Behind C++/WinRT: How does C++/WinRT decide which interfaces are implemented?

devblogs.microsoft.com/oldnewthing/20220324-00

March 24, 2022

Raymond Chen

Last time, we diagnosed a problem by realizing that the `unkwn.h` header had not been included *prior* to including any C++/WinRT headers, and that means that C++/WinRT did not activate its code that supports classic COM interfaces.

This is going to be the first in what will probably be a very sporadic series of looking into the C++/WinRT implementation, as reverse-engineered by me.[1] I'm writing it in part so that I'll be able to refer back to this write-up the next time I have to debug this code. And in part so that there are more people who understand the insides of C++/WinRT and can help support it. (This is the selfish reason for many of the articles I write: I'm writing them in order to reduce my own workload.)

It all hangs on this definition of `is_interface`:

```
#ifdef WINRT_IMPL_IUNKNOWN_DEFINED

template <typename T>
struct is_interface : std::disjunction<
    std::is_base_of<Windows::Foundation::IInspectable, T>,
    std::conjunction<
        std::is_base_of<::IUnknown, T>,
        std::negation<is_implements<T>>>> {};

#else

template <typename T>
struct is_interface :
    std::is_base_of<Windows::Foundation::IInspectable, T> {};

#endif
```

The `WINRT_IMPL_IUNKNOWN_DEFINED` macro is an internal C++/WinRT macro that remembers whether `unknwn.h` has been included. If so, then `::IUnknown` is defined, and C++/WinRT can activate classic COM support. Let's translate the C++ type traits template meta-programming into something we're more familiar with.

One of the main tools of the C++ type traits system is the `std::integral_constant<T, v>`. This is a type that wraps a constant value `v` of type `T`.

```
template<typename T, T v>
struct integral_constant
{
    static constexpr T value = v;
    ... other stuff not relevant here ...
};
```

For example, `std::integral_constant<int, 42>::value` is an integer constant whose value is 42.

This seems pointless, but it's not. Template meta-programming doesn't have variables; it operates on types. The `std::integral_constant` lets you treat a type as if it were a variable whose value is the `integral_constant::value`.

C++ comes with a number of pre-made integral constants. Relevant today are `std::true_type` and `std::false_type`, which wrap a Boolean `true` or `false`, respectively. And it also comes with some pre-made template types that manipulate them:

- `std::conjunction` performs a logical `and` on its arguments.[2]
- `std::disjunction` performs a logical `or` on its arguments.
- `std::negation` performs a logical `not` on its argument.

Okay, now we can start taking apart the first expression.

```
template <typename T>
struct is_interface : std::disjunction<
    std::is_base_of<Windows::Foundation::IInspectable, T>,
    std::conjunction<
        std::is_base_of<::IUnknown, T>,
        std::negation<is_implements<T>>>> {};
```

We mentally convert the `std::disjunction` to `||`, the `std::conjunction` to `&&`, and the `std::negation` to `!`.

```
template <typename T>
struct is_interface is true if
    std::is_base_of<Windows::Foundation::IInspectable, T> ||
    (
        std::is_base_of<::IUnknown, T> &&
        !is_implements<T>);
```

Now we can read out the logic. Something is considered an interface if either

- It derives from `winrt::Windows::Foundation::IInspectable`, or
- It derives from `::IUnknown` and is not an `implements`.

The rejection of `implements` prevents `is_interface` from misdetecting `implements` as a itself being COM interface.

Onward to the `#else`: If `unknwn.h` was not included, then we use a simpler definition of `is_interface` that merely detects derivation from `winrt::Windows::Foundation::IInspectable`.

In order to detect classic COM interfaces, C++/WinRT needs `::IUnknown` to have been defined. Otherwise, it has nothing to test as a base class.

So that's the quick diagnosis of yesterday's problem wherein C++/WinRT failed to recognize classic COM interfaces.

Next time, we'll dig in deeper to how the `is_interface` definition is used to pick out the interfaces.

Bonus chatter: As I noted last time, the requirement that you include `unknwn.h` before including C++/WinRT is no longer present as of C++/WinRT version 2.0.210922.5. The trick is to forward-declare the `::IUnknown` type so that you can talk about it without knowing what it is. The `std::is_base_class` template type requires only that the proposed derived class be complete. The base class (`::IUnknown`) doesn't have to be complete.

Exercise: Why is it okay for `std::is_base_class` to accept an incomplete base class? How can it possibly detect whether something derives from a class which has no definition?

[1] A lot of learning comes from reverse-engineering. When doing debugging, you are pretty much forced into it.

[2] Actually, `std::conjunction` and `std::disjunction` behave more like their JavaScript equivalent operators `&&` and `||` because they short-circuit and support "truthiness".

Answer to exercise: If the base class is incomplete, then nothing can derive from it. You can't derive from an incomplete type.

Raymond Chen

Follow