CHAPTER 2

Simple Program Lifetime Stages

In the previous chapter, you obtained a broad insight into aspects of the modern multitasking operating system's functionality that play a role during program execution. The natural next question that comes to the programmer's mind is what to do, how, and why in order to arrange for the program execution to happen.

Much like the lifetime of a butterfly is determined by its caterpillar stage, the lifetime of a program is greatly determined by the inner structure of the binary, which the OS loader loads, unpacks, and puts its contents into the execution. It shouldn't come as a big surprise that most of our subsequent discussions will be devoted to the art of preparing a blueprint and properly embedding it into the body of the binary executable file(s). We will assume that the program is written in C/C++.

To completely understand the whole story, the details of the rest of the program's lifetime, the loading and execution stage, will be analyzed in great detail. Further discussions will be focused around the following stages of the program's lifetime:

1. Creating the source code
2. Compiling
3. Linking
4. Loading
5. Executing

The truth be told, this chapter will cover far more details about the compiling stage than about the subsequent stages. The coverage of subsequent stages (especially linking stage) only starts in this chapter, in which you will only see the proverbial "tip of the iceberg." After the most basic introduction of ideas behind the linking stage, the remainder of the book will deal with the intricacies of linking as well as program loading and executing.

Initial Assumptions

Even though it is very likely that a huge percentage of readers belong to the category of advanced-to-expert programmers, I will start with fairly simple initial examples. The discussions in this chapter will be pertinent to the very simple, yet very illustrative case. The demo project consists of two simple source files, which will be first compiled and then linked together. The code is written with the intention of keeping the complexity of both compiling and linking at the simplest possible level.

In particular, no linking of external libraries, particularly not dynamic linking, will be taking place in this demo example. The only exception will be the linking with the C runtime library (which is anyways required for the vast majority of programs written in C). Being such a common element in the lifetime of C program execution, for the sake of simplicity I will purposefully turn a blind eye to the particular details of linking with the C runtime library, and assume that the program is created in such a way that all the code from the C runtime library is "automagically" inserted into the body of the program memory map.

By following this approach, I will illustrate the details of program building's quintessential problems in a simple and clean form.

Code Writing

Given that the major topic of this book is the process of program building (i.e., what happens after the source code is written), I will not spend too much time on the source code creation process.

Except in a few rare cases when the source code is produced by a script, it is assumed that a user does it by typing in the ASCII characters in his editor of choice in an effort to produce the written statements that satisfy the syntax rules of the programming language of choice (C/C++ in our case). The editor of choice may vary from the simplest possible ASCII text editor all the way to the most advanced IDE tool. Assuming that the average reader of this book is a fairly experienced programmer, there is really nothing much special to say about this stage of program life cycle.

However, there is one particular programming practice that significantly impacts where the story will be going from this point on, and it is worth of paying extra attention to it. In order to better organize the source code, programmers typically follow the practice of keeping the various functional parts of the code in separate files, resulting with the projects generally comprised of many different source and header files.

This programming practice was adopted very early on, since the time of the development environments made for the early microprocessors. Being a very solid design decision, it has been practiced ever since, as it is proven to provide solid organization of the code and makes code maintenance tasks significantly easier.

This undoubtedly useful programming practice has far reaching consequences. As you will see soon, practicing it leads to certain amount of indeterminism in the subsequent stages of the building process, the resolving of which requires some careful thinking.

Concept illustration: Demo Project

In order to better illustrate the intricacies of the compiling process, as well as to provide the reader with a little hands-on warm-up experience, a simple demo project has been provided. The code is exceptionally simple; it is comprised of no more than one header and two source files. However, it is carefully designed to illustrate the points of extraordinarily importance for understanding the broader picture.

The following files are the part of the project:

- Source file **main.c**, which contains the `main()` function.
- Header file **function.h**, which declares the functions called and the data accessed by the `main()` function.
- Source file **function.c**, which contains the source code implementations of functions and instantiation of the data referenced by the `main()` function.

The development environment used to build this simple project will be based on the **gcc** compiler running on Linux. Listings 2-1 through 2-3 contain the code used in the demo project.

Listing 2-1. `function.h`

```
#pragma once

#define FIRST_OPTION
#ifdef FIRST_OPTION
#define MULTIPLIER (3.0)
#else
#define MULTIPLIER (2.0)#endif

float add_and_multiply(float x, float y);
```

Listing 2-2. function.c

```
int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= MULTIPLIER;
    return z;
}
```

Listing 2-3. main.c

```
#include "function.h"
extern int nCompletionStatus = 0;
int main(int argc, char* argv[])
{
    float x = 1.0;
    float y = 5.0;
    float z;

    z = add_and_multiply(x,y);
    nCompletionStatus = 1;
    return 0;
}
```

Compiling

Once you have written your source code, it is the time to immerse yourself in the process of code building, whose mandatory first step is the compiling stage. Before going into the intricacies of compiling, a few simple introductory terms will be presented first.

Introductory Definitions

Compiling in the broad sense can be defined as the process of transforming source code written in one programming language into another programming language. The following set of introductory facts is important for your overall understanding of the compilation process:

- The process of compiling is performed by the program called the **compiler**.
- The input for the compiler is a **translation unit**. A typical translation unit is a text file containing the source code.
- A program is typically comprised of many translation units. Even though it is perfectly possible and legal to keep all the project's source code in a single file, there are good reasons (explained in the previous section) of why it is typically not the case.

- The output of the compilation is a collection of binary **object files**, one for each of the input translation units.
- In order to become suitable for execution, the object files need to be processed through another stage of program building called **linking**.

Figure 2-1 illustrates the concept of compiling.

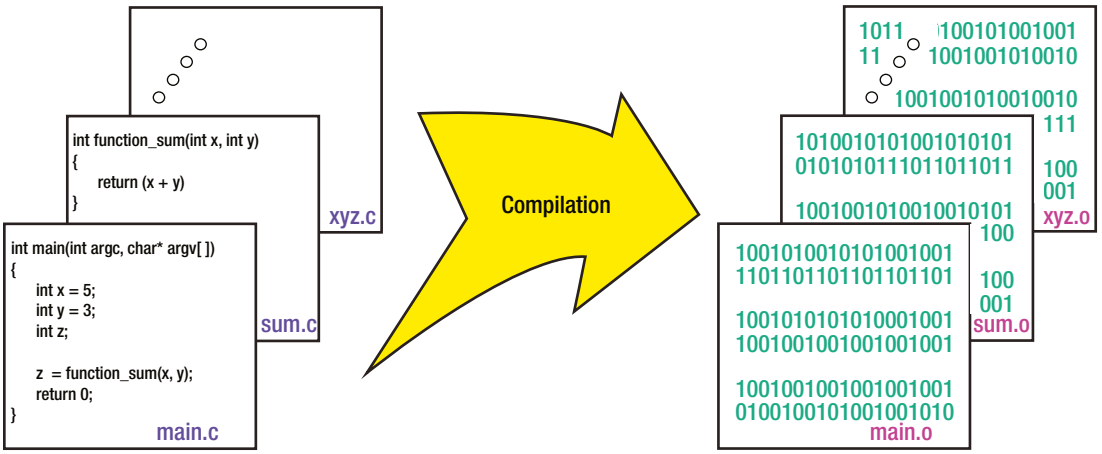

Figure 2-1. *The compiling stage*

Related Definitions

The following variety of compiler use cases is typically encountered:

- **Compilation** in the strict meaning denotes the process of translating the code of a higher-level language to the code of a lower-level language (typically, assembler or even machine code) production files.
- If the compilation is performed on one platform (CPU/OS) to produce code to be run on some other platform (CPU/OS), it is called **cross-compilation**. The usual practice is to use some of the desktop OSes (Linux, Windows) to generate the code for embedded or mobile devices.
- **Decompilation** (disassembling) is the process of converting the source code of a lower-level language to the higher-level language.
- **Language translation** is the process of transforming source code of one programming language to another programming language of the same level and complexity.
- **Language rewriting** is the process of rewriting the language expressions into a form more suitable for certain tasks (such as optimization).

The Stages of Compiling

The compilation process is not monolithic in nature. In fact, it can be roughly divided into the several stages (pre-processing, linguistic analysis, assembling, optimization, code emission), the details of which will be discussed next.

Preprocessing

The standard first step in processing the source files is running them through the special text processing program called a **preprocessor**, which performs one or more of the following actions:

- Includes the files containing definitions (include/header files) into the source files, as specified by the `#include` keyword.
- Converts the values specified by using `#define` statements into the constants.
- Converts the macro definitions into code at the variety of locations in which the macros are invoked.
- Conditionally includes or excludes certain parts of the code, based on the position of `#if`, `#elif`, and `#endif` directives.

The output of the preprocessor is the C/C++ code in its final shape, which will be passed to the next stage, syntax analysis.

Demo Project Preprocessing Example

The gcc compiler provides the mode in which only the preprocessing stage is performed on the input source files:

```
gcc -i <input file> -o <output preprocessed file>.i
```

Unless specified otherwise, the output of the preprocessor is the file that has the same name as the input file and whose file extension is **.i**. The result of running the preprocessor on the file `function.c` looks like that in Listing 2-4.

Listing 2-4. `function.i`

```
# 1 "function.c"
# 1 "
# 1 "
# 1 "function.h" 1

# 11 "function.h"
float add_and_multiply(float x, float y);
# 2 "function.c" 2

int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= MULTIPLIER;
    return z;
}
```

More compact and more meaningful preprocessor output may be obtained if few extra flags are passed to the gcc, like

```
gcc -E -P -i <input file> -o <output preprocessed file>.i
```

which results in the preprocessed file seen in Listing 2-5.

