```
// from within a method of promise_type
std::coroutine_handle<promise_type>::from_promise(*this)
```

We now have everything we need to stick the coroutine handle inside the return object of our new function counter2. Here's the revised example:

```
struct ReturnObject2 {
  struct promise_type {
    ReturnObject2 get_return_object() {
      return {
        // Uses C++20 designated initializer syntax
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void unhandled_exception() {}
  };

  std::coroutine_handle<promise_type> h_;
  operator std::coroutine_handle<promise_type>() const { return h_; }
  // A coroutine_handle<promise_type> converts to coroutine_handle<>
  operator std::coroutine_handle<>() const { return h_; }
};

ReturnObject2
counter2()
{
  for (unsigned i = 0;; ++i) {
    co_await std::suspend_always{};
    std::cout << "counter2: " << i << std::endl;
  }
}

void
main2()
{
  std::coroutine_handle<> h = counter2();
  for (int i = 0; i < 3; ++i) {
    std::cout << "In main2 function\n";
    h();
  }
  h.destroy();
}
```

Output:

```
In main2 function
counter2: 0
In main2 function
counter2: 1
In main2 function
counter2: 2
```

A few things to note about the above code. First, since we no longer need our awaiter to save the coroutine handle (as we've already put the handle into the return object), we just run `co_await std::suspend_always{}`. Second, note that the return object goes out of scope and is destroyed in the fist line of `main2`. However, a `coroutine_handle` is like a C pointer, not like an object. It doesn't matter that we've destroyed the object containing `ReturnObject2::h_`, because we've copied the pointer into `h`. On the other hand, somebody needs to reclaim the space pointed to by `h`, which we do at the end of `main2` by calling `h.destroy()`. In particular, if any code calls `counter2()` and ignores the return value (or otherwise fails to destroy the handle in the `ReturnObject2` object), it create a memory leak.

The promise object

Our examples thus far are a bit unsatisfactory in that even though we can pass control back and forth between a main function and a coroutine, we have not passed any data. It would be great if our counter function, instead of writing to standard output, just returned values to `main`, which could then either print them or use them in calculations.

Since we know the coroutine state includes an instance of `promise_type`, we can add a field `value_` to this type and use that field to transmit values from the coroutine to our main function. How do we get access to the promise type? In the main function, this isn't too hard. Instead of converting our coroutine handle to a `std::coroutine_handle<>`, we can keep it as a `std::coroutine_handle<ReturnObject3::promise_type>`. The method `promise()` on this coroutine handle will return the `promise_type&` that we need.

What about within `counter`—how can a coroutine obtain its own promise object? Recall the `Awaiter` object in our first example, and how it squirreled away a copy of the coroutine handle for `main1`. We can use a similar trick to get the promise within the coroutine: `co_await` on a custom awaiter that gives us the promise object. Unlike our previous type `Awaiter`, however, we don't want this new custom awaiter to suspend the coroutine. After all, until we get our hands on the promise object, we can't stick a valid return value inside it, so wouldn't be returning anything valid from the coroutine.

Even though previously our `Awaiter::await_suspend` method returned `void`, that method is also allowed to return a `bool`. In that case, if `await_suspend` returns false, the coroutine is not suspended after all. In other words, a coroutine isn't actually suspended unless first `await_ready` returns false, then `await_suspend` (if it returns type `bool` instead of `void`) returns true.

We thus define a new awaiter type `GetPromise` that contains a field `promise_type *p_`. We have its `await_suspend` method store the address of the promise object in `p_`, but then return false to avoid actually suspending the coroutine. Until now, we have only seen `co_await` expressions of type `void`. This time, we want our `co_await` to return the address of the promise object, so we also add an `await_resume` function returning `p_`.

```
template<typename PromiseType>
struct GetPromise {
  PromiseType *p_;
  bool await_ready() { return false; } // says yes call await_suspend
  bool await_suspend(std::coroutine_handle<PromiseType> h) {
    p_ = &h.promise();
    return false;     // says no don't suspend coroutine after all
  }
  PromiseType *await_resume() { return p_; }
};
```

In addition to `void` and `bool`, `await_suspend` may also return a `coroutine_handle`, in which case the returned handle is immediately resumed. Instead of returning false, `GetPromise::await_suspend` could alternatively have returned the handle `h` to resume the coroutine immediately, but presumably this would be less efficient.

