Chapter 1

Program Structure

In the beginning there were 0s and 1s….
--GRR

1.1 Introduction

In this chapter we will talk about memory: bits, bytes and how data is represented in the computer. We will also explain what a program is and how they are laid out in both memory and in the executable file. We will conclude by explaining the development cycle of a program.

1.2 The 0s and the 1s

All data in modern computers is represented as 0s and 1s. The main reason for using binary numbers is because it simplifies the electronics that implement digital circuits. For example, a digit 0 can be represented by 0 Volts and a digit of 1 can be represented by +5 Volts. If more digits were used, more levels will be needed increasing the complexity of the electronics that stores and manipulates the data. Binary numbers made of 0s and 1s are the most basic and simple representation that we can have of numbers. [1]

Traditionally computers have used 0V to represent a 0 and +5V to represent a 1, called 5V logic. However, in this age of mobile computing where battery power savings are important, lower voltage levels have been used. For instance, 3.3V logic and 1.8V logic are now the most typical in PCs, phones, tablets, and a multitude of other embedded devices.

1.3 Bits and Bytes

Bits are grouped in bytes, where each byte is made of 8 bits. In modern computers a byte is the smallest unit that can be addressed by a CPU. A byte can be used to store values such as 00000000 (0 in decimal) to 11111111 (255 in decimal). These are very small numbers, so usually larger groups of bytes are used to represent other types of data .

1.4 Representation of Numbers in Memory

[1] You may ask then why we humans use decimal representation. The most likely reason is that we have 5 fingers in each hand for a total of 10 fingers. A long time ago, before any number representation was invented, we humans used our fingers to do computations and count. We can say that the choice of the decimal system was a result of our human anatomy.

Integers are represented in groups of 2 bytes (short int), 4 bytes (int) , 8 bytes (long int) and in some architectures 16 bytes (long long int) variables.

For example, the following 8 bytes represent a long int. Each 1 digit will represent a power of 2^i at that position for that number. You very well could find this sequence of bytes in memory representing this number.

```
00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
63       55       47       39       31       23       15       7      0
```

$$2^{23}+2^{19}+2^{16}+2^{13}+2^{10}+2^{7}+2^{4}+2^{1}= 8987794$$

Negative numbers typically use a representation called “complements of two”, that is, a negative number is obtained by inverting the corresponding positive number and then adding 1. This representation allows using common positive integer arithmetic to do the addition and subtraction operations.

For instance, the number represented above as a negative number can be obtained as:

```
Original: 00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
Negated:  11111111 11111111 11111111 11111111 11111111 01110110 11011011 01101101
Plus 1    11111111 11111111 11111111 11111111 11111111 10001001 00100100 10010010
          63       55       47       39       31       23       15       7      0
```

$$2^{23}+2^{19}+2^{16}+2^{13}+2^{10}+2^{7}+2^{4}+2^{1}= 8987794$$

If we have the binary representation of 8987794 ***added*** to same number in complements of two representing -8987794 we will obtain 0 as expected.

```
 8987794: 00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
+
-8987794: 11111111 11111111 11111111 11111111 11111111 10001001 00100100 10010010
 -------  -----------------------------------------------------------------------
       0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

1.5 Representation of Strings in Memory

Basic strings in C language are represented in memory as a sequence of bytes delimited by a 0 value. Each byte represents a character in ASCII representation. ASCII is the standard that translates characters in the English alphabet to numbers. ASCII stands for American Standard Code for Information Interchange.

