Introduction to Machine-Level Representation of C Programs[1]

Edited by Alberto José Proença
Dep. Informática, Universidade do Minho, Portugal

Contents

1. Information is Bits in Context

A *computer system* is a collection of hardware and software components that work together to run computer programs. Specific implementations of systems change over time, but the underlying concepts do not. All systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to improve at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

In their classic text on the C programming language, Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.

[1] Compiled and adapted from the Beta Draft version book of Randal E. Bryant and David R. O'Hallaron *Computer Systems: A Programmer's Perspective* (Prentice Hall, 2002, final version not yet available). These notes aim to complement the authors' handouts, which introduce their students in Carnegie-Mellon University to *Machine-Level Programs on Linux/IA32*; they are complementary teaching notes for lab classes in Computer Architecture courses lectured at Dep. Informatics, University of Minho (November 2001).

code/intro/hello.c

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

code/intro/hello.c

Figure 1: **The `hello` program.**

Although `hello` is a very simple program, every major part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why, when you run `hello` on your system. We will begin our study of systems by tracing the lifetime of the `hello` program, from the time a programmer creates it, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most modern systems represent text characters using the ASCII standard that represents each character with a unique byte-sized integer value. For example, Figure 2 shows the ASCII representation of the `hello.c` program.

```
  #    i    n    c    l    u    d    e  <sp>   <    s    t    d    i    o    .
 35  105  110   99  108  117  100  101   32   60  115  116  100  105  111   46

  h    >   \n   \n    i    n    t  <sp>   m    a    i    n    (    )   \n    {
104   62   10   10  105  110  116   32  109   97  105  110   40   41   10  123

 \n  <sp> <sp> <sp> <sp>  p    r    i    n    t    f    (    "    h    e    l
 10   32   32   32   32  112  114  105  110  116  102   40   34  104  101  108

  l    o    ,  <sp>   w    o    r    l    d    \    n    "    )    ;   \n    }
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10  125
```

Figure 2: **The ASCII text representation of `hello.c`.**

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '#'. The second byte has the integer value 105, which corresponds to the character '`i`', and so on. Notice that the invisible newline character '\n', which is represented by the integer value 10, terminates each text line. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

2. Programs are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understand by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program*, and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

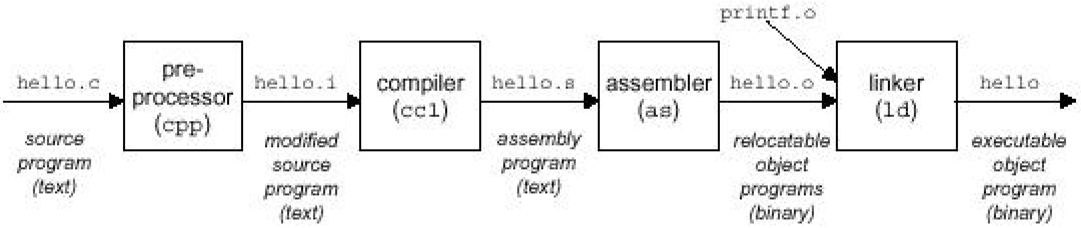


Figure 3: **The compilation system.**

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the `#` character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.
- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.
- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view `hello.o` with a text editor, it would appear to be gibberish.
- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

3. It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a `switch` statement always more efficient than a sequence of `if-then-else` statements? Just how expensive is a function call? Is a `while` loop more efficient than a `do` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times? In Chapter 3 (of the book), we will introduce the Intel IA32 machine language and describe how compilers translate different C constructs into that language. In Chapter 5 (of the book), we will learn how to tune the performance of our C programs by making simple transformations to the C code that help the compiler do its job. And in Chapter 6 (of the book), we will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how our C programs can exploit this knowledge to run more efficiently.

- *Understanding link-time errors.* In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if we define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run-time? We will learn the answers to these kinds of questions in Chapter 7 (of the book).
- *Avoiding security holes.* For many years now, *buffer overflow bugs* have accounted for the majority of security holes in network and Internet servers. These bugs exist because too many programmers are ignorant of the stack discipline that compilers use to generate code for functions. We will describe the stack discipline and buffer overflow bugs in Chapter 3 (of the book), as part of our study of assembly language.

4. C *versus* Assembly Programming

When programming in a high-level language, such as C, we are shielded from the detailed, machine-level implementation of our program. In contrast, when writing programs in assembly code, a programmer must specify exactly how the program manages memory and the low-level instructions the program uses to carry out the computation. Most of the time, it is much more productive and reliable to work at the higher level of abstraction provided by a high-level language. The type checking provided by a compiler helps detect many program errors and makes sure we reference and manipulate data in consistent ways. With modern, optimizing compilers, the generated code is usually at least as efficient as what a skilled, assembly-language programmer would write by hand. Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.

Even though optimizing compilers are available, being able to read and understand assembly code is an important skill for serious programmers. By invoking the compiler with appropriate flags, the compiler will generate a file showing its output in assembly code. Assembly code is very close to the actual machine code that computers execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter 5 (of the book), programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the run-time behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package (covered in Chapter 11 of the book), it is important to know what type of storage is used to hold the different program variables. This information is visible at the assembly code level. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by optimizing compilers.

In these notes we present the details of a particular assembly language and see how C programs get compiled into this form of machine code. Reading the assembly code generated by a compiler involves a different set of skills than writing assembly code by hand. We must understand the transformations typical compilers make in converting the constructs of C into machine code. Relative to the computations expressed in the C code, optimizing compilers can rearrange execution order, eliminate unneeded computations, re-place slow operations such as multiplication by shifts and adds, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can of-ten be a challenge—much like putting together a puzzle having a slightly different design than the picture on the box. It is a form of *reverse engineering*—trying to understand the process by which a system was created by studying the system and working backward. In this case, the system is a machine-generated, assembly-language program, rather than something designed by a human. This simplifies the task of re-verse engineering, because the generated code follows fairly regular patterns, and we can run experiments, having the compiler generate code for many different programs.

A brief history of the Intel architecture is the starting point of the companion document *Machine-Level Programs on Linux/IA32*[2]. Intel processors have grown from rather primitive 16-bit processors in 1978 to the mainstream machines for today's desktop computers. The architecture has grown correspondingly with new features added and the 16-bit architecture transformed to support 32-bit data and addresses. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by GCC and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

The best references on IA32 are from Intel. Two useful references are part of their series on software development: the basic architecture manual gives an overview of the architecture from the perspective of an assembly-language programmer, and the instruction set reference manual gives detailed descriptions of the different instructions. These references (included in the companion document above mentioned) contain far more information than is required to understand Linux code. In particular, with flat mode addressing, all of the complexities of the segmented addressing scheme can be ignored. Also note that the GAS format used by the Linux assembler is very different from the standard format used in Intel documentation and by other compilers (particularly those produced by Microsoft). One main distinction is that the source and destination operands are given in the opposite order On a Linux machine, running the command `info as` will display information about the assembler. One of the subsections documents machine-specific information, including a comparison of GAS with the more standard Intel notation. Note that GCC refers to these machines as "`i386`"—it generates code that could even run on a 1985 vintage machine.

This document also complements the above-mentioned one, namely:

- How control constructs in C, such as if, while, and switch statements, are implemented;
- Examine the problems of out of bounds memory references and the vulnerability of systems to buffer overflow attacks;
- Some tips on using the GDB debugger for examining the runtime behavior of a machine-level program;
- A brief presentation of GCC's support for embedding assembly code within C programs; in some applications, the programmer must drop down to assembly code to access low-level features of the machine; embedded assembly is the best way to do this.

5. Program Encodings

Suppose we write a C program as two files `p1.c` and `p2.c`. We would then compile this code using a Unix command line:

```
unix> gcc -O2 -o p p1.c p2.c
```

The command `gcc` indicates the GNU C compiler GCC. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The flag `-O2` instructs the compiler to apply level-two optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code. Level-two optimization is a good compromise between optimized performance and ease of use. All code in this book was compiled with this optimization level.

[2] To update this document, please add the following Intel family member to the existing list:
Pentium 4: (2001, 42 Mtransistors). Added 8-byte integer and floating-point formats to the vector instructions, along with 144 new instructions for these formats. Intel shifted away from Roman numerals in their numbering convention.

