Chapter 2

Representing and Manipulating Information

Modern computers store and process information represented as two-valued signals. These lowly binary digits, or *bits*, form the basis of the digital revolution. The familiar decimal, or base-10, representation has been in use for over 1000 years, having been developed in India, improved by Arab mathematicians in the 12th century, and brought to the West in the 13th century by the Italian mathematician Leonardo Pisano (c. 1170 – c. 1250), better known as Fibonacci. Using decimal notation is natural for ten-fingered humans, but binary values work better when building machines that store and process information. Two-valued signals can readily be represented, stored, and transmitted, for example, as the presence or absence of a hole in a punched card, as a high or low voltage on a wire, or as a magnetic domain oriented clockwise or counterclockwise. The electronic circuitry for storing and performing computations on two-valued signals is very simple and reliable, enabling manufacturers to integrate millions, or even billions, of such circuits on a single silicon chip.

In isolation, a single bit is not very useful. When we group bits together and apply some *interpretation* that gives meaning to the different possible bit patterns, however, we can represent the elements of any finite set. For example, using a binary number system, we can use groups of bits to encode nonnegative numbers. By using a standard character code, we can encode the letters and symbols in a document. We cover both of these encodings in this chapter, as well as encodings to represent negative numbers and to approximate real numbers.

We consider the three most important representations of numbers. *Unsigned* encodings are based on traditional binary notation, representing numbers greater than or equal to 0. *Two's-complement* encodings are the most common way to represent *signed* integers, that is, numbers that may be either positive or negative. *Floating-point* encodings are a base-two version of scientific notation for representing real numbers. Computers implement arithmetic operations, such as addition and multiplication, with these different representations, similar to the corresponding operations on integers and real numbers.

Computer representations use a limited number of bits to encode a number, and hence some operations can *overflow* when the results are too large to be represented. This can lead to some surprising results. For example, on most of today's computers (those using a 32-bit representation of data type `int`), computing the expression

```
200 * 300 * 400 * 500
```

yields $-884{,}901{,}888$. This runs counter to the properties of integer arithmetic—computing the product of a set of positive numbers has yielded a negative result.

On the other hand, integer computer arithmetic satisfies many of the familiar properties of true integer arithmetic. For example, multiplication is associative and commutative, so that computing any of the following C expressions yields $-884{,}901{,}888$:

```
(500  *  400) * (300 * 200)
((500 *  400) * 300) * 200
((200 *  500) * 300) * 400
400   * (200 * (300 * 500))
```

The computer might not generate the expected result, but at least it is consistent!

Floating-point arithmetic has altogether different mathematical properties. The product of a set of positive numbers will always be positive, although overflow will yield the special value $+\infty$. Floating-point arithmetic is not associative due to the finite precision of the representation. For example, the C expression `(3.14+1e20)-1e20` will evaluate to 0.0 on most machines, while `3.14+(1e20-1e20)` will evaluate to 3.14. The different mathematical properties of integer vs. floating-point arithmetic stem from the difference in how they handle the finiteness of their representations—integer representations can encode a comparatively small range of values, but do so precisely, while floating-point representations can encode a wide range of values, but only approximately.

By studying the actual number representations, we can understand the ranges of values that can be represented and the properties of the different arithmetic operations. This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler. As we will describe, a number of computer security vulnerabilities have arisen due to some of the subtleties of computer arithmetic. Whereas in an earlier era program bugs would only inconvenience people when they happened to be triggered, there are now legions of hackers who try to exploit any bug they can find to obtain unauthorized access to other people's systems. This puts a higher level of obligation on programmers to understand how their programs work and how they can be made to behave in undesirable ways.

Computers use several different binary representations to encode numeric values. You will need to be familiar with these representations as you progress into machine-level programming in Chapter 3. We describe these encodings in this chapter and show you how to reason about number representations.

