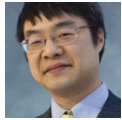


# C++ coroutines: Accepting types via `return_void` and `return_value`

 [devblogs.microsoft.com/oldnewthing/20210407-00](https://devblogs.microsoft.com/oldnewthing/20210407-00)

April 7, 2021



Raymond Chen

Last time, we built the result holder for our simple task coroutine and showed how to move values into the holder and move them back out.

So now we're actually going to move them in.

```
template<typename T>
struct simple_promise : simple_promise_base<T>
{
    // [[implement return_value]] :=
    void return_value(T&& value)
    {
        this->set_value(std::forward<T>(value));
    }

    template<typename Dummy = void>
    std::enable_if_t<!std::is_reference_v<T>, Dummy>
        return_value(T const& value)
    {
        this->set_value(value);
    }
};
```

In the case where `T` is not `void`, we have a few different cases to deal with: `T` could be an lvalue reference, an rvalue reference, or a non-reference.

Let's write out the possibilities. Let `U` the result of removing all references from `T`.

	Object	Lvalue reference	Rvalue reference
<code>T</code>	<code>U</code>	<code>U&amp;</code>	<code>U&amp;&amp;</code>
<code>T&amp;&amp;</code>	<code>U&amp;&amp;</code>	<code>U&amp;</code>	<code>U&amp;&amp;</code>
<code>T const&amp;</code>	<code>U const&amp;</code>	<code>U&amp;</code>	<code>U&amp;</code>

Reference collapsing rules say that adding `&&` to a reference has no effect. Furthermore, references are already immutable, so adding `const` also has no effect. Therefore `T const&` is the same as `T&` when `T` is a reference. The reference collapsing rules say that adding a single `&` converts an rvalue reference to an lvalue reference and leaves lvalue references unchanged.<sup>1</sup>

Okay, so let's apply the above table to our situation.

If `T` is not a reference, then we are in the `U` column, and we want both overloads. The first will move the value and the second will copy it.

If `T` is an lvalue reference, then we are in the `U&` column, so `T&&` and `T const&` are the same thing. The two overloads conflict. We want only the first overload and we should delete the second.

If `T` is an rvalue reference, then we are in the `U&&` column. We don't want to accept lvalue references, because they will create an error deep inside `set_value` which will not be obvious to interpret for those unfamiliar with our implementation. Let's just delete the second overload to convert the error message to a more understandable "cannot bind an lvalue to an rvalue reference." (This is another example of compiler error message metaprogramming.)

Putting this all together says that we want to delete the second overload if `T` is any kind of reference. So we use SFINAE to remove it.

The case where `T` is `void` is much more straightforward:

```
template<>
struct simple_promise<void> : simple_promise_base<void>
{
    // [[implement return_void]] :=
    void return_void()
    {
        this->set_value();
    }
};
```

No weird reference shenanigans. There's only one kind of void, and we want to call `set_value` with nothing.

Note that we had to put an explicit `this->` prefix on our calls to `set_value` to tell the compiler that `set_value` is a dependent name. Otherwise, two-phase lookup would interpret it as a non-dependent name and try to resolve it in the enclosing scope.

Next time, we'll look more closely at the awaiter used when the caller `co_await`s the `simple_task`.

<sup>1</sup> The way I remember the reference collapsing rules is to rephrase it as “lvalues are sticky.” Once there’s an lvalue reference anywhere in the reference chain, the final result is an lvalue.

Raymond Chen

**Follow**