```
32:  48:0 64:@ 80:P 96:`  112:p
33:! 49:1 65:A 81:Q 97:a  113:q
34:" 50:2 66:B 82:R 98:b  114:r
35:# 51:3 67:C 83:S 99:c  115:s
36:$ 52:4 68:D 84:T 100:d 116:t
37:% 53:5 69:E 85:U 101:e 117:u
38:& 54:6 70:F 86:V 102:f 118:v
39:' 55:7 71:G 87:W 103:g 119:w
40:( 56:8 72:H 88:X 104:h 120:x
41:) 57:9 73:I 89:Y 105:i 121:y
42:* 58:: 74:J 90:Z 106:j 122:z
43:+ 59:; 75:K 91:[ 107:k 123:{
44:, 60:< 76:L 92:\ 108:l 124:|
45:- 61:= 77:M 93:] 109:m 125:}
46:. 62:> 78:N 94:^ 110:n 126:~
47:/ 63:? 79:O 95:_ 111:o 127:
```

ASCII Table of printable characters

For example, the string “Hello world is represented by the equivalent ASCII characters delimited by a NULL character.

To be able to represent characters in other languages, the Unicode standard was created. Unicode extends the ASCII standard and it uses two bytes to represent a character instead of one. In unicode it is possible to represent the characters of mostly all languages in the world.

1.5 Memory of a Program

From the point of view of a program, the memory in the computer is an array of bytes that goes from address 0 to 2^{64}-1 (0 to 16 Hexabytes-1, that is 0 to 17 179 869 183 Gigabytes) assuming a 64-bit architecture. This is called the ***Address Space*** of the program.

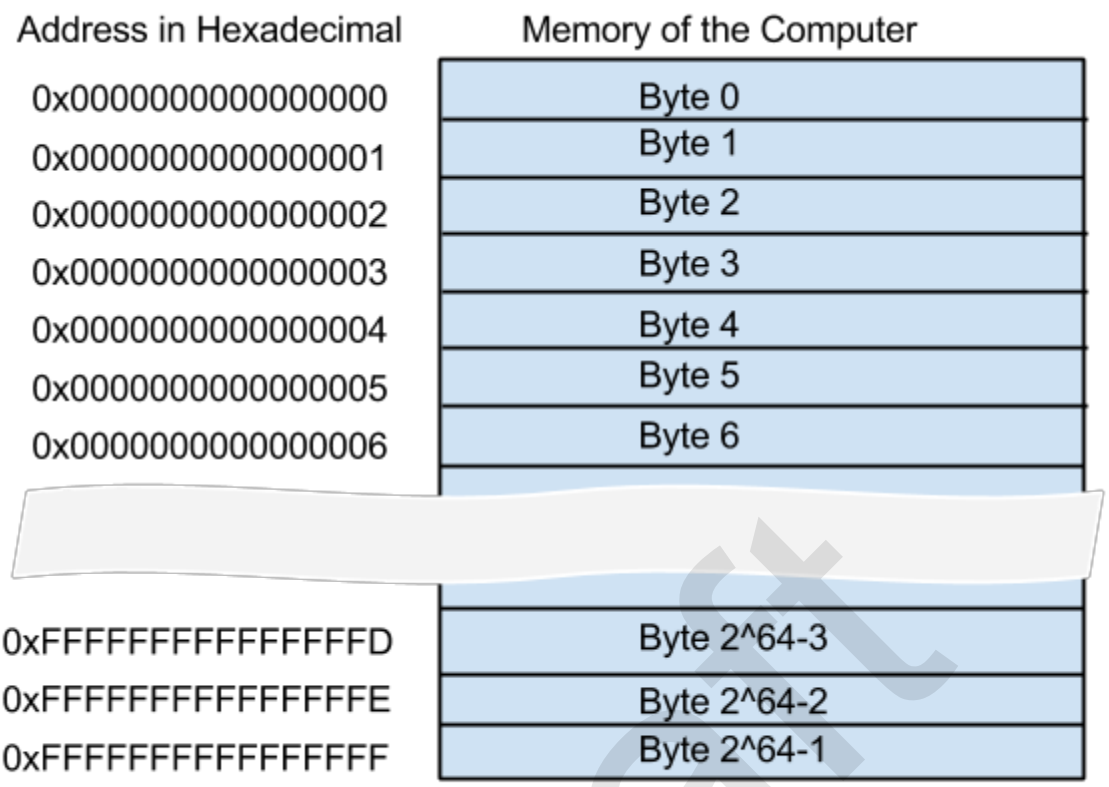


Figure 1.1: Computer Memory as an Array of Bytes (char[])

Every program that runs in memory will see the memory this way. In languages such as C/C++ or assembly language it is possible to access the location of any of these bytes using pointers and pointer dereferencing.

Theoretically a program may access any of these locations. However, there are gaps in the address space. Not all the addresses are "mapped" to physical memory. When accessing memory in these gaps, the program will get an exception called Segmentation Violation or SEGV and the program will crash.

Types such as integers, floats, doubles, or strings are represented as one or more of these bytes in memory. Everything stored in memory is represented with bytes. It is up to the program and the programmer to give meaning to what is stored in these bytes in memory.

