2 Bits, Bytes, and Data Types

In theory, a variable is an abstract concept. It is a placeholder for a value used in some computation. In implementation, a variable occupies a fixed storage. It is important to understand how variables differ in storage because this affects how different variables can be used in different computations. It promotes efficiency in programming by utilizing the correct data types, and hence the correct amount of memory (using 64-bit variables to store whole numbers between 1 and 100 is a waste of space). It promotes correct coding of arithmetic operations (coding floating point computations on whole numbers is a waste of time). It leads to a better understanding of bitwise operations, which are critical for many computing algorithms, such as in graphics. Finally, a proper understanding of memory helps in program design and debugging, especially when more complex data types are used, such as arrays and pointers. This chapter, and the following two chapters, all develop an understanding of C data types based upon an understanding of the actual memory they occupy.

2.1 • Bit Models

In C, what is the difference between an `int` and a `float`? The simple answer is that one stores whole numbers, whereas the other stores real numbers. But how? Or, how about some tougher questions: How does a `double` store higher-precision numbers compared to a `float`? Does it double the numerical range of possible values, or double the precision, or something else? How do the qualifiers `short` or `unsigned` change the way a value is stored? In order to answer these questions, we have to understand the bit models that underlie the data types.

A bit, short for binary digit, is a binary valued variable. The two possible values are typically written as 1 and 0, or true and false. On a computing chip (processor, memory chip, etc.), they are represented by high and low voltages, where a high value is typically 1–5 V and a low value is typically 0.0–0.5 V. Everything in computing is based upon combinations of bits. Everything stored in a computing chip is based upon using a fixed number of bits to model (or represent) the thing of interest.

For our discussion, bits can be either 1 or 0. Since a single bit can only represent only two values, we must group bits together to represent a wider range of numbers. The most common grouping is a byte, which is 8 bits grouped in sequence. For example, the following is a byte:

0	0	0	1	0	0	1	1

When bits are grouped, there are a variety of methods for interpreting them collectively. Each method of interpretation is called a *bit model*. We will now examine several bit models for representing various types and ranges of numbers.

2.1.1 Magnitude-only Bit Model

The simplest bit model is for nonnegative whole numbers. In this case, each bit represents a nonnegative integer power of 2. The place values of the bits are as follows:

example bit value	0	0	0	1	0	0	1	1
place value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
place value (base 10)	128	64	32	16	8	4	2	1

A bit value of 0 indicates a place value of 0; a bit value of 1 indicates a place value as given in the table. The total value of the number represented is found by adding up the place values of all the bits. In the example above, the value represented in the 8 bits (1 byte) is 19:

$$0+0+0+16+0+0+2+1=19$$

Given 8 bits, it is possible to store whole numbers in value up to

$$\sum_{i=0}^{7} 2^i = 2^8 - 1 = 255$$

or in the range 0 to 255.

The bits with the lowest powers of 2 (to the right in the above example) are called the *least significant bits*, or lowest-order bits, because they represent the smallest portions of the number. The bits with the highest powers of two (to the left in the above example) are called the *most significant bits*, or higher-order bits. The significance of bits can be thought of as which digits most change the number. It is common practice to list bits from highest to lowest, left to right, following the same convention used to write base 10 numbers.

Binary addition using this model is done similarly to base 10 addition. The easiest way to compute a sum by hand is to line up vertically the digits with similar powers. The digits in each power (column) are added, starting with the lowest power (rightmost column) and proceeding toward the highest power (leftmost column). If the sum in any single power exceeds the allowed range (9 in base 10, or 1 in base 2), then the sum is carried over to the next highest power. For example:

Base 10 (decimal)	Base 2 (binary)
7	00000111
+4	+00000100
11	00001011

In this example, there was a carry in base 10 from the one's digit to the ten's digit. There was also a carry in base 2 from the 2^2 digit to the 2^3 digit.

In general, storing numbers only within the range 0–255 is not terribly useful. Some things do use this range, such as graphical display pixel values, but obviously a wider range is needed for most computations. This is accomplished by grouping more bits together. For example, by grouping 4 bytes (32 bits) together, the magnitude-only bit model can represent whole numbers in value up to

$$\sum_{i=0}^{31} 2^i = 2^{32} - 1 = 4{,}294{,}967{,}295$$

or in the range 0–4,294,967,295. This number can be related to a size many personal computer enthusiasts are familiar with. At the time of this writing, most common personal computers and workstations can have a maximum of 4 GB of memory. Why? It is a result of the size of the memory bus, which can be thought of as the number of wires from the system to its memory. With 32 wires, a system can address 4,294,967,296 (which is about 4 billion, or 4 "giga") different memory locations. If each memory location is a byte (8 bits), then a "32-bit architecture" can have a maximum of 4 GB of memory.

