How well does MFC IPTR/CIP support class template argument deduction (CTAD)?

devblogs.microsoft.com/oldnewthing/20240313-00

March 13, 2024

Raymond Chen

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

The `CIP` template class fails CTAD because of these constructors:

```
template<class _Interface, const IID* _IID>
class CIP : public _CIP<_Interface, _IID>
{
public:
    typedef _CIP<_Interface, _IID> BC;
    typedef _Interface Interface;

    CIP(Interface* pInterface); // this one
    CIP(Interface* pInterface, BOOL bAddRef); // and this one

    [[ other stuff ]]
};
```

In this case, the compiler can deduce the `_Interface` template parameter, but it doesn't know what `_IID` to use. Again, adding some deduction guides can fix that by saying that it should use the `__uuidof(T)` as the `_IID`.

```
template<typename T> CIP(T*)       -> CIP<T, &__uuidof(T)>;
template<typename T> CIP(T*, BOOL) -> CIP<T, &__uuidof(T)>;
```

As noted before, injecting your own deduction guides into somebody else's types is not recommended because it may conflict with deduction guides added by the type owner in the future.

Fortunately, in the case of MFC `IPTR`/`CIP`, you rarely use `CIP` directly. Instead, you use the `IPTR` that was generated from the `CIP`, and that type already knows what interface it is for, so there's no need for CTAD.

```
using TestPtr = IPTR(Test);

void sample(Test* p)
{
    auto smart = TestPtr(p);
}
```

Okay, but what if you still want to create an MFC smart pointer from a raw pointer without having to specify the interface?

You can do what C++ did before CTAD was invented: "You kids with your CTADs and deduction guides. In my day, we had to use maker template functions,[1] and we liked it!"

```
template<typename T>
CIP<T, &__uuidof(T)> make_CIP(T* p)
{
    return p;
}

template<typename T>
CIP<T, &__uuidof(T)> make_CIP(T* p, BOOL bAddRef)
{
    return { p, bAddRef };
}

void sample(Test* p)
{
    auto smart1 = make_CIP(p);
    auto smart2 = make_CIP(p, FALSE);
}
```

This trick also works for `_com_ptr_t`, which is something I teased last time.

```
template<typename T>
_com_ptr_t<_com_IIID<T, &__uuidof(T)>>
    make_com_ptr_t(T* p)
{
    return p;
}

void sample(Test* p)
{
    auto smart = make_com_ptr_t(p);
}
```

[1] There are maker template functions all over the C++ standard library. `std::back_inserter`, `std::make_reverse_iterator`, `std::make_pair`, *etc*.