# The classical model for linking

devblogs.microsoft.com/oldnewthing/20091012-00

October 12, 2009

Raymond Chen

Commenter Adam wonders why we need import libraries anyway. Why can't all the type information be encoded in the export table?

This goes back to the classical model for linking. This model existed for decades before Microsoft was even founded, so at least this time you don't have Bill Gates to kick around. (Though I'm sure you'll find a way anyway.)

Back in the days when computer programs fit into a single source file, there was only one step in producing an executable file: You compile it. The compiler takes the source code, parses it according to the rules of the applicable language, generates machine code, and writes it to a file, ready to be executed.

This model had quite a following, in large part because it was ridiculously fast if you had a so-called *one-pass* compiler.

When programs got complicated enough that you couldn't fit them all into a single source file, the job was split into two parts. The compiler still did all the heavy lifting: It parsed the source code and generated machine code, but if the source code referenced a symbol that was defined in another source file, the compiler doesn't know what memory address to generate for that reference. The compiler instead generated a placeholder address and included some metadata that said, "Hey, there is a placeholder address at offset XYZ for symbol ABC." And for each symbol in the file, it also generated some metadata that said, "Hey, in case anybody asks, I have a symbol called BCD at offset WXY." These "99%-compiled" files were called *object modules*.

The job of the linker was to finish that last 1%. It took all the object module and glued the dangling bits together. If one object module said "I have a placeholder for symbol ABC," it went and looked for any other object module that said "I have a symbol called ABC," and it filled in the placeholder with the information about ABC, known as *resolving the external reference*.

When all the placeholders got filled in, the linker could then write out the finished executable module. And if there were any placeholders left over, you got the dreaded *unresolved external* error.

Notice that the only information about symbols that is provided in the object module is the symbol name. Older languages trusted the programmer to get everything else right. If your FORTRAN program defined a common block with two integers and a real, and you referenced it from another source file, it was simply a language requirement that when you access the common block, you must treat it as having two integers and a real. The compiler was not under any obligation to verify that your uses of the common block were consistent. Similar, if your C program took a function returning *long* and redeclared it as a function returning *int*, the compiler merely agreed to your little subterfuge, and you were on the hook for the consequences.

Given the classical model for linking, that's pretty much all the language specification could do. All that was shared between object modules was symbol names. And back in the old days, symbol names were restricted to a maximum of eight characters consisting of uppercase letters or digits.

The C++ language came up with a workaround: They encoded the type information in the symbol name, a technique known as *decoration*. Your function which is named `Resolve` in the source code ends up with the name `?Resolve@@YG_NPAGI_N@Z` in the object module, so that it can be matched up against the placeholders which ask for a function named `?Resolve@@YG_NPAGI_N@Z`. The C++ language folks could get away with this because by the time the C++ language rolled around, the maximum length for a symbol was far greater than 8, and the repertoire of valid characters had grown significantly. And if you were one of the dinosaurs using an older system with the 8-character uppercase-only limitation, then you were just out of luck.

But even the greater symbol name length doesn't solve all type mismatches. For example, symbols for structures and unions are not decorated with the members of the structure or union. You can have one C++ file declare a structure called `S` as

```
struct S {
 int i;
 float f;
};
```

and have another C++ file declare it as

```
struct S {
 float f;
 int i;
};
```

and most compilers won't catch the mismatch.

With that historical background, we can begin addressing Adam’s question next time.

**Sidebar**: For those interested in nonclassical linking, there’s this article on changes to linker scalability in Visual C++ 2010.

Raymond Chen

**Follow**