C++ coroutines: Getting started with awaitable objects

devblogs.microsoft.com/oldnewthing/20191209-00

December 9, 2019

Raymond Chen

Coroutines were added to C++20, and Lewis Baker has a nice introduction to them.

But I'm going to write another one, taking a more practical approach: The least you need to know to accomplish various coroutine tasks.

We'll start by looking at awaitable objects: Things that can be passed to `co_await`.

When you do a `co_await x`, the compiler tries to come up with a thing called an *awaiter*.

1. (We're not ready to talk about step 1 yet.)
2. (We're not ready to talk about step 2 yet.)
3. Otherwise, `x` is its own awaiter.

Now that we have an awaiter, we can use it to wait for `x` to complete. I'll start with the basic idea, and then gradually make it more complicated.

The basic idea is that the compiler generates code like this:

	calculate `x` obtain `awaiter`
`co_await`	(We're not ready to talk about this step yet.) save state for resumption `awaiter.await_suspend(handle);` (We're not ready to talk about this step yet.) return to caller
	[Invoking the handle resumes execution here] restore state after resumption `result = awaiter.await_resume();`
	execution continues

The main job of the `await_suspend` method is to arrange *somehow* for the `handle` to be invoked when it's time for the `co_await` to be considered to have completed execution.

The main job of the `await_resume` method is to report the result of the `co_await` operation. If the `await_resume` method returns `void`, then the `co_await` also returns `void`.

You can invoke the `handle` at any time once the `await_suspend` starts. It's even possible (for example, due to race conditions) that the *somehow* caused the `handle` to be invoked even before the `await_suspend` finishes running. The entire function could even have run to completion!

Thread 1	Thread 2
calculate `x` obtain `awaiter`	

save state for resumption
`awaiter.await_suspend`
`{`
schedule `handler` to execute	

	`handler` is invoked restore state after resumption `result = awaiter.await_resume();` execution continues
does final work `}` return to caller	

One of the things that will happen when execution continues is that the awaiter destructs according to the normal rules. In particular, if the awaiter was a temporary (and it almost always is), then it destructs according to the rules for destruction of temporaries.

Observe that the `handler` was invoked before `await_ suspend` could finish running. Any attempt to use members of the temporary `awaiter` will use an object after it has been destructed.

Therefore, it is important that your awaiter not use its `this` pointer once it has arranged for the `handle` to be invoked *somehow*, because the `this` pointer may no longer be valid.

The C++ language coroutine library comes with a predefined awaiter known as `suspend_ always`. Its `await_suspend` throws away the handle without doing anything, which means that the continuation will never run. In other words, `suspend_ always` suspends and never wakes up. Like a dark version of the Snow White fairy tale.

Now, you may think that `suspend_ always` is not particularly useful, seeing as it basically hangs the coroutine. But it's a convenient starting point to build on, because it fills out all the necessary paperwork for being an awaiter. All you have to do is provide a better `await_ suspend` method.

Even with this extremely rudimentary understanding of coroutines, we can already write something interesting.

```
struct resume_new_thread : std::experimental::suspend_always
{
  void await_suspend(
      std::experimental::coroutine_handle<> handle)
  {
    std::thread([handle]{ handle(); }).detach();
  }
};
```

Since is this our first time, let's walk through the steps one at a time.

When you do a

```
co_await resume_new_thread();
```

we start by default-constructing a `resume_ new_ thread` object.

The compiler then sees that you are `co_await` ing it, so it saves the coroutine state, and then step 3 above treats the object as its own awaiter, so the compiler calls the `await _suspend` method.

Our custom awaiter suspends the coroutine by creating a thread, detaching it (so it continues to run after the thread object destructs), and returns.

The thread runs the lambda. The lambda invokes the coroutine handle, which resumes the coroutine.[1]

Upon resumption, the compiler calls the `await_ resume` method to get the result. The built-in `suspend_ always` has an `await_ resume` method that returns nothing, and since we didn't override it, our custom awaiter also returns nothing. In other words, the result of the `co_await` is `void`.

And finally, we have reached the end of the full expression, so the temporary `resume_ new_ thread` object destructs.

The result of this exercise is that if you do a

```
co_await resume_new_thread();
```

your coroutine resumes in a new thread. It's magic![2]

```
winrt::fire_and_forget StartWidget(
    std::shared_ptr<Widget> widget,
    WidgetStartOptions options)
{
    auto ticket = widget->GetStartTicket(options);
    co_await resume_new_thread();
    widget->PlugIn();
    widget->SwitchOn();
    // ticket destructor runs here
}
```

In this example, we have a coroutine that does some up-front validation by trying to obtain a start ticket. And then it moves to a new thread for actually performing the widget operations to get the thing started. At the close-brace, the ticket destructs, which releases the widget to be manipulated by others. Also at the close-brace, the function parameters are destructed. In this case, it means that the `shared_ptr` and `options` destruct.

Note that the destruction of the `ticket`, `shared_ptr`, and `options` all occur on the new thread, not on the original thread.

These simple one-shot awaitables are typically either simple objects or functions that return simple objects. In this case, it was a simple object. Next time, we'll look at the function pattern and compare the two patterns.

Bonus chatter: C++ coroutines are single-use. Once you invoke the handle, it is dead and may not be invoked again.

[1] The `std::thread` constructor accepts any *Callable*, and the `coroutine_handle<>` is itself callable. Therefore, we could have written the function a bit more tersely as

```
void await_suspend(
    std::experimental::coroutine_handle<> handle)
{
  std::thread(handle).detach();
}
```

[2] Observe that in the `resume_new_thread` example, it's possible for the new thread to start up and run to completion before our `await_suspend` finishes. This is an example of the race condition I cautioned about earlier.

Raymond Chen

Follow