COM object destructors are very sensitive functions

devblogs.microsoft.com/oldnewthing/20050927-13

September 27, 2005

Raymond Chen

If you try to do too much, you can find yourself in trouble.

For example, if your destructor hands a reference to itself to other functions, those functions might decide to call your `IUnknown::AddRef` and `IUnknown::Release` methods as part of their internal operations. Consider:

```
ULONG MyObject::Release()
{
 LONG cRef = InterlockedDecrement(&m_cRef);
 if (cRef == 0) {
  delete this;
 }
 return cRef;
}


MyObject::~MyObject()
{
 if (m_fNeedSave) Save();
}
```

That doesn't look so scary now does it? The object saves itself when destructed.

However, the `Save` method might do something like this:

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

On its own, this looks pretty normal. Get a stream and save to it, setting ourselves as its site in case the stream wants to get additional information about the object as part of the saving process.

But in conjunction with the fact that we call it from our destructor, we have a recipe for disaster. Watch what happens when the last reference is released.

- The `Release()` method decrements the reference count to zero and performs a `delete this`.
- The destructor attempts to save the object.
- The `Save()` method obtains the save stream and sets itself as the site. This increments the reference count from zero to one.
- The `SaveToStream()` method saves the object.
- The `Save()` method clears the site on the stream. This decrements the reference count from one back to zero.
- The `Release()` method therefore attempts to destructor the object a second time.

Destructing the object a second time tends to result in widespread mayhem. If you're lucky, you'll crash inside the recursive destruction and identify the source, but if you're not lucky, the resulting heap corruption won't go detected for quite some time, at which point you'll just be left scratching your head.

Therefore, at a minimum, you should assert in your `AddRef()` method that you aren't incrementing the reference count from zero.

```
ULONG MyObject::AddRef()
{
 assert(m_cRef != 0);
 return InterlockedIncrement(&m_cRef);
}
```

This would catch the "case of the mysteriously double-destructed object" much earlier in the game, giving you a fighting chance of identifying the problem. But once you've isolated the problem, what can you do about it? We'll look into that next time.

Raymond Chen

Follow