The general rule for libraries is to place them at the end of the command line. If the members of the different libraries are independent, in that no member references a symbol defined by another member, then the libraries can be placed at the end of the command line in any order. If, on the other hand, the libraries are not independent, then they must be ordered so that for each symbol *s* that is referenced externally by a member of an archive, at least one definition of *s* follows a reference to *s* on the command line. For example, suppose `foo.c` calls functions in `libx.a` and `libz.a` that call functions in `liby.a`. Then `libx.a` and `libz.a` must precede `liby.a` on the command line:

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

Libraries can be repeated on the command line if necessary to satisfy the dependence requirements. For example, suppose `foo.c` calls a function in `libx.a` that calls a function in `liby.a` that calls a function in `libx.a`. Then `libx.a` must be repeated on the command line:

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

Alternatively, we could combine `libx.a` and `liby.a` into a single archive.

Practice Problem 7.3 **(solution page 754)**

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references.

A. `p.o` → `libx.a`

B. `p.o` → `libx.a` → `liby.a`

C. `p.o` → `libx.a` → `liby.a` *and* `liby.a` → `libx.a` → `p.o`

7.7 Relocation

Once the linker has completed the symbol resolution step, it has associated each symbol reference in the code with exactly one symbol definition (i.e., a symbol table entry in one of its input object modules). At this point, the linker knows the exact sizes of the code and data sections in its input object modules. It is now ready to begin the relocation step, where it merges the input modules and assigns run-time addresses to each symbol. Relocation consists of two steps:

1. *Relocating sections and symbol definitions.* In this step, the linker merges all sections of the same type into a new aggregate section of the same type. For example, the `.data` sections from the input modules are all merged into one section that will become the `.data` section for the output executable object

file. The linker then assigns run-time memory addresses to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules. When this step is complete, each instruction and global variable in the program has a unique run-time memory address.

2. *Relocating symbol references within sections.* In this step, the linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses. To perform this step, the linker relies on data structures in the relocatable object modules known as relocation entries, which we describe next.

7.7.1 Relocation Entries

When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate location is unknown, it generates a *relocation entry* that tells the linker how to modify the reference when it merges the object file into an executable. Relocation entries for code are placed in `.rel.text`. Relocation entries for data are placed in `.rel.data`.

Figure 7.9 shows the format of an ELF relocation entry. The `offset` is the section offset of the reference that will need to be modified. The `symbol` identifies the symbol that the modified reference should point to. The `type` tells the linker how to modify the new reference. The `addend` is a signed constant that is used by some types of relocations to bias the value of the modified reference.

ELF defines 32 different relocation types, many quite arcane. We are concerned with only the two most basic relocation types:

`R_X86_64_PC32`. Relocate a reference that uses a 32-bit PC-relative address. Recall from Section 3.6.3 that a PC-relative address is an offset from the current run-time value of the program counter (PC). When the CPU executes an instruction using PC-relative addressing, it forms the *effective address* (e.g., the target of the `call` instruction) by adding the 32-bit value

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

Figure 7.9 ELF relocation entry. Each entry identifies a reference that must be relocated and specifies how to compute the modified reference.

encoded in the instruction to the current run-time value of the PC, which is always the address of the next instruction in memory.

`R_X86_64_32`. Relocate a reference that uses a 32-bit absolute address. With absolute addressing, the CPU directly uses the 32-bit value encoded in the instruction as the effective address, without further modifications.

These two relocation types support the x86-64 *small code model*, which assumes that the total size of the code and data in the executable object file is smaller than 2 GB, and thus can be accessed at run-time using 32-bit PC-relative addresses. The small code model is the default for GCC. Programs larger than 2 GB can be compiled using the `-mcmodel=medium` (*medium code model*) and `-mcmodel=large` (*large code model*) flags, but we won't discuss those.

