Resolving confusion over how to return from a C++ coroutine

devblogs.microsoft.com/oldnewthing/20220114-00

January 14, 2022

Raymond Chen

A customer was having trouble writing a coroutine using C++/WinRT. This function compiled successfully:

```
winrt::IAsyncOperation<bool> HelperFunction()
{
    /* no other co_return statements */

    co_return true;
}
```

But once they added a condition, it stopped compiling successfully:

```
winrt::IAsyncOperation<bool> MainFunction()
{
    ...
    if (condition) {
        ...
        co_return HelperFunction(); // Fails to compile
    }

    co_return false;
}
```

The error message is

```
error C2664: 'void std::experimental::coroutine_traits<winrt::Windows::Foundation::
IAsyncOperation<bool>>::promise_type::return_value(TResult &&) noexcept': cannot
convert argument 1 from 'winrt::Windows::Foundation::IAsyncOperation<bool>' to
'TResult &&'
with
[
    TResult=bool
]
message : Reason: cannot convert from 'winrt::Windows::Foundation::IAsyncOperation<
bool>' to 'TResult'
with
[
    TResult=bool
]
message : No user-defined-conversion operator available that can perform this
conversion, or the operator cannot be called
```

What's going on here?

The `co_return` statement takes the thing being co-returned and passes it to the promise's `return_value` method (or if you `co_return` nothing, calls the promise's `return_void` method with no parameters). Although the language imposes no semantics upon this action, the intent is that this is how you produce the asynchronous result of the coroutine: The asynchronous result of the coroutine is the thing that the caller gets when they `co_await` the coroutine.

Declaration	`IAsyncAction f()`	`IAsyncOperation<T> f()`	`fire_and_forget f()`
Return type	`IAsyncAction`	`IAsyncOperation<T>`	`fire_and_forget`
Using `return T`	`return IAsyncAction(...);`	`return IAsyncOperation<T>(...);`	`return {};`
Result type	`void`	`T`	`void`
Using `co_return T`	`co_return;`	`co_return T(...);`	`co_return;`

If you use the `return` keyword, then you must return the coroutine type. This follows the rules of the C++ language that you're familiar with: If your function says that it returns something, then the thing you `return` needs to be that something (or something convertible to it).

What's new for coroutines is the `co_return` keyword. If you use the `co_return` keyword, then the thing you `co_return` needs to be the coroutine *result* (or something convertible to it).

You have to pick a side: Either `return` everywhere in your function or `co_return` everywhere in your function. You can't mix-and-match. That would result in `MainFunction()` being part-coroutine and part not-coroutine, which the language doesn't support. You're either a coroutine or you're not.

Writing `co_return HelperFunction();` is trying to return an `IAsyncOperation<bool>` as the result of the coroutine. But the coroutine result isn't a `IAsyncOperation<bool>`. It's just a `bool`.

And that's what the compiler error message is trying to say, with compiler-colored glasses: "Cannot convert `IAsyncOperation<bool>` to `bool`." You `co_return`ed an `IAsyncOperation<bool>`, but the only thing that the `IAsyncOperation<bool>` knows how to `co_return` is a `bool`, and the compiler is unable to perform the conversion.

What you need to do is `co_return` a `bool` somehow.

The customer discovered on their own that adding a `co_await` fixed the problem:

```
winrt::IAsyncOperation<bool> MainFunction()
{
    ...
    if (condition) {
        co_return co_await HelperFunction(); // added co_await
    }

    co_return false;
}
```

But the customer was unsure of themselves. "Why is `co_await` needed? Are there any unintended consequences?"

The `co_await` keyword instructs the compiler to generate code to suspend the current coroutine `MainFunction()` and resume execution when `HelperFunction()` produces a result. Since `HelperFunction()` is itself a `IAsyncOperation<bool>`, that result will also be a `bool`. You can then `co_return` that `bool`, which makes it the result of the `MainFunction()` coroutine.

Bonus chatter: The customer also found, in their experimentation, that this version also compiled successfully:

```
winrt::IAsyncOperation<bool> MainFunction()
{
    if (condition) co_return true;
    return false;
}
```

How does this work? It seems to be breaking the rules above, because we are using `return` with the result type, and we're mixing `return` and `co_return` within the same function body.

Yes, this code should not compile.

What you're seeing is a backward compatibility behavior of the Visual C++ compiler: When coroutines were being developed, the original idea was to overload the `return`. If you `return`ed something that matched the declared return type, then it was treated as producing the return value of the function. But if you `return`ed something that matched the result type, then the function transformed into a coroutine, and you were producing the *result* of the coroutine.

My guess is that this syntax was chosen to align with the C# and JavaScript languages, both of which overload the `return` statement in this way.

Ultimately, however, the ambiguity was too much,[1] and the coroutine specification that was ratified created new keywords to make explicit whether the function body was a classic function or a coroutine. The Visual C++ compiler retains the old syntax for backward compatibility with existing code that was written to the pre-ratified standard.

It appears that an artifact of this backward compatibility is that the compiler accepts the reverse error:

```
winrt::IAsyncOperation<bool> MainFunction()
{
    co_return HelperFunction();
}
```

This uses `co_return` with the return type instead of the result type. Somehow, the compiler accepts it even though it's not required by backward compatibility. (My guess is that there's some compatibility code that merges `return` and `co_return`, and while that takes care of the compatibility issue, it also makes the compiler accept other things inadvertently.

It also seems that the `/permissive-` flag doesn't turn off this compatibility behavior.

[1] Consider a class that is designed to be the return type of a coroutine.

```
template<typename T>
class task
{
    /* stuff required to be a coroutine return type */
};

task<int> calculate()
{
    /* do some calculations */
    co_return value;
}
```

This hypothetical `task` type supports being used as the return type of a coroutine, and our sketch of a `calculate()` function calculates a value and `co_return` s it.

But suppose we added a new constructor:

```
template<typename T>
class task
{
public:
    /* create a task that has already completed with a value */
    task(T const& resolved);

    /* existing stuff required to be a coroutine return type */
};
```

This new constructor provides a way to create an already-completed task by passing the result directly to the constructor.

Given this new constructor, the following code would become ambiguous under the pre-standardized version that used `return` for both normal return and coroutine return:

```
task<int> calculate()
{
    /* do some calculations */
    return value;
}
```

Is this a plain non-coroutine function that returns a task with the `resolved` constructor? Or is this a coroutine function that produces a task from the coroutine promise via `return_value()` ? Both interpretations would be valid here.

Changing the keyword to `co_return` for coroutines removes this ambiguity.

Raymond Chen

Follow