Aside Why is uninitialized data called `.bss`?

The use of the term `.bss` to denote uninitialized data is universal. It was originally an acronym for the "block started by symbol" directive from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the `.data` and `.bss` sections is to think of "bss" as an abbreviation for "Better Save Space!"

`.symtab` A *symbol table* with information about functions and global variables that are defined and referenced in the program. Some programmers mistakenly believe that a program must be compiled with the `-g` option to get symbol table information. In fact, every relocatable object file has a symbol table in `.symtab` (unless the programmer has specifically removed it with the STRIP command). However, unlike the symbol table inside a compiler, the `.symtab` symbol table does not contain entries for local variables.

`.rel.text` A list of locations in the `.text` section that will need to be modified when the linker combines this object file with others. In general, any instruction that calls an external function or references a global variable will need to be modified. On the other hand, instructions that call local functions do not need to be modified. Note that relocation information is not needed in executable object files, and is usually omitted unless the user explicitly instructs the linker to include it.

`.rel.data` Relocation information for any global variables that are referenced or defined by the module. In general, any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.

`.debug` A debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the `-g` option.

`.line` A mapping between line numbers in the original C source program and machine code instructions in the `.text` section. It is only present if the compiler driver is invoked with the `-g` option.

`.strtab` A string table for the symbol tables in the `.symtab` and `.debug` sections and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

7.5 Symbols and Symbol Tables

Each relocatable object module, m, has a symbol table that contains information about the symbols that are defined and referenced by m. In the context of a linker, there are three different kinds of symbols:

- *Global symbols* that are defined by module *m* and that can be referenced by other modules. Global linker symbols correspond to *nonstatic* C functions and global variables.
- Global symbols that are referenced by module *m* but defined by some other module. Such symbols are called *externals* and correspond to nonstatic C functions and global variables that are defined in other modules.
- *Local symbols* that are defined and referenced exclusively by module *m*. These correspond to static C functions and global variables that are defined with the `static` attribute. These symbols are visible anywhere within module *m*, but cannot be referenced by other modules.

It is important to realize that local linker symbols are not the same as local program variables. The symbol table in `.symtab` does not contain any symbols that correspond to local nonstatic program variables. These are managed at run time on the stack and are not of interest to the linker.

Interestingly, local procedure variables that are defined with the C `static` attribute are not managed on the stack. Instead, the compiler allocates space in `.data` or `.bss` for each definition and creates a local linker symbol in the symbol table with a unique name. For example, suppose a pair of functions in the same module define a static local variable `x`:

```
1   int f()
2   {
3       static int x = 0;
4       return x;
5   }
6
7   int g()
8   {
9       static int x = 1;
10      return x;
11  }
```

In this case, the compiler exports a pair of local linker symbols with different names to the assembler. For example, it might use `x.1` for the definition in function `f` and `x.2` for the definition in function `g`.

Symbol tables are built by assemblers, using symbols exported by the compiler into the assembly-language `.s` file. An ELF symbol table is contained in the `.symtab` section. It contains an array of entries. Figure 7.4 shows the format of each entry.

The `name` is a byte offset into the string table that points to the null-terminated string name of the symbol. The `value` is the symbol's address. For relocatable modules, the `value` is an offset from the beginning of the section where the object is defined. For executable object files, the `value` is an absolute run-time address. The `size` is the size (in bytes) of the object. The `type` is usually either `data` or `function`. The symbol table can also contain entries for the individual sections

New to C? Hiding variable and function names with `static`

C programmers use the `static` attribute to hide variable and function declarations inside modules, much as you would use *public* and *private* declarations in Java and C++. In C, source files play the role of modules. Any global variable or function declared with the `static` attribute is private to that module. Similarly, any global variable or function declared without the `static` attribute is public and can be accessed by any other module. It is good programming practice to protect your variables and functions with the `static` attribute wherever possible.