This command actually invokes a sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros. Second, the *compiler* generates assembly code versions of the two source files having names `p1.s` and `p2.s`. Next, the *assembler* converts the assembly code into binary object code files `p1.o` and `p2.o`. Finally, the *linker* merges these two object files along with code implementing standard Unix library functions (e.g., `printf`) and generates the final executable file (linking is described in more detail in Chapter 7 of the book; some introductory parts are included at the end of these notes).

6. Machine-Level Code

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes. The assembly code-representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of object code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

The assembly programmer's view of the machine differs significantly from that of a C programmer. Parts of the processor state are visible that are normally hidden from the C programmer:

- The program counter (called %eip) indicates the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing 32-bit values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the local variables of a procedure.
- The condition code registers hold status information about the most recently executed arithmetic instruction. These are used to implement conditional changes in the control flow, such as is required to implement `if` or `while` statements.
- The floating-point register file contains eight locations for storing floating-point data.

Whereas C provides a model where objects of different data types can be declared and allocated in memory, assembly code views the memory as simply a large, byte-addressable array. Aggregate data types in C such as arrays and structures are represented in assembly code as contiguous collections of bytes. Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.

The program memory contains the object code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user, (for example, by using the `malloc` library procedure).

The program memory is addressed using *virtual* addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the 32-bit addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

A single machine instruction performs only a very elementary operation. For example, it might add two numbers stored in registers, transfer data between memory and a register, or conditionally branch to a new instruction address. The compiler must generate sequences of such instructions to implement program constructs such as arithmetic expression evaluation, loops, or procedure calls and returns.

7. Code Examples

Suppose we write a C code file `code.c` containing the following procedure definition:

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

To see the assembly code generated by the C compiler, we can use the "`-S`" option on the command line:

```
unix> gcc -O2 -S code.c
```

This will cause the compiler to generate an assembly file `code.s` and go no further. (Normally it would then invoke the assembler to generate an object code file). The assembly-code file contains various declarations including the set of lines:

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   addl %eax,accum
   movl %ebp,%esp
   popl %ebp
   ret
```

Each indented line in the above code corresponds to a single machine instruction. For example, the `pushl` instruction indicates that the contents of register `%ebp` should be pushed onto the program stack. All information about local variable names or data types has been stripped away. We still see a reference to the global variable `accum`, since the compiler has not yet determined where in memory this variable will be stored.

If we use the '`-c`' command line option, GCC will both compile and assemble the code:

```
unix> gcc -O2 -c code.c
```

This will generate an object code file `code.o` that is in binary format and hence cannot be viewed directly. Embedded within the 852 bytes of the file `code.o` is a 19 byte sequence having hexadecimal representation:

```
55 89 e5 8b 45 0c 03 45 08 01 05 00 00 00 00 89 ec 5d c3
```

This is the object code corresponding to the assembly instructions listed above. A key lesson to learn from this is that the program actually executed by the machine is simply a sequence of bytes encoding a series of instructions. The machine has very little information about the source code from which these instructions were generated.

To inspect the contents of object code files, a class of programs known as *disassemblers* can be invaluable. These programs generate a format similar to assembly code from the object code. With Linux systems, the program OBJDUMP (for "object dump") can serve this role given the '`-d`' command line flag:

```
unix> objdump -d code.o
```

The result is (where we have added line numbers on the left and annotations on the right):

Disassembly of function `sum` in file `code.o`

```
00000000 <sum>:
```

Offset Bytes *Equivalent assembly language*

```
  0:   55                     push   %ebp
  1:   89 e5                  mov    %esp,%ebp
  3:   8b 45 0c               mov    0xc(%ebp),%eax
  6:   03 45 08               add    0x8(%ebp),%eax
  9:   01 05 00 00 00 00      add    %eax,0x0
  f:   89 ec                  mov    %ebp,%esp
 11:   5d                     pop    %ebp
 12:   c3                     ret
 13:   90                     nop
```

On the left we see the 19 hexadecimal byte values listed in the byte sequence earlier, partitioned into groups of 1 to 5 bytes each. Each of these groups is a single instruction, with the assembly language equivalent shown on the right. Several features are worth noting:

- IA32 instructions can range in length from 1 to 15 bytes. The instruction encoding is designed so that commonly used instructions and ones with fewer operands require a smaller number of bytes than do less common ones or ones with more operands.
- The instruction format is designed in such a way that from a given starting position, there is a unique decoding of the bytes into machine instructions. For example, only the instruction `pushl %ebp` can start with byte value `55`.
- The disassembler determines the assembly code based purely on the byte sequences in the object file. It does not require access to the source or assembly-code versions of the program.
- The disassembler uses a slightly different naming convention for the instructions than does GAS. In our example, it has omitted the suffix '`l`' from many of the instructions.
- Compared to the assembly code in `code.s` we also see an additional `nop` instruction at the end. This instruction will never be executed (it comes after the procedure return instruction), nor would it have any effect if it were (hence the name `nop`, short for "no operation" and commonly spoken as "no op"). The compiler inserted this instruction as a way to pad the space used to store the procedure.

Generating the actual executable code requires running a linker on the set of object code files, one of which must contain a function `main`. Suppose in file `main.c` we had the function:

```
int main()
{
    return sum(1, 3);
}
```

Then we could generate an executable program `prog` as follows:

```
unix> gcc -O2 -o prog code.o main.c
```

The file `prog` has grown to 11,667 bytes, since it contains not just the code for our two procedures but also information used to start and terminate the program as well as to interact with the operating system. We can also disassemble the file `prog`:

```
unix> objdump -d prog
```

The disassembler will extract various code sequences, including the following:

Disassembly of function sum in executable file prog

```
080483b4 <sum>:
 80483b4:   55                     push  %ebp
 80483b5:   89 e5                  mov   %esp,%ebp
 80483b7:   8b 45 0c               mov   0xc(%ebp),%eax
 80483ba:   03 45 08               add   0x8(%ebp),%eax
 80483bd:   01 05 64 94 04 08      add   %eax,0x8049464
 80483c3:   89 ec                  mov   %ebp,%esp
 80483c5:   5d                     pop   %ebp
 80483c6:   c3                     ret
 80483c7:   90                     nop
```

Note that this code is almost identical to that generated by the disassembly of `code.c`. One main difference is that the addresses listed along the left are different—the linker has shifted the location of this code to a different range of addresses. A second difference is that the linker has finally determined the location for storing global variable `accum`. On line 5 of the disassembly for `code.o` the address of `accum` was still listed as `0`. In the disassembly of `prog`, the address has been set to `0x8049464`. This is shown in the assembly code rendition of the instruction. It can also be seen in the last four bytes of the instruction, listed from least significant to most as `64 94 04 08`.

8. A Note on Formatting

The assembly code generated by GCC is somewhat difficult to read. It contains some information with which we need not be concerned. On the other hand, it does not provide any description of the program or how it works. For example, suppose file `simple.c` contains the code:

```
int simple(int *xp, int y)
{
    int t = *xp + y;
    *xp = t;
    return t;
}
```

when GCC is run with the ‘`-S`’ flag it generates the following file for `simple.s`.

