C++ coroutines: Constructible awaitable or function returning awaitable?

devblogs.microsoft.com/oldnewthing/20191210-00

December 10, 2019

Raymond Chen

Last time, we learned how to create simple awaitable objects by creating a structure that implements the `await_suspend` method (and relies on `suspend_always` to do the coroutine paperwork for us). We can then construct the awaitable object and then `co_await` on it.

As a reminder, here's our `resume_new_thread` structure:

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

Another option is to write a function that returns a simple awaitable object, and `co_await` on the return value.

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

What's the difference? Which is better?

Both awaitable object patterns let you put instance members on the awaitable object:

```
auto o = blah();
o.configure_something(true);
co_await o;

// fluent interface pattern
co_await blah().configure_something(true);
```

In order to have static members, the type must be publicly visible.

```
// blah can be a struct but not a function
co_await blah::fluffy();
```

Both of the patterns permit the `blah` to be parameterized:

```
co_await blah(1, false);
```

but only the function pattern permits a different awaitable object to be returned based on the parameter types. That's because the function pattern lets you create a different overloaded function for each set of parameters.

```
co_await blah(1);       // awaits whatever blah(int) returns
co_await blah(false);   // awaits whatever blah(bool) returns
```

The function version also supports marking the return value as `[[nodiscard]]`, which recommends that the compiler issue a warning if the return value is not consumed. This avoids a common mistake of writing

```
blah();
```

instead of

```
co_await blah();
```

Let's make a comparison table.

Property	struct	function
Instance members	Yes	Yes
Static members	Yes	No
Allows parameters	Yes	Yes
Different awaitable type depending on parameter types	No	Yes
Different awaitable type depending on parameter values	No	No

Warn if not `co_await`ed	No	Yes

(Note that neither gives you the ability to change the awaitable type based on the parameter *values*.)

Here’s a sketch of how each pattern would implement what it can:

```
struct blah : std::experimental::suspend_always
{
  void await_suspend(
      std::experimental::coroutine_handle<> handle);

  // instance member, fluent interface pattern
  blah& configure_something(bool value);

  // static member
  static blah fluffy();

  // parameterized
  blah();
  blah(int value);
  blah(bool value);
};

// function pattern
[[nodiscard]] auto blah()
{
  struct awaiter : std::experimental::suspend_always
  {
    void await_suspend(
        std::experimental::coroutine_handle<> handle) { ... }

    // instance member, fluent interface pattern
    awaiter& configure_something(bool value) { ... }
  };
  return awaiter{};
}

[[nodiscard]] auto blah(int value)
{
  struct awaiter : std::experimental::suspend_always
  {
    void await_suspend(
        std::experimental::coroutine_handle<> handle) { ... }

    // instance member, used only for blah(int)
    awaiter& configure_int(bool value) { ... }
  };
  return awaiter{};
}

[[nodiscard]] auto blah(bool value)
{
  struct awaiter : std::experimental::suspend_always
  {
    void await_suspend(
        std::experimental::coroutine_handle<> handle) { ... }

    // instance member, used only for blah(bool)
    awaiter& configure_bool(bool value) { ... }
  };
  return awaiter{};
}
```

The upside of the function pattern is that you can have completely different implementations depending on which overload is called. The downside is that you end up repeating yourself a lot. Though you may be able to reduce some of the extra typing by factoring into a base class in an implementation namespace.

Raymond Chen

Follow