In C, several data types use the magnitude-only bit model. An `unsigned char` is a 1-byte (8 bits) variable with a range of 0 to 255. On a 32-bit system, an `unsigned int` is a 4-byte (32 bits) variable with a range of 0 to 4,294,967,295, and an `unsigned short int` is a 2-byte (16 bits) variable with a range of 0 to 65,535. Technically, the C language does not define the size of an `int`. However, the 32-bit usage has become so predominant that we will treat it as the standard within this text. At the time of this writing, as 64-bit architectures are becoming more common, there is debate over whether to standardize the C `int` as 4 bytes or continue to allow it to vary depending on the system architecture. Regardless of size, the concepts taught in this text are valid, but it is easier to learn the concepts by solidifying the size of an `int` at a specific value.

2.1.2 Sign-magnitude Bit Model

In the case where signed whole numbers are desired, a common practice is to allocate the highest-order bit to be the *sign bit*. This is called the sign-magnitude model:

example bit value	1	0	0	1	0	0	1	1
place value	sign	2^6	2^5	2^4	2^3	2^2	2^1	2^0
place value (base 10)	+ or −	64	32	16	8	4	2	1

By common convention, a value of 0 in the sign bit indicates a positive number, while a value of 1 in the sign bit indicates a negative number. In this example, the decimal value represented using the 8 bits is −19. Using the sign-magnitude bit model, it is possible with 8 bits to represent whole numbers in the range −127 to +127.

The sign-magnitude model suffers from two drawbacks. First, notice that there are two possible bit values for zero: 00000000 can be interpreted as "positive zero," while 10000000 is interpreted as "negative zero." This does not make much sense. Even more important, using this bit model makes binary addition somewhat complicated. If we have zero or two negative numbers, we can perform addition exactly as outlined for the magnitude-only bit model. However, if we have one negative number and one positive number, we must instead perform a subtraction. While subtraction is not a terribly difficult task, it would be nice if we could use the same method for binary addition regardless of the signs of the two numbers.

Because of these two drawbacks, no data types in C use the sign-magnitude bit model. It is usually seen only in simple computing circuits, or in design problems where these drawbacks do not present any difficulty. The bit model discussed next was designed to overcome these drawbacks.

2.1.3 Two's Complement Bit Model

Using the two's complement bit model, positive integers (and zero) are represented exactly the same as they are in the magnitude-only bit model. Negative numbers are represented by applying the following sequence of steps:

1. Write the bits for the positive version of the number.
2. Invert (flip) all the bits.
3. Add 1.

For example, to represent −7, we proceed through the following steps:

positive value (+7)	0	0	0	0	0	1	1	1
invert all bits	1	1	1	1	1	0	0	0
add 1	1	1	1	1	1	0	0	1

The process of adding 1 is carried out exactly as described in the previous sections. Based on our example, we find that the two's complement bit representation for −7, using 8 bits, is 11111001.

When a two's complement number has a 1 in the highest bit, it indicates that the number is negative. To find the value, we perform the same steps:

unknown value (−?)	1	1	1	1	1	0	0	1
invert all bits	0	0	0	0	0	1	1	0
add 1	0	0	0	0	0	1	1	1

After performing these steps, the value provides the magnitude of the negative number. In this example, we get a magnitude of 7, so the original bit pattern 11111001 is known to be −7.

The two's complement bit model solves the double zero problem seen with the sign-magnitude model. The only bit pattern for zero is 00000000. Applying the steps that compute the negative value for the pattern yields the original pattern:

0 (base 10)	0	0	0	0	0	0	0	0
invert all bits	1	1	1	1	1	1	1	1
add 1	0	0	0	0	0	0	0	0

Since we only have 8 bits, the carry out of the highest-order bit is lost in this operation; it has no place to go. Therefore trying to compute "negative zero" results in the same bit pattern as "positive zero," thus there is only one bit pattern for zero.

