To build the executable, the linker must perform two main tasks:

Step 1. Symbol resolution. Object files define and reference *symbols*, where each symbol corresponds to a function, a global variable, or a *static variable* (i.e., any C variable declared with the `static` attribute). The purpose of symbol resolution is to associate each symbol *reference* with exactly one symbol *definition*.

Step 2. Relocation. Compilers and assemblers generate code and data sections that start at address 0. The linker *relocates* these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location. The linker blindly performs these relocations using detailed instructions, generated by the assembler, called *relocation entries*.

The sections that follow describe these tasks in more detail. As you read, keep in mind some basic facts about linkers: Object files are merely collections of blocks of bytes. Some of these blocks contain program code, others contain program data, and others contain data structures that guide the linker and loader. A linker concatenates blocks together, decides on run-time locations for the concatenated blocks, and modifies various locations within the code and data blocks. Linkers have minimal understanding of the target machine. The compilers and assemblers that generate the object files have already done most of the work.

7.3 Object Files

Object files come in three forms:

Relocatable object file. Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.

Executable object file. Contains binary code and data in a form that can be copied directly into memory and executed.

Shared object file. A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time.

Compilers and assemblers generate relocatable object files (including shared object files). Linkers generate executable object files. Technically, an *object module* is a sequence of bytes, and an *object file* is an object module stored on disk in a file. However, we will use these terms interchangeably.

Object files are organized according to specific *object file formats*, which vary from system to system. The first Unix systems from Bell Labs used the `a.out` format. (To this day, executables are still referred to as `a.out` files.) Windows uses the Portable Executable (PE) format. Mac OS-X uses the Mach-O format. Modern x86-64 Linux and Unix systems use *Executable and Linkable Format (ELF)*. Although our discussion will focus on ELF, the basic concepts are similar, regardless of the particular format.

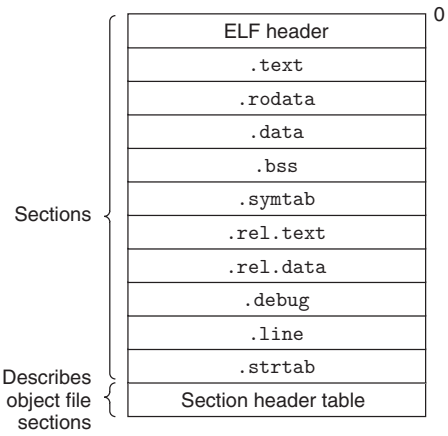

Figure 7.3
Typical ELF relocatable object file.

7.4 Relocatable Object Files

Figure 7.3 shows the format of a typical ELF relocatable object file. The *ELF header* begins with a 16-byte sequence that describes the word size and byte ordering of the system that generated the file. The rest of the ELF header contains information that allows a linker to parse and interpret the object file. This includes the size of the ELF header, the object file type (e.g., relocatable, executable, or shared), the machine type (e.g., x86-64), the file offset of the section header table, and the size and number of entries in the section header table. The locations and sizes of the various sections are described by the *section header table*, which contains a fixed-size entry for each section in the object file.

Sandwiched between the ELF header and the section header table are the sections themselves. A typical ELF relocatable object file contains the following sections:

`.text` The machine code of the compiled program.

`.rodata` Read-only data such as the format strings in `printf` statements, and jump tables for switch statements.

`.data` *Initialized* global and static C variables. Local C variables are maintained at run time on the stack and do *not* appear in either the `.data` or `.bss` sections.

`.bss` *Uninitialized* global and static C variables, along with any global or static variables that are initialized to zero. This section occupies no actual space in the object file; it is merely a placeholder. Object file formats distinguish between initialized and uninitialized variables for space efficiency: uninitialized variables do not have to occupy any actual disk space in the object file. At run time, these variables are allocated in memory with an initial value of zero.