C++ coroutines: Defining the co_await operator

devblogs.microsoft.com/oldnewthing/20191218-00

December 18, 2019

Raymond Chen

At the start of this series, I noted that there are three steps in obtaining an awaiter for an awaitable object. The first two were marked "We're not read to talk about this yet."

Well, now we're ready to talk about one of them.

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. ⇒ If there is a defined `operator co_await` for `x`, then invoke it to obtain the awaiter.
3. Otherwise, `x` is its own awaiter.

The search for a `operator co_await` follows the usual rules for operator overloading: See if the operator is overloaded by the object itself. If not, then look for a free definition.

The case of an awaitable object implementing its own `operator co_await` is rather unusual. After all, if you can add an `operator co_await` to a class, then you may as well just add the `await_*` methods to the class while you're at it.

The more common case is a free `operator co_await`, because that lets you add `co_await` support to a type that wasn't initially defined with coroutines in mind. For example, the C++/WinRT library defines an `operator co_await(std::chrono::duration)`: If you `co_await 30s;`, it will pause the coroutine for 30 seconds.

The `operator co_await` is a unary operator, so the member function version is defined with no parameters (in which case, `this` is the object being awaited), and the free function has one parameter (in which case, the parameter is the object being awaited). In both cases, the return value is the awaiter to use.

Just as an exercise, let's define a `co_await` operator that takes a `CoreDispatcher` and switches to that dispatcher's thread.

We already wrote an awaitable to do this last time:

```
auto ensure_dispatcher_thread(CoreDispatcher dispatcher)
{
  struct awaiter : std::experimental::suspend_always
  {
    CoreDispatcher dispatcher;

    bool await_ready() { return dispatcher.HasThreadAccess(); }

    void await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      dispatcher.RunAsync(CoreDispatcherPriority::Normal,
                          [handle]{ handle(); });
    }
  };
  return awaiter{ {}, std::move(dispatcher) };
}
```

We can add `co_await` support to `CoreDispatcher` by defining an `operator co_await`:

```
auto operator co_await(CoreDispatcher dispatcher)
{
  return ensure_dispatcher_thread(std::move(dispatcher));
}
```

Now you can `co_await` a `CoreDispatcher` directly.

```
co_await this.Dispatcher();
```

The search for an `operator co_await` finds the operator we defined above, so it is invoked to produce the awaiter. The return value is the `awaiter` inside the `ensure_dispatcher_thread` function, so that's what ends up being used to control the suspension and resumption of the coroutine.

Next time, we'll look a bit more at the consequences of the `operator co_await` search algorithm.

Raymond Chen

Follow