Why did I lose the ability to co_await a std::future and concurrency::task?

devblogs.microsoft.com/oldnewthing/20200916-00

September 16, 2020

Raymond Chen

After upgrading to version 2.0.200729.8 of C++/WinRT, some customers reported that they lost the ability to `co_await` a `std::future` or a `Concurrency::task`. What happened?

The relevant change is PR 702 which removed "vestigial support for free awaiters." And that's the part that's relevant here.

"Free awaiters" sounds like a rock album from the 60's, but it's an informal term for a feature that was part of the early explorations of the coroutine TS, before there was even a coroutine TS.

Recall that the algorithm for finding an awaiter for an object x has three steps:

1. (We're not ready to talk about step 1 yet.)[1]
2. If there is a defined `operator co_await` for `x`, then invoke it to obtain the awaiter.
3. Otherwise, `x` is its own awaiter.

In the early coroutine explorations, the concept of an awaiter hadn't yet been invented. Instead, awaiting was described in the form of function calls.

- Call the function `await_ready(x)` to decide whether to suspend the coroutine.
- Call the function `await_suspend(x, handle)` to suspend the coroutine.
- Call the function `await_resume(x)` after the coroutine resumes.

Instead of the functions existing as methods on an awaiter object, they were looked up as free functions via argument-dependent lookup.

It was during this point in the exploration of how coroutines could be implemented that the header files for `std::future` and `Concurrency::task` were frozen for inclusion in Visual Studio 2017. Those classes use the prototype free function pattern for creating awaitable objects.

The C++/WinRT library developed at the same time the coroutine TS was coming together, and it supported the free function pattern as well as the awaiter-based mechanism that made it into the standard. The Microsoft Visual C++ compiler also supports the prototype free function pattern in addition to the standards-based awaiter pattern.

As part of C++/WinRT's continuing move toward standard-conformance,[2] PR 702 removed support for the nonstandard free function pattern for awaitable objects.[3] This means that if you are still using Visual Studio 2017, you lost the ability to `co_await` `std::future` and `Concurrency::task` from a C++/WinRT coroutine.

But all is not lost. If you aren't yet ready to upgrade to Visual Studio 2019 (version 16.8 or higher), you can still recover support for `co_await` 'ing `std::future` and `Concurrency::task` with the help of a modernizer. First, here's the idea kernel:

```
namespace modernizer
{
    template<typename Awaitable>
    struct cpp20_await_adapter
    {
        Awaitable& awaitable;

        bool await_ready()
        { return await_ready(awaitable); }

        template<typename Handle>
        auto await_suspend(Handle handle)
        { return await_suspend(awaitable, handle); }

        auto await_resume()
        { return await_resume(awaitable); }
    };
}
```

The modernizer takes the awaitable object and forward all of the await methods to the corresponding free function, as determined by argument-dependent lookup.

Unfortunately, this doesn't work because of the name conflict: When the `await_ready` method wants to use argument-dependent lookup to find the free `await_ready` function, the thing that is found is... itself! That's because the unqualified call to `await_ready` finds the member function of the same name before trying to look for free functions. The compiler thinks it's a recursive call, and it complains that the parameter list is incorrect.

To fix this, we need to create wrappers with different names so the name search doesn't find ourselves.

```
namespace modernizer
{
    template<typename Awaitable>
    inline auto adl_await_ready(Awaitable& awaitable)
    { return await_ready(awaitable); }

    template<typename Awaitable, typename Handle>
    inline auto adl_await_suspend(Awaitable& awaitable, Handle handle)
    { return await_suspend(awaitable, handle); }

    template<typename Awaitable>
    inline auto adl_await_resume(Awaitable& awaitable)
    { return await_resume(awaitable); }

    template<typename Awaitable>
    struct cpp20_await_adapter
    {
        Awaitable& awaitable;

        bool await_ready()
        { return adl_await_ready(awaitable); }

        template<typename Handle>
        auto await_suspend(Handle handle)
        { return adl_await_suspend(awaitable, handle); }

        auto await_resume()
        { return adl_await_resume(awaitable); }
    };

    template<typename Awaitable>
    auto make_cpp20_await_adapter(Awaitable& awaitable)
    {
        return cpp20_await_adapter<Awaitable>{ awaitable };
    }
}
```

The helper functions with the `adl_` prefix forward to the corresponding unprefixed function via argument-dependent lookup. We can be a little sloppy and use an lvalue reference for the awaitable instead of a forwarding reference because our caller always passes an lvalue.

