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, 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 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 encodings 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, 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 $+\infty$. On the other hand, 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`.

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.

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 give you some practice reasoning 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 when compiling arithmetic expressions.

Our treatment of this material is very mathematical. 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 examine this material from such an abstract viewpoint, because programmers need to have a solid understanding of how computer arithmetic relates to the more familiar integer and real arithmetic. Although it may appear intimidating, the mathematical treatment requires just an understanding of basic algebra. We recommend you work the practice problems as a way to solidify the connection between the formal treatment and some real-life examples.

Aside: How to read this chapter.
If you fi nd 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, try working out a few simple examples (for example, the practice problems) to build your intuition, and then see how the mathematical derivation reinforces your intuition.
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 standard is 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.

2.1 Information Storage

Rather than accessing individual bits in a memory, most computers use blocks of eight 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 10) 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.

One task of a compiler and the run-time system is to subdivide 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 unit—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 fbating-point number) is stored at that location. **End.**

2.1.1 Hexadecimal Notation

A single byte consists of eight 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 (or simply "Hex") uses digits '0' through '9', along with characters 'A' through 'F' to represent 16 possible values. Figure 2.1 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}.

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

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 2.1. 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 four bits each. Note, however, that if the total number of bits is not a multiple of four, you should make the *leftmost* group be the one with fewer than four bits, effectively padding the number with leading 0s. Then you translate each group of four 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. `0x8F7A93` to binary.

B. Binary 1011011110011100 to hexadecimal.

C. `0xC4E5D` to binary.

D. Binary 1101011011011111100110 to hexadecimal.

When a value x is a power of two, that is, $x = 2^n$ for some n, we can readily write x in hexadecimal form by remembering that the binary representation of x is simply 1 followed by n 0s. The hexadecimal digit `0` represents four binary 0s. 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)
11	2048	`0x800`
7		
	8192	
		`0x20000`
16		
	256	
		`0x20`

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	00000000	00
55		
136		
243		
	01010010	
	10101100	
	11100111	
		A7
		3E
		BC

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 from decimal to hexadecimal:

code/data/d2h

```
#!/usr/bin/perl
#!/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]);
}
```

code/data/d2h

Once this fi le 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:

code/data/h2d

```
#!/usr/bin/perl
#!/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);
}
```

code/data/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.

A. `0x502c` $+$ `0x8` $=$

B. `0x502c` $-$ `0x30` $=$

C. `0x502c` $+ 64 =$

D. `0x50da` $-$ `0x502c` $=$

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 an n-bit word size, the virtual addresses can range from 0 to $2^n - 1$, giving the program access to at most 2^n bytes.

Most 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 commonplace as storage costs decrease.

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 `long` and `short`, providing integer representations of various sizes. Figure 2.2 shows the number of bytes allocated for various C data types. The exact number depends on both the machine and the compiler. We show two representative cases: a typical 32-bit machine, and the Compaq Alpha architecture, a 64-bit machine targeting high end applications. Most 32-bit machines use the allocations indicated as "typical." 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.

Figure 2.2 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,

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

Figure 2.2: **Sizes (in bytes) of C numeric data types.** The number of bytes allocated varies with machine and compiler.

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 standard sets 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 for the last 20 years, many programs have been written assuming the allocations listed as “typical 32-bit” in Figure 2.2. Given the increasing prominence of 64-bit machines in the near future, 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 leads to problems on an Alpha.

2.1.4 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what will be the address of the object, and how will we order the bytes in memory. In virtually all machines, a multibyte 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 machines from the former Digital Equipment Corporation (now part of Compaq Corporation), as well as by Intel. 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, Motorola, and Sun Microsystems. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, personal computers manufactured by IBM use Intel-compatible processors and hence are little endian. Many microprocessor chips, including Alpha and the PowerPC by Motorola, can be run in either mode, with the byte ordering convention determined when the chip is powered up.

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 by which end a soft-boiled egg should be opened—the little end or 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 sociopolitical 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 fi ngers. 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 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 [17], 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 is 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 12.

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 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 these tools and how to interpret lines such as this in the next chapter. 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 `0x8049464` to some program value. If we take the the final four 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 2.3 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.

code/data/show-bytes.c

```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 *));
}
```

code/data/show-bytes.c

Figure 2.3: **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.

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 2.3 has the same form as would the declaration of a variable to 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 and '`%f`' to print a floating-point number, and '`%c`' to print a character having the character code given by the argument. **End.**

New to C?: Pointers and arrays.
In function `show_bytes` (Figure 2.3) 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 9. 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 used by the object.

