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

devblogs.microsoft.com/oldnewthing/20240312-00

March 12, 2024

Raymond Chen

We saw earlier that the `_com_ptr_t` class template doesn't work well with class template argument deduction (CTAD).

Template argument deduction fails on this constructor:

```
template<typename _IIID> class _com_ptr_t {
public:
    using ThisIIID = _IIID;
    using Interface = typename _IIID::Interface;

    _com_ptr_t(Interface* pInterface); // this one

    [[ other stuff ]]
};
```

The problem is that the compiler can't figure out what `_IIID` to use that will produce the desired `Interface`. After all, there are an infinite number of such classes.

```
struct SomeRandomClass
{
    using Interface = ::IWidget;
};

struct AnotherRandomClass
{
    using Interface = ::IWidget;
};
```

Both of these random classes are viable candidates for the template type argument `_IIID`. Clearly there is no reason for the compiler to choose one over the other. The general rule is that the compiler will try to deduce the template arguments from the parameter list, but this requires that the parameter list involves the template arguments at all!

The library intends the `_IIID` to be a specialization of a helper type named `_com_IIID`, where `IIID` presumably stands for Interface and Interface identifier.

```
template<typename _Interface, const IID* _IID /*= &__uuidof(_Interface)*/>
class _com_IIID {
public:
    typedef _Interface Interface;

    static const IID& GetIID() noexcept
    {
        return *_IID;
    }

    [[ other stuff ]]
};
```

If the library had the ability to see into the C++17 future, it could have provided a deduction guide:

```
template<typename T> _com_ptr_t(T*)
    -> _com_ptr_t<_com_IIID<T, &__uuidof(T)>>;
```

This tells the compiler that if it sees a `_com_ptr_t(T*)`, it should construct a `_com_ptr_t<_com_IIID<T, &__uuidof(T)>>`.

As a rule of thumb, you shouldn't create deduction guides for types defined in someone else's library, because a future version of the library might add a deduction guide, and now you have a conflict between your deduction guide and the library's. Deduction guides can be thought of as constructor metadata, and the class controls its own constructors.

In practice, this lack of CTAD support is not an issue because you don't use `_com_ptr_t` directly; instead, you use the `_COM_SMARTPTR_TYPEDEF` macro to define a custom type name for your specialized smart pointer, and then you use that custom type name from then on.

```
_COM_SMARTPTR_TYPEDEF(IWidget, __uuidof(IWidget));

IWidget* p;
auto smart = IWidgetPtr(p);
```

So really, this was much ado about nothing.

But what if you really want to avoid typing out the interface name when using smart pointers based on `_com_ptr_t`? We'll find the answer when we look at MFC `IPTR`/`CIP` next time.