C++/WinRT implementation inheritance: Notes on winrt::implements, part 4

devblogs.microsoft.com/oldnewthing/20250224-00

February 24, 2025

Last time, we figured out the rules for inheriting `winrt::implements` in C++/WinRT runtime class implementations: You can use `winrt::implements` in your class hierarchy, as long as you use single inheritance for the `winrt::implements` part. You are not allowed to derive multiply from two different `winrt::implements` base classes.

One case of this is a base class that is configured at construction.

```
struct StringableInt32 :
    winrt::implements<StringableInt32,
        winrt::Windows::Foundation::IStringable>
{
    StringableInt32(int value) : m_value(value) {}

    winrt::hstring ToString()
    { return winrt::to_hstring(m_value); }

private:
    int m_value;
};
```

We can use this by itself:

```
auto o = winrt::make<StringableInt32>(42);
```

But we can also tell the `implements` template that we would like to derive from it.

```
struct Derived :
    winrt::implements<Derived, StringableInt32>
{
    Derived() : StringableInt32(42) {}
};
```

Unfortunately, this doesn't work because `StringableInt32` is an indirect base class, and you can initialize only direct base classes in the constructor. So how do you initialize the `StringableInt32`?

The answer can be found by looking at the definition of `winrt::implements`.

```
template <typename D, typename... I>
struct implements :
    impl::producers<D, I...>,
    impl::base_implements<D, I...>::type
{
protected:

    using base_type = typename
        impl::base_implements<D, I...>::type;
    using root_implements_type = typename
        base_type::root_implements_type;
    using is_factory = typename
        root_implements_type::is_factory;

    using base_type::base_type;
```

We saw last time that `base_implements` looks for the template parameter that is itself derived from `implements`, and makes its `type` member type be that template parameter. We define `base_type` to be that type, which in our case is `StringableInt32`.

The `using base_type::base_type` imports the constructors of `base_type` as constructors of `implements`. And for us, this means that we can use the `StringableInt32(int value)` constructor as if it were a constructor of `implements`.

```
struct Derived :
    winrt::implements<Derived, StringableInt32>
{
    Derived() : implements(42) {}
};
```

The base class can expose other methods to the derived class in the usual way.

```
struct StringableInt32 :
    winrt::implements<StringableInt32,
        winrt::Windows::Foundation::IStringable>
{
    StringableInt32(int value) : m_value(value) {}

    winrt::hstring ToString()
    { return winrt::to_hstring(m_value); }

    void set_value(int value) { m_value = value; }

private:
    int m_value;
};

struct Derived :
    winrt::implements<Derived, StringableInt32>
{
    Derived() : implements(42) {}

    void Reset() { set_value(0); }
};
```

Okay, so this works for base classes that can fully implement an interface. But what if the base class wants to delegate part of its implementation to the derived class? We'll look at that next time.