Listing 2-5. `function.i (Trimmed Down Version)`

```
float add_and_multiply(float x, float y);
int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= 3.0;
    return z;
}
```

Obviously, the preprocessor replaced the symbol `MULTIPLIER`, whose actual value, based on the fact that the `USE_FIRST_OPTION` variable was defined, ended up being 3.0.

Linguistic Analysis

During this stage, the compiler first converts the C/C++ code into a form more suitable for processing (eliminating comments and unnecessary white spaces, extracting tokens from the text, etc.). Such an optimized and compacted form of source code is lexically analyzed, with the intention of checking whether the program satisfies the syntax rules of the programming language in which it was written. If deviations from the syntax rules are detected, errors or warnings are reported. The errors are sufficient cause for the compilation to be terminated, whereas warnings may or may not be sufficient, depending on the user's settings.

More precise insight into this stage of the compilation process reveals three distinct stages:

- **Lexical analysis**, which breaks the source code into non-divisible tokens. The next stage,
- **Parsing/syntax analysis** concatenates the extracted tokens into the chains of tokens, and verifies that their ordering makes sense from the standpoint of programming language rules. Finally,
- **Semantic analysis** is run with the intent to discover whether the syntactically correct statements actually make any sense. For example, a statement that adds two integers and assigns the result to an object will pass syntax rules, but may not pass semantic check (unless the object has overridden assignment operator).

During the linguistic analysis stage, the compiler probably more deserves to be called "complainer," as it tends to more complain about typos or other errors it encounters than to actually compile the code.

Assembling

The compiler reaches this stage only after the source code is verified to contain no syntax errors. In this stage, the compiler tries to convert the standard language constructs into the constructs specific to the actual CPU instruction set. Different CPUs feature different functionality treats, and in general different sets of available instructions, registers, interrupts, which explains the wide variety of compilers for an even wider variety of processors.

Demo Project Assembling Example

The gcc compiler provides the mode of operation in which the input files' source code is converted into the ASCII text file containing the lines of assembler instructions specific to the chip and/or the operating system.

```
$ gcc -S <input file> -o <output assembler file>.s
```

Unless specified otherwise, the output of the preprocessor is the file that has the same name as the input file and whose file extension is **.s**.

The generated file is not suitable for execution; it is merely a text file carrying the human-readable mnemonics of assembler instructions, which can be used by the developer to get a better insight into the details of the inner workings of the compilation process.

In the particular case of the X86 processor architecture, the assembler code may conform to one of the two supported instruction printing formats,

- AT&T format
- Intel format

the choice of which may be specified by passing an extra command-line argument to the gcc assembler. The choice of format is mostly the matter of the developer's personal taste.

AT&T Assembly Format Example

When the file `function.c` is assembled into the AT&T format by running the following command

```
$ gcc -S -masm=att function.c -o function.s
```

it creates the output assembler file, which looks the code shown in Listing 2-6.

Listing 2-6. `function.s (AT&T Assembler Format)`

```
      .file     "function.c"
      .globl    nCompletionStatus
      .bss
      .align 4
      .type     nCompletionStatus, @object
      .size     nCompletionStatus, 4
nCompletionStatus:
      .zero     4
      .text
      .globl    add
      .type     add, @function
```

```
add:
.LFB0:
      .cfi_startproc
      pushl     %ebp
      .cfi_def_cfa_offset 8
      .cfi_offset 5, -8
      movl      %esp, %ebp
      .cfi_def_cfa_register 5
      subl      $20, %esp
      flds      8(%ebp)
      fadds     12(%ebp)
      fstps     -4(%ebp)
      movl      -4(%ebp), %eax
      movl      %eax, -20(%ebp)
      flds      -20(%ebp)
      leave
      .cfi_restore 5
      .cfi_def_cfa 4, 4
      ret
      .cfi_endproc
.LFE0:
      .size     add, .-add
      .globl    add_and_multiply
      .type     add_and_multiply, @function
add_and_multiply:
.LFB1:
      .cfi_startproc
      pushl     %ebp
      .cfi_def_cfa_offset 8
      .cfi_offset 5, -8
      movl      %esp, %ebp
      .cfi_def_cfa_register 5
      subl      $28, %esp
      movl      12(%ebp), %eax
      movl      %eax, 4(%esp)
      movl      8(%ebp), %eax
      movl      %eax, (%esp)
      call      add
      fstps     -4(%ebp)
      flds      -4(%ebp)
      flds      .LC1
      fmulp     %st, %st(1)
      fstps     -4(%ebp)
      movl      -4(%ebp), %eax
      movl      %eax, -20(%ebp)
      flds      -20(%ebp)
      leave
      .cfi_restore 5
      .cfi_def_cfa 4, 4
      ret
      .cfi_endproc
```

```
.LFE1:
      .size     add_and_multiply, .-add_and_multiply
      .section          .rodata
      .align 4
.LC1:
      .long     1077936128
      .ident    "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
      .section          .note.GNU-stack,"",@progbits
```

Intel Assembly Format Example

The same file (`function.c`) may be assembled into the Intel assembler format by running the following command,

```
$ gcc -S -masm=intel function.c -o function.s
```

which results with the assembler file shown in Listing 2-7.

Listing 2-7. `function.s (Intel Assembler Format)`

```
      .file     "function.c"
      .intel_syntax noprefix
      .globl    nCompletionStatus
      .bss
      .align 4
      .type     nCompletionStatus, @object
      .size     nCompletionStatus, 4
nCompletionStatus:
      .zero     4
      .text
      .globl    add
      .type     add, @function
add:
.LFB0:
      .cfi_startproc
      push      ebp
      .cfi_def_cfa_offset 8
      .cfi_offset 5, -8
      mov       ebp, esp
      .cfi_def_cfa_register 5
      sub       esp, 20
      fld       DWORD PTR [ebp+8]
      fadd      DWORD PTR [ebp+12]
      fstp      DWORD PTR [ebp-4]
      mov       eax, DWORD PTR [ebp-4]
      mov       DWORD PTR [ebp-20], eax
      fld       DWORD PTR [ebp-20]
      leave
      .cfi_restore 5
      .cfi_def_cfa 4, 4
      ret
      .cfi_endproc
```

```
.LFE0:
      .size     add, .-add
      .globl    add_and_multiply
      .type     add_and_multiply, @function
add_and_multiply:
.LFB1:
      .cfi_startproc
      push      ebp
      .cfi_def_cfa_offset 8
      .cfi_offset 5, -8
      mov       ebp, esp
      .cfi_def_cfa_register 5
      sub       esp, 28
      mov       eax, DWORD PTR [ebp+12]
      mov       DWORD PTR [esp+4], eax
      mov       eax, DWORD PTR [ebp+8]
      mov       DWORD PTR [esp], eax
      call      add
      fstp      DWORD PTR [ebp-4]
      fld       DWORD PTR [ebp-4]
      fld       DWORD PTR .LC1
      fmulp     st(1), st
      fstp      DWORD PTR [ebp-4]
      mov       eax, DWORD PTR [ebp-4]
      mov       DWORD PTR [ebp-20], eax
      fld       DWORD PTR [ebp-20]
      leave
      .cfi_restore 5
      .cfi_def_cfa 4, 4
      ret
      .cfi_endproc
.LFE1:
      .size     add_and_multiply, .-add_and_multiply
      .section          .rodata
      .align 4
.LC1:
      .long     1077936128
      .ident    "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
      .section          .note.GNU-stack,"",@progbits
```

Optimization

Once the first assembler version corresponding to the original source code is created, the optimization effort starts, in which usage of the registers is minimized. Additionally, the analysis may indicate that certain parts of the code do not in fact need to be executed, and such parts of the code are eliminated.

Code Emission

Finally, the moment has come to create the compilation output: **object files**, one for each translation unit. The assembly instructions (written in human-readable ASCII code) are at this stage converted into the binary values of the corresponding machine instructions (opcodes) and written to the specific locations in the object file(s).

The object file is still not ready to be served as the meal to the hungry processor. The reasons why are the essential topic of this whole book. The interesting topic at this moment is the analysis of an object file.

Being a binary file makes the object file substantially different than the outputs of preprocessing and assembling procedures, both of which are ASCII files, inherently readable by humans. The differences become the most obvious when we, the humans, try to take a closer look at the contents.

Other than obvious choice of using the hex editor (not very helpful unless you write compilers for living), a specific procedure called **disassembling** is taken in order to get a detailed insight into the contents of an object file.

On the overall path from the ASCII files toward the binary files suitable for execution on the concrete machine, the disassembling may be viewed as a little U-turn detour in which the almost-ready binary file is converted into the ASCII file to be served to the curious eyes of the software developer. Fortunately, this little detour serves only the purpose of supplying the developer with better orientation, and is normally not performed without a real cause.

Demo Project Compiling Example

The gcc compiler may be set to perform the complete compilation (preprocessing and assembling and compiling), a procedure that generates the binary object file (standard extension .o) whose structure follows the ELF format guidelines. In addition to usual overhead (header, tables, etc.), it contains all the pertinent sections (.text, .code, .bss, etc.). In order to specify the compilation only (no linking as of yet), the following command line may be used:

```
$ gcc -c <input file> -o <output file>.o
```

Unless specified otherwise, the output of the preprocessor is the file that has the same name as the input file and whose file extension is **.o**.

The content of the generated object file is not suitable for viewing in a text editor. The hex editor/viewer is a bit more suitable, as it will not be confused by the nonprintable characters and absences of newline characters. Figure 2-2 shows the binary contents of the object file `function.o` generated by compiling the file `function.c` of this demo project.