```
   .file "simple.c"
   .version "01.01"
gcc2_compiled.:
.text
   .align 4
.globl simple
   .type simple,@function
simple:
   pushl %ebp
   movl %esp,%ebp
   movl 8(%ebp),%eax
   movl (%eax),%edx
   addl 12(%ebp),%edx
   movl %edx,(%eax)
   movl %edx,%eax
   movl %ebp,%esp
   popl %ebp
   ret
.Lfe1:
   .size simple,.Lfe1-simple
   .ident "GCC: (GNU) 2.95.3 20010315 (release)"
```

The file contains more information than we really require. All of the lines beginning with ‘`.`’ are directives to guide the assembler and linker. We can generally ignore these. On the other hand, there are no explanatory remarks about what the instructions do or how they relate to the source code.

To provide a clearer presentation of assembly code, we will show it in a form that includes line numbers and explanatory annotations. For our example, an annotated version would appear as follows:

```
simple:
  pushl %ebp               Save frame pointer
  movl %esp,%ebp           Create new frame pointer
  movl 8(%ebp),%eax        Get xp
  movl (%eax),%edx         Retrieve *xp
  addl 12(%ebp),%edx       Add y to get t
  movl %edx,(%eax)         Store t at *xp
  movl %edx,%eax           Set t as return value
  movl %ebp,%esp           Reset stack pointer
  popl %ebp                Reset frame pointer
  ret                      Return
```

We typically show only the lines of code relevant to the point being discussed. Each line is numbered on the left for reference and annotated on the right by a brief description of the effect of the instruction and how it relates to the computations of the original C code. This is a stylized version of the way assembly-language programmers format their code.

9. Translating Conditional Branches

To implement the control constructs of C, the compiler must use the different types of jump instructions available in assembly. We will go through the most common constructs, starting from simple conditional branches, and then considering loops and switch statements. Conditional statements in `C` are implemented using combinations of conditional and unconditional jumps. For example, Figure 4 shows the C code for a function that computes the absolute value of the difference of two numbers (a). GCC generates the assembly code shown as (c). We have created a version in C, called `gotodiff` (b), that more closely follows the control flow of this assembly code. It uses the `goto` statement in C, which is similar to the unconditional jump of assembly code. The statement `goto less` on line 6 causes a jump to the label `less` on line 8, skipping the statement on line 7. Note that using `goto` statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of assembly-code programs. We call such C programs "goto code."

code/asm/abs.c

```
1 int absdiff(int x, int y)
2 {
3    if (x < y)
4         return y - x;
5    else
6         return x - y;
7 }
```

code/asm/abs.c

(a) Original C code.

code/asm/abs.c

```
1 int gotodiff(int x, int y)
2 {
3    int rval;
4
5    if (x < y)
6         goto less;
7    rval = x - y;
8    goto done;
9 less:
10   rval = y - x;
11 done:
12   return rval;
13 }
```

code/asm/abs.c

(b) Equivalent goto version of (a).

```
1   movl  8(%ebp),%edx          Get x
2   movl  12(%ebp),%eax         Get y
3   cmpl  %eax,%edx             Compare x:y
4   jl    .L3                   If <, goto less:
5   subl  %eax,%edx             Compute y-x
6   movl  %edx,%eax             Set as return value
7   jmp   .L5                   Goto done:
8 .L3:                        less:
9   subl  %edx,%eax             Compute x-y as return value
10 .L5:                       done: Begin completion code
```

(c) Generated assembly code.

Figure 4: **Compilation of Conditional Statements** C procedure `absdiff` (a) contains an if-else statement. The generated assembly code is shown (c), along with a C procedure `gotodiff` (b) that mimics the control flow of the assembly code. The stack set-up and completion portions of the assembly code have been omitted

The assembly code implementation first compares the two operands (line 3), setting the condition codes. If the comparison result indicates that `x` is less than `y`, it then jumps to a block of code that computes `x-y` (line 9). Otherwise it continues with the execution of code that computes `y-x` (lines 5 and 6). In both cases the computed result is stored in register `%eax`, and ends up at line 10, at which point it executes the stack completion code (not shown).

The general form of an if-else statement in C is given by the `if-else` statement following template:

```
if (test-expr)
   then-statement
else
   else-statement
```

where *test-expr* is an integer expression that evaluates either to 0 (interpreted as meaning “false”) or to a nonzero value (interpreted as meaning “true”). Only one of the two branch statements (*then-statement* or *else-statement*) is executed.

For this general form, the assembly implementation typically follows the form shown below, where we use C syntax to describe the control flow:

```
   t = test-expr;
   if (t)
      goto true;
   else-statement
   goto done;
true:
   then-statement
done:
```

That is, the compiler generates separate blocks of code for *then-statement* and *else-statement*. It inserts conditional and unconditional branches to make sure the correct block is executed.

10. Loops

C provides several looping constructs, namely `while`, `for`, and `do-while`. No corresponding instructions exist in assembly. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Interestingly, most compilers generate loop code based on the `do-while` form of a loop, even though this form is relatively uncommon in actual programs. Other loops are transformed into `do-while` form and then compiled into machine code. We will study the translation of loops as a progression, starting with `do-while` and then working toward ones with more complex implementations.

Do-While Loops

The general form of a `do-while` statement is as follows:

```
do
   body-statement
   while (test-expr);
```

The effect of the loop is to repeatedly execute *body-statement*, evaluate *test-expr* and continue the loop if the evaluation result is nonzero. Observe that *body-statement* is executed at least once. Typically, the implementation of `do-while` has the following general form:

```
loop:
   body-statement
   t = test-expr;
   if (t)
      goto loop;
```

As an example, Figure 5 shows an implementation of a routine to compute the *n*th element in the Fibonacci sequence using a `do-while` loop. This sequence is defined by the recurrence:

$F_1 = 1$
$F_2 = 1$
$F_n = F_{n-2} + F_{n-3}$, n >= 3

For example, the first ten elements of the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, and 55. To implement this using a `do-while` loop, we have started the sequence with values $F_0 = 0$ and $F_1 = 1$, rather than with F_1 and F_2.

The assembly code implementing the loop is also shown, along with a table showing the correspondence between registers and program values. In this example, *body-statement* consists of lines 8 through 11, assigning values to `t`, `val`, and `nval`, along with the incrementing of `i`. These are implemented by lines 2 through 5 of the assembly code. The expression `i < n` comprises *test-expr*. This is implemented by line 6 and by the test condition of the jump instruction on line 7. Once the loop exits, `val` is copy to register `%eax` as the return value (line 8).

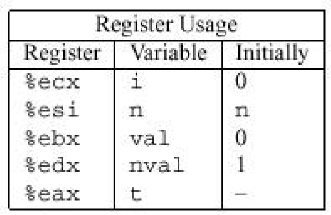
Creating a table of register usage, such as we have shown in Figure 5(b) is a very helpful step in analyzing an assembly language program, especially when loops are present.

code/asm/fib.c

```
1  int fib_dw(int n)
2  {
3     int i = 0;
4     int val = 0;
5     int nval = 1;
6
7     do {
8          int t = val + nval;
9          val = nval;
10         nval = t;
11         i++;
12    } while (i < n);
13
14    return val;
15 }
```

code/asm/fib.c

(a) C code.



Register Usage		
Register	Variable	Initially
%ecx	i	0
%esi	n	n
%ebx	val	0
%edx	nval	1
%eax	t	–

```
1 .L6:                                    loop:
2    leal  (%edx,%ebx),%eax                  Compute t = val + nval
3    movl  %edx,%ebx                         Copy nval to val
4    movl  %eax,%edx                         Copy t to nval
5    incl  %ecx                              Increment i
6    cmpl  %esi,%ecx                         Compare i:n
7    jl    .L6                               If less, goto loop
8    movl  %ebx,%eax                         Set val as return value
```

(b) Corresponding assembly language code.

Figure 5: **C and Assembly Code for Do-While Version of Fibonacci Program.** Only the code inside the loop is shown.

While Loops

The general form of a while statement is as follows:

```
while (test-expr)
body-statement
```

It differs from `do-while` in that *test-expr* is evaluated and the loop is potentially terminated before the first execution of *body-statement*. A direct translation into a form using goto's would be:

```
 loop:
    t = test-expr;
    if (!t)
       goto done;
    body-statement
    goto loop;
 done:
```

This translation requires two control statements within the inner loop—the part of the code that is executed the most. Instead, most C compilers transform the code into a `do-while` loop by using a conditional branch to skip the first execution of the body if needed:

```
    if (!test-expr)
       goto done;
    do
       body-statement
       while (test-expr);
 done:
```

This, in turn, can be transformed into goto code as:

