Creating a task completion source for a C++ coroutine: Producing nothing

devblogs.microsoft.com/oldnewthing/20210325-00

March 25, 2021

Raymond Chen

Last time, we created a `result_holder` that can hold a reference, and we solved it by using a wrapper. But there's another type that we can't put in a `result_holder`, not even with the help of a wrapper. That type is `void`.

```
struct wrapper
{
    void value; // not allowed
};
```

This doesn't work because you cannot have an object of type `void`. You might nevertheless want to have a `result_holder` that "holds" a `void`, because that is basically an event: The result is the fact that something happened.

There are a few ways to work around this problem. One is to redirect `void` to some other type like `bool`, and just ignore the `bool` value. This is the approach that is often used in C# code: A task completion event of `void` is just a task completion event of `bool` where the `bool` is ignored.

But in C++, we have partial specialization, so we can get all fancy-like.

```
template<typename T>
struct wrapper
{
    T value;
    T get_value() { return static_cast<T>(value); }
};

template<>
struct wrapper<void>
{
    void get_value() { }
};
```

In the case of `void`, we use an empty class. This avoids the trouble of having to initialize a dummy `bool` member, and it opens the door to empty base class optimization, although we won't take advantage of EBO here. We then add `get_value` methods to extract the value in a uniform way:

- For `void` it returns nothing.
- For references, it returns the reference.
- For values, it returns a copy of the object.

(Recall that this is intended for an object that can be awaited multiple times, so the underlying object needs to be copyable so that each client that does a `co_await` gets its own copy.)

Now we can revise our code that sets the result so it knows the special way of setting nothing.

```
    template<typename Dummy = void>
    std::enable_if_t<std::is_same_v<T, void>, Dummy>
        set_result(node_list& list)
    {
        if (!ready.load(std::memory_order_relaxed)) {
            new (std::addressof(result.wrap)) wrapper{ };
            ready.store(true, std::memory_order_release);
            this->resume_all(list);
        }
    }
    template<typename Dummy = void>
    std::enable_if_t<!std::is_same_v<T, void>, Dummy>
        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>>
{
    ...

    template<typename Dummy = void>
    std::enable_if_t<std::is_same_v<T, void>, Dummy>
        set_result() const noexcept
    {
        this->action_impl(&state::set_result);
    }
    template<typename Dummy = void>
    std::enable_if_t<!std::is_same_v<T, void>, Dummy>
        set_result(T result) const noexcept
    {
        this->action_impl(&state::set_result,
            std::forward<T>(result));
    }
};
```

Getting the value back out is simpler thanks to our `get_value` helper.

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

Okay, so now we know how to deal with a result of nothing. But how do you report the failure to produce a result at all? We'll look at that next time.

Bonus chatter: While we're at it, we may as well put `[[no_unique_address]]` on the `T value`, in case `T` is an empty class.

```
    template<typename T>
    struct wrapper
    {
        [[no_unique_address]] T value;
        T get_value() { return static_cast<T>(value); }
    };
```

Raymond Chen

Follow