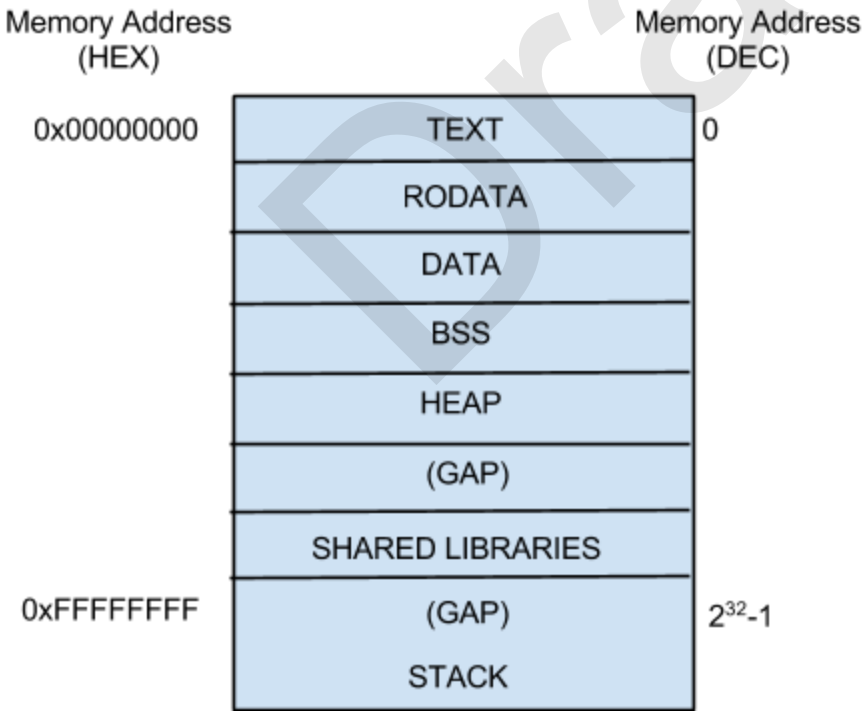
1.5 Sections of a Program

The memory of the computer is used to store both the program code, and the data that the program manipulates. This computer architecture which uses the same memory to store the program, and the data that the program manipulates is called "Von Neumann Architecture" in honor of John Von Neumann that influenced this architecture.

An executable program in memory is divided into sections. This division is due to the fact that different sections of the program need to have different characteristics. For example, the program instructions stored in the TEXT section, need to be in memory that have read and execution permission but it does not have write permissions. The DATA section stores the global variables of a program and it has read and write permissions but not executable permissions.

Here is a list of the different sections of a program. It is not exhaustive but it contains the most important sections.

- TEXT - Instructions that the program runs
- RODATA - Stores Read-only data. These are constants that the program uses, like strings constants or other variables defined like “const int
- DATA – Initialized global variables.
- BSS – Uninitialized global variables. They are initialized to zeroes.
- HEAP – Memory returned when calling malloc/new. It grows upwards.
- SHARED LIBRARIES – Also called dynamic libraries. They are libraries shared with other processes.
- STACK – It stores local variables and return addresses. It grows downwards.



Each section has different permissions: read/write/execute or a combination of them.

Each dynamic library may have its own text, data, and bss sections.

Each program has its own view of the memory that is independent of each other. This view is called the “Address Space” of the program. If a process modifies a byte in its own address space, it will not modify the same location of the address space of another process.

Program hello.c that prints the addresses of variables at every section.