We derive several ways to perform arithmetic operations by directly manipulating the bit-level representations of numbers. Understanding these techniques will be important for understanding the machine-level code generated by compilers in their attempt to optimize the performance of arithmetic expression evaluation.

Our treatment of this material is based on a core set of mathematical principles. We start with the basic definitions of the encodings and then derive such properties as the range of representable numbers, their bit-level representations, and the properties of the arithmetic operations. We believe it is important for you to

C version	GCC command line option
GNU 89	*none*, `-std=gnu89`
ANSI, ISO C90	`-ansi`, `-std=c89`
ISO C99	`-std=c99`
GNU 99	`-std=gnu99`

Figure 2.1: **Specifying different versions of C to GCC**

examine the material from this abstract viewpoint, because programmers need to have a clear understanding of how computer arithmetic relates to the more familiar integer and real arithmetic.

Aside: How to read this chapter.
If you find equations and formulas daunting, do not let that stop you from getting the most out of this chapter! We provide full derivations of mathematical ideas for completeness, but the best way to read this material is often to skip over the derivation on your initial reading. Instead, study the examples provided, and be sure to work *all* of the practice problems. The examples will give you an intuition behind the ideas, and the practice problems engage you in *active learning*, helping you put thoughts into action. With these as background, you will find it much easier to go back and follow the derivations. Be assured, as well, that the mathematical skills required to understand this material are within reach of someone with good grasp of high school algebra. **End Aside.**

The C++ programming language is built upon C, using the exact same numeric representations and operations. Everything said in this chapter about C also holds for C++. The Java language definition, on the other hand, created a new set of standards for numeric representations and operations. Whereas the C standards are designed to allow a wide range of implementations, the Java standard is quite specific on the formats and encodings of data. We highlight the representations and operations supported by Java at several places in the chapter.

Aside: The Evolution of the C Programming Language.
As was described in an aside in Section 1.2, the C programming language was first developed by Dennis Ritchie of Bell Laboratories for use with the Unix operating system (also developed at Bell Labs). At the time, most system programs, such as operating systems, had to be written largely in assembly code, in order to have access to the low-level representations of different data types. For example, it was not feasible to write a memory allocator, such as is provided by the `malloc` library function, in other high-level languages of that era.

The original Bell Labs version of C was documented in the first edition of the book by Brian Kernighan and Dennis Ritchie [57]. Over time, C has evolved through the efforts of several standardization groups. The first major revision of the original Bell Labs C led to the ANSI C standard in 1989, by a group working under the auspices of the American National Standards Institute. ANSI C was a major departure from Bell Labs C, especially in the way functions are declared. ANSI C is described in the second edition of Kernighan and Ritchie's book [58], which is still considered one of the best references on C.

The International Standards Organization took over responsibility for standardizing the C language, adopting a version that was substantially the same as ANSI C in 1990 and hence is referred to as "ISO C90."

This same organization sponsored an updating of the language in 1999, yielding "ISO C99." Among other things this version introduced some new data types and provided support for text strings requiring characters not found in the English language.

The GNU Compiler Collection (GCC) can compile programs according to the conventions of several different versions of the C language, based on different command line options, as shown in Figure 2.1. For example, to compile program `prog.c` according to ISO C99, we could give the command line

```
unix> gcc -std=c99 prog.c
```

The options `-ansi` and `-std=c89` have the same effect—the code is compiled according to the ANSI or ISO C90 standard. (C90 is sometimes referred to as "C89," since its standardization effort began in 1989.) The option `-std=c99` causes the compiler to follow the ISO C99 convention.

