Investigating an argument-dependent lookup issue and working around it

devblogs.microsoft.com/oldnewthing/20250214-00

February 14, 2025

C++/WinRT pull request 1225 fixed a problem with a call to `invoke`. What's the problem, why did it show up all of a sudden, and what can you do if you are stuck on an older version of C++/WinRT?

The problem is at the point in `winrt::impl::promise_base::set_completed` makes an unqualified call to `invoke()`:

```
namespace winrt::impl
{
    ⟦ ... ⟧

    template <typename Delegate, typename... Arg>
    bool invoke(Delegate const& delegate, Arg const&... args) noexcept;

    ⟦ ... ⟧

    template <typename Derived, typename AsyncInterface, typename TProgress = void>
    struct promise_base : implements<Derived, AsyncInterface,
Windows::Foundation::IAsyncInfo>
    {
        ⟦ ... ⟧

        void set_completed() noexcept
        {
            async_completed_handler_t<AsyncInterface> handler;
            AsyncStatus status;

            ⟦ ... ⟧

            if (handler)
            {
                invoke(handler, *this, status);
            }
        }
    }
}
```

The call to `invoke` intends to resolve to `winrt::impl::invoke` but instead, it resolves to `std::invoke`. Why?

Argument-dependent lookup.

If you make a function call through an unqualified name, the compiler starts by looking for a matching class member, a function declaration in block scope, or a non-function. In our case, there is no `invoke` that matches any of those criteria, so we keep going. (The `winrt::impl::invoke` is in namespace scope, not block scope.)

Next, argument-dependent lookup identifies a bunch of namespaces that should be part of the search. For each argument, the namespace containing the type of that argument is added, the class the type is contained in (if it is a nested type), as well as the namespaces of that argument type's base classes, and (if the type is a template type) the namespaces of the template's arguments. There are other rules, but they don't apply here. Some of these rules apply recursively, so the process of gathering all of the associated namespaces can pull in things that are quite distantly removed from where you started.

In our case, argument-dependent lookup finds `std::invoke` because (deep breath)

- The `*this` parameter is a `winrt::impl::promise_base<Args1...>`.
- The first `Args1...` is `std::coroutine_traits::<Args2...>`.
- The namespace for `std::coroutine_traits` is `std`, we look in the `std` namespace and find `std::invoke`.

Okay, so `std::invoke` was found by chasing through a template parameter involved in the type of one of the function parameters.

But how did this ever work, then? Was it always broken?

It used to work because in C++17, coroutines were an experimental feature, so the first type in `Args1...` was `std::experimental::coroutine_traits`. So the search for an `invoke` looked in the `std::experimental` namespace and didn't find one. Promoting coroutines out of the experimental namespace is what broke us, because that added all of the `std` namespace to the argument-dependent lookup search.

Okay, so suppose you have a component that is compiled as C++20 but which is using an older version of C++/WinRT that doesn't have the fix. The obvious solution is to upgrade to a version of C++/WinRT that has the fix, but you may not have that option available to you, say, because you are in the project endgame, and changing a fundamental dependency is a highly disruptive change that comes with a lot of risk. Can you apply a spot fix to pick up just this one fix to C++/WinRT, instead of upgrading all of C++/WinRT and risking a breaking change or regression hiding somewhere in the changelog?

My solution was to beat argument-dependent lookup at its own game. Put an even better `invoke` in the searched-in namespaces.

```
// Must include this before including any C++/WinRT header file

#include <cstdint> // int32_t

namespace winrt::Windows::Foundation
{
    enum class AsyncStatus : int32_t;
    template<typename Handler, typename Sender>
    auto invoke(Handler&& handler, Sender&& sender,
                winrt::Windows::Foundation::AsyncStatus const& status);
}

#include <wil/cppwinrt.h> // if you use wil

#include <winrt/Windows.Foundation.h>

namespace winrt::Windows::Foundation
{
    template <typename Handler, typename Sender>
    auto invoke(Handler && handler, Sender&& sender,
                winrt::Windows::Foundation::AsyncStatus const& status)
    {
        return winrt::impl::invoke(std::forward<Handler>(handler),
                std::forward<Sender>(sender), status);
    }
    // If this assertion fires, then congratulations, you have upgraded to
    // a version of C++/WinRT doesn't need this workaround.
    static_assert(::wil::details::minor_version_from_string(CPPWINRT_VERSION) <
221117);
}
```

First, we declare a version of `invoke` in the `winrt::Windows::Foundation` namespace that will be considered better than `std::invoke` because the final `status` parameter is non-deduced. We have to do this before including `winrt/Windows.Foundation.h` so that its declaration is visible to the compiler at the time the `set_completed()` method is encountered.

If you use WIL, then the next thing to include is `wil/cppwinrt.h`, which must be included before any C++/WinRT header, so that WIL can hook up its C++/WinRT interop.

Then we include `winrt/Windows.Foundation.h`, which consumes our declaration and also provides a definition for `winrt::impl::invoke`.

And then we implement our `invoke` by forwarding everything to `winrt::impl::invoke`.

As a final step, we assert that the C++/WinRT version is one that predates the bug fix. That way, the workaround doesn't hang around forever, long after it has served its purpose.

Bonus chatter: Technically, the `winrt::impl` and `wil::details` namespaces are for internal use, and we aren't supposed to rely on them because they can change at any time. But in this case, the fix is specifically to address an issue in a specific version of the header, so the namespace is stable within the context of this fix, because once you upgrade the header (potentially invalidating our fix), the `static_assert` will fire and tell you to delete the code!