Symbol Resolution

Symbol resolution runs the entire spectrum, from simple and intuitive to complex and perplexing. Resolutions can be carried out silently by the link-editor, can be accompanied by warning diagnostics, or can result in a fatal error condition.

The resolution of two symbols depends on their attributes, the type of file providing the symbol, and the type of file being generated. For a complete description of symbol attributes, see "Symbol Table". For the following discussions, however, it is worth identifying three basic symbol types:

- **Undefined** - Symbols that have been referenced in a file but have not been assigned a storage address.
- **Tentative** - Symbols that have been created within a file but have not yet been sized or allocated in storage. They appear as uninitialized C symbols, or `FORTRAN` `COMMON` blocks within the file.
- **Defined** - Symbols that have been created and assigned storage addresses and space within the file.

In its simplest form, symbol resolution involves the use of a precedence relationship that has **defined** symbols dominating **tentative** symbols, which in turn dominate **undefined** symbols.

The following C code example shows how these symbol types can be generated. Undefined symbols are prefixed with `u_`, tentative symbols are prefixed with `t_`, and defined symbols are prefixed with `d_`.

```
$ cat main.c
extern int      u_bar;
extern int      u_foo();

int             t_bar;
int             d_bar = 1;

d_foo()
{
        return (u_foo(u_bar, t_bar, d_bar));
}
$ cc -o main.o -c main.c
$ nm -x main.o

[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[8]      |0x00000000|0x00000000|NOTY |GLOB |0x0   |UNDEF  |u_foo
[9]      |0x00000000|0x00000040|FUNC |GLOB |0x0   |2      |d_foo
[10]     |0x00000004|0x00000004|OBJT |GLOB |0x0   |COMMON |t_bar
[11]     |0x00000000|0x00000000|NOTY |GLOB |0x0   |UNDEF  |u_bar
[12]     |0x00000000|0x00000004|OBJT |GLOB |0x0   |3      |d_bar
```

Simple Resolutions

Simple symbol resolutions are by far the most common, and result when two symbols with similar characteristics are detected and one symbol takes precedence over the other. This symbol resolution is carried out silently by the link-editor. For example, for symbols with the same binding, a reference to an undefined symbol from one file is bound to, or satisfied by, a defined or tentative symbol definition from another file. Or, a tentative symbol definition from one file is bound to a defined symbol definition from another file.

Symbols that undergo resolution can have either a global or weak binding. Weak bindings have lower precedence than global binding, so symbols with different bindings are resolved according to a slight alteration of the basic rules.

Weak symbols can usually be defined via the compiler, either individually or as aliases to global symbols. One mechanism uses a `#pragma` definition:

```
$ cat main.c
#pragma weak    bar
#pragma weak    foo = _foo

int             bar = 1;

_foo()
{
        return (bar);
}
$ cc -o main.o -c main.c
$ nm -x main.o
[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[7]     |0x00000000|0x00000004|OBJT |WEAK |0x0  |3      |bar
[8]     |0x00000000|0x00000028|FUNC |WEAK |0x0  |2      |foo
[9]     |0x00000000|0x00000028|FUNC |GLOB |0x0  |2      |_foo
```

Notice that the weak alias `foo` is assigned the same attributes as the global symbol `_foo`. This relationship is maintained by the link-editor and results in the symbols being assigned the same value in the output image. In symbol resolution, weak defined symbols are silently overridden by any global definition of the same name.

Another form of simple symbol resolution, interposition, occurs between relocatable objects and shared objects, or between multiple shared objects. In these cases, when a symbol is multiply-defined, the relocatable object, or the first definition between multiple shared objects, is silently taken by the link-editor. The relocatable object's definition, or the first shared object's definition, is said to **interpose** on all other definitions. This interposition can be used to override the functionality provided by one shared object, by a dynamic executable, or by another shared object.

The combination of weak symbols and interposition provides a useful programming technique. For example, the standard C library provides several services that you are allowed to redefine. However, ANSI C defines a set of standard services that must be present on the system and cannot be replaced in a strictly conforming program.