New to C?: Pointer creation and dereferencing.
In lines 15, 20, and 25 of Figure 2.3 we see uses of two operations that are unique to C and C++. The C "address of" operator & creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding 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, it now is a pointer to data of type `unsigned char`. **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 2.4: **Byte representation examples.** This code prints the byte representations of sample data objects.

These procedures use the C operator `sizeof` 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 2.4 on several different machines, giving the results shown in Figure 2.5. The following machines were used:

Linux:	Intel Pentium II running Linux.
NT:	Intel Pentium II running Windows-NT.
Sun:	Sun Microsystems UltraSPARC running Solaris.
Alpha:	Compaq Alpha 21164 running Tru64 Unix.

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, NT, and Alpha, 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 and Sun machines use four-byte addresses, while the Alpha 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
```

Machine	Value	Type	Bytes (hex)
Linux	12,345	`int`	`39 30 00 00`
NT	12,345	`int`	`39 30 00 00`
Sun	12,345	`int`	`00 00 30 39`
Alpha	12,345	`int`	`39 30 00 00`
Linux	$12,345.0$	`float`	`00 e4 40 46`
NT	$12,345.0$	`float`	`00 e4 40 46`
Sun	$12,345.0$	`float`	`46 40 e4 00`
Alpha	$12,345.0$	`float`	`00 e4 40 46`
Linux	`&ival`	`int *`	`3c fa ff bf`
NT	`&ival`	`int *`	`1c ff 44 02`
Sun	`&ival`	`int *`	`ef ff fc e4`
Alpha	`&ival`	`int *`	`80 fc ff 1f 01 00 00 00`

Figure 2.5: **Byte representations of different data values.** Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine-dependent.

```
                 *************
    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 = 0x12345678;
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 would 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 3490593 has hexadecimal representation `0x00354321`, while the fbating-point number 3490593.0 has hexadecimal representation representation `0x4A550C84`.

}

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.

C. How many bits match? 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`?

```
char *s = "ABCDEF";
show_bytes(s, strlen(s));
```

Note that letters '`A`' through '`Z`' have ASCII codes `0x41` through `0x5A`.

Aside: The Unicode character set.
The ASCII character set is suitable for encoding English language documents, but it does not have much in the way of special characters, such as the French 'ç.' It is wholly unsuited for encoding documents in languages such as Greek, Russian, and Chinese. Recently, the 16-bit *Unicode* character set has been adopted to support documents in all languages. This doubling of the character set representation enables a very large number of different characters to be represented. The Java programming language uses Unicode when representing character strings. Program libraries are also available for C that provide Unicode versions of the standard string functions such as `strlen` and `strcpy`. **End Aside.**

2.1.6 Representing Code

Consider the following C function:

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

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Figure 2.6: **Operations of Boolean algebra.** Binary values 1 and 0 encode logic values TRUE and FALSE, while operations ~, &, |, and ^ encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

When compiled on our sample machines, we generate machine code having the following byte representations:

Linux: `55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`

NT: `55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`

Sun: `81 C3 E0 08 90 02 00 09`

Alpha: `00 00 30 42 01 80 FA 6B`

Here we find that the instruction codings are different, except for the NT and Linux machines. Different machine types use different and incompatible instructions and encodings. The NT and Linux machines both have Intel processors and hence support the same machine-level instructions. In general, however, the structure of an executable NT program differs from a Linux program, and hence the machines are not fully binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that a program, from the perspective of the machine, is simply a sequence of bytes. The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging. We will see this more clearly when we study machine-level programming in Chapter 3.

2.1.7 Boolean Algebras and Rings

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole around 1850 and thus is known as *Boolean algebra*. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the properties of propositional logic.

