Issues related to forcing a stub to be created for an imported function

devblogs.microsoft.com/oldnewthing/20060725-00

July 25, 2006

Raymond Chen

I noted last time that you can concoct situations that force the creation of a stub for an imported function. For example, if you declare a global function pointer variable:

```
DWORD (WINAPI *g_pGetVersion)() = GetVersion;
```

then the C compiler is forced to generate the stub and assign the address of the stub to the `g_pGetVersion` variable. That's the best it can do, since the loader will patch up only the imported function address table; it won't patch up anything else in the data segment.

The C++ compiler, on the other hand, can take advantage of some C++ magic and secretly generate a "pseudo global constructor" (I just made up that term so don't go around using it like it's official or something) that copies the value from the imported function address table to the `g_pGetVersion` variable at runtime. Note, however, that since this is happening at runtime, mixed in with all the other global constructors, then the variable might not be set properly if you call it from any code that runs during construction of global objects. Consider the following buggy program made up of two files.

```
// file1.cpp
#include <windows.h>
EXTERN_C DWORD (WINAPI *g_pGetVersion)();
class Oops {
  public: Oops() { g_pGetVersion(); }
} g_oops;
int __cdecl main(int argc, char **argv)
{
  return 0;
}
// file2.cpp
#include <windows.h>
EXTERN_C DWORD (WINAPI *g_pGetVersion)() = GetVersion;
```

The rules for C++ construction of global objects is that global objects within a single translation unit are constructed in the order they are declared (and destructed in reverse order), but there is no enforced order for global objects from separate translation units. But notice that there is an order-of-construction dependency here. The construction of the `g_oops` object requires that the `g_pGetVersion` object be fully constructed, because it's going to call through the pointer when the `Oops` constructor runs.

It so happens that the Microsoft linker constructs global objects in the order in which the corresponding OBJ files are listed in the linker's command line. (I don't know whether this is guaranteed behavior or merely an implementation detail, so I wouldn't rely on it.) Consequently, if you tell the linker to link `file1.obj + file2.obj`, you will crash because the linker will generate a call to the `Oops::Oops()` constructor before it gets around to constructing `g_pGetVersion`. On the other hand, if you list them in the order `file2.obj + file1.obj`, you will run fine.

Even stranger: If you rename `file2.cpp` to `file2.c`, then the program will run fine regardless of what order you give the OBJ files to the linker, because the C compiler will use the stub instead of trying to copy the imported function address at runtime.

But what happens if you mess up and declare a function as `dllimport` when it isn't, or vice versa? We'll look at that next time.

Raymond Chen

Follow