C++ coroutines: The lifetime of objects involved in the coroutine function

devblogs.microsoft.com/oldnewthing/20210412-00

April 12, 2021

Raymond Chen

We finally hooked up the last missing piece of our coroutine promise implementation. Before we can look at the tradeoffs we've made, let's step back and follow the lifetime of the various objects involved in the coroutine function.

Suppose we have a coroutine that goes like this:

```
task SomeCoroutine(Parameter p)
{
    Local1 l1;
    co_await l1.Method();
    co_return l1.Result();
}
```

The coroutine state will look something like this:

	Coroutine state
	bookkeeping
promise	(you write this)
coroutine frame	`Parameter p`
	`Local l1`
	temporaries

The coroutine state consists of three parts: Compiler-defined bookkeeping to keep track of the progress of the coroutine, the promise object (which you define), and the frame (which contains the parameters, locals, and any temporaries).

We've written two kinds of promises, and they look like this:

Coroutine state		Coroutine state
bookkeeping		bookkeeping
`shared_ptr`	→ `ReturnValue`	`ReturnValue`
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

The first kind is the simple one, where the promise object is a `shared_ptr` to a `result_holder` object, and the `result_holder` holds the return value and other internal bookkeeping. In the first kind of promise, there are two allocations, one for the coroutine state, and one for the `result_holder`.

The second kind is the one where we inlined the `result_holder` object into the promise. In the second kind of promise, there is just one allocation, namely for the coroutine state. The promises consists of the `result_holder` itself, so there is no second allocation.

Let's track the memory usage and object lifetimes.

When you call the coroutine, the first thing that happens it that the compiler allocates a coroutine state and sets up some bookkeeping. In the diagrams, boxes shaded in blue contain live data, whereas boxes in light gray represent uninitialized memory. Solid lines delimit allocated memory, and dotted lines go around memory that has not yet been allocated.

Coroutine state		Coroutine state
bookkeeping		bookkeeping
`shared_ptr`	`ReturnValue`	`ReturnValue`
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

At the start of the coroutine, the compiler allocates the coroutine state, but only the compiler bookkeeping portion is live. The promise and the frame remain still uninitialized.

Next, the parameters are moved into the coroutine frame. If a parameter is a reference, then only the reference is moved. In our example, this means that `Parameter` is constructed.

Coroutine state		Coroutine state
bookkeeping		bookkeeping
`shared_ptr`	`ReturnValue`	`ReturnValue`
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

At this point, the coroutine bookkeeping and the `Parameter` in the coroutine state have been initialized.

Next, the compiler constructs the promise object. In our first example, the constructor for the promise object also constructs the `shared_ptr`, which creates the promise state object that holds space for the `ReturnValue` as well as other promise bookkeeping. In our second example, the promise state is inlined into the promise. Again, memory is reserved for the `ReturnValue`, but nothing is in it yet.

This is what we have so far:

Coroutine state			Coroutine state
bookkeeping			bookkeeping
`shared_ptr`	→	`ReturnValue`	`ReturnValue`
`Parameter p`		other stuff	other stuff
`Local l1`			`Parameter p`
temporaries			`Local l1`
			temporaries

In the first example, the `shared_ptr` for the promise object has been initialized, and it points to a newly-allocated `result_holder`. In that `result_holder`, the bookkeeping is initialized, but the return value portion remains uninitialized. The return value won't get

initialized until the coroutine exits either by reaching a `co_return` or by exiting with an exception.

In the second example, the promise object is itself just a `result_holder`; there is no `shared_ptr` and no second allocation. Again, the bookkeeping for the `result_holder` has been initialized, but the portion that holds the return value won't get initialized until there is something to return.

As the coroutine runs, local variables and temporaries are constructed and destructed according to the usual rules of the C++ language. Those local variables and temporaries appear and disappear in the frame. In our example, this means that `Local` gets constructed in the frame, and then a temporary is created to hold the return value from `l1.Method()`. We then go through the standard `co_await` machinery: From that temporary value, we obtain an awaiter, which is another temporary value, which also goes into the frame. The awaiter arranges for the coroutine to resume, and at the resumption, the temporaries are destructed.

