

- 如果当链接器完成对命令行上输入文件的扫描后, U 是非空的, 那么链接器就会输出一个错误并终止。否则, 它会合并和重定位 E 中的目标文件, 构建输出的可执行文件。

不幸的是, 这种算法会导致一些令人困扰的链接时错误, 因为命令行上的库和目标文件的顺序非常重要。在命令行中, 如果定义一个符号的库出现在引用这个符号的目标文件之前, 那么引用就不能被解析, 链接会失败。比如, 考虑下面的命令行发生了什么?

```
linux> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function 'main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to 'addvec'
```

在处理 `libvector.a` 时, U 是空的, 所以没有 `libvector.a` 中的成员目标文件会添加到 E 中。因此, 对 `addvec` 的引用是绝不会被解析的, 所以链接器会产生一条错误信息并终止。


关于库的一般准则是将它们放在命令行的结尾。如果各个库的成员是相互独立的(也就是说没有成员引用另一个成员定义的符号), 那么这些库就可以以任何顺序放置在命令行的结尾处。另一方面, 如果库不是相互独立的, 那么必须对它们排序, 使得对于每个被存档文件的成员外部引用的符号 s , 在命令行中至少有一个 s 的定义是在对 s 的引用之后的。比如, 假设 `foo.c` 调用 `libx.a` 和 `libz.a` 中的函数, 而这两个库又调用 `liby.a` 中的函数。那么, 在命令行中 `libx.a` 和 `libz.a` 必须处在 `liby.a` 之前:

```
linux> gcc foo.c libx.a libz.a liby.a
```

如果需要满足依赖需求, 可以在命令行上重复库。比如, 假设 `foo.c` 调用 `libx.a` 中的函数, 该库又调用 `liby.a` 中的函数, 而 `liby.a` 又调用 `libx.a` 中的函数。那么 `libx.a` 必须在命令行上重复出现:

```
linux> gcc foo.c libx.a liby.a libx.a
```

另一种方法是, 我们可以将 `libx.a` 和 `liby.a` 合并成一个单独的存档文件。

 **练习题 7.3** a 和 b 表示当前目录中的目标模块或者静态库, 而 $a \rightarrow b$ 表示 a 依赖于 b , 也就是说 b 定义了一个被 a 引用的符号。对于下面每种场景, 请给出最小的命令行(即一个含有最少数量的目标文件和库参数的命令), 使得静态链接器能解析所有的符号引用。

- $p.o \rightarrow libx.a$
- $p.o \rightarrow libx.a \rightarrow liby.a$
- $p.o \rightarrow libx.a \rightarrow liby.a$ 且 $liby.a \rightarrow libx.a \rightarrow p.o$

7.7 重定位

一旦链接器完成了符号解析这一步, 就把代码中的每个符号引用和正好一个符号定义(即它的一个输入目标模块中的一个符号表条目)关联起来。此时, 链接器就知道它的输入目标模块中的代码节和数据节的确切大小。现在就可以开始重定位步骤了, 在这个步骤中, 将合并输入模块, 并为每个符号分配运行时地址。重定位由两步组成:

- 重定位节和符号定义。在这一步中, 链接器将所有相同类型的节合并为同一类型的新的聚合节。例如, 来自所有输入模块的 `.data` 节被全部合并成一个节, 这个节成为输出的可执行目标文件的 `.data` 节。然后, 链接器将运行时内存地址赋给新的聚合节, 赋给输入模块定义的每个节, 以及赋给输入模块定义的每个符号。当这一步完成时, 程序中的每条指令和全局变量都有唯一的运行时内存地址了。