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  6c 01 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |l.......4.....(.|
00000030  0d 00 0a 00 55 89 e5 83  ec 14 d9 45 08 d8 45 0c  |....U......E..E.|
00000040  d9 5d fc 8b 45 fc 89 45  ec d9 45 ec c9 c3 55 89  |.]..E..E..E...U.|
00000050  e5 83 ec 1c 8b 45 0c 89  44 24 04 8b 45 08 89 04  |.....E..D$..E...|
00000060  24 e8 fc ff ff ff d9 5d  fc d9 45 fc d9 05 00 00  |$.......]..E.....|
00000070  00 00 de c9 d9 5d fc 8b  45 fc 89 45 ec d9 45 ec  |.....]..E..E..E.|
00000080  c9 c3 00 00 00 00 40 40  00 47 43 43 3a 20 28 55  |......@@.GCC: (U|
00000090  62 75 6e 74 75 2f 4c 69  6e 61 72 6f 20 34 2e 36  |buntu/Linaro 4.6|
000000a0  2e 33 2d 31 75 62 75 6e  74 75 35 29 20 34 2e 36  |.3-1ubuntu5) 4.6|
000000b0  2e 33 00 00 14 00 00 00  00 00 00 00 01 7a 52 00  |.3...........zR.|
000000c0  01 7c 08 01 1b 0c 04 04  88 01 00 00 1c 00 00 00  |.|..............|
000000d0  1c 00 00 00 00 00 00 00  1a 00 00 00 00 41 0e 08  |.............A..|
000000e0  85 02 42 0d 05 56 c5 0c  04 04 00 00 1c 00 00 00  |..B..V..........|
000000f0  3c 00 00 00 1a 00 00 00  34 00 00 00 00 41 0e 08  |<.......4....A..|
00000100  85 02 42 0d 05 70 c5 0c  04 04 00 00 00 2e 73 79  |..B..p........sy|
00000110  6d 74 61 62 00 2e 73 74  72 74 61 62 00 2e 73 68  |mtab..strtab..sh|
00000120  73 74 72 74 61 62 00 2e  72 65 6c 2e 74 65 78 74  |strtab..rel.text|
00000130  00 2e 64 61 74 61 00 2e  62 73 73 00 2e 72 6f 64  |..data..bss..rod|
00000140  61 74 61 00 2e 63 6f 6d  6d 65 6e 74 00 2e 6e 6f  |ata..comment..no|
00000150  74 65 2e 47 4e 55 2d 73  74 61 63 6b 00 2e 72 65  |te.GNU-stack..re|
00000160  6c 2e 65 68 5f 66 72 61  6d 65 00 00 00 00 00 00  |l.eh_frame......|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000190  00 00 00 00 1f 00 00 00  01 00 00 00 06 00 00 00  |................|
000001a0  00 00 00 00 34 00 00 00  4e 00 00 00 00 00 00 00  |....4...N.......|
000001b0  00 00 00 00 04 00 00 00  00 00 00 00 1b 00 00 00  |................|
000001c0  09 00 00 00 00 00 00 00  00 00 00 00 68 04 00 00  |............h...|
000001d0  10 00 00 00 0b 00 00 00  01 00 00 00 04 00 00 00  |................|
000001e0  08 00 00 00 25 00 00 00  01 00 00 00 03 00 00 00  |....%...........|
000001f0  00 00 00 00 84 00 00 00  00 00 00 00 00 00 00 00  |................|
00000200  00 00 00 00 04 00 00 00  00 00 00 00 2b 00 00 00  |............+...|
00000210  08 00 00 00 03 00 00 00  00 00 00 00 84 00 00 00  |................|
00000220  04 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |................|
00000230  00 00 00 00 30 00 00 00  01 00 00 00 02 00 00 00  |....0...........|
00000240  00 00 00 00 84 00 00 00  04 00 00 00 00 00 00 00  |................|
00000250  00 00 00 00 04 00 00 00  00 00 00 00 38 00 00 00  |............8...|
00000260  01 00 00 00 30 00 00 00  00 00 00 00 88 00 00 00  |....0...........|
00000270  2b 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |+...............|
00000280  01 00 00 00 41 00 00 00  01 00 00 00 00 00 00 00  |....A...........|
00000290  00 00 00 00 b3 00 00 00  00 00 00 00 00 00 00 00  |................|
000002a0  00 00 00 00 01 00 00 00  00 00 00 00 55 00 00 00  |............U...|
000002b0  01 00 00 00 02 00 00 00  00 00 00 00 b4 00 00 00  |................|
000002c0  58 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |X...............|
000002d0  00 00 00 00 51 00 00 00  09 00 00 00 00 00 00 00  |....Q...........|
000002e0  00 00 00 00 78 04 00 00  10 00 00 00 0b 00 00 00  |....x...........|
```

Figure 2-2. *Binary contents of an object file*

Obviously, merely taking a look at the hex values of the object file does not tell us a whole lot. The disassembling procedure has the potential to tell us far more.

The Linux tool called **objdump** (part of popular *binutils* package) specializes in disassembling the binary files, among a whole lot of other things. In addition to converting the sequence of binary machine instructions specific to a given platform, it also specifies the addresses at which the instructions reside.

It should not be a huge surprise that it supports both AT&T (default) as well as Intel flavors of printing the assembler code.

By running the simple form of objdump command,

```
$ objdump -D <input file>.o
```

you get the following contents printed on the terminal screen:

disassembled output of function.o (AT&T assembler format)

```
function.o:     file format elf32-i386

Disassembly of section .text:

00000000 <add>:
   0:  55                      push   %ebp
   1:  89 e5                   mov    %esp,%ebp
   3:  83 ec 14                sub    $0x14,%esp
   6:  d9 45 08                flds   0x8(%ebp)
   9:  d8 45 0c                fadds  0xc(%ebp)
   c:  d9 5d fc                fstps  -0x4(%ebp)
   f:  8b 45 fc                mov    -0x4(%ebp),%eax
  12:  89 45 ec                mov    %eax,-0x14(%ebp)
  15:  d9 45 ec                flds   -0x14(%ebp)
  18:  c9                      leave
  19:  c3                      ret

0000001a <add_and_multiply>:
  1a:  55                      push   %ebp
  1b:  89 e5                   mov    %esp,%ebp
  1d:  83 ec 1c                sub    $0x1c,%esp
  20:  8b 45 0c                mov    0xc(%ebp),%eax
  23:  89 44 24 04             mov    %eax,0x4(%esp)
  27:  8b 45 08                mov    0x8(%ebp),%eax
  2a:  89 04 24                mov    %eax,(%esp)
  2d:  e8 fc ff ff ff          call   2e <add_and_multiply+0x14>
  32:  d9 5d fc                fstps  -0x4(%ebp)
  35:  d9 45 fc                flds   -0x4(%ebp)
  38:  d9 05 00 00 00 00       flds   0x0
  3e:  de c9                   fmulp  %st,%st(1)
  40:  d9 5d fc                fstps  -0x4(%ebp)
  43:  8b 45 fc                mov    -0x4(%ebp),%eax
  46:  89 45 ec                mov    %eax,-0x14(%ebp)
  49:  d9 45 ec                flds   -0x14(%ebp)
  4c:  c9                      leave
  4d:  c3                      ret
```

```
Disassembly of section .bss:

00000000 <nCompletionStatus>:
   0:  00 00                   add    %al,(%eax)
       ...

Disassembly of section .rodata:

00000000 <.rodata>:
   0:  00 00                   add    %al,(%eax)
   2:  40                      inc    %eax
   3:  40                      inc    %eax

Disassembly of section .comment:

00000000 <.comment>:
   0:  00 47 43                add    %al,0x43(%edi)
   3:  43                      inc    %ebx
   4:  3a 20                   cmp    (%eax),%ah
   6:  28 55 62                sub    %dl,0x62(%ebp)
   9:  75 6e                   jne    79 <add_and_multiply+0x5f>
   b:  74 75                   je     82 <add_and_multiply+0x68>
   d:  2f                      das
   e:  4c                      dec    %esp
   f:  69 6e 61 72 6f 20 34    imul   $0x34206f72,0x61(%esi),%ebp
  16:  2e 36 2e 33 2d 31 75    cs ss xor %cs:%ss:0x75627531,%ebp
  1d:  62 75
  1f:  6e                      outsb  %ds:(%esi),(%dx)
  20:  74 75                   je     97 <add_and_multiply+0x7d>
  22:  35 29 20 34 2e          xor    $0x2e342029,%eax
  27:  36 2e 33 00             ss xor %cs:%ss:(%eax),%eax

Disassembly of section .eh_frame:

00000000 <.eh_frame>:
   0:  14 00                   adc    $0x0,%al
   2:  00 00                   add    %al,(%eax)
   4:  00 00                   add    %al,(%eax)
   6:  00 00                   add    %al,(%eax)
   8:  01 7a 52                add    %edi,0x52(%edx)
   b:  00 01                   add    %al,(%ecx)
   d:  7c 08                   jl     17 <.eh_frame+0x17>
   f:  01 1b                   add    %ebx,(%ebx)
  11:  0c 04                   or     $0x4,%al
  13:  04 88                   add    $0x88,%al
  15:  01 00                   add    %eax,(%eax)
  17:  00 1c 00                add    %bl,(%eax,%eax,1)
  1a:  00 00                   add    %al,(%eax)
  1c:  1c 00                   sbb    $0x0,%al
  1e:  00 00                   add    %al,(%eax)
  20:  00 00                   add    %al,(%eax)
```