```
    t = test-expr;
    if (!t)
       goto done;
 loop:
    body-statement
    t = test-expr;
    if (t)
       goto loop;
 done:
```

As an example, Figure 6 shows an implementation of the Fibonacci sequence function using a `while` loop (a). Observe that this time we have started the recursion with elements F_1 (`val`) and F_2 (`nval`). The adjacent C function `fib_w_goto` (b) shows how this code has been translated into assembly. The assembly code in (c) closely follows the C code shown in `fib_w_goto`. The compiler has performed several interesting optimizations, as can be seen in the goto code (b). First, rather than using variable `i` as a loop variable and comparing it to `n` on each iteration, the compiler has introduced a new loop variable that we call "`nmi`", since relative to the original code, its value equals $n - i$. This allows the compiler to use only three registers for loop variables, compared to four otherwise. Second, it has optimized the initial test condition `(i < n)` into `(val < n)`, since the initial values of both `i` and `val` are 1. By this means, the compiler has totally eliminated variable `i`. Often the compiler can make use of the initial values of the variables to optimize the initial test. This can make deciphering the assembly code tricky. Third, for successive executions of the loop we are assured that $i =< n$, and so the compiler can assume that `nmi` is nonnegative. As a result, it can test the loop condition as `nmi != 0` rather than `nmi >= 0`. This saves one instruction in the assembly code.

code/asm/fib.c

```
int fib_w(int n)
{
   int i = 1;
   int val = 1;
   int nval = 1;

   while (i < n) {
        int t = val+nval;
        val = nval;
        nval = t;
        i++;
   }

   return val;
}
```

code/asm/fib.c

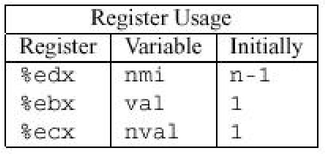
(a) C code.

code/asm/fib.c

```
int fib_w_goto(int n)
{
   int val = 1;
   int nval = 1;
   int nmi, t;

   if (val >= n)
        goto done;
   nmi = n-1;

 loop:
   t = val+nval;
   val = nval;
   nval = t;
   nmi--;
   if (nmi)
        goto loop;

 done:
   return val;
}
```

code/asm/fib.c

(b) Equivalent goto version of (a).



Register Usage		
Register	Variable	Initially
%edx	nmi	n-1
%ebx	val	1
%ecx	nval	1

```
   movl  8(%ebp),%eax          Get n
   movl  $1,%ebx               Set val to 1
   movl  $1,%ecx               Set nval to 1
   cmpl  %eax,%ebx             Compare val:n
   jge   .L9                   If >= goto done:
   leal  -1(%eax),%edx         nmi = n-1
.L10:                        loop:
   leal  (%ecx,%ebx),%eax      Compute t = nval+val
   movl  %ecx,%ebx             Set val to nval
   movl  %eax,%ecx             Set nval to t
   decl  %edx                  Decrement nmi
   jnz   .L10                  If != 0, goto loop:
 .L9:                        done:
```

(c) Corresponding assembly language code.

Figure 6: **C and Assembly Code for While Version of Fibonacci.** The compiler has performed a number of optimizations, including replacing the value denoted by variable `i` with one we call `nmi`.

For Loops

The general form of a `for` loop is as follows:

```
for (init-expr; test-expr; update-expr)
     body-statement
```

The C language standard states that the behavior of such a loop is identical to the following code using a while loop:

```
    init-expr;
    while (test-expr) {
        body-statement
        update-expr;
    }
```

That is, the program first evaluates the initialization expression *init-expr*. It then enters a loop where it first evaluates the test condition *test-expr*, exiting if the test fails, then executes the body of the loop *body-statement*, and finally evaluates the update expression *update-expr*.

The compiled form of this code then is based on the transformation from while to `do-while` described previously, first giving a `do-while` form:

```
    init-expr;
    if (!test-expr)
        goto done;
    do {
        body-statement
        update-expr;
    } while (test-expr);
done:
```

This, in turn, can be transformed into goto code as:

```
    init-expr;
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    update-expr;
    t = test-expr;
    if (t)
        goto loop;
done:
```

As an example, the following code shows an implementation of the Fibonacci function using a `for` loop:

code/asm/fib.c

```
int fib_f(int n)
{
    int i;
    int val = 1;
    int nval = 1;

    for (i = 1; i < n; i++) {
        int t = val+nval;
        val = nval;
        nval = t;
    }

    return val;
}
```

code/asm/fib.c

The transformation of this code into the while loop form gives code identical to that for the function `fib_w` shown in Figure 6. In fact, GCC generates identical assembly code for the two functions.

11. Switch Statements

Switch statements provide a multi-way branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, they also allow an efficient implementation using a data structure called a *jump table*. A jump table is an array where entry *i* is the address of a code segment implementing the action the program should take when the switch index equals *i*.

The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction. The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases. GCC selects the method of translating a switch statement based on the number of cases and the sparsity of the case values. Jump tables are used when there are a number of cases (e.g., four or more) and they span a small range of values.

Figure 7(a) shows an example of a C `switch` statement. This example has a number of interesting features, including case labels that do not span a contiguous range (there are no labels for cases 101 and 105), cases with multiple labels (cases 104 and 106), and cases that “fall through” to other cases (case 102), because the code for the case does not end with a `break` statement. Figure 8 shows the assembly code generated when compiling `switch_eg`. The behavior of this code is shown using an extended form of C as the procedure `switch_eg_impl` in Figure 7(b). We say “extended” because C does not provide the necessary constructs to support this style of jump table, and hence our code is not legal C. The array `jt` contains 7 entries, each of which is the address of a block of code. We extend C with a data type `code` for this purpose.

Lines 1 to 4 set up the jump table access. To make sure that values of `x` that are either less than 100 or greater than 106 cause the computation specified by the `default` case, the code generates an unsigned value `xi` equal to `x-100`. For values of `x` between 100 and 106, `xi` will have values 0 through 6. All other values will be greater than 6, since negative values of `x-100` will wrap around to be very large unsigned numbers. The code therefore uses the `ja` (unsigned greater) instruction to jump to code for the default case when `xi` is greater than 6. Using `jt` to indicate the jump table, the code then performs a jump to the address at entry `xi` in this table. Note that this form of `goto` is not legal C. Instruction 4 implements the jump to an entry in the jump table. Since it is an indirect jump, the target is read from memory. The effective address of the read is determined by adding the base address specified by label `.L10` to the scaled (by 4 since each jump table entry is 4 bytes) value of variable `xi` (in register `%eax`).

In the assembly code, the jump table is indicated by the following declarations, to which we have added comments:

```
.section .rodata
   .align 4                      Align address to multiple of 4
.L10:
   .long .L4                     Case 100: loc_A
   .long .L9                     Case 101: loc_def
   .long .L5                     Case 102: loc_B
   .long .L6                     Case 103: loc_C
   .long .L8                     Case 104: loc_D
   .long .L9                     Case 105: loc_def
   .long .L8                     Case 106: loc_D
```

These declarations state that within the segment of the object code file called “`.rodata`” (for “Read-Only Data”), there should be a sequence of seven “long” (4-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly code labels (e.g., `.L4`). Label `.L10` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (instruction 4).

code/asm/switch.c

```
int switch_eg(int x)
{
   int result = x;

   switch (x) {

   case 100:
        result *= 13;
        break;

   case 102:
        result += 10;
        /* Fall through */

   case 103:
        result += 11;
        break;

   case 104:
   case 106:
        result *= result;
        break;

   default:
        result = 0;
   }

   return result;
}
```

code/asm/switch.c

(a) Switch statement.

code/asm/switch.c

```
/* Next line is not legal C */
code *jt[7] = {
   loc_A, loc_def, loc_B, loc_C,
   loc_D, loc_def, loc_D
};

int switch_eg_impl(int x)
{
   unsigned xi = x - 100;
   int result = x;

   if (xi > 6)
        goto loc_def;

   /* Next goto is not legal C */
   goto jt[xi];

 loc_A: /* Case 100 */
  result *= 13;
  goto done;

 loc_B: /* Case 102 */
  result += 10;
  /* Fall through */

 loc_C: /* Case 103 */
  result += 11;
  goto done;

 loc_D: /* Cases 104, 106 */
  result *= result;
  goto done;

 loc_def: /* Default case*/
  result = 0;

 done:
  return result;
}
```

code/asm/switch.c

(b) Translation into extended C.

Figure 7: **Switch Statement Example with Translation into Extended C.** The translation shows the structure of jump table `jt` and how it is accessed. Such tables and accesses are not actually allowed in C.

The code blocks starting with labels `loc_A` through `loc_D` and `loc_def` in `switch_eg_impl` (Figure 7(b)) implement the five different branches of the switch statement. Observe that the block of code labeled `loc_def` will be executed either when `x` is outside the range 100 to 106 (by the initial range checking) or when it equals either 101 or 105 (based on the jump table). Note how the code for the block labeled `loc_B` falls through to the block labeled `loc_C`.

