# Giving a single object multiple COM identities, part 4

devblogs.microsoft.com/oldnewthing/20211029-00

October 29, 2021

Raymond Chen

One major annoyance of the helper classes for giving a single object multiple COM identities is that you have to create this `MethodForwarder` so that you can talk about methods of the derived class before they exist.

But it turns out that you can avoid that by simply not having to name the functions at all!

We have been using the member function pointer as our way of referring to the function, and that means having to name it. But what if we used something else to refer to the function? Just let the caller pass whatever identifier they like.

```
template<typename Outer, auto Id>
struct CallbackWrapper
{
    struct Wrapper : ICallback
    {
        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;
        }

        ULONG AddRef() { return outer()->AddRef(); }
        ULONG Release() { return outer()->Release(); }

        HRESULT Invoke() noexcept { return outer()->OnInvoke<Id>(); }

        private:
        Outer* outer() {
            return static_cast<Outer*>(
                reinterpret_cast<CallbackWrapper*>(this));
        }
    } callback;

    ICallback* GetCallback() { return &callback; }
};

template<typename Outer, auto... Ids>
struct CallbackWrappers : CallbackWrapper<Derived, Ids>...
{
    template<auto Id>
    ICallback* GetCallback() {
        return static_cast<CallbackWrapper<Derived, Id>*>(this)->GetCallback();
    }

    template<auto Id>
    using CallbackWrapperOuter = CallbackWrapper<Outer, Id>;
    static_assert(((offsetof(CallbackWrapperOuter<Ids>, callback) == 0) && ...));
};
```

The idea here is that instead of passing a list of pointers to member functions, you just pass a list of *whatever you like*. Each *whatever* identifies a callback, and when invoked, it calls the corresponding `OnInvoke` method on the base class with the same *whatever*.

It's not obvious why we had to introduce the `CallbackWrapperOuter` templated type. What was wrong with this?

```
    static_assert(((offsetof(CallbackWrapper<Outer, Ids>, callback) == 0) && ...));
```

What’s wrong with it is the command that separates the two `CallbackWrapper` template parameters. The presence of this comma confuses the `offsetof` macro, since we saw some time ago that the C++ preprocessor doesnt understand anything about C++, and certainly not templates. If the parameter to the `offsetof` macro had been an expression, we could have wrapped an extra set of parentheses around it, but in this case, it is a type, so we need to create a comma-free type alias for it.

You can now derive from the `CallbackWrappers` templated type without needing a method-forwarder helper class.

```
struct Widget final : IWidget
    CallbackWrappers<Widget, 1, 2>
{
    ...
    template<auto Id> HRESULT OnInvoke() = delete;
    template<> HRESULT OnInvoke<1>();
    template<> HRESULT OnInvoke<2>();
    ...

    void RegisterCallbacks()
    {
        SetCallback1(GetCallback<1>(this));
        SetCallback2(GetCallback<2>(this));
    }
};
```

The `Widget` declares a templated member function called `OnInvoke` and immediately deletes it. Deleting the member function means that if you somehow do a

```
        GetCallback<9>(this);
```

you get a compile-time error if there is no 9-specialization. (Without the `= delete`, the error would not be detected until link time.)

You can wrap these names so you don’t have to remember the numbers.

```
    HRESULT OnMicrophoneReady() { return OnInvoke<1>(); }
    ICallback* GetMicrophoneReadyCallback() { return GetCallback<1>(); }
```

But wait, the identifier for the callback doesn’t have to be an integer. It can be anything that is a valid non-type template parameter. And they don’t even all have to be the same type!

```
enum class AudioStreamingCallbackKind
{
    MicrophoneReady,
    RefillOutputBuffer,
};

enum class DecodeCallbackKind
{
    DataAvailable,
};

struct Widget final : IWidget
    CallbackWrappers<Widget,
        AudioStreamingCallbackKind::MicrophoneReady,
        AudioStreamingCallbackKind::RefillOutputBuffer,
        DecodeCallbackKind::DataAvailable,
        &SomeVariable>
{
    template<auto Id> HRESULT OnInvoke() = delete;
    template<> HRESULT OnInvoke<AudioStreamingCallbackKind::MicrophoneReady>();
    template<> HRESULT OnInvoke<AudioStreamingCallbackKind::RefillOutputBuffer>();
    template<> HRESULT OnInvoke<DecodeCallbackKind::DataAvailable>();
    template<> HRESULT OnInvoke<&SomeVariable>();

    void GetReady()
    {

CallWhenMicrophoneReady(GetCallback<AudioStreamingCallbackKind::MicrophoneReady>());
    }

    void StartStreaming()
    {
        CallWhenDecoded(GetCallback<DecodeCallbackKind::DataAvailable>());

CallWhenBufferEmpty(GetCallback<AudioStreamingCallbackKind::RefillOutputBuffer>());
    }

    void EnjoyTheSunshine()
    {
        CallWhenSunny(GetCallback<&SomeVariable>());
    }
};
```

Coming up with an enumeration is certainly more readable than just matching up integers, but it’s kind of annoying having to define the enumeration. The language does not permit string literals to be non-type template parameters, but we can get close, by hashing the strings:

```
constexpr uint64_t callback_id(char const* s)
{
    // Untested implementation of FNV-1a
    uint64_t result = 0xcbf29ce484222325;
    for (size_t index = 0; s[index] != 0; index++) {
        result = (result ^ s[index]) * 1099511628211;
    }
    return result;
}
```

You can use this helper function to generate identifiers on demand:

```
struct Widget final : IWidget
    CallbackWrappers<Widget,
        callback_id("microphoneReady"),
        callback_id("dataAvailable")>
{
    template<auto Id> HRESULT OnInvoke() = delete;
    template<> HRESULT OnInvoke<callback_id("microphoneReady")>();
    template<> HRESULT OnInvoke<callback_id("dataAvailable")>();

    void Initialize()
    {
        CallWhenMicrophoneReady(GetCallback<callback_id("microphoneReady")>());
        CallWhenDataAvailable(GetCallback<callback_id("dataAvailable")>());
    }
};
```

You can save even more typing by using a user-defined literal:

```
constexpr uint64_t operator"" _fnv1a(char const*s, size_t)
{
    return callback_id(s);
}

struct Widget final : IWidget
    CallbackWrappers<Widget,
        "microphoneReady"_fnv1a,
        "dataAvailable"_fnv1a>
{
    template<auto Id> HRESULT OnInvoke() = delete;
    template<> HRESULT OnInvoke<"microphoneReady"_fnv1a>();
    template<> HRESULT OnInvoke<"dataAvailable"_fnv1a>();

    void Initialize()
    {
        CallWhenMicrophoneReady(GetCallback<"microphoneReady"_fnv1a>());
        CallWhenDataAvailable(GetCallback<"dataAvailable"_fnv1a>());
    }
};
```

Okay, that's not so bad. A helper type for generating multiple COM ad-hoc-named identities from a single C++ object, without any overhead beyond the required vtable.

Extending this pattern to support arbitrary callback types is left as a exercise. (You'll want to use templates to deduce the parameters of the Invoke method.)

Even if you don't ever plan on using this thing (I'm not sure I ever would), we at least got a practical tour of quite a few C++ features:

- Aggregating types via variadic templates
- Fold expressions
- CRTP
- Delayed-expansion via templates
- `auto` non-type template parameters
- Working around multi-parameter templates in macro arguments
- Using a template member function as a way to match up pairs of functions
- Designing code so that errors are detected at compile time rather than link time
- User-defined literals

Raymond Chen

**Follow**