Reference counting is hard.

devblogs.microsoft.com/oldnewthing/20040406-00

April 6, 2004

Raymond Chen

One of the big advantages of managed code is that you don't have to worry about managing object lifetimes. Here's an example of some unmanaged code that tries to manage reference counts and doesn't quite get it right. Even a seemingly-simple function has a reference-counting bug.

```
template <class T>
T *SetObject(T **ppt, T *ptNew)
{
 if (*ppt) (*ppt)->Release(); // Out with the old
 *ppt = ptNew; // In with the new
 if (ptNew) (ptNew)->AddRef();
 return ptNew;
}
```

The point of this function is to take a (pointer to) a variable that points to one object and replace it with a pointer to another object. This is a function that sits at the bottom of many "smart pointer" classes. Here's an example use:

```
template <class T>
class SmartPointer {
public:
 SmartPointer(T* p = NULL)
   : m_p(NULL) { *this = p; }
 ~SmartPointer() { *this = NULL; }
 T* operator=(T* p)
   { return SetObject(&m_p, p); }
 operator T*() { return m_p; }
 T** operator&() { return &m_p; }
private:
 T* m_p;
};
void Sample(IStream *pstm)
{
  SmartPointer<IStream> spstm(pstm);
  SmartPointer<IStream> spstmT;
  if (SUCCEEDED(GetBetterStream(&spstmT))) {
   spstm = spstmT;
  }
  ...
}
```

Oh why am I explaining this? You know how smart pointers work.

Okay, so the question is, what's the bug here?

Stop reading here and don't read ahead until you've figured it out or you're stumped or you're just too lazy to think about it.

The bug is that the old object is Release()d before the new object is AddRef()'d. Consider:

```
  SmartPointer<IStream> spstm;
  CreateStream(&spstm);
  spstm = spstm;
```

This assignment statement looks harmless (albeit wasteful). But is it?

The "smart pointer" is constructed with NULL, then the CreateStream creates a stream and assigns it to the "smart pointer". The stream's reference count is now one. Now the assignment statement is executed, which turns into

```
 SetObject(&spstm.m_p, spstm.m_p);
```

Inside the SetObject function, ppt points tp spstm.m_p, and pptNew equals the original value of spstm.m_p.

The first thing that SetObject does is release the old pointer, which now drops the reference count of the stream to zero. **This destroys the stream object**. Then the ptNew parameter (which now points to a freed object) is assigned to spstm.m_p, and finally the ptNew pointer (which still points to a freed object) is AddRef()d. Oops, we're invoking a method on an object that has been freed; no good can come of that.

If you're lucky, the AddRef() call crashes brilliantly so you can debug the crash and see your error. If you're not lucky (and you're usually not lucky), the AddRef() call interprets the freed memory as if it were still valid and increments a reference count somewhere inside that block of memory. Congratulations, you've now corrupted memory. If that's not enough to induce a crash (at some unspecified point in the future), when the "smart pointer" goes out of scope or otherwise changes its referent, the invalid m_p pointer will be Release()d, corrupting memory yet another time.

This is why "smart pointer" assignment functions must AddRef() the incoming pointer before Release()ing the old pointer.

```
template <class T>
T *SetObject(T **ppt, T *ptNew)
{
 if (ptNew) (ptNew)->AddRef();
 if (*ppt) (*ppt)->Release();
 *ppt = ptNew;
 return ptNew;
}
```

If you look at the source code for the ATL function AtlComPtrAssign, you can see that it exactly matches the above (corrected) function.

[Raymond is currently on vacation; this message was pre-recorded.]

Raymond Chen

Follow