Understanding the classical model for linking: Taking symbols along for the ride

devblogs.microsoft.com/oldnewthing/20130108-00

January 8, 2013

Raymond Chen

Last time, we learned the basics of the classical model for linking. Today, we'll look at the historical background for that model, and how the model is exploited by libraries.

In the classical model, compilers and assemblers consume source code and spit out an OBJ file. They do as much as they can, but eventually they get stuck because they don't have the entire module at their disposal. To record the work remaining to be done, the OBJ file contains various sections: a data section, a code section (historically and confusingly called *text*), an uninitialized data section, and so on. The linker resolves symbols, and then for each OBJ file that got pulled into the module, it combines all the code sections into one giant code section, all the data sections into one giant data section, and so on.

One thing you may have noticed is that the unit of consumption is the OBJ file. If an OBJ file is added to the module, the whole thing gets added, even if you needed only a tiny part of the OBJ file. Historically, the reason for this rule is that the compilers and assemblers did not include information in the OBJ file to indicate how to separate all the little pieces. It's like if somebody said, "Can you get me a portable mp3 player?" and the only thing available in the library was a smartphone. Sure, it plays mp3 files, but there's a lot of other electronic junk in there that you *didn't* ask for, but it came along for the ride. And you don't know how to disassemble the smartphone and extract just the mp3-player part.

This behavior is actually exploited as a *feature*, because it allows for tricks like this:

```
/* magicnumber.h */
extern int magicNumber;


/* magicnumber.c */
int magicNumber;


class InitMagicNumber
{
 InitMagicNumber()
 {
    magicNumber = …;
 }
}
g_InitMagicNumber;
```

I'm not going to go into the magic of how the compiler knows to construct the `g_InitMagicNumber` object at module entry; I'll let you read up on that.

The point is that if anybody in the module refers to the `magicNumber` variable, then that causes `magicnumber.obj` to be pulled into the module, which brings in not just the `magicNumber` variable, but also the `g_InitMagicNumber` object, which initializes the magic number when the process starts.

One place the C runtime library took advantage of this was in deciding whether or not to include floating point support.

As you may recall, the 8086 processor did not have native floating support. You had to buy the 8087 coprocessor for that. It was therefore customary for programs of that era to include a floating point library if they did any floating point arithmetic. The library would redirect floating point operations from the coprocessor to the emulator.

The floating point emulation library was pretty hefty, and it would have been a waste to include it for programs that didn't use floating point (which was most of them), so the compiler used a trick to allow it to pull in the floating point library only if the program used floating point: If you used floating point, then the compiler added a *needed* symbol to your OBJ file: `__fltused`.

That magical `__fltused` symbol was marked as *provided* by… the floating point emulation library!

The linker found the symbol in an OBJ in the floating point emulation library, and that served as the loose thread that caused the rest of the floating point emulation library to be pulled into your module.

Next time, we'll look at the interaction between OBJ files and LIB files.

Bonus reading: Larry Osterman gives another example of this trick.

Raymond Chen

Follow