```
  22:  00 00                    add    %al,(%eax)
  24:  1a 00                    sbb    (%eax),%al
  26:  00 00                    add    %al,(%eax)
  28:  00 41 0e                 add    %al,0xe(%ecx)
  2b:  08 85 02 42 0d 05        or     %al,0x50d4202(%ebp)
  31:  56                       push   %esi
  32:  c5 0c 04                 lds    (%esp,%eax,1),%ecx
  35:  04 00                    add    $0x0,%al
  37:  00 1c 00                 add    %bl,(%eax,%eax,1)
  3a:  00 00                    add    %al,(%eax)
  3c:  3c 00                    cmp    $0x0,%al
  3e:  00 00                    add    %al,(%eax)
  40:  1a 00                    sbb    (%eax),%al
  42:  00 00                    add    %al,(%eax)
  44:  34 00                    xor    $0x0,%al
  46:  00 00                    add    %al,(%eax)
  48:  00 41 0e                 add    %al,0xe(%ecx)
  4b:  08 85 02 42 0d 05        or     %al,0x50d4202(%ebp)
  51:  70 c5                    jo     18 <.eh_frame+0x18>
  53:  0c 04                    or     $0x4,%al
  55:  04 00                    add    $0x0,%al
       ...
```

Similarly, by specifying the Intel flavor,

```
$ objdump -D -M intel <input file>.o
```

you get the following contents printed on the terminal screen:

disassembled output of function.o (Intel assembler format)

```
function.o:     file format elf32-i386

Disassembly of section .text:

00000000 <add&gt:
   0:  55                       push   ebp
   1:  89 e5                    mov    ebp,esp
   3:  83 ec 14                 sub    esp,0x14
   6:  d9 45 08                 fld    DWORD PTR [ebp+0x8]
   9:  d8 45 0c                 fadd   DWORD PTR [ebp+0xc]
   c:  d9 5d fc                 fstp   DWORD PTR [ebp-0x4]
   f:  8b 45 fc                 mov    eax,DWORD PTR [ebp-0x4]
  12:  89 45 ec                 mov    DWORD PTR [ebp-0x14],eax
  15:  d9 45 ec                 fld    DWORD PTR [ebp-0x14]
  18:  c9                       leave
  19:  c3                       ret
```

```
0000001a <add_and_multiply>:
  1a:  55                      push   ebp
  1b:  89 e5                   mov    ebp,esp
  1d:  83 ec 1c                sub    esp,0x1c
  20:  8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
  23:  89 44 24 04             mov    DWORD PTR [esp+0x4],eax
  27:  8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
  2a:  89 04 24                mov    DWORD PTR [esp],eax
  2d:  e8 fc ff ff ff          call   2e <add_and_multiply+0x14>
  32:  d9 5d fc                fstp   DWORD PTR [ebp-0x4]
  35:  d9 45 fc                fld    DWORD PTR [ebp-0x4]
  38:  d9 05 00 00 00 00       fld    DWORD PTR ds:0x0
  3e:  de c9                   fmulp  st(1),st
  40:  d9 5d fc                fstp   DWORD PTR [ebp-0x4]
  43:  8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
  46:  89 45 ec                mov    DWORD PTR [ebp-0x14],eax
  49:  d9 45 ec                fld    DWORD PTR [ebp-0x14]
  4c:  c9                      leave
  4d:  c3                      ret

Disassembly of section .bss:

00000000 <nCompletionStatus>:
   0:  00 00                   add    BYTE PTR [eax],al
       ...

Disassembly of section .rodata:

00000000 <.rodata>:
   0:  00 00                   add    BYTE PTR [eax],al
   2:  40                      inc    eax
   3:  40                      inc    eax

Disassembly of section .comment:

00000000 <.comment>:
   0:  00 47 43                add    BYTE PTR [edi+0x43],al
   3:  43                      inc    ebx
   4:  3a 20                   cmp    ah,BYTE PTR [eax]
   6:  28 55 62                sub    BYTE PTR [ebp+0x62],dl
   9:  75 6e                   jne    79 <add_and_multiply+0x5f>
   b:  74 75                   je     82 <add_and_multiply+0x68>
   d:  2f                      das
   e:  4c                      dec    esp
   f:  69 6e 61 72 6f 20 34    imul   ebp,DWORD PTR [esi+0x61],0x34206f72
  16:  2e 36 2e 33 2d 31 75    cs ss xor ebp,DWORD PTR cs:ss:0x75627531
  1d:  62 75
  1f:  6e                      outs   dx,BYTE PTR ds:[esi]
  20:  74 75                   je     97 <add_and_multiply+0x7d>
  22:  35 29 20 34 2e          xor    eax,0x2e342029
  27:  36 2e 33 00             ss xor eax,DWORD PTR cs:ss:[eax]
```

```
Disassembly of section .eh_frame:

00000000 <.eh_frame>:
   0:  14 00                   adc    al,0x0
   2:  00 00                   add    BYTE PTR [eax],al
   4:  00 00                   add    BYTE PTR [eax],al
   6:  00 00                   add    BYTE PTR [eax],al
   8:  01 7a 52                add    DWORD PTR [edx+0x52],edi
   b:  00 01                   add    BYTE PTR [ecx],al
   d:  7c 08                   jl     17 <.eh_frame+0x17>
   f:  01 1b                   add    DWORD PTR [ebx],ebx
  11:  0c 04                   or     al,0x4
  13:  04 88                   add    al,0x88
  15:  01 00                   add    DWORD PTR [eax],eax
  17:  00 1c 00                add    BYTE PTR [eax+eax*1],bl
  1a:  00 00                   add    BYTE PTR [eax],al
  1c:  1c 00                   sbb    al,0x0
  1e:  00 00                   add    BYTE PTR [eax],al
  20:  00 00                   add    BYTE PTR [eax],al
  22:  00 00                   add    BYTE PTR [eax],al
  24:  1a 00                   sbb    al,BYTE PTR [eax]
  26:  00 00                   add    BYTE PTR [eax],al
  28:  00 41 0e                add    BYTE PTR [ecx+0xe],al
  2b:  08 85 02 42 0d 05       or     BYTE PTR [ebp+0x50d4202],al
  31:  56                      push   esi
  32:  c5 0c 04                lds    ecx,FWORD PTR [esp+eax*1]
  35:  04 00                   add    al,0x0
  37:  00 1c 00                add    BYTE PTR [eax+eax*1],bl
  3a:  00 00                   add    BYTE PTR [eax],al
  3c:  3c 00                   cmp    al,0x0
  3e:  00 00                   add    BYTE PTR [eax],al
  40:  1a 00                   sbb    al,BYTE PTR [eax]
  42:  00 00                   add    BYTE PTR [eax],al
  44:  34 00                   xor    al,0x0
  46:  00 00                   add    BYTE PTR [eax],al
  48:  00 41 0e                add    BYTE PTR [ecx+0xe],al
  4b:  08 85 02 42 0d 05       or     BYTE PTR [ebp+0x50d4202],al
  51:  70 c5                   jo     18 <.eh_frame+0x18>
  53:  0c 04                   or     al,0x4
  55:  04 00                   add    al,0x0
       ...
```

Object File Properties

The output of the compilation process is one or more binary object files, whose structure is the natural next topic of interest. As you will see shortly, the structure of object files contains many details of importance on the path of truly understanding the broader picture.

In a rough sketch,

- An object file is the result of translating its original corresponding source file. The result of compilation is the collection of as many object files as there are source files in the project.

  After the compiling completes, the object file keeps representing its original source file in subsequent stages of the program building process.

- The basic ingredients of an object file are the **symbols** (references to the memory addresses in program or data memory) as well as the **sections**.

  Among the sections most frequently found in the object files are the code (.text), initialized data (.data), uninitialized data (.bss), and some of the more specialized sections (debugging information, etc.).

- The ultimate intention behind the idea of building the program is that the sections obtained by compiling individual source files be combined (tiled) together into the single binary executable file.

  Such binary file would contain the sections of the same type (.text, .data, .bss, . . .) obtained by tiling together the sections from the individual files. Figuratively speaking, an object file can be viewed as a simple tile waiting to find its place in the giant mosaic of the process memory map.

- The inner structure of the object file does not, however, suggest where the individual sections will ultimately reside in the program memory map. For that reason, the address ranges of each section in each of the object files is tentatively set to start from a zero value.

  The actual address range at which a section from an object file will ultimately reside in the program map will be determined in the subsequent stages (linking) of program building process.

- In the process of tiling object files' sections into the resultant program memory map, the only truly important parameter is the length of its sections, or to say it more precisely, its address range.

- The object file carries no sections that would contribute to the stack and/or heap. The contents of these two sections of the memory map are completely determined at runtime, and other than the default byte length, require no program-specific initial settings.

- The object file's contribution to the program's .bss (uninitialized data) section is very rudimentary; the .bss section is described merely by its byte length. This meager information is just what is needed for the loader to establish the .bss section as a part of the memory in which some data will be stored.

In general, the information is stored in the object files according to a certain set of rules epitomized in the form of binary format specification, whose details vary across the different platforms (Windows vs. Linux, 32-bit vs. 64-bit, x86 vs. ARM processor family).

Typically, the binary format specifications are designed to support the C/C++ language constructs and the associated implementation problems. Very frequently, the binary format specification covers a variety of binary file modes such as executables, static libraries, and dynamic libraries.

On Linux, the Executable and Linkable Format (ELF) has gained the prevalence. On Windows, the binaries typically conform to the PE/COFF format specification.

Compilation Process Limitations

Step by step, the pieces of the gigantic puzzle of program building process are starting to fall in place, and the broad and clear picture of the whole story slowly emerges. So far, you've learned that the compilation process translates the ASCII source files into the corresponding collection of binary object files. Each of the object files contains sections, the destiny of each is to ultimately become a part of gigantic puzzle of the program's memory map, as illustrated in Figure 2-3.

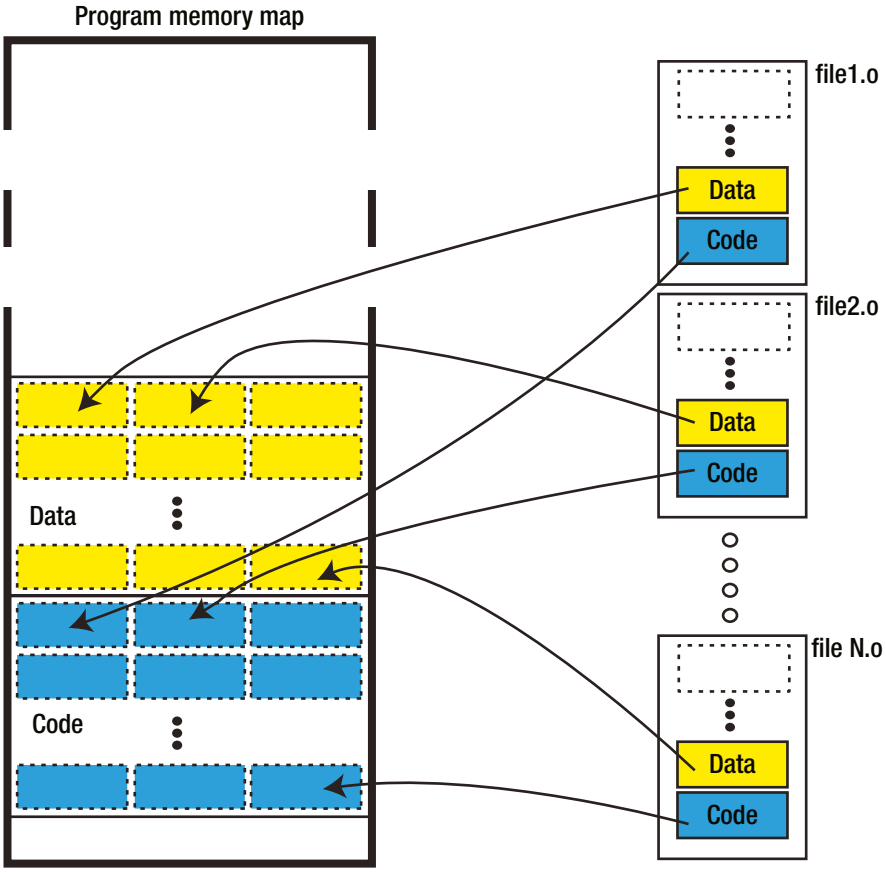

Figure 2-3. *Tiling the individual sections into the final program memory map*

The task that remains is to *tile the individual sections* stored across individual object files together into the body of program memory map. As mentioned in the previous sections, that task needs to be left to another stage of the program building process called **linking**.

The question that a careful observer can't help but asking (before going into the details of linking procedure) is *exactly why do we need a whole new stage of the building process*, or more precisely, *exactly why can't the compilation process described thus far complete the tiling part of the task?*

There are a few very solid reasons for splitting the build procedure, and the rest of this section will try to clarify the circumstances leading to such decision.

In short, the answer can be provided in a few simple statements. First, combining the sections together (especially the code sections) is not always simple. This factor definitely plays certain role, but is not sufficient; there are many programming languages whose program building process can be completed in one step (in other words, they do not require dividing the procedure into the two stages).

Second, the code reuse principle applied to the process of program building (and the ability to combine together the binary parts coming from various projects) definitely affirmed the decision to implement the C/C++ building as a two-step (compiling and linking) procedure.

What Makes Section Combining so Complicated?

For the most part, the translation of source code into binary object files is a fairly simple process. The lines of code are translated into processor-specific machine code instructions; the space for initialized variables is reserved and initial values are written to it; the space for uninitialized variables is reserved and filled out with zeros, etc.

However, there is a part of the whole story which is bound to cause some problems: even though the source code is grouped into the dedicated source files, being part of the same program implies that certain mutual connections must exist. Indeed, the connections between the distinct parts of the code are typically established through either the following two options:

- *Function calls between functionally separate bodies of code*:

  For example, a function in the GUI-related source file of a chat application may call a function in the TCP/IP networking source file, which in turn may call a function located in the encryption source file.

- *External variables*:

  In the domain of the C programming language (substantially less in the C++ domain), it was a usual practice to reserve globally visible variables to maintain the state of interest for various parts of code. A variable intended for broader use is typically declared in one source file as global variable, and referenced from all other source files as extern variable.

  A typical example is the errno variable used in standard C libraries to keep the value of the last encountered error.

In order to access either of the two (which are commonly referred to as **symbols**), their addresses (more precisely, the function's address in the program memory and/or the global variable's address in data memory) must be known.

However, the actual address cannot be known before the individual sections are incorporated into the corresponding program section (i.e., before the section tiling is completed!!!). Until then, a meaningful connection between a function and its caller and/or access to the external variable is impossible to establish, which are both suitably reported as unresolved references. Please notice that this problem does not happen when the function or global variable is referenced from the same source file in which it was defined. In this particular case, both the function/variable and their caller/user end up being the part of the same section, and their positions relative to each other are known before the "grand puzzle completion." In such cases, as soon as the tiling of the sections is completed, the relative memory addresses become concrete and usable.

As mentioned earlier in this section, solving this kind of problem still does not mandate that a build procedure must be divided into two distinct stages. As a matter of fact, many different languages successfully implement a one-pass build procedure. However, the concept of reusing (binary reusing in this case) applied to the realm of building the program (and the concept of libraries) ultimately confirms the decision to split the program building into the two stages (compiling and linking).

Linking

The second stage of the program building process is **linking**. The input to the linking process is the collection of object files created by the previously completed compiling stage. Each object file can be viewed as binary storage of individual source file contributions to the program memory map sections of all kinds (code, initialized data, uninitialized data, debugging information, etc.). The ultimate task of the linker is to form the resultant program memory map section out of individual contributions and to resolve all the references. As a reminder, the concept of virtual memory simplified the task of linker inasmuch as allowing it to assume that the program memory map that the linker needs to populate is a zero-based address range of identical size for each and every program, regardless of what address range the process will be given by the operating system at runtime.

For the sake of simplicity, I will cover in this example the simplest possible case, in which the contributions to the program memory map sections come solely from the files belonging to the same project. In reality, due to advancement of binary reuse concept, this may not be true.

Linking Stages

The linking process happens through a sequence of stages (relocation, reference resolving), which will be discussed in detail next.

Relocation

The first stage of a linking procedure is nothing else than tiling, a process in which sections of various kinds contained in individual object files are combined together to create the program memory map sections (see Figure 2-4). In order to complete this task, the previously neutral, zero-based address ranges of contributing sections get translated into the more concrete address ranges of resultant program memory map.

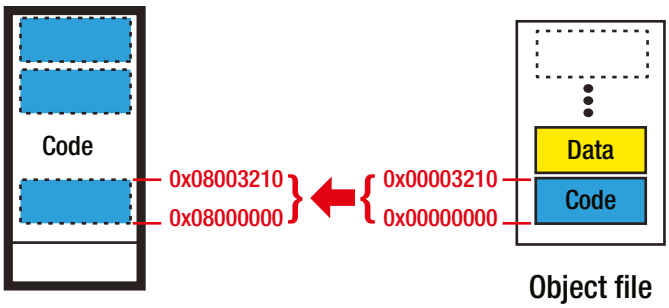

Figure 2-4. *Relocation, the first phase of the linking stage*

The wording "more concrete" is used to emphasize the fact that the resultant program image created by the linker is still neutral by itself. Remember, the mechanism of virtual addressing makes it possible that each and every program has the same, identical, simple view of the program address space (which resides between 0 and 2^N), whereas the real physical address at which the program executes gets determined at runtime by the operating system, invisible to the program and programmer.

Once the relocation stage completes, most (but not all!) of the program memory map has been created.

Resolving References

Now comes the hard part. Taking sections, linearly translating their address ranges into the program memory map address ranges was fairly easy task. A much harder task is to establish the required connections between the various parts of the code, thus making the program homogenous.

Let's assume (rightfully so, given the simplicity of this demo program) that all the previous build stages (complete compilation as well as section relocation) have completed successfully. Now is the moment to point out exactly which kinds of problems are left for the last linking stage to resolve.

As mentioned earlier, the root cause of linking problems is fairly simple: pieces of code originated from different translation units (i.e., source files) and are trying to reference each other, but cannot possibly know where in memory these items will reside up until the object files are tiled into the body of program memory map. The components of the code that cause the most problems are the ones tightly bound to the address in either program memory (function entry points) or in data memory (global/static/extern) variables.

In this particular code example, you have the following situation:

- The function `add_and_multiply` calls the function `add`, which resides in the same source file (i.e., the same translation unit in the same object file). In this case, the address in the program memory of function `add()` is to some extent a known quantity and can be expressed by its relative offset of the code section of the object file `function.o`.
- Now function `main` calls function `add_and_multiply` and also references the extern variable `nCompletionStatus` and has huge problems figuring out the actual program memory address at which they reside. In fact, it only may assume that both of these *symbols* will at some point in the future reside *somewhere* in the process memory map. But, until the memory map is formed, two items cannot be considered as nothing else than *unresolved references*.

The situation is graphically described in Figure 2-5.

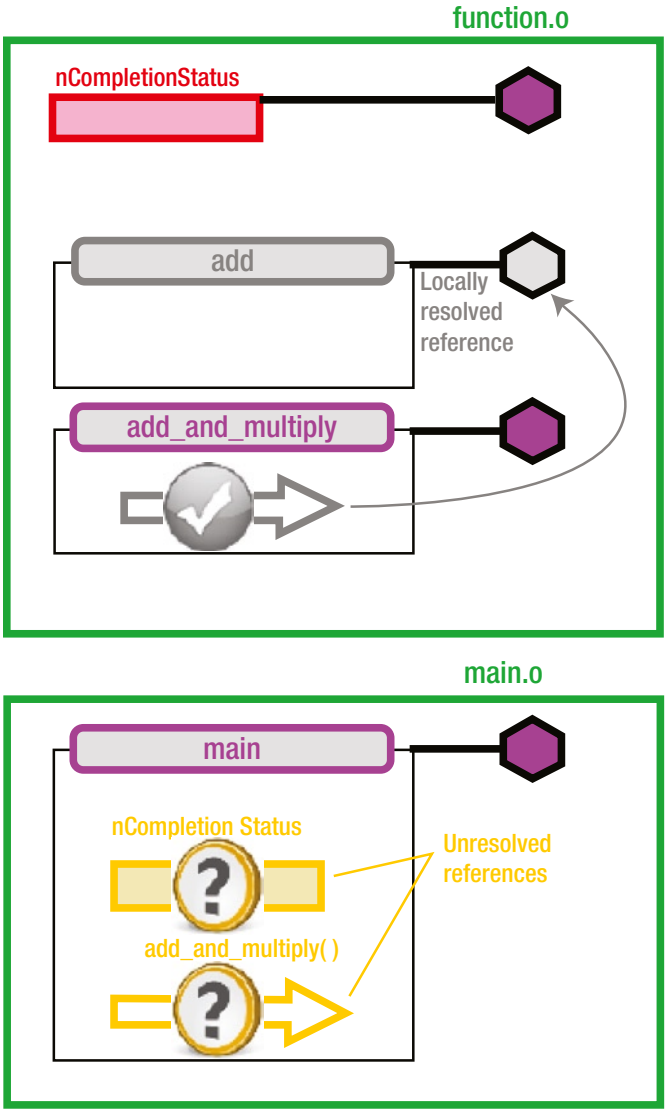

Figure 2-5. *The problem of unresolved references in its essential form*

In order to solve these kinds of problems, a linking stage of resolving the references must happen. What linker needs to do in this situation is to

- Examine the sections already tiled together in the program memory map.
- Find out which part of the code makes calls outside of its original section.
- Figure out where exactly (at which address in the memory map) the referenced part of the code resides.
- And finally, resolve the references by replacing dummy addresses in the machine instructions with the actual addresses of the program memory map.

Once the linker completes its magic, the situation may look like Figure 2-6.

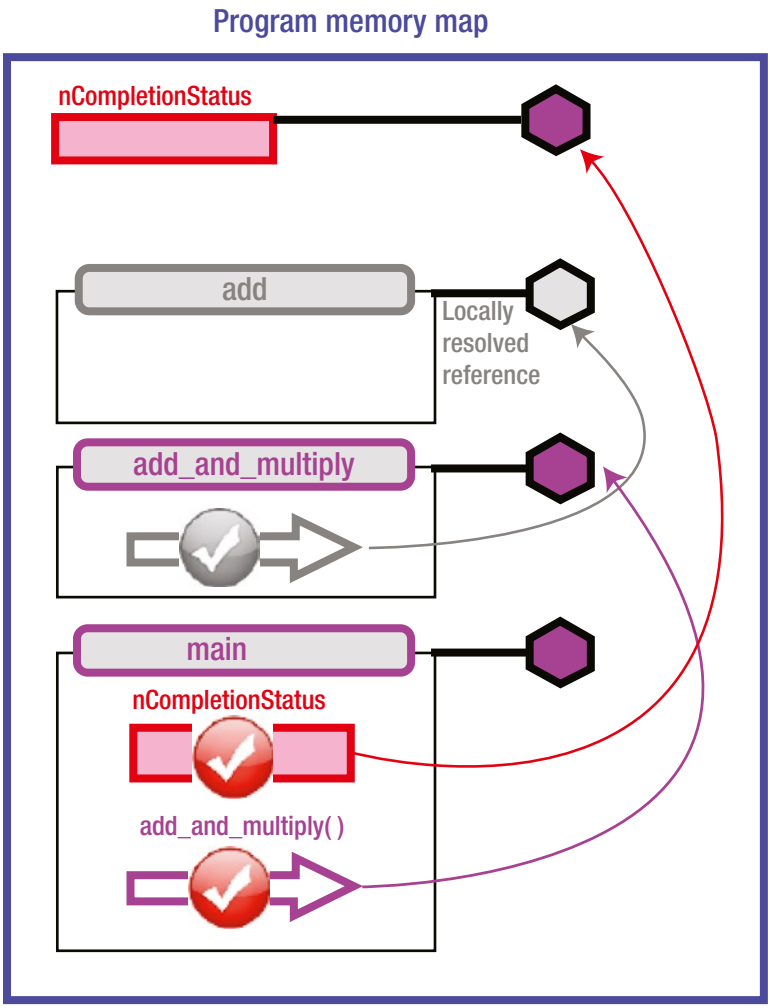

Figure 2-6. *Resolved references*

Demo Project Linking Example

There are two ways to compile and link the complete demo project to create the executable file so that it's ready for running.

In the step-by-step approach, you will first invoke the compiler on both of the source files to produce the object files. In the subsequent step, you will link both object files into the output executable.