Using the sign-magnitude model, the bit pattern 10000000 represented "negative zero." In the two's complement bit model, what does it represent? To find the value, we apply the usual operations for interpreting a two's complement number:

unknown value (−?)	1	0	0	0	0	0	0	0
invert all bits	0	1	1	1	1	1	1	1
add 1	1	0	0	0	0	0	0	0

Since the first bit is 1, we know the number must be negative. After inverting and adding 1, we obtain a magnitude of 128. Therefore, the bit pattern 10000000 using the two's complement bit model represents −128 (base 10). This is called the weird number. When following the two's complement conversion steps, it comes out the same as it started. However, at the beginning, we look only at the highest bit to determine that the number is negative. After converting, we see that the magnitude is 128, and so the complete value is −128. Thus, using the two's complement bit model, the range of integers that can be represented using 8 bits is −128 to +127.

Two's complement also makes addition and subtraction easier to implement because, regardless of the signs of the two numbers, they can always be added. For example, consider 7 + (−5):

	Base 10 (decimal)	**Base 2 (binary)**
carry bits		11111111
	7	00000111
	+ (−5)	+ 11111011
	2	00000010

The bit pattern for −5 was obtained through the two's complement conversion steps: +5 = 00000101; inverted it becomes 11111010; and after adding 1 it be-

comes 11111011. When performing the addition, there was a carry value of 1 in every bit place. The final carry bit at the end was again thrown away because it has no place to go. However, it must be checked. For the sum to be valid, the highest two carry bits must be either 11 or 00. In this example, the final two carry bits were 11, and so the sum is valid.

If the highest two carry bits are either 10 or 01, then the result is an *arithmetic overflow*. Arithmetic overflow is when the process of a calculation results in a number outside the range of values that can be represented by the available bits. For example:

	Base 10 (decimal)	Base 2 (binary)
carry bits		10000000
	(−127)	10000001
	+ (−126)	+ 10000001
	−253	00000011

The bit patterns for both −127 and −126 were computed normally (the reader is encouraged to work these out). Both of these values are fine; the two's complement bit model can represent them using 8 bits. However, the result of the addition is 00000011, which equates to 3 in base 10. This is of course wrong. What has happened is that an overflow has occurred. With 8 bits, we cannot represent −253 (it is outside of the allowed range of −128 to +127). We can see that an overflow has occurred by looking at the highest two carry bits, which in this example are 10.

Several data types in C use the two's complement bit model. A `char` uses 1 byte (8 bits) to represent numbers in the range −128 to +127. An `int` uses 4 bytes (32 bits) to represent numbers in the range −2,147,483,648 to +2,147,483,647. A `short int` uses 2 bytes (16 bits) to represent numbers in the range −32,768 to +32,767. The two's complement bit model is the most commonly seen bit model for representing signed whole numbers in computing systems.

2.1.4 Floating Point Bit Model

For storing real numbers, a method completely different from the previous bit models must be used. Some of the bits must be used to represent the fractional portion of the number. One possibility is to use bits to denote powers of 2 that are negative, and hence fractions. For example, we could use 1 byte (8 bits) as follows:

example bit value	1	0	0	1	0	0	1	1
place value	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
place value (base 10)	16	8	4	2	1	0.5	0.25	0.125

This approach is called a *fixed point* bit model, because the position of the number of digits (in base 10) that can be represented in the fraction are fixed. In our example, the base 10 value that is represented by the 8 bits is 18.375. In this example, only three fractional digits of a base 10 number can be represented. This is generally considered a limitation, and a poor use of available bits.

It is possible to represent a wider range of fractions, using the same number of bits, using a *floating point* bit model. The general idea is to represent the number using scientific notation. For example, in base 10 notation:

$$123.456 = 1.23456 \times 10^2$$

In a computer we have only bits, so we must use base 2 scientific notation. For example, the bits representing 18.375 from above could be written as follows:

$$18.375 \text{ (base 10)} = 10010.011 \text{ (base 2)} = 1.0010011 \times 2^4$$

Since it is known that base 2 is used, we do not need to store the "2" each time; it is implied. Similarly, we can simplify the storage of a number by assuming that the leading mantissa (the value in front of the decimal point) is 1. Therefore the floating point bit model stores numbers in the following form:

$$\pm 1.f \times 2^e$$

The values that must be stored are the bits for the sign, the fraction f, and the exponent e. Obviously, only one bit is needed to store the sign. The remaining available bits can be divided between the fraction and the exponent. Given 32 bits (4 bytes), the standard[1] is to represent the fraction using 23 bits and the exponent using 8 bits:

Total bits	**1**	**8**	**23**
	sign	exponent (e)	fraction (f)
bit places	31	30 . . . 23	22 . . . 0

