

C++ coroutines: Building a result holder for movable types

 devblogs.microsoft.com/oldnewthing/20210406-00

April 6, 2021



Raymond Chen

One of the pieces we need for the simple promise we use to construct a coroutine is what we have been calling the “result holder”. This is basically a variant that starts out empty, and can atomically transition to holding either the result of a successful coroutine, an exception pointer for a failed coroutine.

Our variant is more complicated than a `std::variant` thanks to the atomic nature (since it is used from both the coroutine as well as the awaiter). On the other hand, we’re simpler than a `std::variant` because we do not have to deal with the `valueless__by_exception` state.

The storage of the value result is made complicated by a few things:

- If the promise produces a `void`, we can’t actually store a `void` anywhere because there is no such thing as an object of type `void`. Instead, `void` needs to be represented by the *absence* of an object.
- If the promise produces a reference, we have to be careful to preserve the reference rather than accidentally decaying it to a value.
- If the promise produces a non-reference, then we need to move the value around, in case the value is non-copyable, and more generally to avoid unnecessary copies. Even better is if we just pass the value by reference as much as possible, so that there is only a single move operation to get it to its destination.

As we saw earlier, we can handle the first two cases by wrapping the value inside a structure. And the third case just requires vigilance.

```
// [[simple_promise_result_holder definition]] :=
```

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

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

This is similar to the pattern we've seen before for wrapping something (possibly a reference, possibly a `void`) in a structure. The `get_value` method here is a little different because we want to move the object out when retrieving it.

The first thing to observe is that we cast the value to `T&&`, so everything hinges on what `T&&` is.

In the case where `T` is a reference type, `T&&` collapses back down to just `T`: Adding a `&&` to a reference has no effect. Therefore, if `T` is a reference type, the net effect is to return the `value` as whatever type of reference `T` already is.

In the case where `T` is not a reference, adding `&&` makes it an rvalue reference, so net effect is to return an rvalue reference to the held value. That rvalue reference is then ready to be moved out.

That was a lot to unpack in that one function `get_value`.²

Okay, so here comes the holder itself.

```
template<typename T>
struct simple_promise_result_holder
{
    enum class result_status
    { empty, value, error };
};
```

We define an enumeration that describes what is in the result holder. A status of `empty` means that the coroutine has started but hasn't yet completed, `value` means that it completed with a result, and `error` means that it completed with an error.

Next we have the member variables `simple_promise_result_holder`.

```

std::atomic<result_status> status
{ result_status::empty };

union result_holder
{
    result_holder() { }
    ~result_holder() { }
    simple_promise_value<T> wrapper;
    std::exception_ptr error;
} result;

```

We put the wrapper inside a union, and explicitly give the union trivial constructors and destructors to indicate that we will take responsibility for constructing and destructing the active member of the union.

Okay, so how do we put values into the holder?

```

template<typename...Args>
void set_value(Args&&... args)
{
    new (std::addressof(result.wrapper))
        simple_promise_value<T>
            { std::forward<Args>(args)... };
    status.store(result_status::value,
        std::memory_order_release);
}

```

If we are setting a value into the holder, then use placement new to construct the wrapper, forwarding the parameters into the constructor. If `T` is `void`, then there are no parameters, and we construct an empty wrapper. This seems pointless, but it preserves the invariant that if the status is `succeeded`, then the wrapper is the active member of the union, which saves us some `if constexpr` effort at destruction. In practice, constructing and destructing the empty structure has no code generation effect, so this is just bookkeeping to avoid tripping over undefined behavior.

If `T` is non-`void`, then the parameter is the `T` (or possibly an rvalue or const lvalue reference to `T`) with which to initialize the wrapper.

In both cases, if the wrapper has been successfully constructed, we mark the status as `value` to indicate that we have a value. Note that this must wait until construction is complete for two reasons.

1. If an exception occurs during construction, we don't want to say that we have a working wrapper.
2. We don't want the awaiting thread to see the value until it is fully constructed.

Updating the status with release memory order ensures that the construction of the wrapper is visible to other threads before we set the status.

```

void unhandled_exception() noexcept
{
    new (std::addressof(result.error))
        std::exception_ptr(std::current_exception());
    status.store(result_status::error,
        std::memory_order_release);
}

```

Saving the current exception is entirely analogous. The current exception is obtained from the C++ language itself and used to initialize the `error` member of our holder. Again, we update the status after the exception is stowed, and use release semantics to avoid races.

```

bool is_empty() const noexcept
{
    return status.load(std::memory_order_relaxed) ==
        result_status::empty;
}

```

The `is_empty` method says whether there's anything interesting in the holder yet.

Of course, once we put the value into the holder, we presumably want to be able to get it out again.

```

T get_value()
{
    switch (status.load(std::memory_order_acquire)) {
    case result_status::value:
        return result.wrapper.get_value();
    case result_status::error:
        std::rethrow_exception(
            std::exchange(result.error, {}));
    }
    assert(false);
    std::terminate();
}

```

We check the status to tell us whether the thing inside the holder is a value or an exception. We need to use acquire semantics here, to ensure that we don't read the `T` or `exception_ptr` before it has been completely written by the coroutine.

If the status says that the contents are a value, then we let the wrapper's `get_value` extract the result (or return nothing, if the type is `void`). Reference types come out as themselves, and non-reference types are returned as rvalue references, so that the results can be moved out.

If the status says that we have an error, then we rethrow the stowed exception. We exchange it out so we can detect at destruction if we had an unobserved exception.

If the status is neither of these, then something horrible has happened, and we terminate.

The last step in the lifetime of the holder is destruction:

```
~simple_promise_result_holder()
{
    switch (status.load(std::memory_order_relaxed)) {
        case result_status::value:
            result.wrapper.~simple_promise_value();
            break;
        case result_status::error:
            if (result.error)
                std::rethrow_exception(result.error);
            result.error.~exception_ptr();
    }
};
```

When it's time to clean up, we use the status as a discriminant to figure out what is in the holder and destruct the appropriate member. (Note that we take advantage of the injected class name to save us some `<T>` and `std::` keypresses.)

If we are cleaning an exception and the exception is still there, then that means that an exception occurred in the coroutine but was never observed, much less caught. This is not a great thing, so we rethrow the exception immediately so that the program can crash with an unhandled exception.²

That takes care of the missing pieces related to the result holder. Next time, we'll look at the `return_value` and `return_void` methods which are responsible for mediating the `co_return`.

¹ An earlier version of this code used `std::forward<T>(value)` instead of `static_cast<T&&>(value)`, in the mistaken belief that the two are equivalent. But they're not: `std::forward<T>(value)` contains extra magic in its declaration to reject lvalue-to-rvalue conversion. Mind you, at the end of the day, they are both `static_cast<T&&>(value)`, so it's a toss-up which one you use if you aren't trying to block lvalue-to-rvalue conversion.

² We are explicitly ignoring the guidance not to throw an exception in a destructor, because we really do want to crash.

Raymond Chen

Follow

