What are the duck-typing requirements of C++/WinRT com_ptr?

devblogs.microsoft.com/oldnewthing/20230515-00

May 15, 2023

Raymond Chen

We conclude our survey of duck-typing requirements of various C++ COM smart pointer libraries by looking at C++/WinRT'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 <winrt/base.h>

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

void test()
{
    Test test;

    // Default construction
    TestPtr ptr;

    // Construction from raw pointer
    TestPtr ptr2(&test); // (does not compile)

    // Copy construction
    TestPtr ptr3(ptr2);

    // Attaching and detaching
    auto p = ptr3.detach();
    ptr.attach(p);

    // Assignment from same-type raw pointer
    ptr3.copy_from(&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.put();

    // 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.copy_from(&other);

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

    // Destruction
}
```

C++/WinRT doesn't require that the `Release` method return a reference count, unlike ATL, WRL, and wil. So that's a relief.

As with wil, we have to make a small tweak to the boilerplate by switching to lowercase names for `detach` and `attach`, because that's how C++/WinRT spells them.

Another thing we have to fix is removing construction from raw pointers. C++/WinRT doesn't support the operation of "construct with shared ownership of a raw pointer". It does support "take ownership of a raw pointer" by passing the marker `winrt::take_ownership_of_abi` as a second parameter. However, this is not generally used because it also discards type safety.

Instead of assigning a raw pointer, C++/WinRT uses the `copy_from` method. This makes it clearer that the smart pointer is sharing ownership with the original, rather than taking ownership from it. (The `attach` method takes ownership.)

The only way to receive a pointer in C++/WinRT is to use the `put` method. This releases the old pointer and nulls it out, then returns the address of the pointer so a new value can be placed there. There is no ability to access the inner pointer for in/out use.

C++/WinRT doesn't "color" the return value of the `->` operator, so you don't get protection from signatures, but you also don't get protection from accidentally doing a `ptr->Release()` when you meant to do a `ptr = nullptr`, but the two expressions are so different-looking that you're less likely to confuse them.

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

Finally, so here's the scorecard for `winrt::com_ptr`.

`winrt::com_ptr` **scorecard**	
Default construction	Pass
Construct from raw pointer	Not supported
Copy construction	Pass
Destruction	Pass
Attach and detach	Pass
Assign to same-type raw pointer	Pass (`copy_from`)
Assign to same-type smart pointer	Pass
Fetch the wrapped pointer	`get()`
Access the wrapped object	`->`
Receive pointer via `&`	N/A
Release and receive pointer	`put()`
Preserve and receive pointer	N/A
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

Next time, we’ll capture all these results into a large comparison table and discuss what we find.