7.7.2 Relocating Symbol References

Figure 7.10 shows the pseudocode for the linker's relocation algorithm. Lines 1 and 2 iterate over each section `s` and each relocation entry `r` associated with each section. For concreteness, assume that each section `s` is an array of bytes and that each relocation entry `r` is a `struct` of type `Elf64_Rela`, as defined in Figure 7.9. Also, assume that when the algorithm runs, the linker has already chosen run-time addresses for each section (denoted `ADDR(s)`) and each symbol (denoted `ADDR(r.symbol)`). Line 3 computes the address in the `s` array of the 4-byte reference that needs to be relocated. If this reference uses PC-relative addressing, then it is relocated by lines 5–9. If the reference uses absolute addressing, then it is relocated by lines 11–13.

```
1  foreach section s {
2      foreach relocation entry r {
3          refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5          /* Relocate a PC-relative reference */
6          if (r.type == R_X86_64_PC32) {
7              refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8              *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9          }
10
11         /* Relocate an absolute reference */
12         if (r.type == R_X86_64_32)
13             *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14     }
15 }
```

Figure 7.10 **Relocation algorithm.**

code/link/main-relo.d

```
1  0000000000000000 <main>:
2     0:   48 83 ec 08             sub    $0x8,%rsp
3     4:   be 02 00 00 00          mov    $0x2,%esi
4     9:   bf 00 00 00 00          mov    $0x0,%edi        %edi = &array
5                          a: R_X86_64_32 array           Relocation entry

6     e:   e8 00 00 00 00          callq  13 <main+0x13>   sum()
7                          f: R_X86_64_PC32 sum-0x4        Relocation entry
8    13:   48 83 c4 08             add    $0x8,%rsp
9    17:   c3                      retq
```

code/link/main-relo.d

Figure 7.11 **Code and relocation entries from** `main.o`. The original C code is in Figure 7.1.

Let's see how the linker uses this algorithm to relocate the references in our example program in Figure 7.1. Figure 7.11 shows the disassembled code from `main.o`, as generated by the GNU OBJDUMP tool (`objdump -dx main.o`).

The `main` function references two global symbols, `array` and `sum`. For each reference, the assembler has generated a relocation entry, which is displayed on the following line.[2] The relocation entries tell the linker that the reference to `sum` should be relocated using a 32-bit PC-relative address, and the reference to `array` should be relocated using a 32-bit absolute address. The next two sections detail how the linker relocates these references.

Relocating PC-Relative References

In line 6 in Figure 7.11, function `main` calls the `sum` function, which is defined in module `sum.o`. The `call` instruction begins at section offset `0xe` and consists of the 1-byte opcode `0xe8`, followed by a placeholder for the 32-bit PC-relative reference to the target `sum`.

The corresponding relocation entry `r` consists of four fields:

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -4
```

These fields tell the linker to modify the 32-bit PC-relative reference starting at offset `0xf` so that it will point to the `sum` routine at run time. Now, suppose that the linker has determined that

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

2. Recall that relocation entries and instructions are actually stored in different sections of the object file. The OBJDUMP tool displays them together for convenience.

and

```
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
```

Using the algorithm in Figure 7.10, the linker first computes the run-time address of the reference (line 7):

```
refaddr = ADDR(s)  + r.offset
        = 0x4004d0 + 0xf
        = 0x4004df
```

It then updates the reference so that it will point to the `sum` routine at run time (line 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        = (unsigned) (0x4004e8       + (-4)     - 0x4004df)
        = (unsigned) (0x5)
```

In the resulting executable object file, the `call` instruction has the following relocated form:

```
  4004de:  e8 05 00 00 00          callq  4004e8 <sum>      sum()
```

At run time, the `call` instruction will be located at address `0x4004de`. When the CPU executes the `call` instruction, the PC has a value of `0x4004e3`, which is the address of the instruction immediately following the `call` instruction. To execute the call instruction, the CPU performs the following steps:

1. Push PC onto stack
2. PC $\leftarrow$ PC + 0x5 = 0x4004e3 + 0x5 = 0x4004e8

Thus, the next instruction to execute is the first instruction of the `sum` routine, which of course is what we want!

Relocating Absolute References

Relocating absolute references is straightforward. For example, in line 4 in Figure 7.11, the `mov` instruction copies the address of `array` (a 32-bit immediate value) into register `%edi`. The `mov` instruction begins at section offset `0x9` and consists of the 1-byte opcode `0xbf`, followed by a placeholder for the 32-bit absolute reference to `array`.