The bits in the fraction represent negative powers of 2, as described above. For example, bit 22 represents $2^{-1} = \frac{1}{2} = 0.5$; bit 21 represents $2^{-2} = \frac{1}{4} = 0.25$; bit 20 represents $2^{-3} = \frac{1}{8} = 0.125$; and so on. The 8 bits for the exponent are used to

1. The IEEE 754 floating point standard.

represent whole values in the range −127 to +128. The negative values support the representation of small fractions, and hence larger precision after the decimal point. Conversely, the positive values support the representation of large non-fractional numbers. Thus, there is a choice in how the available bits will be used. For example, it is possible to represent 0.123456 or 123.456 or 123,456.0, but it is not possible to represent 123,456.123456. The total precision available in base 10 is about eight digits using a 4-byte floating point bit model.

The following steps can be used to convert a number from base 10 to binary using the floating point bit model:

1. Write the sign bit.
2. Write the number in fixed point binary, without the sign.
3. Normalize, moving the radix (the decimal point) to just after the first "1" digit.
4. Take f as the values to the right of the radix, zero padded.
5. Take e as the given exponent, biased by adding +127.

The following example demonstrates these steps. The value −118.625 will be encoded:

(1) get sign bit	s = 1
(2) write # in fixed point binary	118.625 = 1110110.101
(3) normalize	$1110110.101 = 1.110110101 \times 2^6$
(4) get fraction from right of radix	f = 110 1101 0100 0000 0000 0000
(5) get exponent, biased +127	e = 6+127 = 133 = 1000 0101

The exponent is biased by adding 127 so that the exponent can be stored as a mangnitude-only, and yet still represent the range −127 to +128. For our example, the final 32-bit floating point representation is:

```
1 1000 0101 110 1101 0100 0000 0000 0000
```

Note that the spaces are for our convenience only; they are not represented in any way in the computing system. The following are some additional examples:

Real number	Bit pattern
4.125	0 1000 0001 000 0100 0000 0000 0000 0000
123456.123	0 1000 1111 111 0001 0010 0000 0001 0000
−6.429678	1 1000 0001 100 1101 1011 1111 1110 1100

In the C language, the `float` data type uses 4 bytes to store a real number, and the `double` data type uses 8 bytes to store a real number. The double still uses 1 bit for sign, but it uses 11 bits to store the exponent and 52 bits to store the fraction, so that it can represent a wider range of numbers.

2.1.5 ASCII and Unicode Bit Models

In order to represent nonnumeric data, another bit model must be used. The ASCII (American Standard Code for Information Interchange) bit model was developed to represent English text symbols, along with control characters necessary to print English text. A partial listing of the ASCII bit model is shown in Table 2.1 (a full listing is in Appendix A). There are 128 total ASCII bit patterns. Each bit pattern uses 7 bits. In order to fill a byte, an eighth bit is added to each pattern. Historically, this extra bit has been used for several purposes, such as parity (a form of error recovery) and extended ASCII sets (providing an additional 128 bit patterns). However, it is also common to leave the bit with a value of zero, in effect padding the 7-bit ASCII patterns with a preceding zero. Most of the bit patterns in ASCII represent printable symbols, such as the lowercase versions of letters in the English alphabet. The patterns for both the lower- and uppercase letters are stored in order, from 97 to 122 and from 65 to 90, respectively. The first 32 bit patterns (from 0000 0000 to 0000 1000) represent control characters, which do not correspond to printable symbols. Instead, these characters control some aspect of printing, such as a backspace, horizontal tab, or carriage return.

In C, both the `char` and `unsigned char` data types use the ASCII bit model. However, because of their 7-bit nature, the eighth bit allows for a second, different use of each data type. If we interpret each of the bit patterns as magnitude-only whole numbers, then we obtain the base 10 values listed in Table 2.1. The unsigned `char` data type can be interpreted in this manner, providing a range of values from 0 to 255. If we interpret each of the bit patterns using the two's complement bit model, then we obtain base 10 values in the range −128 to +127. The `char` data type can be interpreted in this manner. This dual interpretation of the bit patterns can be seen through the following C code:

```
char a;
unsigned char b;
a='A';
b='B';
printf("%c %c %d %d\n",a,b,a,b);
a=183;
b=255;
printf("%d %d\n",a,b);
```

Table 2.1 Partial listing of ASCII bit model.