Here's our new counter code, in which the main function prints out the counter values returned by the coroutine:

```
struct ReturnObject3 {
  struct promise_type {
    unsigned value_;

    ReturnObject3 get_return_object() {
      return ReturnObject3 {
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void unhandled_exception() {}
  };

  std::coroutine_handle<promise_type> h_;
  operator std::coroutine_handle<promise_type>() const { return h_; }
};

ReturnObject3
counter3()
{
  auto pp = co_await GetPromise<ReturnObject3::promise_type>{};

  for (unsigned i = 0;; ++i) {
    pp->value_ = i;
    co_await std::suspend_always{};
  }
}

void
main3()
{
  std::coroutine_handle<ReturnObject3::promise_type> h = counter3();
  ReturnObject3::promise_type &promise = h.promise();
  for (int i = 0; i < 3; ++i) {
    std::cout << "counter3: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

Output:

```
counter3: 0
counter3: 1
counter3: 2
```

One thing to note is that our promise object transmits `i`'s value from the coroutine to the main function by copying it into `promise_type::value_`. Somewhat counterintuitively, we could also have made `value_` an `unsigned *` and returned a *pointer* to the variable `i` inside `counter3`. We can do this because the coroutine's local variables live inside the coroutine state object in the heap, so their memory remains valid across invocations of `co_await` until someone invokes `destroy()` on the coroutine handle. It would be even more convenient to stick `&i` inside the return object, but unfortunately there's no elegant way to do this given the way return objects are constructed.[3]

The `co_yield` operator

The reason it's so clunky for a coroutine to get its own promise object is that the C++ designers had one particular use case in mind and designed for the specific case instead of the general one. However, the specific case is a useful one, namely returning values from coroutines. To that end, the language contains another operator, `co_yield`.

If `p` is the promise object of the current coroutine, the expression "`co_yield e;`" is equivalent to evaluating "`co_await p.yield_value(e);`" Using `co_yeild`, we can simplify the previous example by adding a `yield_value` method to the `promise_type` inside our return object. Since `yield_value` is a method on `promise_type`, we no longer need to jump through hoops to get our hands on the promise object, it's just `this`. Here's what the new code looks like:

```
struct ReturnObject4 {
  struct promise_type {
    unsigned value_;

    ReturnObject4 get_return_object() {
      return {
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void unhandled_exception() {}
    std::suspend_always yield_value(unsigned value) {
      value_ = value;
      return {};
```

[3]Okay, you could use a global-scope `thread_local unsigned **coroutine_resultp` for `get_return_object` to store the address of an `unsigned *value_` field inside the current coroutine return object, then have the coroutine use this pointer to initialize `*coroutine_resultp = &i` before the first call to `co_await`. Pretty gross, though.

```
    }
  };

  std::coroutine_handle<promise_type> h_;
};

ReturnObject4
counter4()
{
  for (unsigned i = 0;; ++i)
    co_yield i;       // co yield i => co_await promise.yield_value(i)
}

void
main4()
{
  auto h = counter4().h_;
  auto &promise = h.promise();
  for (int i = 0; i < 3; ++i) {
    std::cout << "counter4: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

Output:

```
counter4: 0
counter4: 1
counter4: 2
```

The `co_return` operator

So far our coroutines have produced an infinite stream of integers, and our main function has simply destroyed the coroutine state after reading the first three integers. What if instead our coroutine only wants to produce a finite number of values before signaling an end-of-coroutine condition?

To signal the end of a coroutine, C++ adds a new `co_return` operator. There are three ways for a coroutine to signal that it is complete:

1. The coroutine can use "`co_return e;`" to return a final value `e`.
2. The coroutine can use "`co_return`;" with no value (or with a void expression) to end the coroutine without a final value.
3. The coroutine can let execution fall off the end of the function, which is similar to the previous case.

In case 1, the compiler inserts a call to `p.return_value(e)` on the promise object `p`. In cases 2–3, the compiler calls `p.return_void()`. To find out if a coroutine is complete, you can call `h.done()` on its coroutine handle `h`. (Do not confuse `coroutine_handle::done()` with `coroutine_handle::operator bool()`. The latter merely checks whether the coroutine handle contains a non-null pointer to coroutine memory, not whether execution is complete.)

