What are the duck-typing requirements of WRL ComPtr?

devblogs.microsoft.com/oldnewthing/20230511-00

May 11, 2023

Raymond Chen

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

```cpp
// 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 <wrl/client.h>

using TestPtr = Microsoft::WRL::ComPtr<Test>;
using OtherPtr = Microsoft::WRL::ComPtr<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.ReleaseAndGetAddressOf();
    out = ptr3.GetAddressOf();

    // 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
}
```

We encounter the same glitch as we did with ATL `CComPtr`, but this time it happens twice. First, it happens at construction of the WRL `ComPtr`:

```
client.h(235,1): error C2440: '=': cannot convert from 'void' to 'unsigned long'
```

Due to this code:

```
    unsigned long InternalRelease() throw()
    {
        unsigned long ref = 0;
        T* temp = ptr_;

        if (temp != nullptr)
        {
            ptr_ = nullptr;
            ref = temp->Release();
        }

        return ref;
    }
```

WRL wants to propagate the return value of `Release`, and it expects the method to return an `unsigned long`.

The other failure occurs in a familiar place: On the `Attach`.

```
client.h(22,1): error C3313: 'ref': variable cannot have the type 'void'
```

The problem is here:

```
    void Attach(_In_opt_ InterfaceType* other) throw()
    {
        if (ptr_ != nullptr)
        {
            auto ref = ptr_->Release();
            (void)ref;
            // Attaching to the same object only works if duplicate
            // references are being coalesced.  Otherwise
            // re-attaching will cause the pointer to be released and
            // may cause a crash on a subsequent dereference.
            __WRL_ASSERT__(ref != 0 || ptr_ != other);
        }

        ptr_ = other;
    }
```

It’s clear that the WRL code derived from the ATL code, seeing as this is pretty much identical to the ATL code, down to the comments.

So we have to fix our class in the same way we fixed it for ATL: 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; }
};

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

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 WRL. You can use the `&` operator, which is a shorthand for the method call `ReleaseAndGetAddressOf()`, 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 `GetAddressOf()`, which does not release the old pointer. Use `GetAddressOf()` in the cases where the parameter is used as an in/out pointer.

WRL does not use the ATL trick of “coloring” the return value of the `->` operator, so it does not have the ATL requirements that the wrapped class `T` be non-final, or that the `T::AddRef` and `T::Release` methods must have the same signature and calling convention as `IUnknown` if they are virtual.

On the other hand, the lack of “coloring” means that you can accidentally write

```
ptr2->Release();
```

instead of `ptr2.Reset()`. WRL tries to make the problem less likely to occur by using a different name (`Reset`) to try to reduce the chance of confusion.

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

Okay, so here’s the scorecard for `ComPtr`.

`ComPtr` 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	`ReleaseAndGetAddressOf()`
Preserve and receive pointer	`GetAddressOf()`
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.	

Our next smart pointer library will be `com_ptr`.