Bit values	Base 10	Description
0000 0000	0	null character
0000 1000	8	backspace
0000 1001	9	horizontal tab
0000 1010	10	line feed
0000 1101	13	carriage return
0000 1000	27	escape
0010 0000	32	space (space bar)
0010 0001	33	! (exclamation point)
0011 0000	48	0 (numeric zero)
0011 0001	49	1 (numeric one)
0011 1001	57	9 (numeric nine)
0100 0001	65	A
0100 0010	66	B
0101 1010	90	Z
0110 0001	97	a
0110 0010	98	b
0111 1010	122	z

The output of executing this code is as follows:

```
A B 65 66
-73 255
```

Using the `%c` designator, the printf() function call will interpret the variable using the ASCII bit model. Using the `%d` (or `%i`) designator, the printf() function call will interpret the variable as a whole number. In this case, which bit model to use depends upon the data type. For the `char` data type, the printf() function interprets the variable using the two's complement bit model; whereas for the `unsigned char` data type, the printf() function interprets the variable using the magnitude-only bit model. The magnitude-only bit pattern for 183 is 1011 0111, which interpreted using the two's complement bit model is −73 in base 10. This is why the variable `a` appears to change in the second printf() output.

The ASCII bit model has a few problems. Many of the ASCII control characters are obsolete, relics of the earliest days of digital text transmission and printing. The eighth bit is open to interpretation, and has consequently been used in

a variety of methods on different computing systems. Most important, the symbol range covers only the English language. In response to these problems, the Unicode bit model was developed. In its most straightforward implementation, it uses 16-bit patterns to represent 65,536 total symbols. These include the alphabets of many languages besides English. In C, a `short int` is commonly used to store a Unicode bit pattern. Because of the prevalence and continued use of the ASCII bit model, the first 128 bit patterns represent the same symbols in the Unicode bit model as they do in the ASCII bit model. The full details of Unicode can be found in the reference book *The Unicode Standard*, 5th ed., Unicode Consortium, Addison-Wesley, 2006.

2.1.6 Bit Model Summary

To summarize, what is 1100 0010 1110 1101 0100 0000 0000 0000 ?

The answer is that it depends on how the bits are grouped and interpreted. In other words, it depends upon the bit model. Here are some possible answers:

C variable(s)	Bit model	Value(s) or symbol(s)			
4 unsigned char	magnitude-only	194	237	64	0
4 char	two's complement	−62	9	64	0
4 char	ASCII	[x]	[x]	@	[NULL]
1 int	two's complement	−1,024,638,976			
1 unsigned int	magnitude-only	3,270,328,320			
1 float	floating point	−118.625			

The bits can be grouped together 8 at a time (1 byte at a time), or 32 at a time, or other possibilities not listed here. After grouping, different bit models can be applied to determine the value(s) or symbol(s) represented.

The important concept to understand is that bits alone do not provide data. They must be grouped and interpreted according to the appropriate bit model; otherwise they are meaningless. A common pitfall for naive computer operators is to open up any file within a text editor. However, not all bit patterns relate to printable symbols. Looking at the example above, 2 of the 4 bytes do not represent any regular ASCII symbols; 1 of the bytes represents a control value (the NULL character); and only 1 byte represents a printable symbol (the @ symbol). This is why using a text editor to view any random file will often display it as seemingly a bunch of garbage. A text editor interprets all the bits using the ASCII bit model, when the underlying data was not encoded that way.

A common pitfall for naive programmers is to use the wrong data type for the given data or computation. This amounts to using the wrong bit model, with consequences similar to opening a random file using a text editor. When seeing a value that looks very strange (i.e., garbage), one should look to make sure the correct data type is being used. The reason for *type casting* computations to use the same data type is to ensure that all the variables involved are using the same bit model. For example:

```
int    i=3;
double d=7.2;
float  f;
f=(float)i+(float)d;
```

In the last statement, the (float) in front of the variables `i` and `d` converts the bits used to represent the values to the float bit model so that the addition operation can be performed.

While programming, even when it is not necessary to understand how the individual bits are organized in variables, it is often important to understand how many bytes are used by variables. The `sizeof()` operator reports how many bytes a data type, or a variable, is using:

```
int    i;
char   c;
double d;
printf("%d %d %d %d\n",sizeof(i),sizeof(c),
          sizeof(d),sizeof(float));
```

The result of executing this code is:

```
4 1 8 4
```

This matches our expectations: a variable of type char is 1 byte in size; an int and float are 4 bytes each; and a double is 8 bytes. In Chapter 4, when we look at structures and pointers, we will revisit the `sizeof()` operator.