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

Figure 7.4 ELF symbol table entry. The `type` and `binding` fields are 4 bits each.

and for the path name of the original source file. So there are distinct types for these objects as well. The `binding` field indicates whether the symbol is local or global.

Each symbol is assigned to some section of the object file, denoted by the `section` field, which is an index into the section header table. There are three special pseudosections that don't have entries in the section header table: ABS is for symbols that should not be relocated. UNDEF is for undefined symbols—that is, symbols that are referenced in this object module but defined elsewhere. COMMON is for uninitialized data objects that are not yet allocated. For COMMON symbols, the `value` field gives the alignment requirement, and `size` gives the minimum size. Note that these pseudosections exist only in relocatable object files; they do not exist in executable object files.

The distinction between COMMON and `.bss` is subtle. Modern versions of GCC assign symbols in relocatable object files to COMMON and `.bss` using the following convention:

COMMON	Uninitialized global variables
`.bss`	Uninitialized static variables, and global or static variables that are initialized to zero

The reason for this seemingly arbitrary distinction stems from the way the linker performs symbol resolution, which we will explain in Section 7.6.

The GNU READELF program is a handy tool for viewing the contents of object files. For example, here are the last three symbol table entries for the relocatable object file `main.o`, from the example program in Figure 7.1. The first eight entries, which are not shown, are local symbols that the linker uses internally.

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  8: 0000000000000000    24 FUNC    GLOBAL DEFAULT    1 main
  9: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
 10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

In this example, we see an entry for the definition of global symbol `main`, a 24-byte function located at an offset (i.e., `value`) of zero in the `.text` section. This is followed by the definition of the global symbol `array`, an 8-byte object located at an offset of zero in the `.data` section. The last entry comes from the reference to the external symbol `sum`. READELF identifies each section by an integer index. `Ndx=1` denotes the `.text` section, and `Ndx=3` denotes the `.data` section.

Practice Problem 7.1 (solution page 753)

This problem concerns the `m.o` and `swap.o` modules from Figure 7.5. For each symbol that is defined or referenced in `swap.o`, indicate whether or not it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `m.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, `.bss`, or COMMON) it is assigned to in the module.

(a) `m.c` — *code/link/m.c*

```
1  void swap();
2
3  int buf[2] = {1, 2};
4
5  int main()
6  {
7      swap();
8      return 0;
9  }
```

code/link/m.c

(b) `swap.c` — *code/link/swap.c*

```
1  extern int buf[];
2
3  int *bufp0 = &buf[0];
4  int *bufp1;
5
6  void swap()
7  {
8      int temp;
9
10     bufp1 = &buf[1];
11     temp = *bufp0;
12     *bufp0 = *bufp1;
13     *bufp1 = temp;
14 }
```

code/link/swap.c

Figure 7.5 **Example program for Practice Problem 7.1.**

Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf				
bufp0				
bufp1				
swap				
temp				

7.6 Symbol Resolution

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Symbol resolution is straightforward for references to local symbols that are defined in the same module as the reference. The compiler allows only one definition of each local symbol per module. The compiler also ensures that static local variables, which get local linker symbols, have unique names.

Resolving references to global symbols, however, is trickier. When the compiler encounters a symbol (either a variable or function name) that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle. If the linker is unable to find a definition for the referenced symbol in any of its input modules, it prints an (often cryptic) error message and terminates. For example, if we try to compile and link the following source file on a Linux machine,

```
1   void foo(void);
2
3   int main() {
4       foo();
5       return 0;
6   }
```

then the compiler runs without a hitch, but the linker terminates when it cannot resolve the reference to `foo`:

```
linux> gcc -Wall -Og -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function ‘main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to ‘foo'
```

Symbol resolution for global symbols is also tricky because multiple object modules might define global symbols with the same name. In this case, the linker must either flag an error or somehow choose one of the definitions and discard the rest. The approach adopted by Linux systems involves cooperation between the compiler, assembler, and linker and can introduce some baffling bugs to the unwary programmer.