We then revise our await adapter to use the forwarder functions. Basically, we're renaming the function, and it's the renamed functions we use in our adapter.

We also create a `make_` function to simplify our usage later, just in case we are running in C++11 mode, in which case class template argument deduction is not available.

Okay, now that we have the adapter, we can deploy it to bring the old Visual Studio header files into the C++20 coroutine world.

```
#if defined(_MSC_VER) && (_MSC_VER < 1920) && defined(_RESUMABLE_FUNCTIONS_SUPPORTED)
namespace std
{
    template<typename T>
    auto operator co_await(future<T>& x)
    { return ::modernizer::make_cpp20_await_adapter(x); }

    template<typename T>
    auto operator co_await(future<T>&& x)
    { return ::modernizer::make_cpp20_await_adapter(x); }
}

namespace Concurrency
{
    template<typename T>
    auto operator co_await(task<T>& x)
    { return ::modernizer::make_cpp20_await_adapter(x); }

    template<typename T>
    auto operator co_await(task<T>&& x)
    { return ::modernizer::make_cpp20_await_adapter(x); }
}
#endif
```

Bonus chatter: We could try to generalize the wrapper functions to accommodate future additions to the coroutine specification, such as when the `await_suspend` method was extended to allow returning a `bool` to control whether suspension should be bypassed.

```
namespace modernizer
{
    template<typename ...Args>
    inline auto adl_await_ready(Args&&... args)
    { return await_ready(std::forward<Args>(args)...); }

    template<typename ...Args>
    inline auto adl_await_suspend(Args&&... args)
    { return await_suspend(std::forward<Args>(args)...); }

    template<typename ...Args>
    inline auto adl_await_resume(Args&&... args)
    { return await_resume(std::forward<Args>(args)...); }
    template<typename Awaitable>

    struct cpp20_await_adapter
    {
        Awaitable& awaitable;

        template<typename ...Args>
        auto await_ready(Args&&... args)
        { return await_ready(awaitable, std::forward<Args>(args)...); }

        template<typename ...Args>
        auto await_suspend(Args&&... args)
        { return await_suspend(awaitable, std::forward<Args>(args)...); }

        template<typename ...Args>
        auto await_resume(Args&&... args)
        { return await_resume(awaitable, std::forward<Args>(args)...); }
    };
}
```

Note that this version has the wrinkle that if you forgot to create the `adl_` -prefixed wrappers, you sent the compiler into infinite recursion as it expanded `await_ready` to itself with an extra parameter, which then expanded to itself with two extra parameters, and so on until you hit a compiler internal limit or the compiler crashed.

This extra wrinkle is future-proof but also unnecessary, because the point of this adapter is not to anticipate the future but to accommodate the past. Therefore, it need only support what was allowed in the past.

[1] We'll get to step 1 next spring.

[2] Visual Studio has been making steps in that direction as well, and they "recommend that existing coroutine users move to standard coroutines as soon as possible" because new coroutine features will not be backported to legacy mode. We're all moving towards the same goal.[4]

[3] That PR also did some compiler error message metaprogramming so that C++/WinRT gives a more comprehensible error when you try to `co_await` something that isn't `co_await`-able.

[4] Looks like somebody added a business definition for *North Star* to wiktionary in August 2020 but didn't include any citations. I hope nobody adds me as a citation, because that would just make a circular reference.

Raymond Chen

Follow