Here is a new version of counter in which the `counter` function itself decides to produce only 3 values, while the main function just keeps printing values until the coroutine is done. There's one more change we need to make to `promise_type::final_suspend()`, but let's first look at the new code, then discuss the promise object below.

```
struct ReturnObject5 {
  struct promise_type {
    unsigned value_;

    ~promise_type() {
      std::cout << "promise_type destroyed" << std::endl;
    }
    ReturnObject5 get_return_object() {
      return {
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    void unhandled_exception() {}
    std::suspend_always yield_value(unsigned value) {
      value_ = value;
      return {};
    }
    void return_void() {}
  };

  std::coroutine_handle<promise_type> h_;
};

ReturnObject5
counter5()
{
  for (unsigned i = 0; i < 3; ++i)
    co_yield i;
  // falling off end of function or co_return; => promise.return_void();
  // (co_return value; => promise.return_value(value);)
}

void
```

```
main5()
{
  auto h = counter5().h_;
  auto &promise = h.promise();
  while (!h.done()) { // Do NOT use while(h) (which checks h non-NULL)
    std::cout << "counter5: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

Output:

```
counter5: 0
counter5: 1
counter5: 2
promise_type destroyed
```

There are a couple of things to note about `co_return`. Notice that in previous examples, we didn't have a `return_void()` method on our promise object. That's okay as long as we didn't use `co_return`. Otherwise, if you use `co_return` but don't have the appropriate `return_void` or `return_value` method, you will get a compilation error about the missing method. That's the good news. The bad news is that if you fall off the end of a function and your `promise_type` lacks a `return_void` method, you get *undefined behavior*. I'll have more to say about that in the editorial below, but suffice it to say that undefined behavior is really, really bad—like use-after-free or array-bounds-overflow bad. So be careful not to drop off the end of a coroutine whose promise object lacks a `return_void` method!

The other thing to note about `co_return` is that `promise_type::return_void()` and `promise_type::return_value(v)` both return `void`; in particular they don't return awaitable objects. This is presumably out of a desire to unify handling of return values and exceptions (which we'll discuss further down). Nonetheless, there's an important question about what to do at the end of a coroutine. Should the compiler update the coroutine state and suspend the coroutine one final time, so that even after evaluating `co_return`, code in the main function can access the promise object and make sane use of the `coroutine_handle`? Or should returning from a coroutine automatically destroy the coroutine state, like an implicit call to `coroutine_handle::destroy()`?

This question is resolved by the `final_suspend` method on the `promise_type`. The C++ spec says says that a coroutine's *function-body* is effectively wrapped in the following pseudo-code:

```
    {
        promise-type promise promise-constructor-arguments ;
        try {
            co_await promise.initial_suspend() ;
            function-body
        } catch ( ... ) {
            if (!initial-await-resume-called)
                throw ;
            promise.unhandled_exception() ;
        }
    final-suspend :
        co_await promise.final_suspend() ;
    }
    // "The coroutine state is destroyed when control flows
    //  off the end of the coroutine"
```

When a coroutine returns, you implicitly `co_await` the result of `promise.final_suspend()`. If `final_suspend` actually suspends the coroutine, then the coroutine state will be updated one last time and remain valid, and code outside of the coroutine will be responsible for freeing the coroutine object by calling the coroutine handle's `destroy()` method. If `final_suspend` does *not* suspend the coroutine, then the coroutine state will be automatically destroyed.

If you never plan to touch the coroutine state again (maybe because the coroutine just updated some global variable and/or released a semaphore before `co_return`, and that's all you care about), then there's no reason to pay for saving state one last time and worry about manually freeing the coroutine state, so you can have `final_suspend()` return `std::suspend_never`. On the other hand, if you need to access the coroutine handle or promise object after a coroutine returns, you will need `final_suspend()` to return `std::suspend_always` (or some other awaitable object that suspends the coroutine).

To make the point more concrete, here's what happens if we change `ReturnObject5::promise_type::final` to return `std::suspend_never` instead of `std::suspend_always`:

Output:

```
counter5: 0
counter5: 1
counter5: 2
promise_type destroyed
counter5: 2
Segmentation fault
```

The first `co_yield` (before the loop in `main5` even starts) yields 0. The second and third `co_yields`, which correspond to the first and second times we resume `h` in `main5`, yield 1 and 2 without issue. The third time we resume `h`, however, execution falls off the end of the coroutine, destroying the coroutine state. We see that the `promise_type` gets destroyed at this point, leaving `h` effectively a dangling pointer. Yet we call `h.done()` on this dangling pointer, provoking undefined behavior. On my machine, the undefined behavior happens to be `h.done()` returning false. That causes `main5` to stay in the loop and call `h()` once again, only this time it is resuming garbage instead of a valid coroutine state. Not surprisingly, resuming garbage doesn't update `promise.value_`, which remains 2. Also not surprisingly, since we are provoking more and more undefined behavior, our program soon crashes.

Generic generator example

Now we have almost all of the pieces to build a generic generator type, which is the most popular example of C++ coroutines you will find on the web. There are just a couple of remaining topics to cover.

First, up to this point I have been glossing over exceptions. Once a coroutine has been suspended, so you are no longer waiting for the initial call (e.g., `counter()`) to return, resuming a coroutine no longer automatically throws an exception in the main function. Instead, it calls the `unhandled_exception()` method of the promise object. Arguably, we should have been calling `std::terminate()` in that function all along for our examples. (As it is, we suppress any exceptions with an empty function, which makes throwing an exception in a coroutine equivalent to `co_return;`.)

If we want to build a generic generator return object type to help people write coroutines, the most useful approach to exceptions is arguably to re-throw them in the main routine that invokes the generator. We can do that by having `unhandled_exception()` call `std::current_exception` to obtain a `std::exception_ptr` that it stores in the promise object. When this `execption_ptr` is non-NULL, the generator uses `std::rethrow_exception` to propagate the exception in the main function.

Another important point is that up until now, our coroutines have been computing the first value (0) as soon as they are invoked, before the first `co_await`, and hence before the return object is constructed. There are two reasons you might want to defer computation of the first value until after the first coroutine suspension. First, in cases where values are expensive to compute, it may be better to save work in case the coroutine is never resumed (perhaps because of an error in a different coroutine). Second, because of the need to destroy coroutine handles manually, things can get awkward if a coroutine throws an exception before the first time it has been suspended. Take the following example:

```
void
f()
{
  std::vector<std::coroutine_handle<>> coros =
    { mkCoroutineA(), mkCoroutineB() };
  try {
    for (int i = 0; i < 3; ++i)
      for (auto &c : coros)
        if (!c.done())
          c();
  }
  catch (...) {
    for (auto &c : coros)
      c.destroy();
    throw;
  }
  for (auto &c : coros)
```

```
    c.destroy();
}
```

In the example above, suppose `mkCoroutineA()` returns a coroutine handle while `mkCoroutineB()` throws an exception before its first `co_await`. In that case, the coroutine created by `mkCoroutineA()` will never be destroyed. Of course, you could restructure the code to wrap `mkCoroutineB` in it's own try-catch block, but you can see this would quickly get unwieldy when creating many coroutines.

To address these issues, the method `promise_type::initial_suspend()` can return `std::suspend_always`, thereby suspending `mkCoroutineB` immediately on entry, before any code in the coroutine has executed (and hence before said code may throw an exception). We use this technique in our example generator below. It simply means we have to resume the coroutine once before returning the first value from our generator.

So here is our generic generator. A generator producing type `T` must return a `Generator<T>`. The main function uses `operator bool` to determine if the `Generator` still has an output value, and `operator()` to obtain the next value.

```
template<typename T>
struct Generator {
  struct promise_type;
  using handle_type = std::coroutine_handle<promise_type>;

  struct promise_type {
    T value_;
    std::exception_ptr exception_;

    Generator get_return_object() {
      return Generator(handle_type::from_promise(*this));
    }
    std::suspend_always initial_suspend() { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    void unhandled_exception() { exception_ = std::current_exception(); }
    template<std::convertible_to<T> From> // C++20 concept
    std::suspend_always yield_value(From &&from) {
      value_ = std::forward<From>(from);
      return {};
    }
    void return_void() {}
  };

  handle_type h_;

