What are the duck-typing requirements of ATL CComPtr?

devblogs.microsoft.com/oldnewthing/20230510-00

May 10, 2023

Raymond Chen

We continue our survey of duck-typing requirements of various C++ COM smart pointer libraries by looking at ATL's `CComPtr`, 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 <atlbase.h>
#include <atlcom.h>

using TestPtr = CComPtr<Test>;
using OtherPtr = CComPtr<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.p != &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.p;

    // 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
}
```

A glitch in the core functionality tests happens when we call `ptr3.Attach(p)`:

```
atlcomcli.h(250,1): error C2440: 'initializing': cannot convert from 'void' to
'ULONG'
```

The problem is here:

```
    // Attach to an existing interface (does not AddRef)
    void Attach(_In_opt_ T* p2) throw()
    {
        if (p)
        {
            ULONG ref = p->Release();
            (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.
            ATLASSERT(ref != 0 || p2 != p);
        }
        p = p2;
    }
```

ATL expects the `Release` method to return a `ULONG` representing the new reference count. So let’s fix our class to do that.

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

Okay, that gets us past the `Attach`/`Detach` test.

There are two ways to receive a new pointer. You can use the `&` operator directly on the `CComPtr`, which asserts that the smart pointer is already empty. If the pointer is an in/out parameter, then you can take the address of the public `p` member directly, which avoids the assertion check. There is no combined method for “release previous pointer before receiving a new one”.

The comparison tests work as expected. They just compare the wrapped pointers.

The accidental bypass litmus test and the test for accessing the wrapped object via the `->` operator are interesting because `CComPtr` uses a techique that author Jim Springfield called “coloring”:

```
template <class T>
class _NoAddRefReleaseOnCComPtr :
    public T
{
    private:
        STDMETHOD_(ULONG, AddRef)()=0;
        STDMETHOD_(ULONG, Release)()=0;
};

template <class T>
class CComPtrBase
{
    ...

    _NoAddRefReleaseOnCComPtr<T>* operator->() const throw()
    {
        ATLASSERT(p!=NULL);
        return (_NoAddRefReleaseOnCComPtr<T>*)p;
    }

    ...
};
```

The trick here is that instead of returning the wrapped `T*` directly, we pretend that it is a pointer to the `T` portion of a `_NoAddRefReleaseOnCComPtr<T>`, and return a pointer to that derived class. The `_NoAddRefReleaseOnCComPtr<T>` class declares the `AddRef` and `Release` as private, thereby making them inaccessible from the resulting `->` operator:

```
ptr->Release(); // error: Cannot call private method¹
```

In the case where `T` derives from `IUnknown`, these virtual `AddRef` and `Release` methods override the same-signature methods in `IUnknown`. But in the case where `T` does not derive from `IUnknown`, this adds a vtable to `_NoAddRefReleaseOnCComPtr`. Now, this vtable is never materialized, but it nevertheless introduces some pointer arithmetic that the compiler cannot immediately optimize away, because a `static_cast` is not always just a pointer adjustment.

```
; ideally
CComPtr<Test>::operator->()
    lea     rax, [rcx-8]
    ret

; actually
CComPtr<Test>::operator->()
    lea     rdx, [rcx-8]
    neg     rcx
    sbb     rax, rax
    and     rax, rdx
    ret
```

The extra nonsense is there so that the cast from `(T*)` to `(_NoAddRefReleaseOnCComPtr<T>*)` produces `nullptr` when passed `nullptr`. (The "ideal" version would return `-8`.)

I call this a missed optimization because when the compiler inlines the `->` operator, it can see that the resulting pointer is immediately dereferenced, so it cannot be null. Furthermore, the `+8` that comes afterward to convert the `(_NoAddRefReleaseOnCComPtr<T>*)` back to a `(T*)` exactly cancels out the `-8`, so all the pointer nonsense can be optimized out entirely. Bonusly furthermore, `this` can never be `nullptr`; invoking a method on a null pointer is undefined behavior.

```
    ; ptr2->AddressOf()
    ; ideally

    mov     rcx, [ptr2].p
    call    Test::AddressOf

    ; actually
    mov     rcx, [ptr2].p
    mov     eax, 8
    test    rcx, rcx
    cmove   rcx, rax
    call    Test::AddressOf
```

All of these problems could have been avoided if `_NoAddRefReleaseOnCComPtr` had declared the private `AddRef` and `Release` as non-virtual.

```
template <class T>
class _NoAddRefReleaseOnCComPtr :
    public T
{
    private:
        ULONG AddRef();
        ULONG Release();
};
```

This still accomplishes the task of making the `AddRef` and `Release` methods inaccessible, but it doesn't introduce a vtable, which means that the `static_cast` operations do not result in any code generation.

But it's too late to fix that now. That would be a binary breaking change.

Other consequences of "coloring" are that the wrapped class `T` cannot be `final`, and if the `T::AddRef` and `T::Release` methods are virtual, they must return `ULONG` and use `STDMETHODCALLTYPE`.

The `CComPtr` passes the other litmus tests: Most of the operations generate an error complaining that the types do not match. The interesting one is the last one, the assignment of an other-type smart pointer.

```
template <typename Q>
T* operator=(_Inout_ const CComPtr<Q>& lp) throw()
{
    if(!this->IsEqualObject(lp) )
    {
        AtlComQIPtrAssign2((IUnknown**)&this->p, lp, __uuidof(T));
    }
    return *this;
}
```

This one fails because the code uses `CComPtrBase::IsEqualObject`, which in turn does not compile due to lack of `QueryInterface` support. Which is a good thing, because there is a C-style cast to `IUnknown**` in the call to `AtlComQIPtrAssign2`, which requires that our underlying type `T` derive from `IUnknown`.

Okay, so here's the scorecard for `CComPtr`.

`CComPtr` 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	`p`, or implicit conversion
Access the wrapped object	`->` (suboptimal)
Receive pointer via `&`	must be empty
Release and receive pointer	`&`
Preserve and receive pointer	`&p`
Return to empty state	Pass

Comparison	Pass
Accidental bypass	Pass
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` may not be `final`. `T` must have a method of the form `ULONG Release()`. The `T::Release` method must return nonzero if the object is still alive.	

Netx time, we'll look at WRL's `ComPtr`.

[1] Though you can hack around this by forcing a call to the base class version, or just using the raw wrapped pointer.

```
// sneaky! ducking under the ribbon!
ptr->Test::Release();
ptr.p->Release();
```

The purpose of the "coloring" is not be be bulletproof. It's just to prevent you from calling `AddRef` and `Release` by mistake.