What are the duck-typing requirements of _com_ptr_t?

devblogs.microsoft.com/oldnewthing/20230508-00

May 8, 2023

Raymond Chen

The great thing about COM smart pointer classes is that there are so many to choose from.

In general, the smart pointer classes don't even care that the thing you're wrapping isn't even a COM interface pointer. As long as your class supports appropriate method signatures, the smart pointer classes will take them. This is a general property of C++ templates: If the template type parameter supports the operations performed by the template, then it doesn't really matter where they came from. The case where you can run into trouble is when the library does a blind cast of your pointer to `IUnknown*` (either via `reinterpret_cast` or more commonly by a C-style cast), in which case the code is assuming that your class really does derive from `IUnknown`.[1]

So let's look at the minimum requirements for simply using the COM smart pointer wrapper as a way to manage object lifetime, without any actual COM functionality like changing interfaces via `QueryInterface`. Here are the operations we're looking for:

- Default construction,
- Construction from a raw pointer (`AddRef`),
- Copy construction (`AddRef`),
- Destruction (`Release`),
- Attaching and detaching (adopting and relinquishing ownership),
- Assignment to same-type raw pointer (`AddRef`),
- Assignment to same-type smart pointer (`AddRef`),
- Accessing the wrapped object,
- Returning to empty state (`Release`),
- Receiving a new pointer,
- Bonus: Comparison.
- Litmus test: Accidentally bypassing the wrapper.
- Litmus test: Construction from other-type raw pointer.
- Litmus test: Construction from other-type smart pointer.
- Litmus test: Assignment from other-type raw pointer.
- Litmus test: Assignment from other-type smart pointer.

In addition to some core operations, we have a bonus operation, as well as some litmus tests to see what happens if you make a mistake.

We’ll start with the `_com_ptr_t` class that comes with the `#import` directive. All of our tests will use the same basic framework. For the most part, our changes will be focused on the highlighted portions.

```
// Dummy implementations of AddRef and Release for
// testing purposes only. In real code, they would
// manage the object reference count.
struct Test
{
    void AddRef() {}
    void Release() {}
    Test* AddressOf() { return this; }
};

struct Other
{
    void AddRef() {}
    void Release() {}
};

// Pull in the smart pointer library
// (this changes based on library)
#include <comdef.h>
extern IID IID_NeverDefined; // never defined
_COM_SMARTPTR_TYPEDEF(Test, IID_NeverDefined);
_COM_SMARTPTR_TYPEDEF(Other, IID_NeverDefined);

void test()
{
    Test test;

    // Default construction
    TestPtr ptr;

    // Construction from raw pointer
    TestPtr ptr2(&test);

    // Copy construction
    TestPtr ptr3(ptr2);

    // Attaching and detaching
    auto p = ptr3.Detach();
    ptr.Attach(p);

    // Assignment from same-type raw pointer
    ptr3 = &test;

    // Assignment from same-type smart pointer
    ptr3 = ptr;

    // Accessing the wrapped object
    // (this changes based on library)
    if (ptr.GetInterfacePtr() != &test) {
        std::terminate(); // oops
    }
    if (ptr->AddressOf() != &test) {
        std::terminate(); // oops
    }

    // Returning to empty state
    ptr3 = nullptr;

    // Receiving a new pointer
    // (this changes based on library)
    Test** out = &ptr3;

    // Bonus: Comparison.
    if (ptr == ptr2) {}
    if (ptr != ptr2) {}
    if (ptr < ptr2) {}

    // Litmus test: Accidentally bypassing the wrapper
    ptr->AddRef();
    ptr->Release();

    // Litmus test: Construction from other-type raw pointer
    Other other;
    TestPtr ptr4(&other);

    // Litmus test: Construction from other-type smart pointer
    OtherPtr optr;
    TestPtr ptr5(optr);

    // Litmus test: Assignment from other-type raw pointer
    ptr = &other;

    // Litmus test: Assignment from other-type smart pointer
    ptr = optr;

    // Destruction
}
```

The `_COM_SMARTPTR_TYPEDEF` macro that comes with the `_com_ptr_t` library requires you to specify the `IID` that the alleged interface responds to. We give it a dummy interface ID in the form of a reference to a variable that is never defined. This ensures a linker error if the code ever tries to use that fake interface ID.

The `_com_ptr_t` passes the core functionality tests: The expected `AddRef` and `Release` operations are performed.

Our test for receiving a new pointer simulates calling a function prototyped as

```
void GetTest(Test** out);
```

but instead of having to write a function that produces a `Test` object, we just put the parameter in a variable. That way, we can just step through our test in the debugger to confirm that everything behaves as desired.

To receive a pointer into the wrapper, use the `&` operator. This operator releases any previous wrapped pointer before receiving the new one.

The `_com_ptr_t` fails the bonus test. The comparison operations fail with a compiler error saying that it couldn't find a `QueryInterface` method. That's because `_com_ptr_t` uses COM identity, so it needs to `QueryInterface` for the canonical unknown before performing the comparison.

The `_com_ptr_t` fails the "accidental bypass" litmus test. There are two ways of releasing the object that are dangerously similar:

```
ptr2.Release(); // good
ptr2->Release(); // bad
```

The good way asks the wrapper to release the pointer. This nulls out the wrapped pointer in addition to calling its `Release()` method. On the other hand, the bad way asks the wrapper for the wrapped pointer, and then calls the `Release()` method directly on the wrapped pointer. This doesn't update the wrapper, so when the wrapper destructs, you get a double-release.

The fact that the member function to release the wrapped pointer is also called `Release` makes this an easy trap to fall into.

The `_com_ptr_t` passes the litmus test: Both attempts to convert from another type of smart pointer generate an error which complains that there is no `QueryInterface` method.

So let's keep a little scorecard.

`_com_ptr_t` scorecard	
Default construction	Pass
Construct from raw pointer	Pass
Copy construction	Pass
Destruction	Pass
Attach and detach	Pass
Assign to same-type raw pointer	Pass

Assign to same-type smart pointer	Pass
Fetch the wrapped pointer	`GetInterfacePtr()`
Access the wrapped object	`->`
Receive pointer via `&`	release old
Release and receive pointer	`&`
Preserve and receive pointer	N/A
Return to empty state	Pass
Comparison	Not supported
Accidental bypass	Fail
Construct from other-type raw pointer	Pass
Construct from other-type smart pointer	Pass
Assign from other-type raw pointer	Pass
Assign from other-type smart pointer	Pass

Next time, we'll look at MFC's `IPTR`.

[1] Many of these libraries are quite old and predate the fancy-pants `reinterpret_cast` keyword. The only cast operator available at the time was the C-style cast, so that's what they used.