I'd like an IUnknown, I know you have many, I'll take any of them

devblogs.microsoft.com/oldnewthing/20210101-00

January 1, 2021

Raymond Chen

A concrete implementation of a COM object may implement multiple interfaces. If you have a pointer to the concrete implementation, and you pass to a function that expects an `IUnknown`, you will probably get an error complaining that `IUnknown` is an ambiguous base, or that there is an ambiguous conversion to `IUnknown`.

```
void DoSomething(IUnknown* unk);

class MyClass : public IFred, public IBarney
{
    ...

    void SomeMethod()
    {
        DoSomething(this); // fails to compile
    }
};

using namespace Microsoft::WRL;

class MyWrlClass :
    RuntimeClass<RuntimeClassFlags<ClassicCom>,
                 IFred, IBarney>
{
    ...
    void SomeMethod()
    {
        DoSomething(this); // fails to compile
    }
};
```

The problem is that when you call `DoSomething(this)`, the compiler doesn't know whether you want to pass the `IUnknown` that is a base class of `IFred`, or the `IUnknown` that is a base class of `IBarney`.

What we know and the compiler doesn't know is that it doesn't matter which one you pass. They are functionally equivalent. (The only time you are particular about which one you get is if you are trying to get the canonical unknown, but that is typically only something needed by the `QueryInterface` method.)

The usual solution is to cast the implementation pointer to one of the interfaces that it unambiguously implements, and then let the compiler convert that interface pointer to `IUnknown`. This does require you to know which interfaces the object implements, which could be a source of fragility if the object gains or loses interfaces over time.

From a code size point of view, you want to choose the interface that is the first base class, assuming that the first base class is an interface. That way, the conversion is a nop.

```
        DoSomething(static_cast<IFred>(this));
```

If you use the WRL template library to create your COM objects, then there's a handy helper function: `CastToUnknown`. This takes the implementation pointer and casts it to `IUnknown`, saving you the trouble of having to decide which of the many possible paths to `IUnknown` to use. This is a protected method, so you can use it from within the class, but not from the outside.

```
class MyWrlClass :
    RuntimeClass<RuntimeClassFlags<ClassicCom>,
                 IFred, IBarney>
{
    ...
    void SomeMethod()
    {
        DoSomething(this->CastToUnknown());
    }
};

void SomeFunction(MyWrlClass* p)
{
  // this doesn't compile
  DoSomething(p->CastToUnknown());
}
```

The call to `CastToUnknown` from the `SomeFunction` is disallowed because the `CastTo-Unknown` method is protected.

But you could choose to unprotected it.

```
class MyWrlClass :
    RuntimeClass<RuntimeClassFlags<ClassicCom>,
                 IFred, IBarney>
{
public:
    using RuntimeClass::CastToUnknown;
    ...
};
```

The `using` statement imports the base class's `CastToUnknown` method, and since the `using` statement is in the `public` section, the imported function is now `public`.

But really, the point of this article is to call out the existence of the `CastToUnknown` method. It's really handy when you need it, such as when you want to extend your object's lifetime:

```
Callback<ISomething>(
    [lifetime = ComPtr<IUnknown>(this->CastToUnknown())]()
    {
      ...
    });
```

Unfortunately, it's still quite a bit of a mouthful. You can factor it out to avoid having to type the whole thing out all the time.

```
template<typename T> ComPtr<T> AsComPtr(T* p) { return p; }

Callback<ISomething>(
    [lifetime = AsComPtr(this)]()
    {
      ...
    });
```

Note that this isn't quite the same as the previous version because the resulting `ComPtr` is a `ComPtr<MyWrlClass>` rather than a `ComPtr<IUnknown>`, but that works just as well for the purpose of extending the object's lifetime.

WRL was written when the latest version of the C++ language was C++11, so it doesn't doesn't have access to CTAD. If CTAD were around, it could have had a deduction guide:

```
template<typename T> ComPtr(T*) -> ComPtr<T>;
```

That would avoid the need for the `AsComPtr` helper function.

```
Callback<ISomething>(
    [lifetime = ComPtr(this)]()
    {
      ...
    });
```

Raymond Chen

Follow