As of the writing of this book, when no option is specified, the program will be compiled according to a version of C based on ISO C90, but including some features of C99, some of C++, and others specific to GCC. This version can be specified explicitly using the option `-std=gnu89`. The GNU project is developing a version that combines ISO C99, plus other features, that can be specified with command line option `-std=gnu99`. (Currently, this implementation is incomplete.) This will become the default version. **End Aside.**

2.1 Information Storage

Rather than accessing individual bits in memory, most computers use blocks of 8 bits, or *bytes*, as the smallest addressable unit of memory. A machine-level program views memory as a very large array of bytes, referred to as *virtual memory*. Every byte of memory is identified by a unique number, known as its *address*, and the set of all possible addresses is known as the *virtual address space*. As indicated by its name, this virtual address space is just a conceptual image presented to the machine-level program. The actual implementation (presented in Chapter 9) uses a combination of random-access memory (RAM), disk storage, special hardware, and operating system software to provide the program with what appears to be a monolithic byte array.

In subsequent chapters, we will cover how the compiler and run-time system partitions this memory space into more manageable units to store the different *program objects*, that is, program data, instructions, and control information. Various mechanisms are used to allocate and manage the storage for different parts of the program. This management is all performed within the virtual address space. For example, the value of a pointer in C—whether it points to an integer, a structure, or some other program object—is the virtual address of the first byte of some block of storage. The C compiler also associates *type* information with each pointer, so that it can generate different machine-level code to access the value stored at the location designated by the pointer depending on the type of that value. Although the C compiler maintains this type information, the actual machine-level program it generates has no information about data types. It simply treats each program object as a block of bytes, and the program itself as a sequence of bytes.

New to C?: The role of pointers in C.
Pointers are a central feature of C. They provide the mechanism for referencing elements of data structures, including arrays. Just like a variable, a pointer has two aspects: its *value* and its *type*. The value indicates the location of some object, while its type indicates what kind of object (e.g., integer or floating-point number) is stored at that location. **End.**

2.1.1 Hexadecimal Notation

A single byte consists of 8 bits. In binary notation, its value ranges from 00000000_2 to 11111111_2. When viewed as a decimal integer, its value ranges from 0_{10} to 255_{10}. Neither notation is very convenient for describing bit patterns. Binary notation is too verbose, while with decimal notation, it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or *hexadecimal* numbers. Hexadecimal

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.2: **Hexadecimal notation.** Each Hex digit encodes one of 16 values.

(or simply "hex") uses digits '0' through '9' along with characters 'A' through 'F' to represent 16 possible values. Figure 2.2 shows the decimal and binary values associated with the 16 hexadecimal digits. Written in hexadecimal, the value of a single byte can range from 00_{16} to FF_{16}.

In C, numeric constants starting with `0x` or `0X` are interpreted as being in hexadecimal. The characters 'A' through 'F' may be written in either upper or lower case. For example, we could write the number $FA1D37B_{16}$ as `0xFA1D37B`, as `0xfa1d37b`, or even mixing upper and lower case, e.g., `0xFa1D37b`. We will use the C notation for representing hexadecimal values in this book.

A common task in working with machine-level programs is to manually convert between decimal, binary, and hexadecimal representations of bit patterns. Converting between binary and hexadecimal is straightforward, since it can be performed one hexadecimal digit at a time. Digits can be converted by referring to a chart such as that shown in Figure 2.2. One simple trick for doing the conversion in your head is to memorize the decimal equivalents of hex digits `A`, `C`, and `F`. The hex values `B`, `D`, and `E` can be translated to decimal by computing their values relative to the first three.

For example, suppose you are given the number `0x173A4C`. You can convert this to binary format by expanding each hexadecimal digit, as follows:

Hexadecimal	`1`	`7`	`3`	`A`	`4`	`C`
Binary	0001	0111	0011	1010	0100	1100

This gives the binary representation 000101110011101001001100.