```
  Set up the jump table access
    leal  -100(%edx),%eax           Compute xi = x-100
    cmpl  $6,%eax                   Compare xi:6
    ja    .L9                       If >, goto done
    jmp   *.L10(,%eax,4)            Goto jt[xi]

  Case 100
 .L4:                              loc A:
    leal  (%edx,%edx,2),%eax        Compute 3*x
    leal  (%edx,%eax,4),%edx        Compute x+4*3*x
    jmp   .L3                       Goto done

  Case 102
 .L5:                              loc B:
    addl  $10,%edx                  result += 10, Fall through

  Case 103
 .L6:                              loc C:
    addl  $11,%edx                  result += 11
    jmp   .L3                       Goto done

  Cases 104, 106
 .L8:                              loc D:
    imull %edx,%edx                 result *= result
    jmp   .L3                       Goto done

  Default case
 .L9:                              loc def:
    xorl  %edx,%edx                 result = 0

  Return result
 .L3:                              done:
    movl  %edx,%eax                 Set result as return value
```

Figure 8: **Assembly Code for Switch Statement Example in Figure 1.7.**

12. Life in the Real World: Using the GDB Debugger

The GNU debugger GDB provides a number of useful features to support the run-time evaluation and analysis of machine-level programs. With the examples and exercises in this book, we attempt to infer the behavior of a program by just looking at the code. Using GDB, it becomes possible to study the behavior by watching the program in action, while having considerable control over its execution.

Figure 9 shows examples of some GDB commands that help when working with machine-level, IA32 programs. It is very helpful to first run OBJDUMP to get a disassembled version of the program. Our examples were based on running GDB on the file `prog`, described and disassembled on page 8. We would start GDB with the command line:

```
unix> gdb prog
```

The general scheme is to set breakpoints near points of interest in the program. These can be set to just after the entry of a function, or at a program address. When one of the breakpoints is hit during program execution, the program will halt and return control to the user. From a breakpoint, we can examine different registers and memory locations in various formats. We can also single-step the program, running just a few instructions at a time, or we can proceed to the next breakpoint.

As our examples suggests, GDB has an obscure command syntax, but the online help information (invoked within GDB with the `help` command) overcomes this shortcoming.

Command		**Effect**
Starting and Stopping		
quit		Exit GDB
run		Run your program (give command line arguments here)
kill		Stop your program
Breakpoints		
break	*sum*	Set breakpoint at entry to function `sum`
break	**0x80483c3*	Set breakpoint at address `0x80483c3`
delete	1	Delete breakpoint 1
delete		Delete all breakpoints
Execution		
stepi		Execute one instruction
stepi	*4*	Execute four instructions
nexti		Like `stepi`, but proceed through function calls
continue		Resume execution
finish		Run until current function returns
Examining code		
disas		Disassemble current function
disas	*sum*	Disassemble function sum
disas	*0x80483b7*	Disassemble function around address `0x80483b7`
disas	*0x80483b7 0x80483c7*	Disassemble code within specified address range
print	*/x $eip*	Print program counter in hex
Examining data		
print	*$eax*	Print contents of `%eax` in decimal
print	*/x $eax*	Print contents of `%eax` in hex
print	*/t $eax*	Print contents of `%eax` in binary
print	*0x100*	Print decimal representation of `0x100`
print	*/x 555*	Print hex representation of 555
print	*/x ($ebp+8)*	Print contents of `%ebp` plus 8 in hex
print	**(int *) 0xbffff890*	Print integer at address `0xbffff890`
print	**(int *) ($ebp+8)*	Print integer at address `%ebp` + 8
x/2w	*0xbffff890*	Examine two (4-byte) words starting at address 0xbffff890
x/20b	*sum*	Examine first 20 bytes of function `sum`
Useful information		
info	*frame*	Information about current stack frame
info	*registers*	Values of all the registers
help		Get information about GDB

Figure 9: **Example** GDB **Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

13. Out-of-Bounds Memory References and Buffer Overflow

We have seen that C does not perform any bounds checking for array references, and that local variables are stored on the stack along with state information such as register values and return pointers. This combination can lead to serious program errors, where the state stored on the stack gets corrupted by a write to an out-of-bounds array element. When the program then tries to reload the register or execute a `ret` instruction with this corrupted state, things can go seriously wrong.

A particularly common source of state corruption is known as *buffer overflow*. Typically some character array is allocated on the stack to hold a string, but the size of the string exceeds the space allocated for the array. This is demonstrated by the following program example.

```
/* Implementation of library function gets() */
char *gets(char *s)
{
  int c;
  char *dest = s;
  while ((c = getchar()) != '\n' && c != EOF)
      *dest++ = c;
  *dest++ = '\0'; /* Terminate String */
  if (c == EOF)
      return NULL;
  return s;
}

/* Read input line and write it back */
void echo()
{
  char buf[4]; /* Way too small! */
  gets(buf);
  puts(buf);
}
```

The above code shows an implementation of the library function `gets` to demonstrate a serious problem with this function. It reads a line from the standard input, stopping when either a terminating newline character or some error condition is encountered. It copies this string to the location designated by argument `s`, and terminates the string with a null character. We show the use of `gets` in the function `echo`, which simply reads a line from standard input and echoes it back to standard output.

The problem with `gets` is that it has no way to determine whether sufficient space has been allocated to hold the entire string. In our `echo` example, we have purposely made the buffer very small - just four characters long. Any string longer than three characters will cause an out-of-bounds write.

Examining a portion of the assembly code for `echo` shows how the stack is organized.

