Giving a single object multiple COM identities, part 3

devblogs.microsoft.com/oldnewthing/20211028-00

October 28, 2021

Raymond Chen

Last time, we left off our investigation of how to give a single object multiple COM identities without any data overhead, by tricking the compiler into generating the adjustor thunks automatically. We had managed to build the callbacks into base classes, using base classes means that we're back to the problem of having multiple implementations of `IUnknown`, which the C++ language does not permit to implement separately. We saw that the standard workaround for this is to move the `ICallback` into a member variable.

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

    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()->*Callback)(); }

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

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

This code makes the assumption that the first instance data member of a class with no virtual methods be pointer-interconvertible with the class itself. According to the standard, this assumption is valid only for standard-layout types, and `CallbackWrapper` is not a standard layout type because it has a member `callback` which is not a standard-layout type. On the other hand, in practice the assumption holds. We can add some extra assertions to verify this:

```
template<auto Callback>
struct CallbackWrapper
{
    ...

    CallbackWrapper() {
        assert(reinterpret_cast<void*>(this) == &callback);
    }
};

template<auto Callbacks...>
struct CallbackWrapper : CallbackWrapper<Callbacks>...
{
    static_assert(((offsetof(CallbackWrapper<Callbacks>, callback) == 0) && ...));
    template<auto Callback>
    ICallback* GetCallback() {
        return static_cast<CallbackWrapper<Callback>*>(this);
    }
};
```

We add a static assertion that verifies that the member variable `callback` has the same address as the containing class `CallbackWrapper` in all of the base classes. The proliferation of parentheses comes from the language rules for fold expressions:

```
    (expr && ...)
```

The outer parentheses around the fold expression are mandatory. Furthermore, the `expr` must be a *cast-expression*. The equality comparison operator is lower precedence than a cast, so we need to parenthesize it to turn it into a higher-priority *primary-expression*. And the final extra set of parentheses come from the syntax of `static_assert` itself, which requires that its argument be parenthesized.

Now, this compile-time check is not standard-conforming, because the `offsetof` macro supports only standard-layout types (although it works well enough in practice), so we supplement it with a runtime assertion.[1]

With all these changes, the resulting code generation is quite efficient:

```
CallbackWrapper<&WidgetImpl<Widget>::OnCallback1>::OnCallback1:
    sub     rcx, 8
    jmp     Widget::OnCallback1
```

If you throw in link-time code generation, then the compiler will even notice that `Widget::OnCallback1` is called from only one place, so it will inline it into the `CallbackWrapper`, resulting in the wrapper ending up with no cost at all. (The adjustment of the `this` pointer can be reduced to zero cost by folding it into the subsequent member offsets.)

Having to create the `MethodWrapper` class is a bit of an annoyance, though. You have to put into that class every function you might want to forward. (Fortunately, it's okay to have methods corresponding to functions you never use, since they will never be instantiated, and therefore the compiler will never notice that the forwarded-to function doesn't exist.)

We'll try to simplify that next time.

[1] The standard does require that a union be pointer-interconvertible with its members, so we could have made the `CallbackWrapper` be a union with the `Wrapper` as its sole member. However, unions cannot be base classes, so that messes up the "Derive from this thing" step.

Raymond Chen

Follow