Conversely, given a binary number 1111001010110110110011, you convert it to hexadecimal by first splitting it into groups of 4 bits each. Note, however, that if the total number of bits is not a multiple of 4, you should make the *leftmost* group be the one with fewer than 4 bits, effectively padding the number with leading zeros. Then you translate each group of bits into the corresponding hexadecimal digit:

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	`3`	`C`	`A`	`D`	`B`	`3`

Practice Problem 2.1:

Perform the following number conversions:

A. `0x39A7F8` to binary

B. Binary 1100100101111011 to hexadecimal

C. `0xD5E4C` to binary

D. Binary 1001101110011110110101 to hexadecimal

When a value x is a power of two, that is, $x = 2^n$ for some nonnegative integer n, we can readily write x in hexadecimal form by remembering that the binary representation of x is simply 1 followed by n zeros. The hexadecimal digit `0` represents four binary zeros. So, for n written in the form $i + 4j$, where $0 \leq i \leq 3$, we can write x with a leading hex digit of `1` ($i = 0$), `2` ($i = 1$), `4` ($i = 2$), or `8` ($i = 3$), followed by j hexadecimal `0`s. As an example, for $x = 2048 = 2^{11}$, we have $n = 11 = 3 + 4 \cdot 2$, giving hexadecimal representation `0x800`.

Practice Problem 2.2:

Fill in the blank entries in the following table, giving the decimal and hexadecimal representations of different powers of 2:

n	2^n (Decimal)	2^n (Hexadecimal)
9	512	`0x200`
19		
	16,384	
		`0x10000`
17		
	32	
		`0x80`

Converting between decimal and hexadecimal representations requires using multiplication or division to handle the general case. To convert a decimal number x to hexadecimal, we can repeatedly divide x by 16, giving a quotient q and a remainder r, such that $x = q \cdot 16 + r$. We then use the hexadecimal digit representing r as the least significant digit and generate the remaining digits by repeating the process on q. As an example, consider the conversion of decimal 314156:

$$\begin{array}{rcll}
314156 & = & 19634 \cdot 16 + 12 & (\texttt{C}) \\
19634 & = & 1227 \cdot 16 + 2 & (\texttt{2}) \\
1227 & = & 76 \cdot 16 + 11 & (\texttt{B}) \\
76 & = & 4 \cdot 16 + 12 & (\texttt{C}) \\
4 & = & 0 \cdot 16 + 4 & (\texttt{4})
\end{array}$$

From this we can read off the hexadecimal representation as `0x4CB2C`.

Conversely, to convert a hexadecimal number to decimal, we can multiply each of the hexadecimal digits by the appropriate power of 16. For example, given the number `0x7AF`, we compute its decimal equivalent as $7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1792 + 160 + 15 = 1967$.

Practice Problem 2.3:

A single byte can be represented by two hexadecimal digits. Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns:

Decimal	Binary	Hexadecimal
0	0000 0000	0x00
167		
62		
188		
	0011 0111	
	1000 1000	
	1111 0011	
		0x52
		0xAC
		0xE7

Aside: Converting between decimal and hexadecimal.
For converting larger values between decimal and hexadecimal, it is best to let a computer or calculator do the work. For example, the following script in the Perl language converts a list of numbers (given on the command line) from decimal to hexadecimal: *bin/d2h*

```
#!/usr/local/bin/perl
# Convert list of decimal numbers into hex

for ($i = 0; $i < @ARGV; $i++) {
    printf("%d\t= 0x%x\n", $ARGV[$i], $ARGV[$i]);
}
```

bin/d2h Once this file has been set to be executable, the command

```
unix> ./d2h 100 500 751
```

yields output:

```
100 = 0x64
500 = 0x1f4
751 = 0x2ef
```

Similarly, the following script converts from hexadecimal to decimal: *bin/h2d*

```
#!/usr/local/bin/perl
# Convert list of hex numbers into decimal

for ($i = 0; $i < @ARGV; $i++) {
  $val = hex($ARGV[$i]);
  printf("0x%x = %d\n", $val, $val);
}
```

bin/h2d **End Aside.**

Practice Problem 2.4:

Without converting the numbers to decimal or binary, try to solve the following arithmetic problems, giving the answers in hexadecimal. **Hint:** Just modify the methods you use for performing decimal addition and subtraction to use base 16.