```
echo:
  pushl  %ebp              Save %ebp on stack
  movl   %esp,%ebp
  subl   $20,%esp          Allocate space on stack
  pushl  %ebx              Save %ebx
  addl   $-12,%esp         Allocate more space on stack
  leal   -4(%ebp),%ebx     Compute buf as %ebp-4
  pushl  %ebx              Push buf on stack
  call   gets              Call gets
```

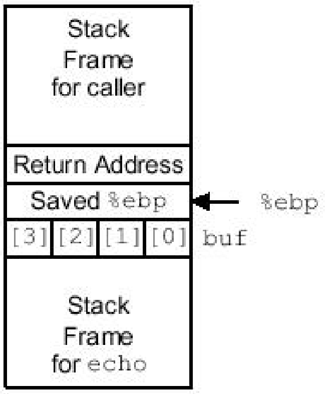


Figure 10: **Stack Organization for `echo` Function.** Character array `buf` is just below part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

We can see in this example that the program allocates a total of 32 bytes (lines 4 and 6) for local storage. However, the location of character array `buf` is computed as just four bytes below `%ebp` (line 7). Figure 10 shows the resulting stack structure. As can be seen, any write to `buf[4]` through `buf[7]` will cause the saved value of `%ebp` to be corrupted. When the program later attempts to restore this as the frame pointer, all subsequent stack references will be invalid. Any write to `buf[8]` through `buf[11]` will cause the return address to be corrupted. When the `ret` instruction is executed at the end of the function, the program will "return" to the wrong address. As this example illustrates, buffer overflow can cause a program to seriously misbehave.

Our code for `echo` is simple but sloppy. A better version involves using the function `fgets`, which includes as an argument a count on the maximum number bytes to read. One homework problem asks you to write an echo function that can handle an input string of arbitrary length. In general, using `gets` or any function that can overflow storage is considered a bad programming practice. The C compiler even produces the following error message when compiling a file containing a call to `gets`: "the `gets` function is dangerous and should not be used."

A more pernicious use of buffer overflow is to get a program to perform a function that it would otherwise be unwilling to do. This is one of the most common methods to attack the security of a system over a computer network. Typically, the program is fed with a string that contains the byte encoding of some executable code, called the *exploit code*, plus some extra bytes that overwrite the return pointer with a pointer to the code in the buffer. The effect of executing the `ret` instruction is then to jump to the exploit code.

In one form of attack, the exploit code then uses a system call to start up a shell program, providing the attacker with a range of operating system functions. In another form, the exploit code performs some otherwise unauthorized task, repairs the damage to the stack, and then executes `ret` a second time, causing an (apparently) normal return to the caller.

As an example, the famous Internet worm of November, 1988 used four different ways to gain access to many of the computers across the Internet. One was a buffer overflow attack on the finger daemon `fingerd`, which serves requests by the FINGER command. By invoking FINGER with an appropriate string, the worm could make the daemon at a remote site have a buffer overflow and execute code that gave the worm access to the remote system. Once the worm gained access to a system, it would replicate itself and consume virtually all of the machine's computing resources. As a consequence, hundreds of machines were effectively paralyzed until security experts could determine how to eliminate the worm. The author of the worm was caught and prosecuted. He was sentenced to three years probation, 400 hours of community service, and a $10,500 fine. Even to this day, however, people continue to find security leaks in systems that leave them vulnerable to buffer overflow attacks. This highlights the need for careful programming. Any interface to the external environment should be made "bullet proof" so that no behavior by an external agent can cause the system to misbehave.

Aside: Worms and viruses

Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [69], a *worm* is a program that can run by itself and can propagate a fully working version of itself to other machines. A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently. In the popular press, the term "virus" is used to refer to a variety of different strategies for spreading attacking code among systems, and so you will hear people saying "virus" for what more properly should be called a "worm." **End Aside**.

In one of the suggested homework exercises, you can gain first-hand experience at mounting a buffer overflow attack. Note that we do not condone using this or any other method to gain unauthorized access to a system. Breaking into computer systems is like breaking into a building—it is a criminal act even when the perpetrator does not have malicious intent. We give this problem for two reasons. First, it requires a deep understanding of machine language programming, combining such issues as stack organization, byte ordering, and instruction encoding. Second, by demonstrating how buffer overflow attacks work, we hope you will learn the importance of writing code that does not permit such attacks.

14. Embedding Assembly Code in C Programs

In the early days of computing, most programs were written in assembly code. Even large-scale operating systems were written without the help of high-level languages. This becomes unmanageable for programs of significant complexity. Since assembly code does not provide any form of type checking, it is very easy to make basic mistakes, such as using a pointer as an integer rather than dereferencing the pointer. Even worst, writing in assembly code locks the entire program into a particular class of machine. Rewriting an assembly language program to run on a different machine can be as difficult as writing the entire program from scratch.

Early compilers for higher-level programming languages did not generate very efficient code and did not provide access to the low-level object representations, as is often required by systems programmers. Programs requiring maximum performance or requiring access to object representations were still often written in assembly code. Nowadays, however, optimizing compilers have largely removed performance optimization as a reason for writing in assembly code. Code generated by a high quality compiler is generally as good or even better than what can be achieved manually. The C language has largely eliminated machine access as a reason for writing in assembly code. The ability to access low-level data representations through unions and pointer arithmetic, along with the ability to operate on bit-level data representations, provide sufficient access to the machine for most programmers. For example, almost every part of a modern operating system such as Linux is written in C.

Nonetheless, there are times when writing in assembly code is the only option. This is especially true when implementing an operating system. For example, there are a number of special registers storing process state information that the operating system must access. There are either special instructions or special memory locations for performing input and output operations. Even for application programmers, there are some machine features, such as the values of the condition codes, which cannot be accessed directly in C.

The challenge then is to integrate code consisting mainly of C with a small amount written in assembly language. One method is to write a few key functions in assembly code, using the same conventions for argument passing and register usage as are followed by the C compiler. The assembly functions are kept in a separate file, and the compiled C code is combined with the assembled assembly code by the linker. For example, if file `p1.c` contains C code and file `p2.s` contains assembly code, then the compilation command:

```
unix> gcc -o p p1.c p2.s
```

will cause file `p1.c` to be compiled, file `p2.s` to be assembled, and the resulting object code to be linked to form an executable program `p`.

Basic Inline Assembly

With GCC, it is also possible to mix assembly with C code. Inline assembly allows the user to insert assembly code directly into the code sequence generated by the compiler. Features are provided to specify instruction operands and to indicate to the compiler which registers are being overwritten by the assembly instructions. The resulting code is, of course, highly machine-dependent, since different types of machines do not have compatible machine instructions. The `asm` directive is also specific to GCC, creating an incompatibility with many other compilers. Nonetheless, this can be a useful way to keep the amount of machine-dependent code to an absolute minimum.

Inline assembly is documented as part of the GCC information archive. Executing the command `info gcc` on any machine with GCC installed will give a hierarchical document reader. Inline assembly is documented by first following the link titled “C Extensions” and then the link titled “Extended Asm.” Unfortunately, the documentation is somewhat incomplete and imprecise.

The basic form of inline assembly is to write code that looks like a procedure call:

```
asm( code-string );
```

where *code-string* is an assembly code sequence given as a quoted string. The compiler will insert this string verbatim into the assembly code being generated, and hence the compiler-supplied and the user-supplied assembly will be combined. The compiler does not check the string for errors, and so the first indication of a problem might be an error report from the assembler.

We illustrate the use of `asm` by an example where having access to the condition codes can be useful. Consider functions with the following prototypes:

```
int ok_smul(int x, int y, int *dest);
```

```
int ok_umul(unsigned x, unsigned y, unsigned *dest);
```

Each is supposed to compute the product of arguments `x` and `y` and store the result in the memory location specified by argument `dest`. As return values, they should return 0 when the multiplication overflows and 1 when it does not. We have separate functions for signed and unsigned multiplication, since they overflow under different circumstances.

Examining the documentation for the IA32 multiply instructions `mul` and `imul`, we see that both set the carry flag `CF` when they overflow. Examining Figure 3.9, we see that the instruction `setae` can be used to set the low-order byte of a register to 0 when this flag is set and to 1 otherwise. Thus, we wish to insert this instruction into the sequence generated by the compiler.

In an attempt to use the least amount of both assembly code and detailed analysis, we attempt to implement `ok_smul` with the following code:

code/asm/okmul.c

```
/* First attempt. Does not work */
int ok_smul1(int x, int y, int *dest)
{
   int result = 0;

   *dest = x*y;
   asm("setae %al");
   return result;
}
```

code/asm/okmul.c

The strategy here is to exploit the fact that register `%eax` is used to store the return value. Assuming the compiler uses this register for variable `result`, the first line will set the register to 0. The inline assembly will insert code that sets the low-order byte of this register appropriately, and the register will be used as the return value.

Unfortunately, GCC has its own ideas of code generation. Instead of setting register `%eax` to 0 at the beginning of the function, the generated code does so at the very end, and so the function always returns 0. The fundamental problem is that the compiler has no way to know what the programmer’s intentions are, and how the assembly statement should interact with the rest of the generated code. By a process of trial and error (we will develop more systematic approaches shortly), we were able to generate working, but less than ideal code as follows:

code/asm/okmul.c

```
/* Second attempt. Works in limited contexts */
int dummy = 0;

int ok_smul2(int x, int y, int *dest)
{
   int result;

   *dest = x*y;
   result = dummy;
   asm("setae %al");
   return result;
}
```

code/asm/okmul.c

This code uses the same strategy as before, but it reads a global variable `dummy` to initialize `result` to 0. Compilers are typically more conservative about generating code involving global variables, and therefore less likely to rearrange the ordering of the computations. The above code depends on quirks of the compiler to get proper behavior. In fact, it only works when compiled with optimization enabled (command line flag `-O`). When compiled without optimization, it stores `result` on the stack and retrieves its value just before returning, overwriting the value set by the `setae` instruction. The compiler has no way of knowing how the inserted assembly language relates to the rest of the code, because we provided the compiler no such information.

Extended Form of `asm`

GCC provides an extended version of the `asm` that allows the programmer to specify which program values are to be used as operands to an assembly code sequence and which registers are overwritten by the assembly code. With this information the compiler can generate code that will correctly set up the required source values, execute the assembly instructions, and make use of the computed results. It will also have information it requires about register usage so that important program values are not overwritten by the assembly code instructions.

The general syntax of an extended assembly sequence is as follows:

`asm(` *code-string* `:` *output-list* `:` *input-list* `:` *overwrite-list* `);`

where the square brackets denote optional arguments. The declaration contains a string describing the assembly code sequence, followed by optional lists of outputs (i.e., results generated by the assembly code), inputs (i.e., source values for the assembly code), and registers that are overwritten by the assembly code. These lists are separated by the colon (‘`:`’) character. As the square brackets show, we only include lists up to the last nonempty list.

The syntax for the code string is reminiscent of that for the format string in a `printf` statement. It consists of a sequence of assembly code instructions separated by the semicolon (‘`;`’) character. Input and output operands are denoted by references `%0`, `%1`, and so on, up to possibly `%9`. Operands are numbered, according to their ordering first in the output list and then in the input list. Register names such as “`%eax`” must be written with an extra ‘`%`’ symbol, e.g., “`%%eax`.”

The following is a better implementation of `ok_smul` using the extended assembly statement to indicate to the compiler that the assembly code generates the value for variable `result`:

code/asm/okmul.c