The function fread(3C), for example, is an ANSI C library function, whereas the system function read(2) is not. A conforming ANSI C program must be able to redefine read(2) and still use fread(3C) in a predictable way.

The problem here is that read(2) underlies the fread(3C) implementation in the standard C library. Therefore a program that redefines read(2) might confuse the fread(3C) implementation. To guard against this occurrence, ANSI C states that an implementation cannot use a name that is not reserved for it. Using the following `#pragma` directive you can define just such a reserved name, and from it generate an alias for the function read(2).

```
#pragma weak read = _read
```

Thus, you can quite freely define your own **read()** function without compromising the fread(3C) implementation, which in turn is implemented to use the **_read()** function.

The link-editor will not have difficulty with your redefinition of **read()**, either when linking against the shared object or archive version of the standard C library. In the former case, interposition takes its course. In the latter case, the fact that the C library's definition of read(2) is weak allows that definition to be quietly overridden.

You can use the link-editor's **-m** option to write a list of all interposed symbol references, along with section load address information, to the standard output.

Complex Resolutions

Complex resolutions occur when two symbols of the same name are found with differing attributes. In these cases, the link-editor selects the most appropriate symbol and generates a warning message indicating the symbol, the attributes that conflict, and the identity of the file from which the symbol definition is taken. In the following example two files with a definition of the data item array have different size requirements.

```
$ cat foo.c
int array[1];

$ cat bar.c
int array[2] = { 1, 2 };

$ cc -dn -r -o temp.o foo.c bar.c
ld: warning: symbol `array' has differing sizes:
        (file foo.o value=0x4; file bar.o value=0x8);
        bar.o definition taken
```

A similar diagnostic is produced if the symbol's alignment requirements differ. In both of these cases, the diagnostic can be suppressed by using the link-editor's **-t** option.

Another form of attribute difference is the symbol's type. In the following example the symbol **bar()** has been defined as both a data item and a function.

```
$ cat foo.c
bar()
{
        return (0);
}
$ cc -o libfoo.so -G -K pic foo.c
$ cat main.c
int     bar = 1;

main()
{
        return (bar);
}
$ cc -o main main.c -L. -lfoo
ld: warning: symbol `bar' has differing types:
        (file main.o type=OBJT; file ./libfoo.so type=FUNC);
        main.o definition taken
```

Note -

Symbol types in this context are classifications that can be expressed in ELF. They are not related to the data types as employed by the programming language, except in the crudest fashion.

In cases like the previous example, the relocatable object definition is taken when the resolution occurs between a relocatable object and a shared object, or the first definition is taken when the resolution occurs between two shared objects. When such resolutions occur between symbols of different bindings (weak global), a warning is also produced.

Inconsistencies between symbol types are not suppressed by the link-editor's **-t** option.

Fatal Resolutions

Symbol conflicts that cannot be resolved result in a fatal error condition. In this case, an appropriate error message is provided indicating the symbol name together with the names of the files that provided the symbols, and no output file is generated. Although the fatal condition is sufficient to terminate the link-edit, all input file processing is first completed. In this manner, all fatal resolution errors can be identified.

The most common fatal error condition exists when two relocatable objects both define symbols of the same name, and neither symbol is a weak definition:

```
$ cat foo.c
int bar = 1;

$ cat bar.c
bar()
{
        return (0);
}

$ cc -dn -r -o temp.o foo.c bar.c
ld: fatal: symbol `bar' is multiply-defined:
        (file foo.o and file bar.o);
ld: fatal: File processing errors. No output written to int.o
```

`foo.c` and `bar.c` have conflicting definitions for the symbol `bar`. Because the link-editor cannot determine which should dominate, the link-edit usually terminates with an error message. You can use the link-editor's **`-z muldefs`** option to suppress this error condition, and allow the first symbol definition to be taken.

- *Previous*: Input File Processing
- *Next*: Undefined Symbols