C++/WinRT implementation inheritance: Notes on winrt::implements, part 5

devblogs.microsoft.com/oldnewthing/20250225-00

February 25, 2025

Last time, we saw how you can derive from `winrt::implements` to form a new base class, in the case that the new base class is self-contained.

But what if the new base class isn't self-contained? For example, maybe you want the base class to do the boring paperwork and defer to the derived class for the actual work.

```
namespace winrt
{
    using namespace winrt::Windows::Foundation;
    using namespace winrt::Windows::Web::Http;
    using namespace winrt::Windows::Web::Http::Filters;
}

struct ObserverFilter :
    winrt::implements<ObserverFilter, winrt::IHttpFilter>
{
    ObserverFilter(winrt::IHttpFilter const& downstream) :
        m_downstream(downstream) {}

    winrt::IAsyncActionWithProgress<
        winrt::HttpResponseMessage, winrt::HttpProgress>
        SendRequestAsync(winrt::HttpRequestMessage const& message)
    {
        ⟦ let derived class observe the message ⟧
        return m_downstream.SendRequestAsync(message);
    }
private:
    winrt::IHttpFilter m_downstream;
};
```

This `ObserverFilter` base class does the grunt work of being an HTTP filter (managing the downstream), with a place for the derived class to plug in and do the observing.

You can solve this problem by simply setting aside that you're working in C++/WinRT and solving it using standard C++. In standard C++, you would use a pure virtual method.

```
struct ObserverFilter :
    winrt::implements<ObserverFilter, winrt::IHttpFilter>
{
    ObserverFilter(winrt::IHttpFilter const& downstream) :
        m_downstream(downstream) {}

    winrt::IAsyncActionWithProgress<
        winrt::HttpResponseMessage, winrt::HttpProgress>
        SendRequestAsync(winrt::HttpRequestMessage const& message)
    {
        ObserveMessage(message);
        return m_downstream.SendRequestAsync(message);
    }

    virtual void ObserveMessage(
        winrt::HttpRequestMessage const& message) = 0;
private:
    winrt::IHttpFilter m_downstream;
};
```

The derived class would then implement the `ObserveMessage` method.

```
struct Derived : winrt::implements<Derived, ObserverFilter>
{
    virtual void ObserveMessage(
        winrt::HttpRequestMessage const& message) override
    {
        ⟦ observe the message ⟧
    }
};
```

If the helper base class wants to leave one of the interface methods completely to the derived class, you still have to define it in the base class. You can't just leave it blank and say "Don't worry, my derived class will take care of it."

```
// A simple copy drop target provides no custom feedback
// and accepts anything by copy.
struct SimpleCopyDropTarget :
    winrt::implements<SimpleCopyDropTarget,
    winrt::ICoreDropOperationTarget>
{
    winrt::IAsyncOperation<winrt::DataPackageOperation>
        EnterAsync(winrt::CoreDragInfo const& info,
                   winrt::CoreDragUIOverride const&)
    {
        co_return GetOperation(info);
    }

    winrt::IAsyncOperation<winrt::DataPackageOperation>
        OverAsync(winrt::CoreDragInfo const& info,
                  winrt::CoreDragUIOverride const&)
    {
        co_return GetOperation(info);
    }

    winrt::IAsyncAction
        LeaveAsync(winrt::CoreDragInfo const&)
    {
        co_return;
    }

    // I want to let the derived class implement DropAsync

private:
    winrt::DataPackageOperation GetOperation(
        winrt::CoreDragInfo const& info)
    {
        return info.AllowedOperations() &&
               winrt::DataPackageOperation::Copy;
    }
};
```

This doesn’t work because C++/WinRT expects that if you declare that you implement an interface (in this case, `ICoreDropOperationTarget`), then you implement all of its methods. It requires this so that it can generate the `ICoreDropOperationTarget` vtable entries.

One solution is to declare the method as pure virtual, saying, “Trust me, the derived class will do it.”

```
struct SimpleCopyDropTarget :
    winrt::implements<SimpleCopyDropTarget,
    winrt::ICoreDropOperationTarget>
{
    ⟦ ... ⟧

    virtual winrt::IAsyncOperation<DataPackageOperation>
        DropAsync(winrt::CoreDragInfo const& info) = 0;

private:
    ⟦ ... ⟧
};
```

This does have the downside of requiring the derived class to accept the parameters exactly as you specified, even if they might prefer a different form. For example, since `OverAsync` is a coroutine, the derived class might prefer to accept the `info` by value so that it can carry it across a suspension point.

There are many solutions to this problem. We'll look at one of the next time.