Creating a task completion source for a C++ coroutine: Producing a result with references

devblogs.microsoft.com/oldnewthing/20210324-00

March 24, 2021

Raymond Chen

Last time, we created a `result_holder` that can be awaited until it is assigned a result, and I noted that the code was broken.

One of the reasons that it's broken is that it doesn't handle references properly.

```
result_holder<int&> counter; // errors!
```

If `T` is a reference, we run into trouble trying to put it into a union:

```
union optional
{
    optional() {}
    ~optional() {}

    T value; // oops
} result;
```

References may not be members of a union. There's also the problem that certain C++/CX types also cannot be members of a union. So what do you do if `T` is one of those "forbidden in a union" types?

A common workaround is to wrap the illegal type inside a legal one: Create a wrapper structure that has a single member, namely the type that can't go into a union. Then put the structure in the union.

```
    struct wrapper
    {
        T value;
    };

    union optional
    {
        optional() {}
        ~optional() {}

        wrapper wrap;
    } result;
```

And now every reference to the wrapped value must go through the wrapper.

```
    ~result_holder_state()
    {
        if (ready.load(std::memory_order_relaxed)) {
            result.wrap.~wrapper();
        }
    }

    ...

    void set_result(node_list& list, T v)
    {
        if (!ready.load(std::memory_order_relaxed)) {
            new (std::addressof(result.wrap))
                wrapper{ std::forward<T>(v) };
            ready.store(true, std::memory_order_release);
            this->resume_all(list);
        }
    }
    ...
};

template<typename T>
struct result_holder
    : async_helpers::awaitable_sync_object<
        result_holder_state<T>>
{
    ...

    void set_result(T result) const noexcept
    {
        this->action_impl(&state::set_result,
            std::forward<T>(result));
    }
};
```

Note also that we use `std::forward` instead of `std::move` to construct the wrapper. Forwarding a reference preserves reference-ness, and forwarding a non-reference moves it. (I always have to go back and work out the cases by hand to convince myself that this is true.)

Okay, so that's how we can get the reference into the result holder. But how do we get it back out?

```
    T get_result()
    {
        return result.wrap.value;
    }
};
```

Now that `get_result` returns a reference, we have to make sure that the reference doesn't get decayed to a value as it propagates out of `get_result` back to the awaiter and ultimately to the caller of `co_await`:

```
    template<typename State>
    class awaitable_state
    {
        ...

        decltype(auto) await_resume(
            impl::node<extra_await_data>& node) noexcept
        {
            node.handle = nullptr;
            return parent().get_result();
        }
        ...
    };

    template<typename State>
    class awaitable_sync_object
    {
        ...

        struct awaiter
        {
            ...

            decltype(auto) await_resume()
            { return s.await_resume(node); }

            ...
        }
    };
```

The `decltype(auto)` specifier allows you to forward a return type perfectly, without incurring the decay that normally occurs if you had used `auto`.

Okay, so now we can put a reference in our `result_holder`. There's another thing we can't put in our `result_holder`: We'll look at it next time.

Raymond Chen

Follow