```
#include <stdio.h>
int a = 5; // Stored in data section
int b[20]; // Stored in bss
const char * hello = "Hello world";

int main() { // Stored in text
     int x; // Stored in stack
     int *p =(int*) malloc(sizeof(int)); //Stored in heap
     printf("(Data) &a=0x%lx\n", &a)
     printf("(Bss) &b[0]=0x%lx\n", &b[0]);
     printf("(Stack) &x=0x%lx\n", &x);
     printf("(Heap) p=0x%lx\n", p);
     printf("(ROData) "Hello"=0x%lx\n", hello);
     printf("(TEXT) main=0x%lx\n", main);
}
```


1.6 Memory Gaps

Between each memory section there may be gaps that do not have any memory mapping. For example, before the beginning of the memory and the TEXT section usually there is a gap. If the program tries to access a memory gap, the OS will send a SEGV signal. A signal is a type of interrupt that the OS sends to a program. That is why writing to or reading through a pointer that is initialized to NULL (address 0) causes the program to get a SEGV from the OS and get terminated. The default behavior of a SEGV is to kill the program and write a core file in the current directory. The core file contains the value of the variables global and local at the time of the SEGV. The core file can be used for “post mortem” debugging to find out where the program crashed.

```
bash> gdb program-name core
gdb> where
```

1.7 What is a program?

A program is a file in a special format that contains all the necessary information to load an application into memory and make it run. A program file includes:

- machine instructions - This is the code that will make the TEXT section in the program.

- Initialized data - This is the list of initialized variables and string constants that are defined in the program. The initialized data is stored in the DATA and RODATA sections.
- List of shared libraries that the program needs to execute.
- List of zero initialized memory sections BSS, STACK, HEAP etc that need to be allocated before the
- List of defined and undefined symbols (variables and function names) in the executable. Some of these symbols do not have a value yet and will be known only until the Operating System loads the program and links it with other shared libraries when the program is loaded into memory.

The most important types of executable file formats are:

> **ELF** – Executable and Link Format. It is used in most UNIX systems (Solaris, Linux)
> **PE/COFF** – Portable Executable/Common Object File Format. It is used in Windows systems.
> **a.out** – Used in BSD (Berkeley Standard Distribution) and early UNIX. It was very restrictive. It is not used anymore. It is now the default filename when you compile a file on unix without specifying an output filename.

BSD UNIX and AT&T UNIX are the predecessors of the modern UNIX flavors like Solaris and Linux.

These formats start with a well defined header. The first few bytes of the header contain a “Magic Number” that is used as one of the tests to differentiate the type of executable and the integrity of the executable. For example, ELF files start with the sequence:

```
#define ELFMAG      "\177ELF"
```

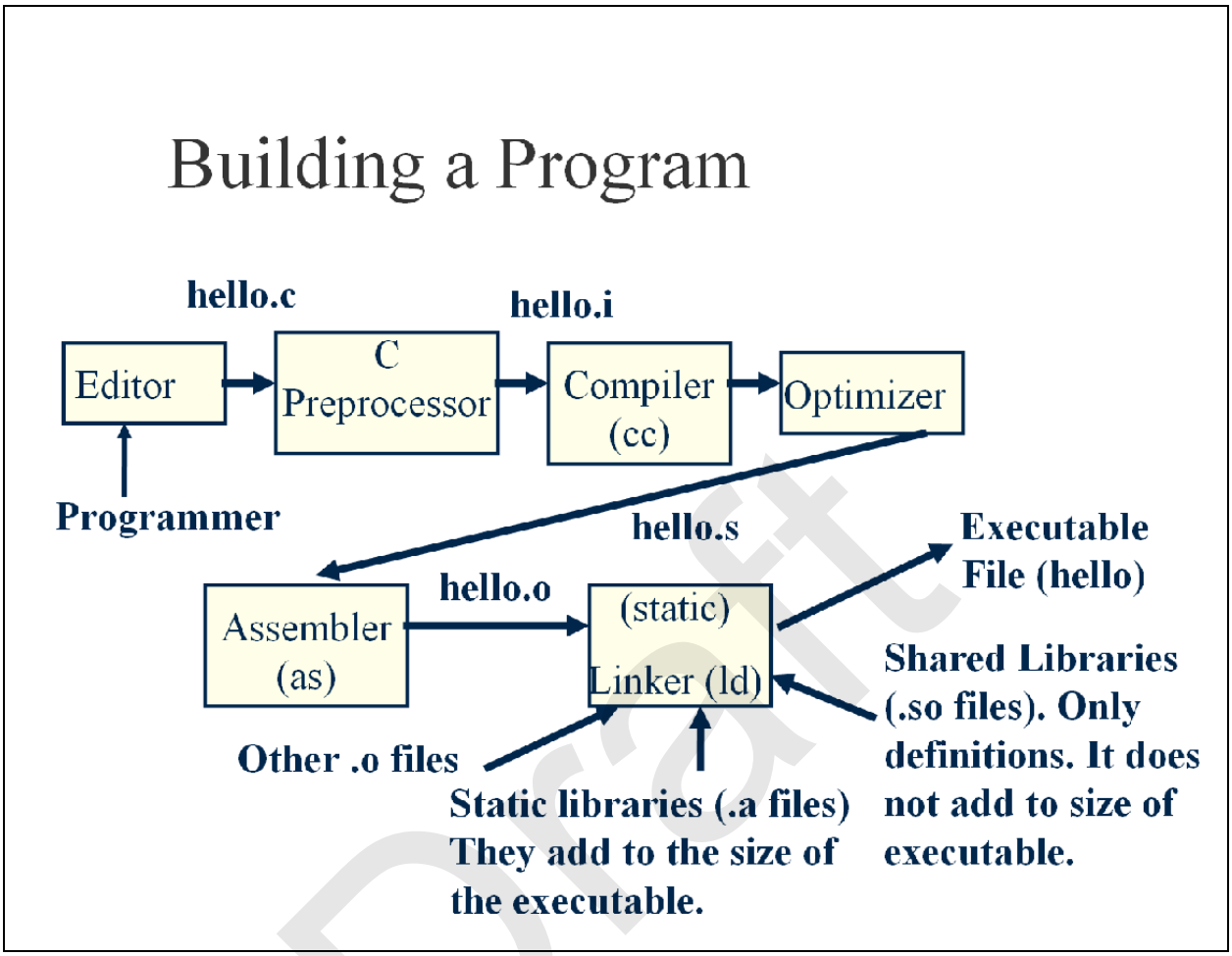
For more information on the ELF format type “man elf” in a Unix System.

1.8 Building a Program

The programmer first writes a program such as ***hello.c*** using an editor program. Then the programmer compiles the program.

During the compilation process the compiler performs the following steps. These steps may be executed by a single program or by multiple separate programs coordinated by the compiler.

Compilation Steps
1. C-Preprocessor
2. Compilation/Optimization
3. Assembly
4. Linking



C Preprocessor

The pre processor executes the instructions such as #define, #include, #ifdef etc operators and generates a ***hello.i*** temporary file. The operation #include<stdio.h> reads the file in /usr/include/stdio.h and includes the file in hello.i. If the included file as other #include operators , these files are included as well. The #define operators defines macros that are expanded by the processor when the names of these macros appear in the source code. For example, the C preprocessor, given the #define A(x) (1+x) will substitute A(5) by(1+5) in the file hello.i .The resulting file is stored in a temporary file ***hello.i.*** The name of the preprocessor program in UNIX systems is called cpp. You may pass the flag "-E" to tell the gcc compiler to stop just after preprocessing the program to be able to analyze the .i temporary file. Notice the expansion of A(x) below marked in red.

```
hello.c:
```

```
#include <stdio.h>
#define A(x) (1 + x)

int main()
{
        printf("Hello world\n");
        printf("A=%d\n", A(5));
}
```

Stop file just after preprocessor:

```
bash$  gcc -E hello.c > hello.i
bash$ cat hello.i
```

```
hello.i:
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
extern int fflush (FILE *__stream);
...
# 252 "/usr/include/stdio.h" 3 4
extern int fflush_unlocked (FILE *__stream);
...
# 266 "/usr/include/stdio.h" 3 4
...
extern FILE *fopen (const char *__restrict __filename,
      const char *__restrict __modes) ;
extern int fclose (FILE *__stream);
...
extern void funlockfile (FILE *__stream) __attribute__
((__nothrow__ , __leaf__));
# 943 "/usr/include/stdio.h" 3 4
# 3 "hello.c" 2
int main()
{
 printf("Hello world\n");
 printf("A=%d\n", (1 + 5));
}
```

Compilation

The compiler compiles hello.i, optimizes it and generates an assembly instruction listing ***hello.s.***

You may use the flag “-S”o stop the gcc compiler after generating the assembly output .

```
bash$ gcc -S hello.c
```

```
hello.s:
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $6, %esi
        movl    $.LC1, %edi
        movl    $0, %eax
        call    printf
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Gentoo 4.7.3-r1 p1.3, pie-0.5.5) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```


Assembly

The assembler (usr/bin/as) assembles hello.s and generates an object file hello.o
The file hello.o, also called “Object File”, is not yet an executable file. It has to be linked together with other libraries such as the C standard library in /usr/lib/libc.a that provides implementation of printf, malloc and other standard C functions.