```
/* Uses the extended assembly statement to get reliable code */
int ok_smul3(int x, int y, int *dest)
{
   int result;

   *dest = x*y;

   /* Insert the following assembly code:
        setae %bl                  # Set low-order byte
        movzbl %bl, result         # Zero extend to be result
   */
   asm("setae %%bl; movzbl %%bl,%0"
         : "=r" (result)      /* Output */
         :                    /* No inputs */
         : "%ebx"             /* Overwrites */
         );

   return result;
}
```

code/asm/okmul.c

The first assembly instruction stores the test result in the single-byte register `%bl`. The second instruction then zero-extends and copies the value to whatever register the compiler chooses to hold `result`, indicated by operand `%0`. The output list consists of pairs of values separated by spaces. (In this example there is only a single pair). The first element of the pair is a string indicating the operand type, where '`r`' indicates an integer register and '`=`' indicates that the assembly code assigns a value to this operand. The second element of the pair is the operand enclosed in parentheses. It can be any assignable value (known in C as an *lvalue*). The input list has the same general format, while the overwrite list simply gives the names of the registers (as quoted strings) that are overwritten.

The code shown above works regardless of the compilation flags. As this example illustrates, it may take a little creative thinking to write assembly code that will allow the operands to be described in the required form. For example, there are no direct ways to specify a program value to use as the destination operand for the `setae` instruction, since the operand must be a single byte. Instead, we write a code sequence based on a specific register and then use an extra data movement instruction to copy the resulting value to some part of the program state.

One would expect the same code sequence could be used for `ok_umul`, but GCC uses the `imull` (signed multiply) instruction for both signed and unsigned multiplication. This generates the correct value for either product, but it sets the carry flag according to the rules for signed multiplication. We therefore need to include an assembly-code sequence that explicitly performs unsigned multiplication using the `mull` instruction, as follows:

code/asm/okmul.c

```
/* Uses the extended assembly statement */
int ok_umul(unsigned x, unsigned y, unsigned *dest)
{
   int result;

   /* Insert the following assembly code:
       movl x,%eax           # Get x
       mull y                # Unsigned multiply by y
       movl %eax, *dest      # Store low-order 4 bytes at dest
       setae %dl             # Set low-order byte
       movzbl %dl, result    # Zero extend to be result
   */
   asm("movl %2,%%eax; mull %3; movl %%eax,%0;
         setae %%dl; movzbl %%dl,%1"
       : "=r" (*dest), "=r" (result)  /* Outputs */
       : "r" (x), "r" (y)             /* Inputs */
       : "%eax", "%edx"               /* Overwrites */
       );

   return result;
}
```

code/asm/okmul.c

Recall that the `mull` instruction requires one of its arguments to be in register `%eax` and is given the second argument as an operand. We indicate this in the `asm` statement by using a `movl` to move program value `x` to `%eax` and indicating that program value `y` should be the argument for the `mull` instruction. The instruction then stores the 8-byte product in two registers with `%eax` holding the low-order 4 bytes and `%edx` holding the high-order bytes. We then use register `%edx` to construct the return value. As this example illustrates, comma ('`,`') characters are used to separate pairs of operands in the input and output lists, and register names in the overwrite list. Note that we were able to specify `*dest` as an output of the second `movl` instruction, since this is an assignable value. The compiler then generates the correct machine code to store the value in `%eax` at this memory location.

Although the syntax of the `asm` statement is somewhat arcane, and its use makes the code less portable, this statement can be very useful for writing programs that accesses machine-level features using a minimal amount of assembly code. We have found that a certain amount of trial and error is required to get code that works. The best strategy is to compile the code with the `-S` switch and then examine the generated assembly code to see if it will have the desired effect. The code should be tested with different settings of switches such as with and without the `-O` flag.

15. Introduction to Linkers

Linking is the process of collecting and combining the various pieces of code and data that a program needs in order to be *loaded* (copied) into memory and executed. Linking can be performed at *compile tim*e, when the source code is translated into machine code, at *load tim*e, when the program is loaded into memory and executed by the *loade*r, and even at *run tim*e, by application programs. On early computer systems, linking was performed manually. On modern systems, linking is performed automatically by programs called *linker*s.

Linkers play a crucial role in software development because they enable *separate compilatio*n. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files. Linking is usually handled quietly by the linker, and is not an important issue for students who are building small programs in introductory programming classes. So why bother learning about linking?

- *Understanding linkers will help you build large programs.* Programmers who build large programs often encounter linker errors caused by missing modules, missing libraries, or incompatible library versions. Unless you understand how a linker resolves references, what a library is, and how a linker uses a library to resolve references, these kinds of errors will be baffling and frustrating.
- *Understanding linkers will help you avoid dangerous programming errors.* The decisions that Unix linkers make when they resolve symbol references can silently affect the correctness of your pro-grams. Programs that incorrectly define multiple global variables pass through the linker without any warnings in the default case. The resulting programs can exhibit baffling run-time behavior and are extremely difficult to debug. We will show you how this happens and how to avoid it.
- *Understanding linking will help you understand how language scoping rules are implemented.* For example, what is the difference between global and local variables? What does it really mean when you define a variable or function with the `static` attribute?
- *Understanding linking will help you understand other important systems concepts.* The executable object files produced by linkers play key roles in important systems functions such as loading and running programs, virtual memory, paging, and memory mapping.
- *Understanding linking will enable you to exploit shared libraries.* For many years, linking was considered to be fairly straightforward and uninteresting. However, with the increased importance of shared libraries and dynamic linking in modern operating systems, linking is a sophisticated process that provides the knowledgeable programmer with significant power. For example, many software products use shared libraries to upgrade shrink-wrapped binaries at run time. Also, most Web servers rely on dynamic linking of shared libraries to serve dynamic content.

Chapter 7 in the book[3] is a thorough discussion of all aspects of linking, from traditional static linking, to dynamic linking of shared libraries at load time, to dynamic linking of shared libraries at run time. We will describe the basic mechanisms using real examples, and we will identify situations where linking issues can affect the performance and correctness of your programs. To keep things concrete and understandable, we will couch our discussion in the context of an IA32 machine running a version of Unix, such as Linux or Solaris, that uses the standard ELF object file format. However, it is important to realize that the basic concepts of linking are universal, regardless of the operating system, the ISA, or the object file format. Details may vary, but the concepts are the same.

[3] From where these notes were taken.

16 Compiler Drivers

Consider the C program in Figure 11. It consists of two source files, `main.c` and `swap.c`. Function `main()` calls `swap`, which swaps the two elements in the external global array `buf`. Granted, this is a strange way to swap two numbers, but it will serve as a small running example throughout this chapter that will allow us to make some important points about how linking works.

Most compilation systems provide a *compiler driver* that invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user. For example, to build the example program using the GNU compilation system, we might invoke the GCC driver by typing the following command to the shell:

```
unix> gcc -O2 -g -o p main.c swap.c
```

Figure 12 summarizes the activities of the driver as it translates the example program from an ASCII source file into an executable object file. (If you want to see these steps for yourself, run GCC with the `-v` option.) The driver first runs the C preprocessor (`cpp`), which translates the C source file `main.c` into an ASCII intermediate file `main.i`:

```
cpp [other arguments] main.c /tmp/main.i
```

Next, the driver runs the C compiler (`cc1`), which translates `main.i` into an ASCII assembly language file `main.s`.