2.2 • Bitwise Operations

In some situations, a single bit can store all the information needed for a computation. For example, a file can be write-protected, or not. A user can be logged in, or not. A program can be currently running, or not. In many advanced algorithms, bits are frequently used in computations and data structures. For example, compression/decompression routines (codecs) commonly manipulate bit

patterns to reduce the necessary storage size. Sorting methods commonly use trees and other binary structures during operation. Bitwise operations are also common in device drivers and graphics programming.

In the C language, the smallest data type available is the `char`, which is 1 byte (8 bits). How then can one store or manipulate a single bit? One possibility is to use an entire char to store the single bit. This is of course wasteful and, if used indiscriminately, would seriously raise the amount of memory necessary to operate a modern computing system. Another option is to figure out how to manipulate just 1 bit within a char, or within any other data type. In the C language, this is accomplished using bitwise operations.

In this section, we first describe the basic logic underlying bitwise operations. We then look at the C bit operators that are used to program bitwise operations. Finally, we look at bit masking, which is the most common type of coding problem involving bit manipulations. In later chapters in this text, we will again see bit coding problems when we look at file attributes (Chapter 5) and some graphics operations (Chapter 8).

2.2.1 Binary Logic Operations

Recall that a single bit can have a value of either 1 or 0, which can also represent true or false. The three most basic logic operations are AND, OR, and NOT. The operations work as follows:

AND	OR	NOT
0 AND 0 = 0	0 OR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	

The result of an AND operation is true only if both input values are true. The result of an OR operation is true if either input value is true. The result of a NOT operation is to reverse the bit value. Logic operations are independent of any particular programming language used to implement them. There are several other logic operations, for example NOR and NAND, which are commonly used in circuits. In this text, we are concerned only with the basic three operations and how to implement and use them in C programming.

2.2.2 Bit Operators

Bit operators are the symbols or syntax used in a programming language to affect individual bits within a variable. In the C programming language, there are six bit operators:

Operator	Symbol name	Action
~	tilde	bitwise NOT
&	ampersand	bitwise AND
|	vertical bar	bitwise OR
^	caret	bitwise XOR
>>	greater-than greater-than	right-shift
<<	less-than less-than	left-shift

The bit operators are intended to be used only on the whole number data types, `char` and `int`, and the related extensions (e.g., `unsigned char`). Although bits can be manipulated in any variable, it is rare to see bit operators applied to the real number data types.

The bitwise NOT operator inverts every bit in a variable. It is written using the tilde symbol (~). For example:

```
unsigned char a;
a=17;
a=~a;
printf("%d\n",a);
```

The result of executing this code is:

```
238
```

At the bit level, the result of the code is:

C code	Bits in variable a	Base 10 value
a=17;	00010001	17
a=~a;	11101110	238

The bitwise AND operator performs an AND between two variables, independently at every bit. It is written using the ampersand symbol (&). For example:

```
unsigned char a,b;
a=17;
b=22;
a=a & b;
printf("%d\n",a);
```

The result of executing this code is:

```
16
```

At the bit level, the result of the code is:

C code	Bits in variable a	Base 10 value
a=17;	00010001	17
b=22;	00010110	22
a=a & b;	00010000	16

Prior to the bitwise AND, only one bit position had a value of 1 in both variables a and b. Therefore, after the bitwise AND, this is the only bit position with a value of 1.

The bitwise OR operator performs an OR between two variables, independently at every bit. It is written using the vertical bar symbol (|). For example:

```
unsigned char a,b;
a=17;
b=22;
a=a | b;
printf("%d\n",a);
```

The result of executing this code is:

```
23
```

At the bit level, the result of the code is:

C code	Bits in variable a	Base 10 value
a=17;	00010001	17
b=22;	00010110	22
a=a | b;	00010111	23

After the bitwise OR, the variable a has a value of 1 in any bit position in which either of the two input variables a or b had a value of 1.

