# Giving a single object multiple COM identities, part 1

devblogs.microsoft.com/oldnewthing/20211026-00

October 26, 2021

Raymond Chen

Last time, we looked at patterns for COM objects that need to hand out references to themselves. One common pattern is to make the callbacks separate objects that retain strong references to (and forward the callbacks to) the main object. An alternative pattern is to have the main object implement all of the necessary callback interfaces, but also observed that this had some downsides: The clients as well as the event sources have full access to the object, including interfaces that they weren't intended to access, and there's no way to register the object against multiple different event sources that use the same callback interface (but against which you want to register different handlers).

So it looks like we're stuck with the pattern of making the callback be a separate object, and paying for the extra allocations. This can be quite cumbersome if you need to generate these callbacks frequently, say, because they are completion callbacks for operations you perform frequently. This happens a lot in multimedia, where you might say "Go read some samples from the audio card, and then call this callback when you're done." You'd rather not have to keep creating these one-time callback objects.

Fortunately, the situation is not hopeless. You can still keep your callback in the main object. You just have to give it a separate COM identity.

| | | | | | |
|---|---|---|---|---|---|
| | | Widget | | | |
| client | → | IUnknown | IWidget | | |
| | | refcount | | | |
| | | IUnknown | ICallback | ← | event source |
| | | state data | | | |

The idea here is that the main object and the callback share the same reference count, but they expose separate COM identities. The `IUnknown` given to the client responds to `Query-Interface` with itself or the `IWidget`. The one given to the event source responds to itself or the client's `ICallback`. We want to write something like this:

```
struct Widget : public IWidget, public ICallback
{
    HRESULT IWidget::IUnknown::QueryInterface(REFIID riid, void** ppv)
    {
        if (riid == IID_IUnknown || riid == IID_IWidget) {
            *ppv = static_cast<IWidget*>(this);
            AddRef();
            return S_OK;
        }
        *ppv = nullptr;
        return E_NOINTERFACE;
    }

    HRESULT IWidget::WidgetMethod() { ... }

    HRESULT ICallback::IUnknown::QueryInterface(REFIID riid, void** ppv)
    {
        if (riid == IID_IUnknown || riid == IID_ICallback) {
            *ppv = static_cast<ICallback*>(this);
            AddRef();
            return S_OK;
        }
        *ppv = nullptr;
        return E_NOINTERFACE;
    }

    HRESULT ICallback::CallbackMethod() { ... }

    // AddRef and Release are shared by IWidget and ICallback
    ULONG AddRef() { ... }
    ULONG Release() { ... }
}
```

(I'm removing some of the macros for declaring COM interfaces for expository purposes.)

You can't actually write this, though. If a method is inherited from multiple base classes, you cannot override each one separately. Your only choice is to override all of them with the same function.

The standard workaround for this is to demote one of the base classes to a member. We'll demote the callback.

```
template<typename D>
struct CallbackWrapper : public ICallback
{
    D* m_outer;

    CallbackWrapper(D* outer) : m_outer(outer) { }

    HRESULT QueryInterface(REFIID riid, void** ppv)
    {
        if (riid == IID_IUnknown || riid == IID_ICallback) {
            *ppv = static_cast<ICallback*>(this);
            AddRef();
            return S_OK;
        }
        *ppv = nullptr;
        return E_NOINTERFACE;
    }

    // Forward everything else to the outer class
    D* outer() { return m_outer; }
    ULONG AddRef() { return outer()->AddRef(); }
    ULONG Release() { return outer()->Release(); }

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

struct Widget : public IWidget
{
    HRESULT QueryInterface(REFIID riid, void** ppv)
    {
        if (riid == IID_IUnknown || riid == IID_IWidget) {
            *ppv = static_cast<IWidget*>(this);
            AddRef();
            return S_OK;
        }
        *ppv = nullptr;
        return E_NOINTERFACE;
    }

    ULONG AddRef() { ... }
    ULONG Release() { ... }

    HRESULT WidgetMethod() { ... }
    HRESULT CallbackMethod() { ... }


    CallbackWrapper<Widget> m_wrapper{ this };
};
```

