Giving a single object multiple COM identities, part 2

devblogs.microsoft.com/oldnewthing/20211027-00

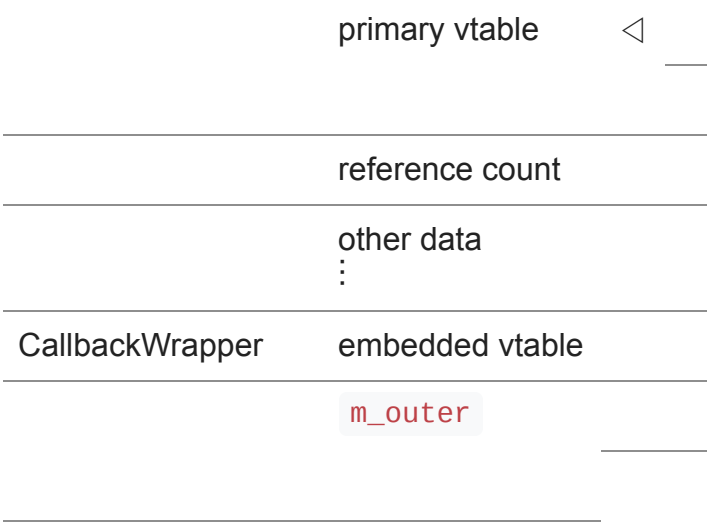
October 27, 2021

Raymond Chen

Last time, we looked at how you can give a single object multiple COM identities so that different clients can maintain references to different parts of the object without knowing about the other parts.

The idea here is that each identity is its own separate collection of `IUnknown` interfaces which are interconnected via `QueryInterface` to each other, but all of the identities share the same `AddRef` and `Release`, so that the lifetime of the collective ends only when all the identities run down.

Our previous implementation gave each embedded identity a pointer to the outer object so it could access the shared `AddRef` and `Release` methods, as well as forward any calls to the outer object for processing.



But it turns out that this pointer is redundant: Recovering the outer object from the embedded object can be done from the embedded object's `this` pointer because the embedded object is at a fixed compile-time offset from the start of the outer object. All we have to do is adjust the embedded object's `this` pointer to get a pointer to the outer object.

```
ULONG CallbackWrapper::AddRef()
{
    Outer* outer =  reinterpret_cast<Outer*>(
        reinterpret_cast<uintptr_t>(this) - offsetof(Outer, m_wrapper));
    return outer->AddRef();
}
```

One major problem is that the `CallbackWrapper` class doesn't know what member name to use as the second parameter to the `offsetof` macro. You could establish a convention that "Oh, the name must be `m_wrapper`," but that would prevent you from having more than one wrapper.

Even if you knew the name of the member variable, you can't calculate the offset of a member variable before the type is complete.

```
struct Widget final : IWidget, ...
{
    ...
    HRESULT OnCallback();
    CallbackWrapper<&Widget::OnCallback> m_wrapper;
    ...
};
```

At the point `CallbackWrapper<;&Widget::OnCallback1>` is instantiated, not only is the `Widget` type incomplete, it doesn't even have a member named `m_wrapper` yet!

But you know what you can do before a type is complete? You can `static_cast` between the type and its base types. The compiler will do the work of calculating and applying the pointer adjustment (once it is known).

So let's try it:

```
ULONG CallbackWrapper::AddRef()
{
    Outer* outer = static_cast<Outer*>(this);
    return outer->AddRef();
}

struct Widget final : IWidget, CallbackWrapper<&Widget::OnCallback>
{
    ...
    HRESULT OnCallback();
    ...
};
```

This doesn't work because we are now talking about `Widget::OnCallback` before it has been declared. Unfortunately, you cannot forward-declare a member function, so we will have to inject a helper forwarder class. To make it easier to find the ultimate target, we remember it as the nested type name `Outer`.

```
template<typename OuterType>
struct MethodForwarder
{
    using Outer = OuterType;

    auto outer() { return static_cast<Outer*>(this); }

    HRESULT OnCallback() { return outer()->OnCallback(); }
};

struct Widget final : IWidget,
    CallbackWrapper<&MethodForwarder<Widget>::OnCallback>,
{
    ...
    HRESULT OnCallback();
    ...

    HRESULT RegisterCallback()
    {
        SetCallback(this);
    }
};
```

There is a unique conversion from `Widget` to `ICallback` (namely through the `CallbackWrapper`), but if you need multiple callbacks, you're in for a lot more typing to resolve the ambiguous conversion:

```
template<typename Outer>
struct MethodForwarder
{
    auto outer() { return static_cast<Outer*>(this); }

    HRESULT OnCallback1() { return outer()->OnCallback1(); }
    HRESULT OnCallback2() { return outer()->OnCallback2(); }
};

struct Widget final : IWidget,
    CallbackWrapper<&MethodForwarder<Widget>::OnCallback1>,
    CallbackWrapper<&MethodForwarder<Widget>::OnCallback2>,
{
    ...
    HRESULT OnCallback1();
    HRESULT OnCallback2();
    ...

    HRESULT RegisterCallback()
    {

SetCallback1(static_cast<CallbackWrapper<&MethodForwarder<Widget>::OnCallback1>*>

(this));

SetCallback2(static_cast<CallbackWrapper<&MethodForwarder<Widget>::OnCallback2>*>

(this));
    }
};
```

We can do some things to help a little with the verbosity: We can create a type alias, a variadic aggregator, and a helper function for casting to the correct base class:

```
template<auto... Callbacks>
struct CallbackWrappers : CallbackWrapper<Callbacks>...
{
    template<auto Callback>
    ICallback* GetCallback() {
        return static_cast<CallbackWrapper<Callback>*>(this);
    }
};

struct Widget;
using WidgetForwarder = MethodForwarder<Widget>;

struct Widget final : IWidget,
    CallbackWrappers<&WidgetForwarder::OnCallback1,
                     &WidgetForwarder::OnCallback2>
{
    ...
    HRESULT OnCallback1();
    HRESULT OnCallback2();
    ...

    HRESULT RegisterCallback()
    {
        SetCallback1(GetCallback<&WidgetForwarder::OnCallback1>(this));
        SetCallback2(GetCallback<&WidgetForwarder::OnCallback2>(this));
    }
};
```

An alternative to using the method forwarder would be to split the Widget into two parts, one that contains the bulk of the implementation and one that adds the callbacks.

```
struct WidgetImpl : IWidget
{
    ...
    HRESULT OnCallback1();
    HRESULT OnCallback2();
    ...
};

struct Widget final : WidgetImpl,
    CallbackWrappers<&WidgetImpl::OnCallback1,
                     &WidgetImpl::OnCallback2>
{
};
```

This alternative version solves one problem but adds a new one: How does `WidgetImpl` access the callbacks? One idea is to use the CRTP pattern to allow `WidgetImpl` to cast to its derived type:

```
template<typename D>
struct WidgetImpl : IWidget
{
    ...
    HRESULT OnCallback1();
    HRESULT OnCallback2();
    ...
};

struct Widget final : WidgetImpl<Widget>,
    CallbackWrappers<&WidgetImpl<Widget>::OnCallback1,
                     &WidgetImpl<Widget>::OnCallback2>
{
};
```

But this is just about as clunky, so it doesn't really save us much.

We're nowhere near done yet, because this still doesn't work. We'll pick up the investigation next time.

Raymond Chen

Follow