Do you know when your destructors run? Part 1.

devblogs.microsoft.com/oldnewthing/20040520-00

May 20, 2004

Raymond Chen

Larry Osterman discussed the importance of knowing when your global destructors run, but this problem is not exclusive to global objects. You need to take care even with local objects. Consider:

```
void Sample()
{
  if (SUCCEEDED(CoInitialize(NULL))) {
    CComPtr<IXMLDOMDocument> p;
    if (SUCCEEDED(p.CoCreateInstance(CLSID_IXMLDOMDocument))) {
     ...
    }
    CoUninitialize();
  }
}
```

Easy as pie. And there's a bug here.

When does the destructor for that smart-pointer run?

Answer: When the object goes out of scope, which is at the closing brace of the outer `if` statement, **after** the `CoUninitialize` call.

So you shut down COM, and then try to access a pointer to a COM object. This is not good. (Or as Larry describes it, "Blam!")

To fix this problem, you have to release all your COM pointers before the `CoUninitialize`. One way would be to insert a `p.Release()` at the end of the inner `if`. (But of course, if you're going to do that, then why bother using a smart pointer?)

Another fix would be to introduce a seemingly unnecessary scope:

```
void Sample()
{
  if (SUCCEEDED(CoInitialize(NULL))) {
    {
      CComPtr<IXMLDOMDocument> p;
      if (SUCCEEDED(p.CoCreateInstance(CLSID_IXMLDOMDocument))) {
       ...
      }
    } // ensure p is destructed before the CoUninit
    CoUninitialize();
  }
}
```

Make sure you leave that comment there or the next person to come across this code is going to "clean it up" by removing the "redundant" braces.

Of course, this is still too subtle. Here's another solution: Put the CoUninitialize inside a destructor of its own!

```
class CCoInitialize {
public:
 CCoInitialize() : m_hr(CoInitialize(NULL)) { }
 ~CCoInitialize() { if (SUCCEEDED(m_hr)) CoUninitialize(); }
 operator HRESULT() const { return m_hr; }
 HRESULT m_hr;
};
void Sample()
{
  CCoInitialize init;
  if (SUCCEEDED(init)) {
    CComPtr<IXMLDOMDocument> p;
    if (SUCCEEDED(p.CoCreateInstance(CLSID_IXMLDOMDocument))) {
     ...
    }
  }
} // CoUninitialize happens here
```

This works even if you put the smart pointer at the same scope, as long as you put it **after the CCoInitialize object**:

```
void Sample()
{
  CCoInitialize init;
  CComPtr<IXMLDOMDocument> p;
  if (SUCCEEDED(init) &&
      SUCCEEDED(p.CoCreateInstance(CLSID_IXMLDOMDocument))) {
   ...
  }
}
```

This works because objects with automatic storage duration are destructed in reverse order of declaration, so the object `p` wil be destructed first, then the object `init`.

Mind you, this is basically subtle no matter now you slice it. Nobody said programming was easy.

Tomorrow, part 2.

Raymond Chen

Follow