There is an infinite number of different Boolean algebras, where the simplest is defined over the two-element set $\{0, 1\}$. Figure 2.6 defines several operations in this Boolean algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations, as will be discussed later. The Boolean operation ~ corresponds to the logical operation NOT, denoted in propositional logic as $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice-versa. Correspondingly, ~p equals 1 when p equals 0, and vice-versa. Boolean operation & corresponds to the logical operation AND, denoted in propositional

Shared properties

Property	Integer ring	Boolean algebra
Commutativity	$a + b = b + a$ $a \times b = b \times a$	$a \mid b = b \mid a$ $a \,\&\, b = b \,\&\, a$
Associativity	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a \mid b) \mid c = a \mid (b \mid c)$ $(a \,\&\, b) \,\&\, c = a \,\&\, (b \,\&\, c)$
Distributivity	$a \times (b + c) = (a \times b) + (a \times c)$	$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$
Identities	$a + 0 = a$ $a \times 1 = a$	$a \mid 0 = a$ $a \,\&\, 1 = a$
Annihilator	$a \times 0 = 0$	$a \,\&\, 0 = 0$
Cancellation	$-(-a) = a$	$\tilde{}(\tilde{}a) = a$

Unique to Rings

Inverse	$a + -a = 0$	—

Unique to Boolean Algebras

Distributivity	—	$a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$
Complement	— —	$a \mid \tilde{}a = 1$ $a \,\&\, \tilde{}a = 0$
Idempotency	— —	$a \,\&\, a = a$ $a \mid a = a$
Absorption	— —	$a \mid (a \,\&\, b) = a$ $a \,\&\, (a \mid b) = a$
DeMorgan's laws	— —	$\tilde{}(a \,\&\, b) = \tilde{}a \mid \tilde{}b$ $\tilde{}(a \mid b) = \tilde{}a \,\&\, \tilde{}b$

Figure 2.7: **Comparison of integer ring and Boolean algebra.** The two mathematical structures share many properties, but there are key differences, particularly between $-$ and ~.

logic as $\wedge$. We say that $P \wedge Q$ holds when both P and Q are true. Correspondingly, $p \& q$ equals 1 only when $p = 1$ and $q = 1$. Boolean operation | corresponds to the logical operation OR, denoted in propositional logic as $\vee$. We say that $P \vee Q$ holds when either P or Q are true. Correspondingly, $p \mid q$ equals 1 when either $p = 1$ or $q = 1$. Boolean operation ^ corresponds to the logical operation EXCLUSIVE-OR, denoted in propositional logic as $\oplus$. We say that $P \oplus Q$ holds when either P or Q are true, but not both. Correspondingly, p ^ q equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon, who later founded the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 master's thesis, he showed that Boolean algebra could be applied to the design and analysis of networks of electromechanical relays. Although computer technology has advanced considerably since, Boolean algebra still plays a central role in the design and analysis of digital systems.

There are many parallels between integer arithmetic and Boolean algebra, as well as several important differences. In particular, the set of integers, denoted $\mathcal{Z}$, forms a mathematical structure known as a *ring*, denoted $\langle \mathcal{Z}, +, \times, -, 0, 1\rangle$, with addition serving as the *sum* operation, multiplication as the *product* operation, negation as the additive inverse, and elements 0 and 1 serving as the additive and multiplicative identities. The Boolean algebra $\langle \{0,1\}, |, \&, \tilde{}, 0, 1\rangle$ has similar properties. Figure 2.7 highlights properties of these two structures, showing the properties that are common to both and those that are unique to one or the other. One important difference is that $\tilde{}a$ is not an inverse for a under $|$.

Aside: What good is abstract algebra?
Abstract algebra involves identifying and analyzing the common properties of mathematical operations in different domains. Typically, an algebra is characterized by a set of elements, some of its key operations, and some important elements. As an example, modular arithmetic also forms a ring. For modulus n, the algebra is denoted $\langle \mathcal{Z}_n, +_n, \times_n, -_n, 0, 1\rangle$, with components defined as follows:

$$\begin{aligned} \mathcal{Z}_n &= \{0, 1, \ldots, n-1\} \\ a +_n b &= a + b \bmod n \\ a \times_n b &= a \times b \bmod n \\ -_n a &= \begin{cases} 0, & a = 0 \\ n - a, & a > 0 \end{cases} \end{aligned}$$

Even though modular arithmetic yields different results from integer arithmetic, it has many of the same mathematical properties. Other well-known rings include rational and real numbers. **End Aside.**

If we replace the OR operation of Boolean algebra by the EXCLUSIVE-OR operation, and the complement operation $\tilde{}$ with the identity operation I—where $I(a) = a$ for all a—we have a structure $\langle \{0,1\}, \hat{}, \&, I, 0, 1\rangle$. This structure is no longer a Boolean algebra—in fact it's a ring. It can be seen to be a particularly simple form of the ring consisting of all integers $\{0, 1, \ldots, n-1\}$ with both addition and multiplication performed modulo n. In this case, we have $n = 2$. That is, the Boolean AND and EXCLUSIVE-OR operations correspond to multiplication and addition modulo 2, respectively. One curious property of this algebra is that every element is its own additive inverse: $a \mathbin{\hat{}} I(a) = a \mathbin{\hat{}} a = 0$.