C declaration	32-bit	64-bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char *`	4	8
`float`	4	4
`double`	8	8

Figure 2.3: **Sizes (in bytes) of C numeric data types.** The number of bytes allocated varies with machine and compiler. This chart shows the values typical of 32-bit and 64-bit machines.

A. `0x503c` $+$ `0x8` $=$

B. `0x503c` $-$ `0x40` $=$

C. `0x503c` $+ 64 =$

D. `0x50ea` $-$ `0x503c` $=$

2.1.2 Words

Every computer has a *word size*, indicating the nominal size of integer and pointer data. Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is the maximum size of the virtual address space. That is, for a machine with a w-bit word size, the virtual addresses can range from 0 to $2^w - 1$, giving the program access to at most 2^w bytes.

Most personal computers today have a 32-bit word size. This limits the virtual address space to 4 gigabytes (written 4 GB), that is, just over 4×10^9 bytes. Although this is ample space for most applications, we have reached the point where many large-scale scientific and database applications require larger amounts of storage. Consequently, high-end machines with 64-bit word sizes are becoming increasingly common as storage costs decrease. As hardware costs drop over time, even desktop and laptop machines will switch to 64-bit word sizes, and so we will consider the general case of a w-bit word size, as well as the special cases of $w = 32$ and $w = 64$.

2.1.3 Data Sizes

Computers and compilers support multiple data formats using different ways to encode data, such as integers and floating point, as well as different lengths. For example, many machines have instructions for manipulating single bytes, as well as integers represented as two-, four-, and eight-byte quantities. They also support floating-point numbers represented as four and eight-byte quantities.

The C language supports multiple data formats for both integer and floating-point data. The C data type `char` represents a single byte. Although the name "`char`" derives from the fact that it is used to store a single character in a text string, it can also be used to store integer values. The C data type `int` can also be prefixed by the qualifiers `short`, `long`, and recently `long long`, providing integer representations of various sizes. Figure 2.3 shows the number of bytes allocated for different C data types. The exact number depends on both the machine and the compiler. We show typical sizes for 32-bit and 64-bit machines. Observe that "short" integers have two-byte allocations, while an unqualified `int` is 4 bytes. A "long" integer uses the full word size of the machine. The "long long" integer data type, introduced in ISO C99, allows the full range of 64-bit integers. For 32-bit machines, the compiler must compile operations for this data type by generating code that performs sequences of 32-bit operations.

Figure 2.3 also shows that a pointer (e.g., a variable declared as being of type "`char *`") uses the full word size of the machine. Most machines also support two different floating-point formats: single precision, declared in C as `float`, and double precision, declared in C as `double`. These formats use four and eight bytes, respectively.

New to C?: Declaring pointers.
For any data type T, the declaration

T `*p;`

indicates that `p` is a pointer variable, pointing to an object of type T. For example

```
char *p;
```

is the declaration of a pointer to an object of type `char`. **End.**

Programmers should strive to make their programs portable across different machines and compilers. One aspect of portability is to make the program insensitive to the exact sizes of the different data types. The C standards set lower bounds on the numeric ranges of the different data types, as will be covered later, but there are no upper bounds. Since 32-bit machines have been the standard since around 1980, many programs have been written assuming the allocations listed for this word size in Figure 2.3. Given the increasing availability of 64-bit machines, many hidden word size dependencies will show up as bugs in migrating these programs to new machines. For example, many programmers assume that a program object declared as type `int` can be used to store a pointer. This works fine for most 32-bit machines, but it leads to problems on a 64-bit machine.

2.1.4 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what the address of the object will be, and how we will order the bytes in memory. In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used. For example, suppose a variable `x` of type `int` has address `0x100`, that is, the value of the address expression `&x` is `0x100`. Then the four bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.

