On objects with a reference count of zero

devblogs.microsoft.com/oldnewthing/20050929-10

September 29, 2005

Raymond Chen

One commenter claimed that

> When the object is first constructed, the reference count should be 0 and AddRef should be called at some point (probably via QueryInterface) to increment the reference count.

If you construct your object with a reference count of zero, you are playing with matches. For starters, when the object is created, there reference count is not zero – the person who created the object has a reference! Remember the COM rule for references: If a function produces a reference (typically an interface pointer), the reference count is incremented to account for the produced reference. If you consider the constructor to be a function, then it needs to return with an incremented reference count to account for the produced object.

If you prefer to play with matches, you can end up burning yourself with code like the following:

```
// A static creator method
HRESULT MyObject::Create(REFIID riid, void **ppvObj)
{
 *ppvObj = NULL;
 MyObject *pobj = new MyObject();
 HRESULT hr = pobj ? S_OK : E_OUTOFMEMORY;
 if (SUCCEEDED(hr)) {
  hr = pobj->Initialize(); // dangerous!
  if (SUCCEEDED(hr)) {
   hr = pobj->QueryInterface(riid, ppvObj);
  }
  if (FAILED(hr)) {
   delete pobj;
  }
 }
 return hr;
}
```

Notice that you're initializing the object while its reference count is zero. This puts you in the same "limbo zone" as cleaning up an object while its reference count is zero, and therefore exposes you to the same problems:

```
HRESULT MyObject::Load()
{
 CComPtr<IStream> spstm;
 HRESULT hr = GetLoadStream(&spstm);
 if (SUCCEEDED(hr)) {
  CComQIPtr<IObjectWithSite, &IID_IObjectWithSite> spows(spstm);
  if (spows) spows->SetSite(this);
  hr = LoadFromStream(spstm);
  if (spows) spows->SetSite(NULL);
 }
 return hr;
}


HRESULT MyObject::Initialize()
{
 return Load();
}
```

An object that saves itself during destruction is very likely to load itself during creation. And you run into exactly the same problem. The call to `IObjectWithSite::SetSite(this)` increments the reference count of the object from zero to one, and the call to The call to `IObjectWithSite::SetSite(NULL)` decrements it back to zero. When the reference count decrements to zero, this destroys the object, resulting in the object being inadvertently destroyed by the `MyObject::Load()` method.

The `MyObject::Create` static method doesn't realize that this has happened and proceeds to call the `QueryInterface` method to return a pointer back to the caller, expecting it to increment the reference count from zero to one. Unfortunately, it's doing this to an object that has already been destroyed.

That's what happens when you play with an object whose reference count is zero: It can disappear the moment you relinquish control. Objects should be created with a reference count of one, not zero.

ATL prefers to play with matches, using the moral equivalent of the above `MyObject::Create` function in its object construction:

```
void InternalFinalConstructAddRef() {}
void InternalFinalConstructRelease()
{
    ATLASSERT(m_dwRef == 0);
}


static HRESULT WINAPI CreateInstance(void* pv, REFIID riid, LPVOID* ppv)
{
    ATLASSERT(*ppv == NULL);
    HRESULT hRes = E_OUTOFMEMORY;
    T1* p = NULL;
    ATLTRY(p = new T1(pv))
    if (p != NULL)
    {
        p->SetVoid(pv);
        p->InternalFinalConstructAddRef();
        hRes = p->FinalConstruct();
        p->InternalFinalConstructRelease();
        if (hRes == S_OK)
            hRes = p->QueryInterface(riid, ppv);
        if (hRes != S_OK)
            delete p;
    }
    return hRes;
}
```

ATL hands you a set of matches by calling your `FinalConstruct` method with a reference count of zero. If you know that you're going to get burned, you can use the `DECLARE_PROTECT_FINAL_CONSTRUCT` macro to change the `InternalFinalConstructAddRef` and `InternalFinalConstructRelease` methods to versions that actually increment the reference count temporarily during the call to `FinalConstruct`, then drop the reference count back to zero (without destructing the object) prior to the `QueryInterface` call.

It works, but in my opinion it relies too much on programmer vigilance. The default for ATL is to hand programmers matches and relying on programmers "knowing" that something dangerous might happen inside the `FinalConstruct` and having the presence of mind to ask for `DECLARE_PROTECT_FINAL_CONSTRUCT`. In other words, it chooses the dangerous default, and programmers must explicitly ask for the safe version. But programmers have a lot of things on their mind, and forcing them to consider the consequences of the transitive closure of every operation performed in the `FinalConstruct` method is an unresonable requirement.

Consider our example above. When the code was originally written, the `Load` method may have been the much simpler

```
HRESULT MyObject::Load()
{
 CComPtr<IStream> spstm;
 HRESULT hr = GetLoadStream(&spstm);
 if (SUCCEEDED(hr)) {
  hr = LoadFromStream(spstm);
 }
 return hr;
}
```

It wasn't until a month or two later that somebody added site support to the `Load` and `Save` methods. This seemingly simple and isolated change, adhering perfectly to the COM rules for reference counting, had ripple effects back through the object creation and destruction code paths. If you put four levels of function calls between the `FinalConstruct` and the `Load`, this fourth-level-caller effect can very easily be overlooked. I suspect that these nonlocal effects are one of the most significant sources of code defects. ATL was being clever and optimized out an increment and a decrement (something which the compiler most likely could optimize out on its own), but in return, you got handed a book of matches.

(I don't mean to be picking on ATL here, so don't go linking to this article with the title "Raymond rails into ATL as a poorly-designed pile of dung". ATL is trying to be small and fast, but the cost is added complexity, often subtle.)

Raymond Chen

Follow