How do I convert from the C++/WinRT projection type to the C++/WinRT implementation type?

devblogs.microsoft.com/oldnewthing/20200828-00

August 28, 2020

Raymond Chen

Last time, we looked at converting from the C++/WinRT implementation type to the corresponding C++/WinRT projection type. Going from the projection back to the implementation is a little trickier and relies on your own vigilance. There's no a priori guarantee that the projected pointer actually refers to your implementation. For example, if all you have is an `IInspectable`, that could be anybody. Even if you have a concrete `Sample::Class`, the projected type could have been implemented by somebody else.[1]

But assume that you have other ways of knowing that the `IInspectable` or other projected type really is backed by the implementation you claim. Typically, it's because you put it there yourself originally.

```
// Go from the projection to the implementation.
implementation::Class*p = get_self<implementation::Class>(o);
```

The `get_self` function assumes that the thing you're providing is backed by the implementation you specify, and it returns a pointer to the that implementation type. The lifetime of the pointer is controlled by the projected object you obtained the pointer from, so you now have to keep both `o` and `p` around, which can be a hassle.

There's a nasty gotcha with `get_self` beyond the fact that you had better have the right implementation: You also have to have the right interface!

If the object you pass is an interface, then the implementation cannot implement that interface multiple times, or there will be an ambiguity over how to convert from that interface to the implementation. This is a problem in standard C++ as well:

```
struct B {};
struct D1 : B {};
struct D2 : B {};
struct C : D1, D2 {};

C* c;
B* b = c; // error: 'B' is an ambiguous base of 'C'
```

The C++/WinRT `get_self` function is more restrictive than C++ casts, because it's a conversion, not a cast. The thing you're converting from must be listed as one of the implemented interfaces.

```
struct C : implements<C, ISomething>
{
};

IInspectable something;
C* p = get_self<C>(something);
// error: 'static_cast': cannot convert from
// winrt::impl::producer<D, I, void> *' to 'D *'
// with D = C, I = IInspectable
```

You can't convert from `IInspectable` since the class `C` doesn't list `IInspectable` as one of its explicitly-implemented interfaces. (It is an implicitly-implemented interface because `IInspectable` is the base of all Windows Runtime interfaces.)

You can avoid this problem by converting the `IInspectable` to an explicitly-implemented interface first.

```
C* p = get_self<C>(something.as<ISomething>());
```

To avoid having to remember what interfaces your object implements, you can use the `default_interface` helper template type.

```
C* p = get_self<C>(something.as<winrt::default_interface<C>>());
```

But wait, you're not out of the woods yet.

If you mistakenly implement `IInspectable` as well as a Windows Runtime interface, then you run into another ambiguity: If recovering the object from an `IInspectable`, is the `IInspectable` the explicitly-implemented `IInspectable` or the implicitly-implemented `IInspectable` that came along for the ride as a base class of the Windows Runtime interface?

```
struct C : implements<C, ISomething, IInspectable>
{
};

IInspectable something;
C* p = get_self<C>(something); // 50% chance of working
```

C++/WinRT assumes that you gave it the `IInspectable` that came from the explicitly-implemented interface, which has a 50% chance of being correct. If it's incorrect, you will get the wrong pointer back and corrupt memory and be very sad.

The best way to fix is to “stop holding it wrong”. Remove the redundant `IInspectable` from your list of explicitly-implemented interfaces.

A less-good (but still effective) fix is to use the default interface trick we saw above.

```
C* p = get_self<C>(something.as<winrt::default_interface<C>>());
```

Note that the value returned by `get_self` is a raw pointer. The lifetime of the object is still controlled by whatever you passed to `get_self`. This can get annoying, since you now have to carry two things around: You need to carry the raw pointer so you can access your implementation, and you need to carry the original object that manages the lifetime. Here’s a helper function which converts the projected type into a reference-counted `com_ptr`. That way, you don’t have to carry two objects around. While I’m at it, I’ll fix the nasty gotcha (though really, you should just fix it by removing the redundant `IInspectable`).

```
template<typename D, typename T>
winrt::com_ptr<D> as_self(T&& o)
{
    winrt::com_ptr<D> result;
    if constexpr (std::is_same_v<std::remove_reference_t<T>,
                                 winrt::Windows::Foundation::IInspectable>)
    {
        auto temp = o.as<winrt::default_interface<D>>();
        result.attach(winrt::get_self<D>(temp));
        winrt::detach_abi(temp);
    }
    else if constexpr (std::is_rvalue_reference_v<T&&>)
    {
        result.attach(winrt::get_self<D>(o));
        winrt::detach_abi(o);
    }
    else
    {
        result.copy_from(winrt::get_self<D>(o));
    }

    return result;
}
```

The basic idea is to use `get_self` to obtain the raw pointer and use that pointer to initialize the resulting `com_ptr`. Depending on the circumstances, we might steal the reference count associated with the pointer, or we might just copy it.

If the inbound parameter is an `IInspectable` (either lvalue or rvalue reference), then we are in the gotcha case, and we will make an explicit conversion to the default interface before calling `get_self`. We can use attach/detach semantics because the temporary default interface is going out of scope soon, so we can steal its reference.

Otherwise, the inbound parameter is something other than `IInspectable`, so we are not in the gotcha case. For rvalue references, we can use use attach/detach semantics because the rvalue reference allows us to steal its reference. For lvalue references, we use copy semantics because the original retains its reference.

[1] It is legal for a projected type to have multiple implementations. This actually happens on occasion. For example there are different implementations of `PointerPoint` depending on which kind of device the point came from. The `PointerPoint` itself is a smart pointer to an unknown implementation.

If you aren't sure whether the implementation is yours, you can create a private marker interface to identify your own objects.

Raymond Chen

Follow