For ordering the bytes representing an object, there are two common conventions. Consider a w-bit integer having a bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit, and x_0 is the least. Assuming w is a multiple of eight, these bits can be grouped as bytes, with the most significant byte having bits $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$, the least significant byte having bits $[x_7, x_6, \ldots, x_0]$, and the other bytes having bits from the middle. Some machines choose to store the object in memory ordered from least significant byte to most, while other machines store them from most to least. The former convention—where the least significant byte comes first—is referred to as *little endian*. This convention is followed by most Intel-compatible machines. The latter convention—where the most significant byte comes first—is referred to as *big endian*. This convention is followed by most machines from IBM and Sun Microsystems. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, both IBM and Sun manufacture machines that use Intel-compatible processors and hence are little endian. Many recent microprocessors are *bi-endian*, meaning that they can be configured to operate as either little- or big-endian machines.

Continuing our earlier example, suppose the variable `x` of type `int` and at address `0x100` has a hexadecimal value of `0x01234567`. The ordering of the bytes within the address range `0x100` through `0x103` depends on the type of machine:

Big endian

	0x100	0x101	0x102	0x103	
···	01	23	45	67	···

Little endian

	0x100	0x101	0x102	0x103	
···	67	45	23	01	···

Note that in the word `0x01234567` the high-order byte has hexadecimal value `0x01`, while the low-order byte has value `0x67`.

People get surprisingly emotional about which byte ordering is the proper one. In fact, the terms "little endian" and "big endian" come from the book *Gulliver's Travels* by Jonathan Swift, where two warring factions could not agree as to how a soft-boiled egg should be opened—by the little end or by the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, and hence the arguments degenerate into bickering about socio-political issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

Aside: Origin of "endian."
Here is how Jonathan Swift, writing in 1726, described the history of the controversy between big and little endians:

> ...Lilliput and Blefuscu ...have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. It began upon the following occasion. It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account; wherein one emperor lost his life, and another his crown. These civil commotions were constantly fomented by the monarchs of Blefuscu; and when they were quelled, the exiles always fled for refuge to that empire. It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have

```c
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}

void show_int(int x) {
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
    show_bytes((byte_pointer) &x, sizeof(void *));
}
```

Figure 2.4: **Code to print the byte representation of program objects.** This code uses casting to circumvent the type system. Similar functions are easily defined for other data types.

> been published upon this controversy: but the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments.

In his day, Swift was satirizing the continued conflicts between England (Lilliput) and France (Blefuscu). Danny Cohen, an early pioneer in networking protocols, first applied these terms to refer to byte ordering [25], and the terminology has been widely adopted. **End Aside.**

For most application programmers, the byte orderings used by their machines are totally invisible; programs compiled for either class of machine give identical results. At times, however, byte ordering becomes an issue. The first is when binary data are communicated over a network between different machines. A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice versa, leading to the bytes within the words being in reverse order for the receiving program. To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation. We will see examples of these conversions in Chapter 11.

A second case where byte ordering becomes important is when looking at the byte sequences representing integer data. This occurs often when inspecting machine-level programs. As an example, the following line occurs in a file that gives a text representation of the machine-level code for an Intel IA32 processor:

```
80483bd:  01 05 64 94 04 08       add    %eax,0x8049464
```

This line was generated by a *disassembler*, a tool that determines the instruction sequence represented by an executable program file. We will learn more about disassemblers and how to interpret lines such as this in Chapter 3. For now, we simply note that this line states that the hexadecimal byte sequence `01 05 64 94 04 08` is the byte-level representation of an instruction that adds a word of data to the value stored at address `0x8049464`. If we take the final 4 bytes of the sequence: `64 94 04 08`, and write them in reverse order, we have `08 04 94 64`. Dropping the leading 0, we have the value `0x8049464`, the numeric value written on the right. Having bytes appear in reverse order is a common occurrence when reading machine-level program representations generated for little-endian machines such as this one. The natural way to write a byte sequence is to have the lowest numbered byte on the left and the highest on the right, but this is contrary to the normal way of writing numbers with the most significant digit on the left and the least on the right.

A third case where byte ordering becomes visible is when programs are written that circumvent the normal type system. In the C language, this can be done using a *cast* to allow an object to be referenced according to a different data type from which it was created. Such coding tricks are strongly discouraged for most application programming, but they can be quite useful and even necessary for system-level programming.