Aside: Who, besides mathematicians, care about Boolean rings?
Every time you enjoy the clarity of music recorded on a CD or the quality of video recorded on a DVD, you are taking advantage of Boolean rings. These technologies rely on *error-correcting codes* to reliably retrieve the bits from a disk even when dirt and scratches are present. The mathematical basis for these error-correcting codes is a linear algebra based on Boolean rings. **End Aside.**

We can extend the four Boolean operations to also operate on bit vectors, i.e., strings of 0s and 1s of some fixed length w. We define the operations over bit vectors according their applications to the matching elements of the arguments. For example, we define $[a_{w-1}, a_{w-2}, \ldots, a_0]\&[b_{w-1}, b_{w-2}, \ldots, b_0]$ to be $[a_{w-1}\& b_{w-1}, a_{w-2} \& b_{w-2}, \ldots, a_0 \& b_0]$, and similarly for operations $\tilde{}$, $|$, and $\hat{}$. Letting $\{0,1\}^w$ denote the set of all strings of 0s and 1s having length w, and a^w denote the string consisting of w repetitions of symbol a, then one can see that the resulting algebras: $\langle \{0,1\}^w, |, \&, \tilde{}, 0^w, 1^w\rangle$ and $\langle \{0,1\}^w, \hat{}, \&, I, 0^w, 1^w\rangle$ form Boolean algebras and rings, respectively. Each value of w defines a different Boolean algebra and a different Boolean ring.

Aside: Are Boolean rings the same as modular arithmetic?
The two-element Boolean ring $\langle \{0,1\}, \hat{}, \&, I, 0, 1\rangle$ is identical to the ring of integers modulo two $\langle \mathcal{Z}_2, +_2, \times_2, -_2, 0, 1\rangle$. The generalization to bit vectors of length w, however, yields a very different ring from modular arithmetic. **End Aside.**

Practice Problem 2.8:

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

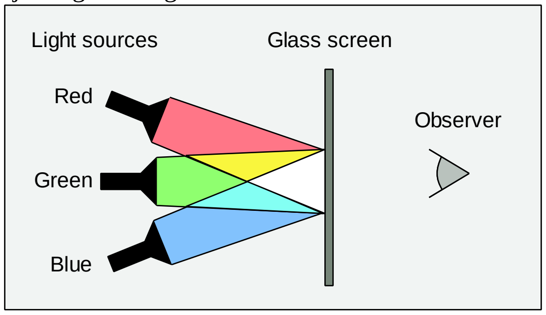
Operation	Result
a	$[01101001]$
b	$[01010101]$
~a	
~b	
a & b	
a | b	
a ^ b	

One useful application of bit vectors is to represent finite sets. For example, we can denote any subset $A \subseteq \{0, 1, \ldots, w-1\}$ as a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where $a_i = 1$ if and only if $i \in A$. For example, (recalling that we write a_{w-1} on the left and a_0 on the right), we have $a = [01101001]$ representing the set $A = \{0, 3, 5, 6\}$, and $b = [01010101]$ representing the set $B = \{0, 2, 4, 6\}$. Under this interpretation, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. For example, the operation a & b yields bit vector $[01000001]$, while $A \cap B = \{0, 6\}$.

In fact, for any set S, the structure $\langle \mathcal{P}(S), \cup, \cap, \bar{}, \emptyset, S \rangle$ forms a Boolean algebra, where $\mathcal{P}(S)$ denotes the set of all subsets of S, and $\bar{}$ denotes the set complement operator. That is, for any set A, its complement is the set $\overline{A} = \{a \in S | a \notin A\}$. The ability to represent and manipulate finite sets using bit vector operations is a practical outcome of a deep mathematical principle.

Practice Problem 2.9:

Computers generate color pictures on a video screen or liquid crystal display by mixing three different colors of light: red, green, and blue. Imagine a simple scheme, with three different lights, each of which can be turned on or off, projecting onto a glass screen:



We can then create eight different colors based on the absence (0) or presence (1) of light sources R, G, and B:

R	G	B	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

This set of colors forms an eight-element Boolean algebra.

A. The complement of a color is formed by turning off the lights that are on and turning on the lights that are off. What would be the complements of the eight colors listed above?

