How well does WRL ComPtr support class template argument deduction (CTAD)?

devblogs.microsoft.com/oldnewthing/20240315-00

March 15, 2024

Raymond Chen

Continuing our investigation of which C++ COM wrappers support class template argument deduction (CTAD), next up is WRL's `ComPtr`.

WRL fails CTAD because it tries too hard. The from-raw-pointer constructor is this one:

```
template <typename T>
class ComPtr
{
public:
    typedef T InterfaceType;

protected:
    InterfaceType *ptr_;

public:
    template<class U>
    ComPtr(_In_opt_ U *other) throw() : ptr_(other) // this one
    {
        InternalAddRef();
    }

    ⟦ other stuff ⟧
};
```

WRL thinks is being helpful here in that it lets you, for example, initialize a `ComPtr<IBase>` not only with a pointer to `IBase` but also a pointer to anything that derives from `IBase`. If you pass a pointer to a derived class, it upcasts it into a pointer to the base class.

This attempt at being helpful prevents CTAD from working, because the compiler can't figure out which `T` to use when it's given just a `U`.

WRL could just have written the simple, straightforward constructor:

```
    ComPtr(_In_opt_ T *other) throw() : ptr_(other)
    {
        InternalAddRef();
    }
```

If the caller passes a pointer to something that doesn't derive from `T`, then the compiler tells you that it can't find a matching constructor.

This fancy template constructor is a case of what the Germans call *Verschlimmbesserung*, which means "making something worse in a well-intentioned but failed attempt to make it better."

But I think I know why the authors of WRL wrote their raw-pointer constructor this way, and it's a common problem to library authors: Error message metaprogramming.

If somebody tries to construct a `ComPtr<T>` from a pointer to something unrelated to `T`, the fancy templated version gives the error message

```
error C2440: 'initializing': cannot convert from 'U *' to 'IBase *'
with
    [
        U=INotDerived
    ]
```

Without the fancy template, the error message would have been

```
error C2440: '<function-style-cast>': cannot convert from 'INotDerived *' to 'Microsoft::
WRL::ComPtr<IBase>'
'Microsoft::WRL::ComPtr<IBase>::ComPtr' no overloaded function could convert all the argument
types
could be 'ComPtr<IBase>::ComPtr(const ComPtr<U> &amp;,typename Details::EnableIf<Details::
IsConvertible<U*, T*>::value, void *>::type *)'
with
    [
        T=IBase
    ]
or 'ComPtr<IBase>::ComPtr(nullptr)'
'ComPtr<IBase>::ComPtr(nullptr)': cannot convert argument 1 from 'INotDerived *' to 'nullptr'
or
    [[ repeat for the other constructors ]]
```

A simple error on the client turns into an explosive and incomprehensible error message.

The library author can get the best of both worlds by providing both the fancy templated constructor (to improve the error message) and a simple, straightforward constructor (for CTAD):

```
template <typename T>
class ComPtr
{
public:
    typedef T InterfaceType;

protected:
    InterfaceType *ptr_;

public:
    // This one for CTAD
    ComPtr(_In_opt_ T *other) throw() : ptr_(other)
    {
        InternalAddRef();
    }

    // This one to improve error message
    template<class U>
    ComPtr(_In_opt_ U *other) throw() :
        // ComPtr(static_cast<T*>(other)) {}
        ComPtr(MustDeriveFromT(other)) {}

    ⟦ other stuff ⟧

private:
    constexpr static T* MustDeriveFromT(T* p) { return p; }
};
```

Or the library author can provide a deduction guide to steer CTAD to the correct type:

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

Since WRL was written for C++11, you'd have to put the deduction guide inside an `#ifdef` to hide it from pre-C++17 compilers.

As a consumer, though, you shouldn't be creating deduction guides for somebody else's classes. Instead, you can use a maker function.

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

Next time, we'll look at wil, which has a different category of problems.