  Generator(handle_type h) : h_(h) {}
  ~Generator() { h_.destroy(); }
  explicit operator bool() {
    fill();
    return !h_.done();
  }
  T operator()() {
    fill();
    full_ = false;
    return std::move(h_.promise().value_);
  }

private:
  bool full_ = false;

  void fill() {
    if (!full_) {
      h_();
      if (h_.promise().exception_)
        std::rethrow_exception(h_.promise().exception_);
      full_ = true;
    }
  }
};

Generator<unsigned>
counter6()
{
  for (unsigned i = 0; i < 3;)
    co_yield i++;
}

void
main6()
{
  auto gen = counter6();
  while (gen)
    std::cout << "counter6: " << gen() << std::endl;
}
```

Output:

```
counter6: 0
counter6: 1
counter6: 2
```

One final point to note here is that we now destroy the `coroutine_hande` inside the destructor for `Generator`, since in our specific use case we know the coroutine handle is no longer needed once the `Generator` is gone.

Editorial

You probably already got the sense that I'm happy to see coroutines in C++, but sad that the design was so clunky. I think the `co_await` operator is reasonably well thought out, but the return object design is a complete mess. All you really need is something simple: simultaneous access to local variables in the coroutine and the coroutine handle while creating the return object. Yet the interfaces are both convoluted *and* prevent you from accessing all the necessary variables at the same time.

Obviously I've only thought about C++ coroutines for a couple of days, but it seems to me that the fundamental interface should have been two operators, `co_await` (more or less as-is) and `co_init` for allocating the coroutine handle and creating the return object. `std::coroutine_handle` should not even need to be a template, as any notion of a promise object should just be layered on top of whatever primitives the language provides. Something like:

```
template<typename T>
struct Yield {
  T *target_;
  Yield(T &t) : target_(&t) {}
  std::suspend_always operator()(const T &t) { *target_ = t; }
  std::suspend_always operator()(T &&t) { *target_ = std::move(t); }
};

template<typename T, bool Suspend = true>
struct ResumeWith {
  T value_;
  ResumeWith(const T &v) : value_(v) {}
  ResumeWith(T &&v) : value_(std::move(v)) {}
  constexpr bool await_ready() const noexcept { return !Suspend; }
  void await_suspend(std::coroutine_handle) {}
  T await_resume() { return std::move(value_); }
};

struct HypotheticalReturnObject {
  std::coroutine_handle h;
  bool done = false;
  unsigned val;

  ResumeWith<Yield<unsigned>> operator co_init(std::coroutine_handle hh) {
    h = hh;
    return ResumeWith(Yield(val));
  }
  std::suspend_always operator co_return() {
    done = true;
    return {};
  }
};

HypotheticalReturnObject &
hypothetical_counter()
{
  auto yield = co_init HypotheticalReturnObject{};

  for (unsigned i = 0; i < 3; ++i)
    co_await yield(i);
}
```

But with this design, you'd have more flexibility. For instance you could alternatively declare `unsigned i` first, and stick the address `&i` inside the return object, because everything is in scope when you are constructing the return object.

Obviously this isn't perfect, as I just started looking at C++ coroutines and am ignorant of the design history. The hypothetical design doesn't tell you what to do about exceptions. Still, I have a hard time believing it's not possible to do a lot better than the current design, and come up with something involving fewer and simpler low-level concepts that are nonetheless more expressive.

Another source of clunkiness that really baffles me is the undefined behavior when falling off the end of a coroutine without a `return_void()` method on the promise object. Undefined behavior is incredibly bad. Why would you do this to programmers? The only justification I can think of is cases where the programmer knows execution will not fall off the end of a function, but the compiler can't figure it out. In those cases, the compiler might need to generate a few bytes of dead code to handle the impossible case. But even if it's so important to optimize that fringe case, don't make undefined behavior the default! For example, why not allow the `[[noreturn]]` tag on coroutines, or allow a coroutine to end with a `[[fallthrough]];` statement, and say the behavior is undefined only when you fall off the end of a coroutine and one of these tags is present? That would satisfy the tiny minority of people who need to optimize this case without creating easy opportunities for the vast majority of programmers to shoot themselves in the foot.

Big picture, of course, clunky coroutines are still a lot better than no coroutines. I anticipate C++20 coroutines will significantly change the way I program going forward, and may prove an even bigger deal than lambda expressions.