B. What colors correspond to Boolean values 0^w and 1^w for this algebra?

C. Describe the effect of applying Boolean operations on the following colors:

Blue	|	Red	=
Magenta	&	Cyan	=
Green	^	White	=

2.1.8 Bit-Level Operations in C

One useful feature of C is that it supports bit-wise Boolean operations. In fact, the symbols we have used for the Boolean operations are exactly those used by C: | for OR, & for AND, ~ for NOT, and ^ for EXCLUSIVE-OR. These can be applied to any "integral" data type, that is, one declared as type `char` or `int`, with or without qualifiers such as `short`, `long`, or `unsigned`. Here are some examples of expression evaluation:

C expression	Binary expression	Binary result	C result
`~0x41`	~[01000001]	[10111110]	`0xBE`
`~0x00`	~[00000000]	[11111111]	`0xFF`
`0x69 & 0x55`	[01101001] & [01010101]	[01000001]	`0x41`
`0x69 | 0x55`	[01101001] | [01010101]	[01111101]	`0x7D`

As our examples show, the best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

Practice Problem 2.10:

To show how the ring properties of ^ can be useful, consider the following program:

```
void inplace_swap(int *x, int *y)
{
    *x = *x ^ *y;  /* Step 1 */
    *y = *x ^ *y;  /* Step 2 */
    *x = *x ^ *y;  /* Step 3 */
}
```

As the name implies, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables `x` and `y`. Note that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is no performance advantage to this way of swapping. It is merely an intellectual amusement.

Starting with values a and b in the locations pointed to by `x` and `y`, respectively, fi ll in the table that follows giving the values stored at the two locations after each step of the procedure. Use the ring properties to show that the desired effect is achieved. Recall that every element is its own additive inverse (that is, a ^ $a = 0$).

Step	`*x`	`*y`
Initially	a	b
Step 1		
Step 2		
Step 3		

One common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that indicates a selected set of bits within a word. As an example, the mask `0xFF` (having 1s for the least significant eight bits) indicates the low-order byte of a word. The bit-level operation `x & 0xFF` yields a value consisting of the least significant byte of `x`, but with all other bytes set to 0. For example, with `x` = `0x89ABCDEF`, the expression would yield `0x000000EF`. The expression `~0` will yield a mask of all 1s, regardless of the word size of the machine. Although the same mask can be written `0xFFFFFFFF` for a 32-bit machine, such code is not as portable.

Practice Problem 2.11:

Write C expressions for the following values, with the results for `x` = `0x98FDECBA` and a 32-bit word size shown in square brackets:

A. The least signifi cant byte of `x`, with all other bits set to 1 [`0xFFFFFFBA`].

B. The complement of the least signifi cant byte of `x`, with all other bytes left unchanged [`0x98FDEC45`].

C. All but the least signifi cant byte of `x`, with the least signifi cant byte set to 0 [`0x98FDEC00`].

Although our examples assume a 32-bit word size, your code should work for any word size $w \geq 8$.

Practice Problem 2.12:

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations AND and OR, it had instructions `bis` (bit set) and `bic` (bit clear). Both instructions take a data word `x` and a mask word `m`. They generate a result `z` consisting of the bits of `x` modifi ed according to the bits of `m`. With `bis`, the modifi cation involves setting `z` to 1 at each bit position where `m` is 1. With `bic`, the modifi cation involves setting `z` to 0 at each bit position where `m` is 1.

We would like to write C functions `bis` and `bic` to compute the effect of these two instructions. Fill in the missing expressions in the following code using the bit-level operations of C:

```
/* Bit Set */
int bis(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ___________;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  /* Write an expression in C that computes the effect of bit clear */
  int result = ___________;
  return result;
}
```

2.1.9 Logical Operations in C

C also provides a set of *logical* operators `||`, `&&`, and `!`, which correspond to the OR, AND, and NOT operations of propositional logic. These can easily be confused with the bit-level operations, but their function is quite different. The logical operations treat any nonzero argument as representing TRUE and argument 0 as representing FALSE. They return either 1 or 0, indicating a result of either TRUE or FALSE, respectively. Here are some examples of expression evaluation:

Expression	Result
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x41`	`0x01`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Observe that a bit-wise operation will have behavior matching that of its logical counterpart only in the special case in which the arguments are restricted to 0 or 1.