```
$ gcc -c function.c main.c
$ gcc function.o main.o -o demoApp
```

In the all-at-once approach, the same operation may be completed by invoking the compiler and linker with just one command.

```
$ gcc function.c main.c -o demoApp
```

For the purposes of this demo, let's take the step-by-step approach, as it will generate the `main.o` object file, which contains very important details that I want to demonstrate here.

The disassembling of the file `main.o`,

```
$ objdump -D -M intel main.o
```

reveals that it contains unresolved references.

disassembled output of main.o (Intel assembler format)

```
main.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:  55                    push   ebp
   1:  89 e5                 mov    ebp,esp
   3:  83 e4 f0              and    esp,0xfffffff0
   6:  83 ec 20              sub    esp,0x20
   9:  b8 00 00 80 3f        mov    eax,0x3f800000
   e:  89 44 24 14           mov    DWORD PTR [esp+0x14],eax
  12:  b8 00 00 a0 40        mov    eax,0x40a00000
  17:  89 44 24 18           mov    DWORD PTR [esp+0x18],eax
  1b:  8b 44 24 18           mov    eax,DWORD PTR [esp+0x18]
  1f:  89 44 24 04           mov    DWORD PTR [esp+0x4],eax
  23:  8b 44 24 14           mov    eax,DWORD PTR [esp+0x14]
  27:  89 04 24              mov    DWORD PTR [esp],eax
  2a:  e8 fc ff ff ff        call   2b  <main + 0x2b>
  2f:  d9 5c 24 1c           fstp   DWORD PTR [esp+0x1c]
  33:  c7 05 00 00 00 00 01  mov    DWORD PTR ds:0x0,0x1
  3a:  00 00 00
  3d:  b8 00 00 00 00        mov    eax,0x0
  42:  c9                    leave
  43:  c3                    ret   :
```

Line 2a features a call instruction that jumps to itself (strange, isn't it?) whereas line 33 features the access of the variable residing at the address 0x0 (even more strange). Obviously, these two obviously strange values were inserted by the linker purposefully.

The disassembled output of the output executable, however, shows that not only the contents of the `main.o` object file have been relocated to the address range starting at the address 0x08048404, but also these two troubled spots have been resolved by the linker.

```
$ objdump -D -M intel demoApp
```

disassembled output of demoApp (Intel assembler format)

```
080483ce <add_and_multiply>:
 80483ce:       55                          push   ebp
 80483cf:       89 e5                       mov    ebp,esp
 80483d1:       83 ec 1c                    sub    esp,0x1c
 80483d4:       8b 45 0c                    mov    eax,DWORD PTR [ebp+0xc]
 80483d7:       89 44 24 04                 mov    DWORD PTR [esp+0x4],eax
 80483db:       8b 45 08                    mov    eax,DWORD PTR [ebp+0x8]
 80483de:       89 04 24                    mov    DWORD PTR [esp],eax
 80483e1:       e8 ce ff ff ff              call   80483b4 <add>
 80483e6:       d9 5d fc                    fstp   DWORD PTR [ebp-0x4]
 80483e9:       d9 45 fc                    fld    DWORD PTR [ebp-0x4]
 80483ec:       d9 05 20 85 04 08           fld    DWORD PTR ds:0x8048520
 80483f2:       de c9                       fmulp  st(1),st
 80483f4:       d9 5d fc                    fstp   DWORD PTR [ebp-0x4]
 80483f7:       8b 45 fc                    mov    eax,DWORD PTR [ebp-0x4]
 80483fa:       89 45 ec                    mov    DWORD PTR [ebp-0x14],eax
 80483fd:       d9 45 ec                    fld    DWORD PTR [ebp-0x14]
 8048400:       c9                          leave
 8048401:       c3                          ret
 8048402:       90                          nop
 8048403:       90                          nop

08048404 <main>:
 8048404:       55                          push   ebp
 8048405:       89 e5                       mov    ebp,esp
 8048407:       83 e4 f0                    and    esp,0xfffffff0
 804840a:       83 ec 20                    sub    esp,0x20
 804840d:       b8 00 00 80 3f              mov    eax,0x3f800000
 8048412:       89 44 24 14                 mov    DWORD PTR [esp+0x14],eax
 8048416:       b8 00 00 a0 40              mov    eax,0x40a00000
 804841b:       89 44 24 18                 mov    DWORD PTR [esp+0x18],eax
 804841f:       8b 44 24 18                 mov    eax,DWORD PTR [esp+0x18]
 8048423:       89 44 24 04                 mov    DWORD PTR [esp+0x4],eax
 8048427:       8b 44 24 14                 mov    eax,DWORD PTR [esp+0x14]
 804842b:       89 04 24                    mov    DWORD PTR [esp],eax
 804842e:       e8 9b ff ff ff              call   80483ce <add_and_multiply>
 8048433:       d9 5c 24 1c                 fstp   DWORD PTR [esp+0x1c]
 8048437:       c7 05 18 a0 04 08 01        mov    DWORD PTR ds:0x804a018,0x1
 804843e:       00 00 00
 8048441:       b8 00 00 00 00              mov    eax,0x0
 8048446:       c9                          leave  t:
```

The line at the memory map address 0x8048437 references the variable residing at address 0x804a018. The only open question now is what resides at that particular address?

The versatile objdump tool may help you get the answer to that question (a decent part of subsequent chapters is dedicated to this exceptionally useful tool).

By running the following command

```
$ objdump -x -j .bss demoApp
```

you can disassemble the .bss section carrying the uninitialized data, which reveals that your variable `nCompletionStatus` resides exactly at the address 0x804a018, as shown in Figure 2-7.