Like the arithmetic or logic operators in C, the bit operators can be applied to either variables or constants. For example:

```
char x,y;
x=7;
y=6;
x=x&y;
y=x|16;
printf("%d %d\n",x,y);
```

The result of executing this code is:

```
6 22
```

At the bit level, the result of the code is:

C code	Bits in variable a	Base 10 value
x=7;	00000111	7
y=6;	00000110	6
x=x & y;	00000110	6
y=x | 16;	00010110	22

The left-shift and right-shift bit operators move bits into higher-order and lower-order bit positions, respectively. When a bit pattern is written out horizontally, this is equivalent to moving the bits toward the left, or toward the right, respectively. A left-shift is written in C using two consecutive less-than symbols (<<) and a right-shift is written using two consecutive greater-than symbols (>>). The value following the shift operator indicates how many bit positions to move. For example:

```
unsigned char a,b;
a=17;
a=a << 2;
b=64;
b=b >> 3;
printf("%d %d\n",a,b);
```

The result of executing this code is:

```
68 8
```

At the bit level, the result of the code is:

C code	Bits in variable a	Base 10 value
a=17;	0 0 0 1 0 0 0 1	17
a=a << 2;	0 1 0 0 0 1 0 0	68
b=64;	0 1 0 0 0 0 0 0	64
b=b >> 3;	0 0 0 0 1 0 0 0	8

After the left-shift, every bit value in the variable a has moved to the left (to a higher-order bit) by two bit positions. After the right-shift, every bit value in the variable b has moved to the right (to a lower-order bit) by three bit positions. Any bits that move beyond the highest or lowest available bit are discarded. For example:

Original bits	Shifting	Discarded bits	Result
01101111	left 3	011	01111 000
01101111	right 3	111	000 01101

The new bit values that take up residence in the now vacated bit positions are given values that depend upon the bit model used by the variable. For a magnitude-only bit model, the new bit values are always zero, as shown above. For a two's complement bit model, the new bit values are zero for left-shifts, and copies of the original highest-order bit for right-shifts. The latter maintains the original sign of the value, while shifting the negative number in a manner synonymous with positive numbers. For example:

```
char a,b;
a=17;
a=a >> 2;
b=-65;
b=b >> 2;
printf("%d %d\n",a,b);
```

The result of executing this code is:

```
4 -17
```

At the bit level, the result of the code is:

C code	Bits in variable a	Base 10 value
`a=17;`	00010001	17
`a=a >> 2;`	00000100	4
`b= -65;`	10111111	−65
`b=b >> 2;`	11101111	−17

2.2.3 Bitmask Operations

Bitmasking is perhaps the most common type of bitwise operation. It involves using a *bitmask* to change or query one or more designated bits within a variable. The bitmask indicates which bits are to be affected by the operation. The idea is to operate on a variable, changing or affecting only the bits indicated by the bitmask:

variable → bitmask (bit N) → variable (only bit N changed)

The bitmask indicated that bit N should be changed (N is the bit position, which is also the power of 2 of the bit). The following are examples of bitmasks for an 8-bit variable (e.g., unsigned char):

Bitmask	Base 10 value	Indicated bits to work on
00000001	1	bit 0
00010000	16	bit 4
10101100	172	bits 2, 3, 5, and 7

A bitmask can indicate that any number of bits are to be affected. The three most common bitmask operations work on a single bit: set the bit, clear the bit, or query the value of the bit. Each of these operations can be accomplished through the following logic:

Operation	Logic
set Nth bit	x = x OR 2^N
clear Nth bit	x = x AND NOT(2^N)
read Nth bit	= x AND 2^N

The act of setting a bit gives the bit a value of 1, regardless of its initial value. At the same time, it leaves all the other bits unchanged. The act of clearing a

bit gives the bit a value of 0, regardless of its initial value, and similarly leaves all the other bits unchanged. The act of querying a bit determines the current value of a bit, leaving all bit values unchanged. Each of these operations can be implemented in C as follows:

Operation	C code
set Nth bit	`x = x | (1<<N);`
clear Nth bit	`x = x & (~(1<<N));`
read Nth bit	`(x & (1<<N)) >>N`

When reading the Nth bit, the final right-shift operation results in a value of 1 or 0 regardless of which bit is being read. Without that final right-shift, the read value will be equal to the value of the bit position (the power of 2 of the bit place).

