C++ coroutines: Promise constructors

devblogs.microsoft.com/oldnewthing/20210504-00

May 4, 2021

Raymond Chen

So far, all of our coroutine promises have had only a default constructor. But the standard actually gives the promise access to the coroutine parameters, if it wants them.[1]

If somebody declares a coroutine that uses your promise, say,

```
simple_task<void> Awesome(int x, int y)
{
    ...
}
```

the compiler first looks for a promise constructor that accepts those parameters, prefixed if applicable by the hidden `*this` parameter. In this example, it tries to construct a `simple_promise(x, y)`. Standard overload rules apply, so the actual constructor could take two integer lvalues, or one integer lvalue and one integer by value, or maybe it takes two `long`s, since integers are implicitly convertible to `long`. This gives your coroutine an opportunity to snoop on the parameters. For example, you might have a promise that detects that one of the parameters is a `Logger`, in which case it uses that logging object for its own internal logging.

If no suitable constructor can be found, then the compiler falls back to using the default constructor for the promise.

You might say, “Well, that’s interesting, but it has no effect on me because my only constructor is the default constructor, so that’s the only one the compiler will ever use.”

You’d be wrong.

Because the compiler will autogenerate a copy constructor.

Somebody could create a coroutine like this:

```
simple_task<void> Weirdo(simple_promise<void> wha)
{
    ...
}
```

If they do that, then the compiler will look for a promise constructor that takes a `simple_promise<void>` parameter, and it will find one: The copy constructor. The promise for the coroutine will therefore be *copy-constructed* from the `wha` parameter, which is probably not what you were expecting.

On the other hand, the fact that they are passing your private promise type as a parameter suggests that they are intentionally messing with the internals and therefore deserve what they get.

However, an unwitting developer might stumble into this case if they create a generic type similar to `std::any`:

```
struct Object
{
    template<typename T>
    operator T() { return std::any_cast<T>(o); }

    template<typename T>
    Object& operator=(T&& other)
    { o = std::forward<T>(other); return *this; }

private:
    std::any o;
};
```

This is a generic type that can hold any value, and you can get the same value out by converting to the thing you hope is inside.

Which means that it can try to convert to `simple_promise`.

```
simple_task<void> Print(Object o)
{
    ...
}
```

The compiler will see that an `Object` can be passed to the `simple_promise` copy constructor, which will try to convert the `Object` to a `simple_promise` in order to copy it. The conversion will (probably) fail with a `std::bad_any_cast`, and your program crashes for a totally mysterious reason. You'll be looking at the crash dumps wondering, "Why is this code trying to convert my `Object` to a `simple_promise`?"

Let's fix that by explicitly denying copying.

```
template<typename T>
struct simple_promise_base
{
    ...

    simple_promise_base() = default;
    simple_promise_base(simple_promise_base const&) = delete;
    void operator=(simple_promise_base const&) = delete;

    ...
};
```

I'm going to declare this the nominal end of what turned into a 47-part series on coroutines,[2] because I'm pretty sure you're all sick of coroutines by now. There are still some other topics related to coroutines that aren't connected to this series, so you're not out of the woods yet. And there's generators, which is deserving of its own series, but I'll wait until the outrage dies down.

[1] Be aware that this is a dark corner of the language specification that not all implementations agree on. The specification says that the parameters are passed as lvalues, but gcc passes them as their original reference class, and MSVC doesn't pass them at all until you upgrade to version 16.8 or higher, set `/std:c++latest`, and omit the legacy `/await` flag.

I get the impression that the gcc behavior is a bug, rather than a feature, because setting `-pedantic` does not cause gcc to switch to the standard-conforming behavior.

[2] Or 48 parts if you count the prologue article about `E_ILLEGAL_DELEGATE_ASSIGNMENT`.

Raymond Chen

Follow