You can pass the `m_wrapper` to the event source to pass the callback.

```
EventSource::RegisterCallback(&m_wrapper);
```

Demoting it to a member also means that you can create multiple wrappers for different callbacks.

```
template<auto Method>
struct CallbackWrapper : public ICallback
{
    template<typename Outer> static Outer outer_type(HRESULT(Outer::*)());
    using Outer = decltype(outer_type(Callback));

    Outer* m_outer;

    CallbackWrapper(Outer* outer) : m_outer(outer) { }

    HRESULT QueryInterface(REFIID riid, void** ppv)
    {
        if (riid == IID_IUnknown || riid == IID_ICallback) {
            *ppv = static_cast<ICallback*>(this);
            AddRef();
            return S_OK;
        }
        *ppv = nullptr;
        return E_NOINTERFACE;
    }

    // Forward everything else to the outer class
    Outer* outer() { return m_outer; }
    ULONG AddRef() { return outer()->AddRef(); }
    ULONG Release() { return outer()->Release(); }

    HRESULT CallbackMethod() { return (outer()->*Callback)(); }
};

struct Widget : public IWidget
{
    HRESULT QueryInterface(REFIID riid, void** ppv)
    {
        if (riid == IID_IUnknown || riid == IID_IWidget) {
            *ppv = static_cast<IWidget*>(this);
            AddRef();
            return S_OK;
        }
        *ppv = nullptr;
        return E_NOINTERFACE;
    }

    HRESULT WidgetMethod() { ... }

    ULONG AddRef() { ... }
    ULONG Release() { ... }

    HRESULT CallbackMethod1() { ... }
    HRESULT CallbackMethod2() { ... }

    CallbackWrapper<&Widget::CallbackMethod1> m_wrapper1{ this };
    CallbackWrapper<&Widget::CallbackMethod2> m_wrapper2{ this };
};
```

This is important if your Widget performs multiple operations that all use the same callback interface. You need a way to distinguish the different instances of the callback.

There's some template magic going on here, taking advantage of the `auto` non-type template parameter introduced in C++17. Prior to C++17, you would have had to declare the `Outer` type explicitly:

```
template<typename Outer, HRESULT(Outer::*Method)()>
{
    ...
};

...
    CallbackWrapper<Widget, &Widget::CallbackMethod1> m_wrapper1{ this };
    CallbackWrapper<Widget, &Widget::CallbackMethod2> m_wrapper2{ this };
```

With C++17, we can accept just the function pointer and then infer the associated class with some template magic:

```
    template<typename Outer> static Outer outer_type(HRESULT(Outer::*)());
    using Outer = decltype(outer_type(Callback));
```

The `outer_type` function takes a callback function and returns the associated class. The method is never called, so it needs no implementation. We are just using it for the template inference. The existence of a function declared but never implemented might trigger some warnings, so you can work around it in a few different ways.

One is to give it an implementation (that is nevertheless never called). For example, you might try to use `std::declval` :

```
    template<typename Outer> static Outer outer_type(HRESULT(Outer::*)())
    { return std::declval<Outer>(); }
    using Outer = decltype(outer_type(Callback));
```

The `outer_type` function is called only in a non-evaluated context, so it is not odr-used. On the other hand, I'm not sure whether the instantiation of the templated `outer_type` function causes its body to become instantiated, which in turn triggers the odr-usage of `std::declval` , which is not allowed.

A safer solution would be

```
    template<typename Outer> static Outer* outer_type(HRESULT(Outer::*)())
    { return nullptr; }
    using Outer = decltype(std::remove_pointer_t(outer_type(Callback)));
```

I’ve been showing an implementation in straight C++, but it also works in a framework, as long as you have some way of using the outer object to (1) adjust the reference count and (2) call the method you want to forward to. In most frameworks, the most-derived class provides all you need, so the above code works even if you declare the COM class with ATL or WRL or C++/WinRT.

One annoyance here is that we have to remember the outer object’s `this` pointer, even though it’s something that could be hard-coded by taking advantage of the fixed layout of the `Widget` object. We’ll look at removing that `m_outer` member next time.

Raymond Chen

**Follow**