C++ coroutines: Basic implementation of a promise type

devblogs.microsoft.com/oldnewthing/20210330-00

March 30, 2021

Raymond Chen

Last time, we diagrammed out how the pieces of a coroutine fit together. Today we'll fill in the diagram with code.

Fortunately, most of the hard work has already been done for us by the `result_holder` class we already wrote. We just need to adapt it to the format required by the coroutine specification.

```
namespace std::experimental
{
    template <typename T, typename... Args>
    struct coroutine_traits<result_holder<T>, Args...>
    {
        struct promise_type
        {
            result_holder<T> holder;

            result_holder<T> get_return_object() const noexcept
            {
                return holder;
            }

            void return_value(T const& v) const
            {
                holder.set_result(v);
            }

            void unhandled_exception() const noexcept
            {
                holder.set_exception(std::current_exception());
            }

            suspend_never initial_suspend() const noexcept
            {
                return{};
            }

            suspend_never final_suspend() const noexcept
            {
                return{};
            }
        };
    };
}
```

When the compiler encounters a function which contains a `co_await` or `co_return`, it realizes that it's dealing with a coroutine. It collects the following:

- The return type of the function.
- The type of `*this`, if it's defined as an instance member function.
- The types of the parameters, if any.

It takes all of these types and looks for a `coroutine_traits` specialization that matches it. For example, given

```
ReturnType FreeFunction(ArgType1& arg1, ArgType2 arg2);
```

the compiler looks for the type

```
std::experimental::coroutine_traits<
    ReturnType, ArgType1&, ArgType2>
```

For an instance method

```
ReturnType SomeClass::Member(ArgType1& arg1, ArgType2 arg2) const;
```

the compiler looks for

```
std::experimental::coroutine_traits<
    ReturnType, SomeClass const&, ArgType1&, ArgType2>
```

In practice, few coroutines care about anything other than the return type, so you will generally see the second and subsequent template type parameters declared but ignored.[1]

The specialized `coroutine_traits` must have a nested type called `promise_type`. This could be defined inline or it could be an alias (via `typedef` or `using`) for another type define elsewhere.

The `promise_type` is the promise object that is stored inside the coroutine state. The `get_return_object()` method is called to create the thing that is returned to the caller of the coroutine. In our case, we want to return a copy of our `result_holder`: The `result_holder` state becomes the way that the coroutine and the caller communicate with each other:

	Coroutine state				Caller
	bookkeeping				
promise	`holder`	→	`result_holder` state	←	`holder`
	stack frame				

When the coroutine reaches its `co_return`, the compiler calls `p.return_value()` with the returned value, which we pass onward to the holder by calling `set_value`. (If there is no returned value, then the compiler uses `p.return_void()`.)[2] That will in turn update the state, and the state will release any waiting coroutines, which resumes the caller.

On the other hand, if an exception escapes the coroutine, then the compiler will call `p.unhandled_exception()`, and we deal with the exception by stowing it in the `holder`. Again, this will update the state, and the state will release any waiting coroutines, which resumes the caller. And when the caller performs an `await_result` to obtain the result of the `co_await`, our `state` object rethrows the exception.

So that's the magic of how the result of the coroutine (either a return value or an exception) gets transferred from the coroutine back to the awaiter.

Wait, what are these last two method `initial_suspend` and `final_suspend`? We'll look at those next time.

[1] You could create fancy coroutines that change their implementation depending on what the parameters are. For example, you might define a marker type like

```
struct with_sugar_t {};
inline constexpr with_sugar_t with_sugar{};
```

and then have a special version of the `result_holder` coroutine promise that is used if the coroutine function is declared as

```
result_holder<int> Nicely(with_sugar_t);
```

Okay, so I don't have a really good motivation for this feature, but it does exist.

[2] It is illegal to have both `return_value` and `return_void` in a promise type, even if one of them is removed by SFINAE:

```
template<typename Dummy = void>
std::enable_if_t<std::is_same_v<T, void>, Dummy>
    return_void() const
{
    holder.set_result();
}

template<typename Dummy = void>
std::enable_if_t<!std::is_same_v<T, void>, Dummy>
    return_value(T const& value) const
{
    holder.set_result(value);
}
```

The reason is that in order for the compiler to perform substitution in order to determine which methods are callable, it needs to know what was passed to all of the `co_return` statements, so it can try substituting them into `return_value`'s parameter and see which ones succeed. But it hasn't started compiling the coroutine function body yet, so it doesn't know what to try to substitute for `value`.

This is a frustrating limitation because it prevents you from writing a single promise that covers both `void`-completing coroutines and value-completing coroutines. You always have to make two types, one for `return_void` and another for `return_value`.

Raymond Chen

Follow