Linking

The linker or also called “Static Linker” is used to put together object files ending with *.o and static libraries ending with *.a.

The linker also takes the definitions in shared libraries and verifies that the symbols (functions and variables) needed by the program are completely satisfied. If there is symbol that is not

defined in either the executable or shared libraries, the linker will give an error.

Static Libraries (*.a)
A collection of object files (*.o) that contain function definitions and variables. The static libraries often have the suffix .a from “archive”. When linked, the linker searches through the object files to find what objects satisfy the symbols used by the program that is linked and it will link these object files in the collection. Static Libraries add to the total size of the program in disk.

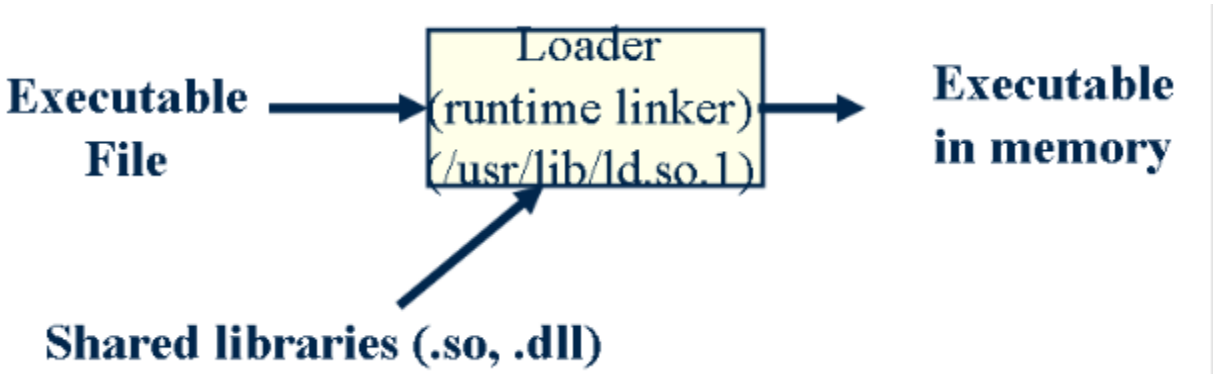
Dynamic Libraries (*.so, *.sl, *.dll)
These are also called “Shared Objects”, or “Shared Libraries”, or “Dynamically Linked Libraries”. These libraries are loaded only once for the whole computer, and they are shared across multiple programs that run at the same time. This saves memory space since only one shared library instance is loaded for all existing programs that use it. For this reason, shared libraries do not add to the total size of the program.

Shared libraries are preferred to static libraries since they are more space efficient and makes the program load faster in most cases. Static and shared libraries are found in /usr/lib on unix systems.

The compiler (cc or gcc) by default hides all these intermediate steps. You can use compiler options to run each step independently.

Loading a Program

The loader is a program that is used to run an executable file in a process. Before the program starts running, the loader allocates space for all the sections of the executable file (text, data, bss etc). It loads into memory the executable and shared libraries (if not loaded yet). It also writes (resolves) any values in the executable to point to the functions/variables in the shared libraries.(E.g. calls to printf in hello.c). Once the memory image is ready, the loader jumps to the _start entry point that calls init() of all libraries and initializes static constructors. Then it calls main() and the program begins. _start also calls exit() when main() returns. The loader is also called ”runtime linker” and it is also a shared library itself.



Listing Shared Library Dependencies
Use the "ldd" command to list the shared libraries that an executable needs.

```
bash$ ldd hello
        linux-vdso.so.1 (0x00007fff709ff000)
        libc.so.6 => /lib64/libc.so.6 (0x00007ff729340000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff7296e7000)
```


Chapter Summary Questions

1. What two basic values are used to represent all data in your computer?
2. Name five of the memory sections in an executable file and briefly describe their purpose.
3. What are the four main steps in building or compiling a program, focusing on the ones that you don't have interaction with (i.e. not the editor or running the program).
4. Many novice programmers believe that the main() function is where the program immediately begins. What is the actual entry point for any program?