Just before the `co_return`, we have this:

Coroutine state		**Coroutine state**
bookkeeping		bookkeeping
`shared_ptr` →	`ReturnValue`	`ReturnValue`
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

In both versions of the coroutine state, the compiler bookkeeping remains initialized, as well as the bookkeeping for the `result_holder`, the captured parameter, and the local variable. In the first example, we also have a `shared_ptr` to the `result_holder`, but in the second example, `result_holder` is embedded in the coroutine state, so no pointer or second allocation is required. In the frame, the parameter remains initialized as well as the local. The temporary is now uninitialized because the lifetime of the temporary has ended.

We now reach the `co_return`. The `co_return`'d value is Let's say that the `Result()` method returns a `ReturnValue` object by value. That rvalue reference to the `ReturnValue` is passed to the promise's `return_value` method. In our first example, the promise move-constructs the `ReturnValue` in the external promise state. In our second example, the promise move-constructs it in the embedded promise state.

Coroutine state			Coroutine state
bookkeeping			bookkeeping
`shared_ptr`	→	`ReturnValue`	`ReturnValue`
`Parameter p`		other stuff	other stuff
`Local l1`			`Parameter p`
temporaries			`Local l1`
			temporaries

At this point, the `ReturnValue` object in the `result_holder` is initialized. In the first example, the `result_holder` is external to the coroutine state; in the second example, it is embedded inside the coroutine state.

At the end of the `co_return` statement, the temporary `ReturnValue` is destructed. This `ReturnValue` does not have a lifetime that extends across a suspension point, so it doesn't have to go into the coroutine frame. It can just go on the regular stack.

The coroutine now completes, and any remaining local variables are destructed. In this case, it means that the `Local` in the coroutine frame is now destructed. The coroutine frame now consists only of the compiler bookkeeping, the promise, and the parameters. All of the locals and temporaries have been destructed. The memory for them is just sitting around unused.

Coroutine state			Coroutine state
bookkeeping			bookkeeping
`shared_ptr`	→	`ReturnValue`	`ReturnValue`
`Parameter p`		other stuff	other stuff
`Local l1`			`Parameter p`
temporaries			`Local l1`
			temporaries

The code paths diverge significally when the coroutine completes. In our first version, our `final_suspend` allows the coroutine state to be destroyed. Destroying the coroutine state destructs the promise, then the parameters, then the bookkeeping, following the usual C++ rule of destructing in reverse order of construction.

In our second version, our `final_suspend` keeps the coroutine state alive because the caller has a reference to it via the task. The objects now look like this:

Coroutine state		Coroutine state
bookkeeping		bookkeeping
`shared_ptr`	`ReturnValue`	`ReturnValue`
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

In the first version, the entire coroutine state has been destroyed and deallocated, leaving only the `result_holder` with the `ReturnValue` and a little bit of result holder bookkeeping. But in the second version, the coroutine state is still allocated, with live compiler bookkeeping, a live promise, and the live captured parameter. The live compiler bookeeping says "This coroutine has suspended at its `final_suspend` . If you resume the coroutine, it will self-destruct."

In both cases, the `result_holder` (either external or embedded) is kept alive by the `task` that we returned to our caller.

The caller's `co_await` copies (in the first case) or moves (in the second case) the result out of the `ReturnValue` .

Coroutine state		Coroutine state
bookkeeping		bookkeeping
`shared_ptr`	`ReturnValue`	(`ReturnValue`)
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

There is no change to the first version of our coroutine implementation, because copying the `ReturnValue` doesn't modify the original. But in the second version, moving the `ReturnValue` to the caller causes the `ReturnValue` inside the embedded `result_holder` to become empty. It is still initialized, but it holds nothing of interest.

And finally, the caller destructs the task, which causes the last bit to disappear. In the first case, the shared promise state destructs, which means that the result holder bookkeeping and `ReturnValue` both destruct, and then the memory for the `result_holder` is freed. In the second case, we go through coroutine state destruction, which as we noted earlier destructs the coroutine state destructs the promise, then the parameters, then the bookkeeping.

Coroutine state		Coroutine state
bookkeeping		bookkeeping
`shared_ptr`	`ReturnValue`	(`ReturnValue`)
`Parameter p`	other stuff	other stuff
`Local l1`		`Parameter p`
temporaries		`Local l1`
		temporaries

Next time, we'll compare the two designs and identify pros and cons.

Raymond Chen

Follow