What are the duck-typing requirements of wil com_ptr?

devblogs.microsoft.com/oldnewthing/20230512-00

May 12, 2023

Raymond Chen

We continue our survey of duck-typing requirements of various C++ COM smart pointer libraries by looking at wil's `com_ptr`, running it through our standard tests.

```
// 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 <wil/com.h>

using TestPtr = wil::com_ptr<Test>;
using OtherPtr = wil::com_ptr<Other>;

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.get() != &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;
    out = ptr3.put();
    out = ptr3.addressof();

    // 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
}
```

Once again, we encounter the same glitch as we did with ATL `CComPtr` and WRL `ComPtr`:

```
com.h(363,1): error C2440: '=': cannot convert from 'void' to 'ULONG'
```

It’s coming from this code:

```
    void attach(pointer other) WI_NOEXCEPT
    {
        auto ptr = m_ptr;
        m_ptr = other;
        if (ptr)
        {
            ULONG ref;
            ref = ptr->Release();
            WI_ASSERT_MSG(((other != ptr) || (ref > 0)), "Bug: Attaching the same
already assigned, destructed pointer");
        }
    }
```

The code peeks at the reference count of the outgoing object and confirms that we didn't attach a smart pointer to itself.

As usual, the fix is to make the `Release` method return a `ULONG` representing the new reference count.

```
struct Test
{
    void AddRef() { }
    // Dummy implementation for testing purposes only.
    ULONG Release() { return 1; }
};
```

We have to make a small tweak to the boilerplate by switching to lowercase names for `detach` and `attach`, because that's how wil spells them.

Once we fix that up, the basic tests all pass. The comparison tests compare the wrapped pointers.

There are three ways to receive a pointer in wil. You can use the `&` operator, which is a shorthand for the method call `put()`, which releases the old pointer and nulls it out, then returns the address of the pointer so a new value can be placed there. Alternatively, you can use `addressof()`, which does not release the old pointer. Use `addressof()` in the cases where the parameter is used as an in/out pointer.

wil does not use the ATL trick of "coloring" the return value of the `->` operator, so you don't have all the hassles of matching the signatures, but you also don't get protection from accidentally doing a `ptr->Release()` when you meant to do a `ptr.reset()`. Fortunately, there is no `ptr.release()` method, so the mistake is a little less likely.

The other-type litmus tests all pass. They all result in various types of compile-time errors.

Okay, so here's the scorecard for `wil::com_ptr`.

`wil::com_ptr` **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	`get()`
Access the wrapped object	`->`
Receive pointer via `&`	release old
Release and receive pointer	`put()`
Preserve and receive pointer	`addressof()`
Return to empty state	Pass
Comparison	Pass
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
Notes: `T` must have a method of the form `ULONG Release()`. The `T::Release` method must return nonzero if the object is still alive.	

Next time, we'll finish our tour of COM smart pointer classes by looking at C++/WinRT's `com_ptr`.