The corresponding relocation entry `r` consists of four fields:

```
r.offset = 0xa
r.symbol = array
r.type   = R_X86_64_32
r.addend = 0
```

These fields tell the linker to modify the absolute reference starting at offset `0xa` so that it will point to the first byte of `array` at run time. Now, suppose that the linker has determined that

(a) Relocated `.text` section

```
1  00000000004004d0 <main>:
2    4004d0:  48 83 ec 08           sub    $0x8,%rsp
3    4004d4:  be 02 00 00 00        mov    $0x2,%esi
4    4004d9:  bf 18 10 60 00        mov    $0x601018,%edi    %edi = &array
5    4004de:  e8 05 00 00 00        callq  4004e8 <sum>      sum()
6    4004e3:  48 83 c4 08           add    $0x8,%rsp
7    4004e7:  c3                    retq

8  00000000004004e8 <sum>:
9    4004e8:  b8 00 00 00 00        mov    $0x0,%eax
10   4004ed:  ba 00 00 00 00        mov    $0x0,%edx
11   4004f2:  eb 09                 jmp    4004fd <sum+0x15>
12   4004f4:  48 63 ca              movslq %edx,%rcx
13   4004f7:  03 04 8f              add    (%rdi,%rcx,4),%eax
14   4004fa:  83 c2 01              add    $0x1,%edx
15   4004fd:  39 f2                 cmp    %esi,%edx
16   4004ff:  7c f3                 jl     4004f4 <sum+0xc>
17   400501:  f3 c3                 repz retq
```

(b) Relocated `.data` section

```
1  0000000000601018 <array>:
2    601018:  01 00 00 00 02 00 00 00
```

Figure 7.12 **Relocated `.text` and `.data` sections for the executable file `prog`.** The original C code is in Figure 7.1.

```
ADDR(r.symbol) = ADDR(array) = 0x601018
```

The linker updates the reference using line 13 of the algorithm in Figure 7.10:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend)
        = (unsigned) (0x601018       + 0)
        = (unsigned) (0x601018)
```

In the resulting executable object file, the reference has the following relocated form:

```
  4004d9:  bf 18 10 60 00         mov    $0x601018,%edi   %edi = &array
```

Putting it all together, Figure 7.12 shows the relocated `.text` and `.data` sections in the final executable object file. At load time, the loader can copy the bytes from these sections directly into memory and execute the instructions without any further modifications.

Practice Problem 7.4 (solution page 754)

This problem concerns the relocated program in Figure 7.12(a).

A. What is the hex address of the relocated reference to `sum` in line 5?

B. What is the hex value of the relocated reference to `sum` in line 5?

Practice Problem 7.5 **(solution page 754)**

Consider the call to function `swap` in object file `m.o` (Figure 7.5).

```
9:   e8 00 00 00 00          callq  e <main+0xe>      swap()
```

with the following relocation entry:

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

Now suppose that the linker relocates `.text` in `m.o` to address `0x4004d0` and `swap` to address `0x4004e8`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

7.8 Executable Object Files

We have seen how the linker merges multiple object files into a single executable object file. Our example C program, which began life as a collection of ASCII text files, has been transformed into a single binary file that contains all of the information needed to load the program into memory and run it. Figure 7.13 summarizes the kinds of information in a typical ELF executable file.

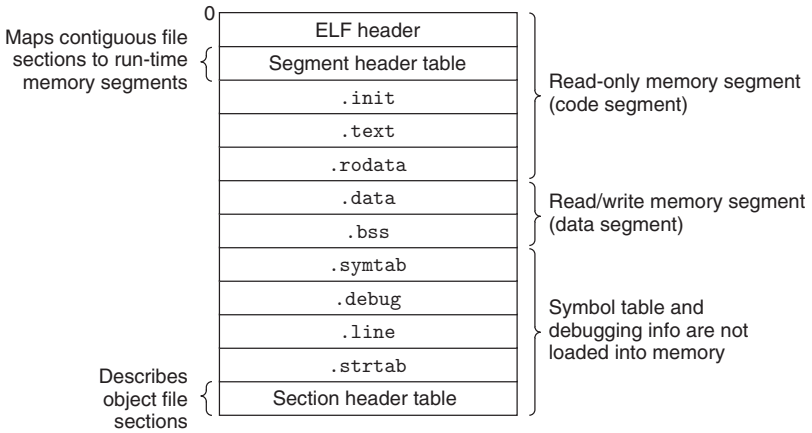

Figure 7.13 Typical ELF executable object file.