```
milan@milan$ objdump -x -j .bss demoApp
                                        o
                                        o
SYMBOL TABLE:                           o
0804a010 l    d  .bss   00000000              .bss
0804a010 l     O .bss   00000001              completed.6159
0804a014 l     O .bss   00000004              dtor_idx.6161
0804a018 g     O .bss   00000004              nCompletionStatus
```

Figure 2-7. *bss disassembled*

Linker's Viewpoint

"When you've got a hammer in your hand, everything looks like a nail"

—Handy Hammer Syndrome

But seriously, folks....

Now that you know the intricacies of the linking task, it helps to zoom out a little bit and try to summarize the philosophy that guides the linker while running its usual tasks. As a matter of fact, the linker is a specific tool, which, unlike its older brother the compiler, is not interested in the minute details of the written code. Instead, it views the world as a set of object files that (much like puzzle pieces) are about to be combined together in a wider picture of program memory map, as illustrated by Figure 2-8.

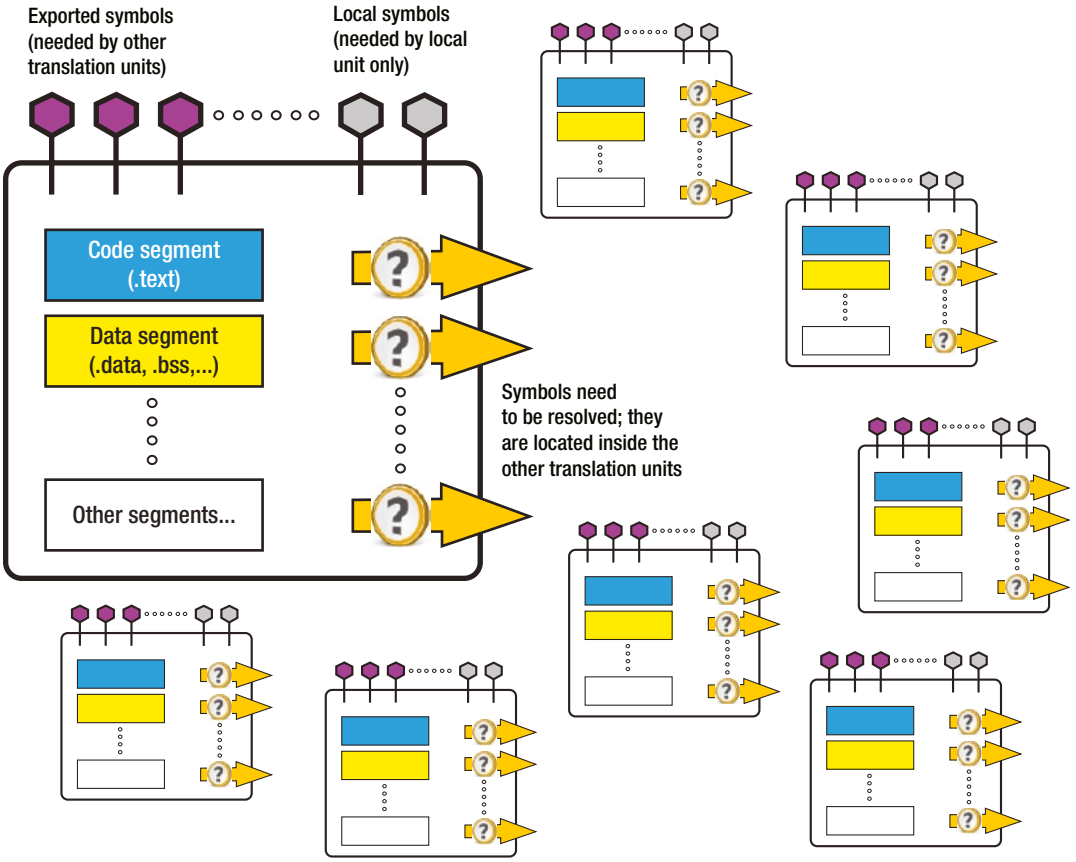

Figure 2-8. *The linker's view of the world*

It does not take a whole lot of imagination to find out that Figure 2-8 has a lot of resemblance with the left part of Figure 2-9, whereas the linker's ultimate task could be represented by the right part of the same figure.

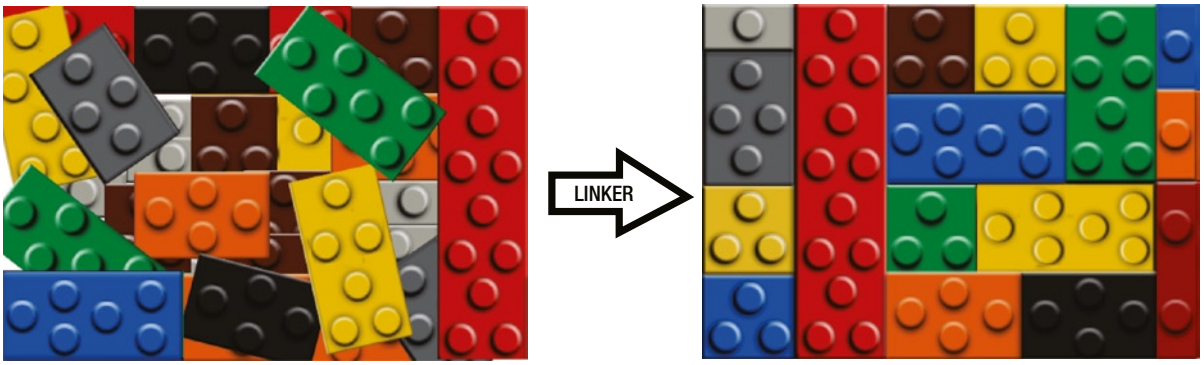

Figure 2-9. *Linker's view of the world as seen by humans*

Executable File Properties

The ultimate result of the linking procedure is the binary executable file, whose layout follows the rules of the executable format suitable for the target platform. Regardless of the actual format differences, the executable file typically contains the resultant sections (.text, .data, .bss, and many more narrowly specialized ones) created by combining the contributions from individual object files. Most notably, the code (.text) section not only contains the individual tiles from object files, but the linker modified it to make sure that all the references between the individual tiles have been resolved, so that the function calls between different parts of code as well as variable accesses are accurate and meaningful.

Among all the symbols contained in the executable file, a very unique place belongs to the `main` function, as from the standpoint of C/C++ programs it is the function from which the entire program execution starts. However, this is not the very first part of code that executes when the program starts.

An exceptionally important detail that needs to be pointed out is that the executable file is not entirely made of code compiled from the project source files. As a matter of fact, a strategically important piece of code responsible for starting the program execution is added at the linking stage to the program memory map. This object code, which linker typically stores at the beginning of the program memory map, comes in two variants:

- **crt0** is the "plain vanilla" entry point, the first part of program code that gets executed under the control of kernel.
- **crt1** is the more modern startup routine with support for tasks to be completed before the `main` function gets executed and after the program terminates.

Having these details in mind, the overall structure of the program executable may be symbolically represented by Figure 2-10.

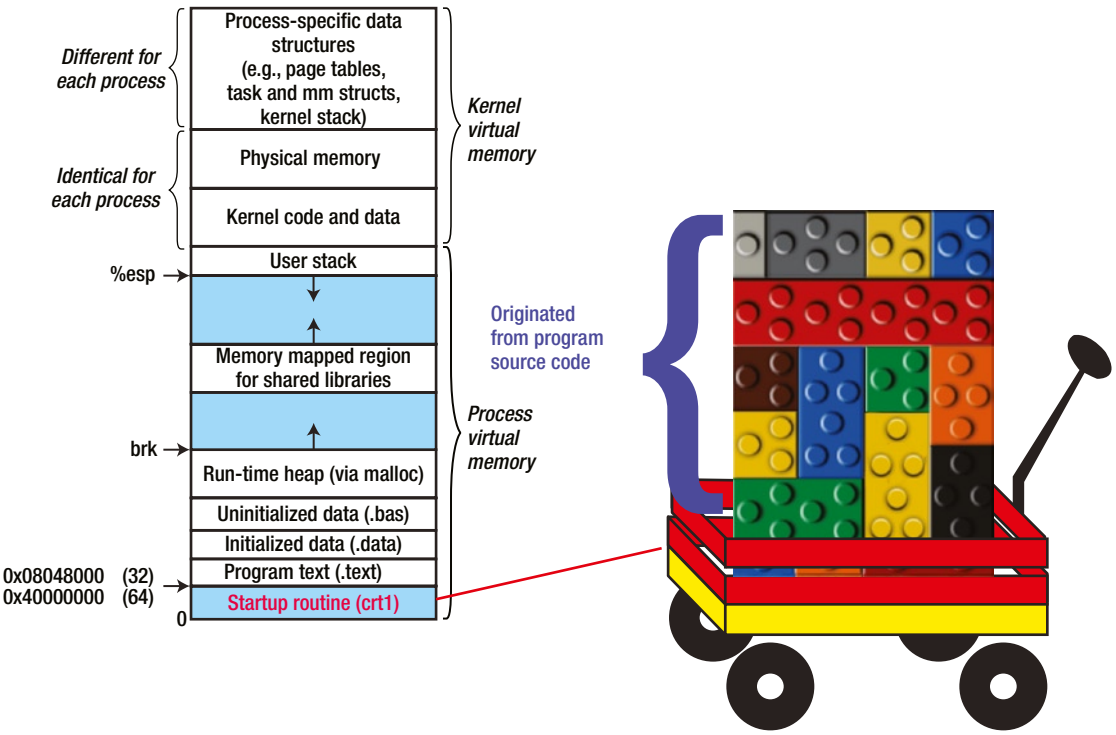

Figure 2-10. *Overall structure of an executable file*

As you will see later in the chapter devoted to dynamic libraries and dynamic linking, this extra piece of code, which gets provided by the operating system, makes almost all the difference between an executable and a dynamic library; the latter does not have this particular part of the code.

More details of the sequence of steps happening when a program execution starts will be discussed in the next chapter.

Variety of Section Types

Much like running an automobile cannot be imagined without the motor and a set of four wheels, executing the program cannot be imagined without the code (.text) and the data (.data and/or .bss) sections. These ingredients are naturally the quintessential part of the most basic program functionality.

However, much like the automobile is not only the motor and four wheels, the binary file contains many more sections. In order to finely synchronize the variety of operational tasks, the linker creates and inserts into the binary file many more different section types.

By convention, the section name starts with the dot (.) character. The names of the most important section types are platform independent; they are called the same regardless of the platform and the binary format it belongs to.

Throughout the course of this book, the meanings and roles of certain section types in the overall scheme of things will be discussed at length. Hopefully, by the time the book is read through, the reader will have a substantially wider and more focused understanding of the binary file sections.

In Table 2-1, the specification of the Linux' prevalent ELF binary format brings the following (`http://man7.org/linux/man-pages/man5/elf.5.html`) list of various section types provided in the alphabetical order. Even though the descriptions of individual sections are a bit meager, taking a glance at the variety of sections at this point may give reader fairly good idea about the variety available.

Table 2-1. *Linker Section Types*

Section Name	Description
.bss	This section holds the uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. This section is of type `SHT_NOBITS`. The attribute types are `SHF_ALLOC` and `SHF_WRITE`.
.comment	This section holds version control information. This section is of type `SHT_PROGBITS`. No attribute types are used.
.ctors	This section holds initialized pointers to the C++ constructor functions. This section is of type `SHT_PROGBITS`. The attribute types are `SHF_ALLOC` and `SHF_WRITE`.
.data	This section holds initialized data that contributes to the program's memory image. This section is of type `SHT_PROGBITS`. The attribute types are `SHF_ALLOC` and `SHF_WRITE`.
.data1	This section holds initialized data that contributes to the program's memory image. This section is of type `SHT_PROGBITS`. The attribute types are `SHF_ALLOC` and `SHF_WRITE`.
.debug	This section holds information for symbolic debugging. Thecontents are unspecified. This section is of type `SHT_PROGBITS`. No attribute types are used.
.dtors	This section holds initialized pointers to the C++ destructor functions. This section is of type `SHT_PROGBITS`. The attribute types are `SHF_ALLOC` and `SHF_WRITE`.
.dynamic	This section holds dynamic linking information. The section's attributes include the `SHF_ALLOC` bit. Whether the `SHF_WRITE` bit is set is processor-specific. This section is of type `SHT_DYNAMIC`. See the attributes above

(*continued*)

Table 2-1. (*continued*)

Section Name	Description
.dynstr	This section holds the strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. This section is of type SHT_STRTAB. The attribute type used is SHF_ALLOC.
.dynsym	This section holds the dynamic linking symbol table. This section is of type SHT_DYNSYM. The attribute used is SHF_ALLOC.
.fini	This section holds executable instructions that contribute to the process termination code. When a program exits normally, the system arranges to execute the code in this section. This section is of type SHT_PROGBITS. The attributes used are SHF_ALLOC and SHF_EXECINSTR.
.gnu.version	This section holds the version symbol table, an array of ElfN_Half elements. This section is of type SHT_GNU_versym. The attribute type used is SHF_ALLOC.
.gnu.version_d	This section holds the version symbol definitions, a table of ElfN_Verdef structures. This section is of type SHT_GNU_verdef. The attribute type used is SHF_ALLOC.
.gnu_version.r	This section holds the version symbol needed elements, a table of ElfN_Verneed structures. This section is of type SHT_GNU_versym. The attribute type used is SHF_ALLOC.
.got	This section holds the global offset table. This section is of type SHT_PROGBITS. The attributes are processor-specific.
.got.plt	This section holds the procedure linkage table. This section is of type SHT_PROGBITS. The attributes are processor-specific.
.hash	This section holds a symbol hash table. This section is of type SHT_HASH. The attribute used is SHF_ALLOC.
.init	This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point. This section is of type SHT_PROGBITS. The attributes used are SHF_ALLOC and SHF_EXECINSTR.
.interp	This section holds the pathname of a program interpreter. If the file has a loadable segment that includes the section, the section's attributes will include the SHF_ALLOC bit. Otherwise, that bit will be off. This section is of type SHT_PROGBITS.
.line	This section holds the line number information for symbolic debugging, which describes the correspondence between the program source and the machine code. The contents are unspecified. This section is of type SHT_PROGBITS. No attribute types are used.
.note	This section holds information in the "Note Section" format. This section is of type SHT_NOTE. No attribute types are used. OpenBSD native executables usually contain a .note. openbsd.ident section to identify themselves, for the kernel to bypass any compatibility ELF binary emulation tests when loading the file.
.note.GNU-stack	This section is used in Linux object files for declaring stack attributes. This section is of type SHT_PROGBITS. The only attribute used is SHF_EXECINSTR. This indicates to the GNU linker that the object file requires an executable stack.
.plt	This section holds the procedure linkage table. This section is of type SHT_PROGBITS. The attributes are processor-specific.

(*continued*)

Table 2-1. (*continued*)

Section Name	Description
.relNAME	This section holds relocation information as described below. If the file has a loadable segment that includes relocation, the section's attributes will include the SHF_ALLOC bit. Otherwise, the bit will be off. By convention, "NAME" is supplied by the section to which the relocations apply. Thus a relocation section for .text normally would have the name .rel.text. This section is of type SHT_REL.
.relaNAME	This section holds relocation information as described below. If the file has a loadable segment that includes relocation, the section's attributes will include the SHF_ALLOC bit. Otherwise, the bit will be off. By convention, "NAME" is supplied by the section to which the relocations apply. Thus, a relocation section for .text normally would have the name .rela.text. This section is of type SHT_RELA.
.rodata	This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type SHT_PROGBITS. The attribute used is SHF_ALLOC.
.rodata1	This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type SHT_PROGBITS. The attribute used is SHF_ALLOC.
.shrstrtab	This section holds section names. This section is of type SHT_STRTAB. No attribute types are used.
.strtab	This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the SHF_ALLOC bit. Otherwise the bit will be off. This section is of type SHT_STRTAB.
.symtab	This section holds a symbol table. If the file has a loadable segment that includes the symbol table, the section's attributes will include the SHF_ALLOC bit. Otherwise, the bit will be off. This section is of type SHT_SYMTAB.
.text	This section holds the "text," or executable instructions, of a program. This section is of type SHT_PROGBITS. The attributes used are SHF_ALLOC and SHF_EXECINSTR.

A Variety of Symbol Types

The ELF format provides a vast variety of linker symbol types, far larger than you can imagine at this early stage on your path toward understanding the intricacies of the linking process. At the present moment, you can clearly distinguish that symbols can be of either local scope or of broader visibility, typically needed by the other modules. Throughout the book material later on, the various symbol types will be discussed in substantially more detail.

Table 2-2 features the variety of symbol types, as shown in the man pages (http://linux.die.net/man/1/nm) of the useful ***nm*** symbol examination utility program. As a general rule, unless explicitly indicated (like in the case of "U" vs. "u"), the small letter denotes local symbols, whereas the capital letter indicates better symbol visibility (extern, global).

Table 2-2. *Linker Symbol Types*

Symbol Type	Description
"A"	The symbol's value is absolute, and will not be changed by further linking.
"B" or "b"	The symbol is in the uninitialized (.bss) data section.
"C"	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
"D" or "d"	The symbol is in the initialized data section.
"G" or "g"	The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global `int` variable as opposed to a large global array.
"i"	For PE format files, this indicates that the symbol is in a section specific to the implementation of DLLs. For ELF format files, this indicates that the symbol is an indirect function. This is a GNU extension to the standard set of ELF symbol types. It indicates a symbol that, if referenced by a relocation, does not evaluate to its address, but instead must be invoked at runtime. The runtime execution willthen return the value to be used in the relocation.
"N"	The symbol is a debugging symbol.
"p"	The symbols is in a stack unwind section.
"R" or "r"	The symbol is in a read only data section.
"S" or "s"	The symbol is in an uninitialized data section for small objects.
"T" or "t"	The symbol is in the text (code) section.
"U"	The symbol is undefined. In fact, this binary does not define this symbol, but expects that it eventually appears as the result of loading the dynamic libraries.
"u"	The symbol is a unique global symbol. This is a GNU extension to the standard set of ELF symbol bindings. For such a symbol, the dynamic linker will make sure that in the entire process there is just one symbol with this name and type in use.
"V" or "v"	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error. On some systems, uppercase indicates that a default value has been specified.
"W" or "w"	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal definedsymbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the symbol is determined in a system-specific manner without error.On some systems, uppercase indicates that a default value has been specified.
"-"	The symbol is a stabs symbol in an `a.out` object file. In this case, the next values printed are the stabs other field, the stabs desc field, and the stab type. Stabs symbols are usedto hold debugging information.
"?"	The symbol type is unknown, or object file format-specific.