Figure 2.4 shows C code that uses casting to access and print the byte representations of different program objects. We use `typedef` to define data type `byte_pointer` as a pointer to an object of type "`unsigned char`." Such a byte pointer references a sequence of bytes where each byte is considered to be a nonnegative integer. The first routine `show_bytes` is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count. It prints the individual bytes in hexadecimal. The C formatting directive "`%.2x`" indicates that an integer should be printed in hexadecimal with at least two digits.

New to C?: Naming data types with `typedef`.
The `typedef` declaration in C provides a way of giving a name to a data type. This can be a great help in improving code readability, since deeply nested type declarations can be difficult to decipher.

The syntax for `typedef` is exactly like that of declaring a variable, except that it uses a type name rather than a variable name. Thus, the declaration of `byte_pointer` in Figure 2.4 has the same form as the declaration of a variable of type "`unsigned char *`."

For example, the declaration:

```
typedef int *int_pointer;
int_pointer ip;
```

defines type "`int_pointer`" to be a pointer to an `int`, and declares a variable `ip` of this type. Alternatively, we could declare this variable directly as:

```
int *ip;
```

End.

New to C?: Formatted printing with `printf`.
The `printf` function (along with its cousins `fprintf` and `sprintf`) provides a way to print information with considerable control over the formatting details. The first argument is a *format string*, while any remaining arguments are values to be printed. Within the format string, each character sequence starting with '`%`' indicates how to format the next argument. Typical examples include '`%d`' to print a decimal integer, '`%f`' to print a floating-point number, and '`%c`' to print a character having the character code given by the argument. **End.**

code/data/show-bytes.c

```
void test_show_bytes(int val) {
    int ival = val;
    float fval = (float) ival;
    int *pval = &ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
}
```

code/data/show-bytes.c

Figure 2.5: **Byte representation examples.** This code prints the byte representations of sample data objects.

New to C?: Pointers and arrays.
In function `show_bytes` (Figure 2.4) we see the close connection between pointers and arrays, as will be discussed in detail in Section 3.8. We see that this function has an argument `start` of type `byte_pointer` (which has been defined to be a pointer to `unsigned char`), but we see the array reference `start[i]` on line 8. In C, we can dereference a pointer with array notation, and we can reference array elements with pointer notation. In this example, the reference `start[i]` indicates that we want to read the byte that is `i` positions beyond the location pointed to by `start`. **End.**

Procedures `show_int`, `show_float`, and `show_pointer` demonstrate how to use procedure `show_bytes` to print the byte representations of C program objects of type `int`, `float`, and `void *`, respectively. Observe that they simply pass `show_bytes` a pointer `&x` to their argument `x`, casting the pointer to be of type "`unsigned char *`." This cast indicates to the compiler that the program should consider the pointer to be to a sequence of bytes rather than to an object of the original data type. This pointer will then be to the lowest byte address occupied by the object.

New to C?: Pointer creation and dereferencing.
In lines 13, 17, and 21 of Figure 2.4 we see uses of two operations that give C (and therefore C++) its distinctive character. The C "address of" operator `&` creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding the object indicated by variable `x`. The type of this pointer depends on the type of `x`, and hence these three pointers are of type `int *`, `float *`, and `void **`, respectively. (Data type `void *` is a special kind of pointer with no associated type information.)

The cast operator converts from one data type to another. Thus, the cast `(byte_pointer) &x` indicates that whatever type the pointer `&x` had before, the program will now reference a pointer to data of type `unsigned char`. The casts shown here do not change the actual pointer; they simply direct the compiler to refer to the data being pointed to according to the new data type. **End.**

These procedures use the C `sizeof` operator to determine the number of bytes used by the object. In general, the expression `sizeof(`T`)` returns the number of bytes required to store an object of type T. Using `sizeof` rather than a fixed value is one step toward writing code that is portable across different machine types.

