Q: I've recently upgraded Release 2 and I've recompiled my applications, including all my libraries. Now I can't link my application because of undefined symbols. This worked in Release 1. Is this a bug?

Q: I'm porting my C code from another UNIX environment. This code compiled and linked fine in the old environment, but fails to link on the NeXT. What's going on?

A: The GNU C compiler used by the NeXT was chosen, in part, because it complies with the ANSI C standard. Under Release 2 and later, the loader is more strictly ANSI compliant than under Release 1. The older BSD environment used by many UNIX systems tolerates lazy coding practices that ANSI does not.

To verify that this is your problem use the nm(1) command on the undefined symbols in your program (see the man page for more info):

```
localhost> nm -o myLib.a | grep mySymbol
myLib.a:foo.o: 000000004 C _mySymbol
```

Useful tip: To print the table of contents for a library, use otool(1):

```
localhost> otool -Sv myLib.a
```

Note the 'C' in front of the symbol '_mySymbol'; this indicates that it is a common symbol. The problem revolves around libraries that declare variables, but do not bother to initialize them. A common example is errno. We have all seen code that will declare errno in different files like:

```
int errno;
```

We all know that errno is defined somewhere, so we leave it to the linker to figure out which .o actually owns the variable to link it in. The linker assumes that the object file that not only declares the variable but also initializes it (called a Data Symbol), owns it and links in that module. If a module merely declares the variable, then this is called a Common Symbol.

Now comes the difference between the way ANSI looks at this versus generic BSD environments. What if no module actually initializes the variable in question. Then every

module has this symbol declared as a Common Symbol. What is the linker supposed to do at this point? Make a guess and pick the first module that references this Common Symbol and link it and any accompanying code that comes with it? Chances are you may never actually call the code in that module, but since this module declared knowledge of this Common Symbol, the loader has no other choice but to link it in. Your final executable file will have this extra code along with any other modules that this unwanted code may reference.

In an effort to make executables smaller and more efficient, to reduce paging and link and loading times, our ld complies with ANSI and ANSI considers this to be an error. This is why when you were linking your code, your variables could not be found.

The solution: The right way to fix this would be to initialize your variables in the library module that owns the data. If this is not feasible, then ranlib(1) knows how to revert to the older BSD-like behavior (see the -c option in the man page):

```
localhost> ranlib -c myLib.a
```

Q: Does it make sense to use the -c and -s options for ranlib simultaneously? As in:

```
localhost> ranlib -s -c myLib.a
```

A: Yes, but sometimes (lots of times) it does not work. What you are trying to do here is to speed up the link editor. This is accomplished by having ranlib build the table of contents in sorted order; this in turn is used when the link editor searches for an object file in a library that defines an undefined symbol. Since a sorted list ONLY works when there is exactly ONE object file in the library that defines each symbol this may fail when the -c flag is turned on because there maybe may the same common symbol in many of the object files in that library. In this case ranlib prints a message and an unsorted table of contents (like the -a option) is produced. Then, every time the link editor is used on that library it issues a warning about the table of contents not being sorted and that slower link editing will result. There are some uncommon uses of libraries that must use this unsorted symbol table to get the functionality required.

The key thing to remember in all of this is that the link editor (ld) does NOT search the object file's symbol tables of the library members to determine what symbols are defined. It ONLY searches the table of contents produced by ranlib.

QA691

Valid for 2.0, 3.0