```
cc1 /tmp/main.i main.c -O2 [other arguments] -o /tmp/main.s
```

Then, the driver runs the assembler (`as`), which translates `main.s` into a *relocatable object file* `main.o`:

```
as [other arguments] -o /tmp/main.o /tmp/main.s
```

code/link/main.c

```
/* main.c */
void swap();

int buf[2] = {1, 2};

int main()
{
   \swap();
   return 0;
}
```

code/link/main.c

(a) `main.c`

code/link/swap.c

```
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
   int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

code/link/swap.c

(b) `swap.c`

Figure 11: **Example program 1:** The example program consists of two source files, `main.c` and `swap.c`. The `main` function initializes a two-element array of ints, and then calls the `swap` function to swap the pair.

The driver goes through the same process to generate `swap.o`. Finally it runs the linker program `ld`, which combines `main.o` and `swap.o`, along with the necessary system object files, to create the *executable object file* p:

```
ld -o p [system object files and args] /tmp/main.o /tmp/swap.o
```

To run the executable p, we type its name on the Unix shell's command line:

```
unix> ./p
```

The shell invokes a function in the operating system called the *loade*r, which copies the code and data in the executable file `p` into memory, and then transfers control to the beginning of the program.

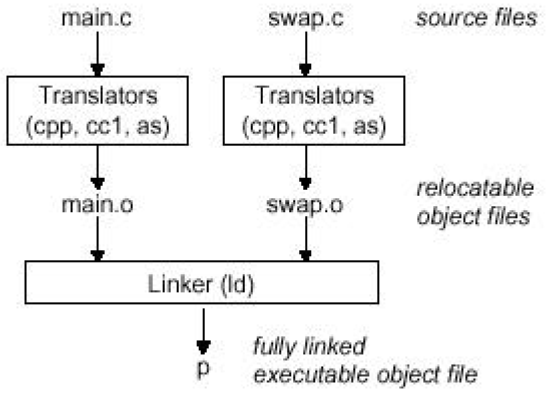

main.c
swap.c
source files
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
swap.o
relocatable object files
Linker (ld)
p
fully linked executable object file

Figure 12: **Static linking.** The linker combines relocatable object files to form an executable object file p.

17. Static Linking

Static linkers such as the Unix `ld` program take as input a collection of relocatable object files and command line arguments and generate as output a fully linked executable object file that can be loaded and run. The input relocatable object files consist of various code and data sections. Instructions are in one section, initialized global variables are in another section, and uninitialized variables are in yet another section.

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

- *Symbol resolution.* Object files define and reference *symbol*s. The purpose of symbol resolution is to associate each symbol reference with exactly one symbol definition.
- *Relocation.* Compilers and assemblers generate code and data sections that start at address zero. 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 following sections describe these tasks in more detail. As you read, keep in mind the basic facts of 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.

18. 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 file formats 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.) Early versions of System V Unix used the Common Object File format (COFF). Windows NT uses a variant of COFF called the Portable Executable (PE) format. Modern Unix systems — such as Linux, later versions of System V Unix, BSD Unix variants, and Sun Solaris — use the Unix *Executable and Linkable Format (ELF*). Although our discussion will focus on ELF, the basic concepts are similar, regardless of the particular format.

Relocatable Object Files

Figure 13 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., IA32) the file offset of the *section header tabl*e, 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 tabl*e, which contains a fixed sized entry for each section in the object file.

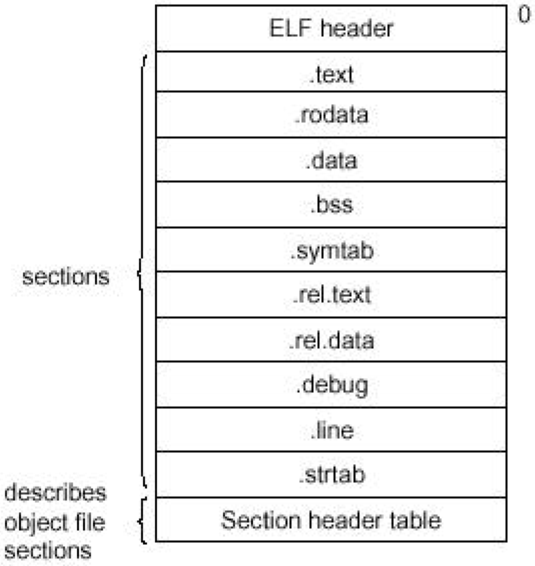


Figure 13: **Typical ELF relocatable object file.**

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 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 C variables. This section occupies no actual space in the object file; it is merely a place holder. 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.

`.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`. 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.

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 Storage Start" instruction 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!". **End Aside.**

Executable Object Files

We have seen how the linker merges multiple object modules into a single executable object file. Our 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 14 summarizes the kinds of information in a typical ELF executable file.

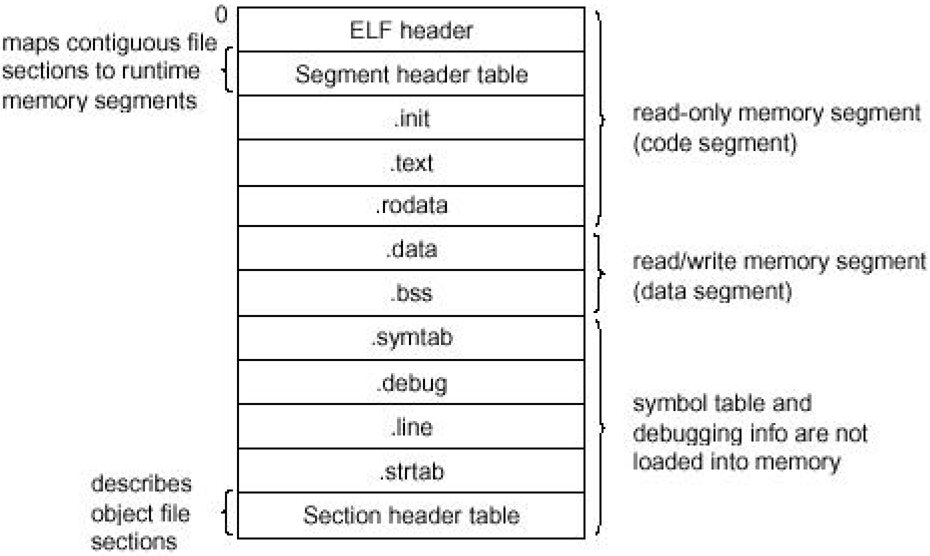

0
maps contiguous file sections to runtime memory segments
ELF header
Segment header table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header table
describes object file sections
read-only memory segment (code segment)
read/write memory segment (data segment)
symbol table and debugging info are not loaded into memory

Figure 14: **Typical ELF executable object file**

The format of an executable object file is similar to that of a relocatable object file. The ELF header describes the overall format of the file. It also includes the program's *entry point*, which is the address of the first instruction to execute when the program runs. The `.text`, `.rodata`, and `.data` sections are similar to those in a relocatable object file, except that these sections have been relocated to their eventual run-time memory addresses. The `.init` section defines a small function, called `init`, that will be called by the program's initialization code. Since the executable is *fully linked* (relocated), it needs no `.relo` sections.

19. Tools for Manipulating Object Files

There are a number of tools available on Unix systems to help you understand and manipulate object files. In particular, the GNU *binutils* package is especially helpful and runs on every Unix platform.

AR**:** Creates static libraries, and inserts, deletes, lists, and extracts members.

STRINGS**:** Lists all of the printable strings contained in an object file.

STRIP**:** Deletes symbol table information from an object file.

NM**:** Lists the symbols defined in the symbol table of an object file.

SIZE**:** Lists the names and sizes of the sections in an object file.

READELF**:** Displays the complete structure of an object file, including all of the information encoded in the ELF header. Subsumes the functionality of SIZE and NM.

OBJDUMP**:** The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the `.text` section.

Unix systems also provide the `ldd` program for manipulating shared libraries:

LDD**:** Lists the shared libraries that an executable needs at run time.