What is the underlying object behind a COM interface pointer?

devblogs.microsoft.com/oldnewthing/20070424-00

April 24, 2007

Raymond Chen

When you're debugging, you might have a pointer to a COM interface and want to know what the underlying object is. Now, sometimes this trick won't work because the interface pointer actually points to a stub or proxy, but in the case where no marshalling is involved, it works great. (This technique also works for many C++ compilers for any object that has virtual methods and therefore a vtable.)

Recall that the layout of a COM object requires that the pointer to a COM interface point to the object's vtable, and it's the vtable that is the key.

```
0:000> dv
            pstm = 0x000c7568
0:000> dt psf
Local var @ 0x7cc2c Type IStream*
0x000c7568
   +0x000 __VFN_table : 0x1c9c8e84
```

Okay, so far all we know is that our `IStream *` lives at `0x000c7568` and its vtable is `0x1c9c8e84`. Whose stream implementation is it?

```
0:000> ln 0x1c9c8e84
(1c9c8e84)   ABC!CAlphaStream::`vftable'
```

Aha, it's a `CAlphaStream` from `ABC.DLL`. Let's take a look at it:

```
0:000> dt ABC!CAlphaStream 0x000c7568
   +0x000 __VFN_table : 0x1c9c8e84 // our vtable
   +0x004 m_cRef            : 480022128
   +0x008 lpVtbl            : 0x1c9d2d30
   +0x00c lpVtbl            : 0x00000014
   +0x010 m_pszName         : 0x000c7844 "??????????"
   +0x014 m_dwFlags         : 0x3b8
   +0x018 m_pBuffer         : 0x00000005
   +0x01c m_cbBuffer        : 705235565
   +0x020 m_cbPos           : 2031674
```

“Hey, how did you get the debugger to dump `m_pszName` as a string?” If you issue the `.enable_unicode 1` command, then the debugger will treat pointers to `unsigned short` as if they were pointers to Unicode strings. (By default, only pointers to `wchar_t` are treated as pointers to Unicode strings.)

Okay, back to the structure dump. It doesn’t look right at all. The reference count is some absurd value, the vtable at offset `0x00c` is a bogus pointer, the name in `m_pszName` is garbage, pretty much every field aside from the initial vtable and the vtable at offset `0x008` is blatantly wrong.

What happened? Well, clearly we were given a “ `q` ” pointer; i.e., a pointer to one of the vtables other than the first one. We have to adjust the pointer so it points to the start of the object instead of the middle.

How do we do this adjustment? There’s the methodical way and the quick-and-dirty way.

The methodical way is to use the adjustor thunks to tell you how much the pointer needs to be adjusted in order to move from a secondary vtable to the primary one. (This assumes that the primary `IUnknown` implementation is the first base class. This is not guaranteed to be the case but it usually is.)

```
0:000> dps 1c9c8e84 l1
1c9c8e84  1c9eb08e ABC![thunk]:CAlphaStream::QueryInterface`adjustor{8}'
```

Aha, this adjustors adjust by eight bytes, so we just need to subtract eight from our pointer to get the object’s starting address.

```
0:000> dt ABC!CAlphaStream 0x000c7560-8
   +0x000 __VFN_table : 0x1c9c8ee8
   +0x004 m_cRef           : 2
   +0x008 lpVtbl           : 0x1c9c8e84
   +0x00c lpVtbl           : 0x1c9c8e70
   +0x010 m_pszName        : 0x1c9d2d30 "Scramble"
   +0x014 m_dwFlags        : 0x14
   +0x018 m_pBuffer        : 0x000c7844
   +0x01c m_cbBuffer       : 952
   +0x020 m_cbPos          : 5
```

Ah, that looks much nicer. Notice that the reference count is a more reasonable value of two, the name pointer looks good, the buffer size and position appear to be much more realistic.

Now, I don’t bother with the whole adjustor thunk thing. Instead I rely on the principle of “Assume it’s mostly correct“: Assume that the object is not corrupted and just adjust the pointer by eye until the fields line up. Let’s take another look at the original (bad) dump:

```
0:000> dt ABC!CAlphaStream 0x000c7568
   +0x000 __VFN_table : 0x1c9c8e84
   +0x004 m_cRef           : 480022128
   +0x008 lpVtbl           : 0x1c9d2d30
   +0x00c lpVtbl           : 0x00000014
   +0x010 m_pszName        : 0x000c7844 "??????????"
   +0x014 m_dwFlags        : 0x3b8
   +0x018 m_pBuffer        : 0x00000005
   +0x01c m_cbBuffer       : 705235565
   +0x020 m_cbPos          : 2031674
```

This obviously doesn't smell right, but what do we have to do to get things to line up? Well, we know that the vtable we have must go into one of the other two vtable slots, either the one at offset `0x008` or the one at offset `0x00c`. If we moved it to offset `0x00c`, then that would move the `0x00000014` currently at offset `0x00c` down twelve bytes, placing it at offset `0x018`, right at `m_pBuffer`. But obviously `0x00000014` is not a valid buffer pointer, so `0x00c` can't be the correct adjustment. On the other hand, if we put our vtable at offset `0x008`, then that would move `0x000c7844` into the `m_pBuffer` position, which is not too unreasonable. Therefore, I would guess that the adjustor is eight, yielding the same structure dump that we got by dumping the vtable to see the adjustor.

In real life, I tend to pay attention to the vtables, the reference count, and any string members because it's usually pretty easy to see whether you got them right. (Vtables reside in code. Reference counts tend to be small integers. Strings are, well, strings.)

Raymond Chen

Follow