The following code demonstrates setting, clearing, and reading bits:

```
char a;
int i;
a=17;
a=a | (1 << 3);      /* set 3rd bit */
printf("%d\n",a);
a=a & (~(1<<4));     /* clear 4th bit */
printf("%d\n",a);
for (i=7; i>=0; i--)
  printf("%d ",(a&(1<<i)) >> i);  /* read i'th bit */
printf("\n");
```

The result of executing this code is:

```
25
9
0 0 0 0 1 0 0 1
```

At the bit level, the result of the code is:

C code	Operation	Bits in variable a	Base 10 value
`a=17;`		00010001	17
`a=a | (1 << 3);`	set bit 3	00011001	25
`a=a & (~(1 << 4));`	clear bit 4	00001001	9

2.3 • Memory Map

In this section, we introduce the *memory map* concept. A memory map is a table, listing all the variables in a piece of code. The table includes the variable names, values, and memory addresses. The addresses indicate the sizes of the variables, in bytes. Each address is assumed to represent 1 byte. For example, consider the following code:

```
char a,b,c;
a=7;
b=-13;
c=0;
```

The memory map for this code could be written as follows:

Label	Address	Value
a	400	7
b	401	-13
c	402	0

The starting address is not important; 400 was picked so that the range of values in the addresses differs significantly from the values in the variables, so that they are easier to read. The important thing to note in the address column is that each variable occupies only 1 byte. This is because a char variable is only 1 byte in size.

If variables of different types are used, then the address column should reflect their relative sizes. For example, consider the following code:

```
char a;
int b;
float c;
double d;
a=7;
b=-13;
c=0.1;
d=42.5;
```

The memory map for this code could be written as follows:

Label	Address	Value
a	400	7
b	401–404	-13
c	405–408	0.1
d	409–416	42.5

In this example, we see the appropriate sizes of the different data types reflected in the address column. A char is 1 byte, an int and a float are both 4 bytes, and a double is 8 bytes in size.

It is important to remember that what resides in memory are bit patterns. In order to be interpreted as a value (or symbol), the appropriate bit model must be applied. Which model is determined by the data types. For example, consider the following code:

```
char a;
short int b;
char c;
a=6;
b=13;
c='6';
```

The memory map for this code could be written as follows:

Label	Address	Bits	Value/symbol
a	400	0000 0110	6
b	401–402	0000 0000 0000 1101	13
c	403	0011 0110	'6'

This example emphasizes the difference between the symbol '6' and the integer value 6, which have different bit patterns. In most cases, it is not necessary to include the bit patterns in a memory map. One can differentiate between symbols and values using appropriate notation, such as enclosing symbols within single quotes (as in C coding).

A memory map can be used in a dynamic manner, keeping track of variable values as a piece of code executes. As a tool, it lets a programmer work through a piece of code to determine its result. For example, consider the following code:

```
int i,n;
n=0;
for (i=1; i<=4; i++)
  n=n+i;
```

The memory map for this code could be written as follows:

Label	Address	Value
i	400–403	1 2 3 4 5
n	404–407	0 1 3 6 10

Note that the final value of `i` in the memory map is 5, which is 1 beyond the loop bounds. This is because the loop is not terminated until the tested condition `i<=4` is false, which is when i=5. This is a common programming mistake (recognizing the value of a loop counter after the loop terminates) that can be caught by working through code using a memory map.

The memory map can point out another common programming mistake. Consider the following code, which is intended to compute the sum of the first five even integers:

```
int i,sum;
for (i=1; i<=10; i++)
  if (i%2 == 0)
    sum=sum+i;
```

Have you spotted the coding error? Through a partial execution of the program, the memory map could be written as follows:

Label	Address	Value
i	400–403	1 2
sum	404–407	?

In the first iteration, i=1, `i%2` is equal to 1, and so the if statement fails. In the second iteration, i=2, `i%2` is equal to 0, and so the if statement is true. The program now comes to the line that adds i to sum. However, sum has not previously been given a value. What is its current value? The answer is that the value is unknown. Every variable always has a value. There is no such thing as a "blank" variable; the wires storing the bits for the variable must have either high

or low voltages, corresponding to 1's and 0's. However, because the program did not previously give this variable a value, the current value is unknown. It could be zero, or it could be anything. We can change the code to see this effect:

```
int i,sum;
printf("%d\n",sum);
for (i=1; i<=10; i++)
if (i%2 == 0)
  sum=sum+i;
printf("%d\n",sum);
```

Executing this code yields (results will vary from machine to machine, and even moment to moment):

```
3457056
3457086
```

Executing the program again yields:

```
14852128
14852158
```

Additional executions of the program yield similarly strange numbers, always having a difference of 30. That difference is due to the desired calculation, while the rest of it is due to the original unknown value in the variable when the program started each time.

The examples shown in this section are relatively simple. The true power of a memory map becomes apparent when it is used to work with pointers, arrays, and structures. In Chapters 3 and 4, the memory map will be used repeatedly to look at these constructs.

Questions and Exercises