A second important distinction between the logical operators `&&` and `||` versus their bit-level counterparts `&` and `|` is that the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument. Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Practice Problem 2.13:

Suppose that `x` and `y` have byte values `0x66` and `0x93`, respectively. Fill in the following table indicating the byte values of the different C expressions:

Expression	Value	Expression	Value
`x & y`		`x && y`	
`x | y`		`x || y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

Practice Problem 2.14:

Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. In other words, it will return 1 when `x` and `y` are equal and 0 otherwise.

2.1.10 Shift Operations in C

C also provides a set of *shift* operations for shifting bit patterns to the left and to the right. For an operand `x` having bit representation $[x_{n-1}, x_{n-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{n-k-1}, x_{n-k-2}, \ldots, x_0, 0, \ldots 0]$. That is, `x` is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k 0s. The shift amount should be a value between 0 and $n-1$. Shift operations group from left to right, so `x << j << k` is equivalent to `(x << j) << k`. Be careful about operator precedence: `1<<5 - 1` is evaluated as `1 << (5-1)`, not as `(1<<5) - 1`.

There is a corresponding right shift operation `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift: *logical* and *arithmetic*. A logical right shift fills the left end with k 0s, giving a result $[0, \ldots, 0, x_{n-1}, x_{n-2}, \ldots x_k]$. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{n-1}, \ldots, x_{n-1}, x_{n-1}, x_{n-2}, \ldots x_k]$. This convention might seem peculiar, but as we will see it is useful for operating on signed integer data.

The C standard does not precisely define which type of right shift should be used. For unsigned data (i.e., integral objects declared with the qualifier `unsigned`), right shifts must be logical. For signed data (the default), either arithmetic or logical shifts may be used. This unfortunately means that any code assuming one form or the other will potentially encounter portability problems. In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case.

Practice Problem 2.15:

Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.

`x`		`x << 3`		`x >> 2` (Logical)		`x >> 2` (Arithmetic)	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
`0xF0`							
`0x0F`							
`0xCC`							
`0x55`							

C declaration	Guaranteed		Typical 32-bit	
	Minimum	Maximum	Minimum	Maximum
`char`	-127	127	-128	127
`unsigned char`	0	255	0	255
`short [int]`	$-32{,}767$	32,767	$-32{,}768$	32,767
`unsigned short [int]`	0	65,535	0	65,535
`int`	$-32{,}767$	32,767	$-2{,}147{,}483{,}648$	2,147,483,647
`unsigned [int]`	0	65,535	0	4,294,967,295
`long [int]`	$-2{,}147{,}483{,}647$	2,147,483,647	$-2{,}147{,}483{,}648$	$2, 147, 483, 647$
`unsigned long [int]`	0	4,294,967,295	0	4,294,967,295

Figure 2.8: **C Integral data types.** Text in square brackets is optional.

2.2 Integer Representations

In this section we describe two different ways bits can be used to encode integers—one that can only represent nonnegative numbers, and one that can represent negative, zero, and positive numbers. We will see later that they are strongly related both in their mathematical properties and their machine-level implementations. We also investigate the effect of expanding or shrinking an encoded integer to fit a representation with a different length.

2.2.1 Integral Data Types

C supports a variety of *integral* data types—ones that represent a finite range of integers. These are shown in Figure 2.8. Each type has a size designator: `char`, `short`, `int`, and `long`, as well as an indication of whether the represented number is nonnegative (declared as `unsigned`), or possibly negative (the default). The typical allocations for these different sizes were given in Figure 2.2. As indicated in Figure 2.8, these different sizes allow different ranges of values to be represented. The C standard defines a minimum range of values each data type must be able to represent. As shown in the figure, a typical 32-bit machine uses a 32-bit representation for data types `int` and `unsigned`, even though the C standard allows 16-bit representations. As described in Figure 2.2, the Compaq Alpha uses a 64-bit word to represent `long` integers, giving an upper limit of over 1.84×10^{19} for unsigned values, and a range of over $\pm 9.22 \times 10^{18}$ for signed values.

New to C?: Signed and unsigned numbers in C, C++, and Java.
Both C and C++ support signed (the default) and unsigned numbers. Java supports only signed numbers. **End.**

2.2.2 Unsigned and Two's-Complement Encodings

Assume we have an integer data type of w bits. We write a bit vector as either $\vec{x}$, to denote the entire vector, or as $[x_{w-1}, x_{w-2}, \ldots, x_0]$ to denote the individual bits within the vector. Treating $\vec{x}$ as a number written in binary notation, we obtain the *unsigned* interpretation of $\vec{x}$. We express this interpretation as a function