How do UNIX linkers work with archive libraries, and why the order of objects and libraries on the link line matters.

I originally wrote this article in 2001 in gnu.gcc.help.
Here is a link to it.
Since then, I had to refer many people to it, and so decided that a local copy (with a shorter URL) is in order.
Here it is [slightly edited for clarity on 2006-01-10]:

```
"Robert Kraus" <robkra@gmx.net> wrote in message
news:F%HA7.11606$PV1.303467@news.chello.at...

> I often have linking problems, when I build a project with multiple static
> libraries.

You are not alone: I see many people who do

   gcc -lfoo main.o

>
> Sometimes these problems can get resolved by changing the order of the
> linked libraries.
>
> What causes these errors ? Why can it be resolved by rearranging -
> does this mean, that all includes, definitions, declarations etc.
> are correct ?

I find the following analogy most useful when explaining
how the linker works with archive libraries:

- think of an archive library as a bookshelf, with
  some books on it (the separate .o files).
- some books may refer you to other books
  (via unresolved symbols), which may be on the
  same, or on a different bookshelf.

A librarian comes to the first bookshelf (the first archive
library on your link line) with two lists -- a list of things
she still needs (unresolved symbols), and a list of things
she already has (symbols already defined).

Usually, she will already have crt0.o (added to the link
line by the compiler driver), which refers to main.
So her "need" list contains main. Often she will
have main.o as well, which means she no longer needs
main, but probably needs malloc, free, printf, etc.

Now she starts to scan one bookshelf at a time looking for books
that define symbols she needs. If a book defines one of
the symbols on the "need" list, she takes that book from
the shelf, scratches off all symbols defined in that book
from the "need" list, adds all defined symbols to the "have" list,
and adds all symbols used by that book and not already in the
"have" list to the "need" list.

If a book does not define any symbols currently in the "need"
list, she does not take it (even though this book may come in
```

handy later).

If a book she just took defines a symbol she already has, there is "multiply-defined symbol" problem.

If she took any books from the current shelf, she re-scans the shelf again, looking for more books to take (because the books she just took may need other books on the current shelf).

Once there are no more books she needs on the current shelf, she is done with that shelf, and she does not return to it (unless the library corresponding to it is listed on the command line several times).

If, after searching all the bookshelves listed on command line (as well as libc which is added by the compiler driver), she still has entries in the "need" list, there is an "undefined symbol" error.

Clearly, if many of your biology books refer to chemistry books, and you've got a biology paper to write, you better start searching the biology bookshelf before the chemistry one.

Sometimes, two archive libraries are inter-dependent: objects from one depend on objects from the other, and vice versa.

This means, that there is no correct order of linking such libraries, that both libraries are poorly structured, and that in order to successfully link an executable, you may need to list both of these libraries several times, as in:

```
   gcc main.o -lfoo -lbar -lfoo -lbar -lfoo
```

Jane Anderson (j.anderson@geac.com) adds, that on systems using GNU ld one can use the --start-group and --end-group command-line options to work around such inter-library dependencies:

```
   gcc main.o '-Wl,-(' -lfoo -lbar '-Wl,-)'
```

Finally, there is no reason a linker could not pass over all the objects and libraries twice: first to build the two lists, then to actually pull all the needed .o files from their libraries.

This is certainly user-friendly (you never get the types of errors that could be solved by rearranging library order).

I believe 2-pass linkers were common some 10-20 years ago. I believe the reasons they are not used much anymore are:

- they consume twice the amount of I/O and significant additional amounts of memory, and
- they force users with well-structured libraries (who can and do arrange them properly on the link line) to pay this price so that users with poorly-structured libraries can still link successfully.

The linker on AIX is the only UNIX linker I know that uses a 2-pass approach.

There is a good Linkers and Loaders book by John R. Levine.

```
      An early draft of it is available online.
```