We ran the code shown in Figure 2.5 on several different machines, giving the results shown in Figure 2.6. The following machines were used:

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	`39 30 00 00`
Windows	12,345	`int`	`39 30 00 00`
Sun	12,345	`int`	`00 00 30 39`
Linux 64	12,345	`int`	`39 30 00 00`
Linux 32	$12,345.0$	`float`	`00 e4 40 46`
Windows	$12,345.0$	`float`	`00 e4 40 46`
Sun	$12,345.0$	`float`	`46 40 e4 00`
Linux 64	$12,345.0$	`float`	`00 e4 40 46`
Linux 32	`&ival`	`int *`	`e4 f9 ff bf`
Windows	`&ival`	`int *`	`b4 cc 22 00`
Sun	`&ival`	`int *`	`ef ff fa 0c`
Linux 64	`&ival`	`int *`	`b8 11 e5 ff ff 7f 00 00`

Figure 2.6: **Byte representations of different data values.** Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine dependent.

Linux 32: Intel IA32 processor running Linux.

Windows: Intel IA32 processor running Windows.

Sun: Sun Microsystems SPARC processor running Solaris.

Linux 64: Intel x86-64 processor running Linux.

Our argument 12,345 has hexadecimal representation `0x00003039`. For the `int` data, we get identical results for all machines, except for the byte ordering. In particular, we can see that the least significant byte value of `0x39` is printed first for Linux 32, Windows, and Linux 64, indicating little-endian machines, and last for Sun, indicating a big-endian machine. Similarly, the bytes of the `float` data are identical, except for the byte ordering. On the other hand, the pointer values are completely different. The different machine/operating system configurations use different conventions for storage allocation. One feature to note is that the Linux 32, Windows, and Sun machines use four-byte addresses, while the Linux 64 machine uses eight-byte addresses.

Observe that although the floating-point and the integer data both encode the numeric value 12,345, they have very different byte patterns: `0x00003039` for the integer, and `0x4640E400` for floating point. In general, these two formats use different encoding schemes. If we expand these hexadecimal patterns into binary form and shift them appropriately, we find a sequence of 13 matching bits, indicated by a sequence of asterisks, as follows:

```
   0   0   0   0   3   0   3   9
00000000000000000011000000111001
                   *************
             4   6   4   0   E   4   0   0
          01000110010000001110010000000000
```

This is not coincidental. We will return to this example when we study floating-point formats.

Practice Problem 2.5:

Consider the following three calls to `show_bytes`:

```
int val = 0x87654321;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate which of the following values will be printed by each call on a little-endian machine and on a big-endian machine:

A.	Little endian:	Big endian:
B.	Little endian:	Big endian:
C.	Little endian:	Big endian:

Practice Problem 2.6:

Using `show_int` and `show_float`, we determine that the integer 3510593 has hexadecimal representation `0x00359141`, while the floating-point number 3510593.0 has hexadecimal representation `0x4A564504`.

A. Write the binary representations of these two hexadecimal values.

B. Shift these two strings relative to one another to maximize the number of matching bits. How many bits match?

C. What parts of the strings do not match?

2.1.5 Representing Strings

A string in C is encoded by an array of characters terminated by the null (having value 0) character. Each character is represented by some standard encoding, with the most common being the ASCII character code. Thus, if we run our routine `show_bytes` with arguments `"12345"` and `6` (to include the terminating character), we get the result `31 32 33 34 35 00`. Observe that the ASCII code for decimal digit x happens to be `0x3`x, and that the terminating byte has the hex representation `0x00`. This same result would be obtained on any system using ASCII as its character code, independent of the byte ordering and word size conventions. As a consequence, text data is more platform-independent than binary data.

Aside: Generating an ASCII table.
You can display a table showing the ASCII character code by executing the command `man ascii`. **End Aside.**

Practice Problem 2.7:

What would be printed as a result of the following call to `show_bytes`?