C# static constructors are called on demand, not at startup

devblogs.microsoft.com/oldnewthing/20070815-00

August 15, 2007

Raymond Chen

One of the differences between C++ and C# is when static constructors run. In C++, static constructors are the first thing run in a module, even before the `DllMain` function runs.[1] In C#, however, static constructors don't run until you use the class for the first time. If your static constructor has side effects, you may find yourself experiencing those side effects in strange ways.

Consider the following program. It's rather contrived and artificial, but it's based on an actual program that encountered the same problem.

```
using System;
using System.Runtime.InteropServices;
class Program {
 [DllImport("kernel32.dll", SetLastError=true)]
 public static extern bool SetEvent(IntPtr hEvent);
 public static void Main()
 {
  if (!SetEvent(IntPtr.Zero)) {
   System.Console.WriteLine("Error: {0}", Trace.GetLastErrorFriendlyName());
  }
 }
}
```

This program tries to set an invalid event, so the call to `SetEvent` is expected to fail with an invalid handle error. We print the last error code using a function in this helper class: The details of this method aren't important. In fact, for illustrative purposes, I'm going to skip the call to `FormatMessage` and just return an ugly name.[2]

```
class Trace {
 public static string GetLastErrorFriendlyName()
 {
  return Marshal.GetLastWin32Error().ToString();
 }
}
```

Run this program, and you should get this output:

```
Error: 6
```

Six is the expected error code, since that is the numeric value of `ERROR_INVALID_HANDLE`.

You don't think much of this program until one day you run it and instead of getting error 6, you get something like this:

```
Error: 126
```

What happened?

While you weren't paying attention, somebody decided to do some enhancements to the `Trace` class, maybe added some new methods and stuff, and in particular, a static constructor got added:

```
class Trace {
 public static string GetLastErrorFriendlyName()
 {
  return Marshal.GetLastWin32Error().ToString();
 }
 [DllImport("kernel32.dll", SetLastError=true, CharSet=CharSet.Auto)]
 public static extern IntPtr LoadLibrary(string dll);
 static Trace() { LoadLibrary("enhanced_logging.dll"); }
}
```

It's not important what the static constructor does; the point is that we have a static constructor now. In this case, the static constructor tries to load a helper DLL which presumably does something fancy so we can get better trace logging, something like that, the details aren't important.

The important thing is that the constructor has a side effect. Since it uses a p/invoke, the value of `Marshal.GetLastWin32Error()` is overwritten by the error code returned by the `LoadLibrary`, which in our case is error 126, `ERROR_MOD_NOT_FOUND`.

Now let's look at what happens in our program.

First, we call `SetEvent`, which fails and sets the Win32 error code to 6. Next, we call `Trace.GetLastErrorFriendlyName`, but wait! This is the first call to a method in the `Trace` class, so we have to run the static constructor first.

The static constructor tries to load the `enhanced_logging.dll` module, and it fails, setting the last error code to 126. This *overwrites the previous value*.

After the static constructor returns, we return to our program already in progress and call `Trace.GetLastErrorFriendlyName`, but it's too late. The damage has been done. The last error code has been corrupted.

And that's why we get 126 instead of 6.

What's really scary is that problems with static constructors running at inopportune times are often extremely hard to identify. For one thing, there is no explicit indication in the source code that there's any static constructor funny business going on. Indeed, somebody could just recompile the assembly containing the `Trace` class without modifying your program, and the problem will rear its head. "But I didn't change anything. The timestamp on `program.exe` is the same as the one that still works!"

A side effect you might not consider is synchronization. If the static constructor takes any locks, you have to keep an eye on your lock hierarchy, or one of those locks might trigger a deadlock. This is insidious, because you can stare at the code all you want; you won't see anything. You'll have a method like

```
class Trace {
 ...
 public static string GetFavoriteColor() { return "blue"; }
}
```

and yet when you try to step over a call to `Trace.GetFavoriteColor`, your program hangs! "This makes no sense. How can `Trace.GetFavoriteColor` hang? It just returns a constant!"

Another factor that makes this problem baffling is that the problem occurs only the first time you call a method in the `Trace` class. We saw it here only because the very first thing we did with `Trace` was display an error. If you happened to call, say, `Trace.GetFavoriteColor()` before calling `Trace.GetLastErrorFriendlyName()`, then you wouldn't have seen this problem. In fact, that's how the program that inspired today's entry stumbled across this problem. They deleted a call into the `Trace` class from some unrelated part of the program, which meant that the static constructor ran at a different time than it used to, and unfortunately, the new time was less hospitable to static construction.

"I'm sorry, did I call you at a bad time?"

Footnotes[3]

[1]This is not strictly true. In reality, it's a bit of sleight-of-hand performed by the C runtime library.[4]

[2]For a less ugly name, you can use this class instead:

```
class Trace {
 [DllImport("kernel32.dll", SetLastError=true)]
 public static extern IntPtr LocalFree(IntPtr hlocal);
 [DllImport("kernel32.dll", SetLastError=true, CharSet=CharSet.Auto)]
 public static extern int FormatMessage(int flags, IntPtr unused1,
    int error, int unused2, ref IntPtr result, int size, IntPtr unused3);
 static int FORMAT_MESSAGE_ALLOCATE_BUFFER = 0x00000100;
 static int FORMAT_MESSAGE_IGNORE_INSERTS  = 0x00000200;
 static int FORMAT_MESSAGE_FROM_SYSTEM     = 0x00001000;
 public static string GetLastErrorFriendlyName()
 {
  string result = null;
  IntPtr str = IntPtr.Zero;
  if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                    FORMAT_MESSAGE_IGNORE_INSERTS  |
                    FORMAT_MESSAGE_FROM_SYSTEM, IntPtr.Zero,
                    Marshal.GetLastWin32Error(), 0,
                    ref str, 0, IntPtr.Zero) > 0) {
   try {
    result = Marshal.PtrToStringAuto(str);
   } finally {
    LocalFree(str);
   }
  }
  return result;
 }
}
```

Note that there may be better ways of accomplishing this. I'm not the expert here.

[3]Boring footnote symbols from now on. You guys sure know how to take the fun out of blogging. (I didn't realize that blogs were held to academic writing standards. Silly me.) Now you can go spend your time telling Scoble that he wrote a run-on sentence or something.

[4]Although this statement is written as if it were a fact, it is actually my interpretation of how the C runtime works and is not an official position of the Visual Studio team nor Microsoft Corporation, and that interpretation may ultimately prove incorrect. Similar remarks apply to other statements of fact in this article.

Postscript: Before you start pointing fingers and saying, "Hah hah, we don't have this problem in Win32!"—it turns out that you do! As we noted in the introduction, static constructors run when the DLL is loaded. The granularity in Win32 is not as fine, being at the module level rather than the class level, but the problem is still there. If you use delay-loading, then the first call to a function in a delay-loaded DLL will load the target DLL, and its static constructors will run, possibly when your program wasn't expecting it.

Raymond Chen

Follow