Class template argument deduction (CTAD) and C++ COM wrappers: Initial explorations

devblogs.microsoft.com/oldnewthing/20240311-00

March 11, 2024

Raymond Chen

A while back, we studied the duck typing requirements of C++ COM wrappers and summarized them in a table. Today we'll look at a smaller point of comparison: Class template argument deduction, also known as CTAD, introduced in C++17.

CTAD lets you omit the `<...>` arguments of a class template under certain circumstances. For example, you can write

```
auto v = std::vector({ 1, 2, 3 });
```

instead of

```
auto v = std::vector<int>({ 1, 2, 3 });
```

You may even have been using this feature without realizing it:

```
auto lock1 = std::lock_guard(m_mutex1);
std::lock_guard lock2(m_mutex2);
```

These are shorthand for

```
auto lock1 = std::lock_guard<std::mutex>(m_mutex1);
std::lock_guard<std::mutex>( lock2(m_mutex2);
```

For C++ COM wrappers, a common pattern is constructing a smart pointer from a raw pointer. Let's see how well these wrapper classes handle CTAD.

```
IWidget* p;

// _com_ptr_t: nope
auto smart = _com_ptr_t(p); // does not compile

// MFC IPTR/CIP: nope
auto smart = CIP(p); // does not compile
auto smart = CIP(p, TRUE); // does not compile

// ATL CComPtr: yes
auto smart = CComPtr(p); // deduces CComPtr<IWidget>

// WRL ComPtr: nope
auto smart = ComPtr(p): // does not compile

// wil com_ptr: maybe
auto smart = wil::com_ptr(p); // requires C++20

// C++/WinRT com_ptr: nope
auto smart = winrt::com_ptr(p); // does not compile
```

Note that these tests are unfair, because all of these libraries predate C++17!

We'll spend the next few days looking at why CTAD doesn't work, how the library authors could have supported CTAD (had they known about it), and what we as library consumers can do about it.