15

Bit Fiddling

You will learn about the following in this chapter:

- Operators:

  ```
  ~ & |^
  ```

  ```
  >> <<
  ```

  ```
  &= |= ^= >>= <<=
  ```

- Binary, octal, and hexadecimal number notations (a review)
- Two C facilities for handling the individual bits in a value: bitwise operators and bit fields
- Keywords:

  ```
  _Alignas, _Alignof
  ```

With C, you can manipulate the individual bits in a variable. Perhaps you are wondering why anyone would want to. Be assured that sometimes this ability is necessary, or at least useful. For example, a hardware device is often controlled by sending it a byte or two in which each bit has a particular meaning. Also, operating system information about files often is stored by using particular bits to indicate particular items. Many compression and encryption operations manipulate individual bits. High-level languages generally don't deal with this level of detail; C's ability to provide high-level language facilities while also being able to work at a level typically reserved for assembly language makes it a preferred language for writing device drivers and embedded code.

We'll investigate C's bit powers in this chapter after we supply you with some background about bits, bytes, binary notation, and other number bases.

Binary Numbers, Bits, and Bytes

The usual way to write numbers is based on the number 10. For example, 2157 has a 2 in the thousands place, a 1 in the hundreds place, a 5 in the tens place, and a 7 in the ones place. This means you can think of 2157 as being the following:

```
2 x 1000 + 1 x 100 + 5 x 10 + 7 x 1
```

However, 1000 is 10 cubed, 100 is 10 squared, 10 is 10 to the first power, and, by convention, 1 is 10 (or any positive number) to the zero power. Therefore, you can also write 2157 as this:

$$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

Because our system of writing numbers is based on powers of 10, we say that 2157 is written in *base 10*.

Presumably, the decimal system evolved because we have 10 fingers. A computer bit, in a sense, has only two fingers because it can be set only to 0 or 1, off or on. Therefore, a *base 2* system is natural for a computer. It uses powers of two instead of powers of 10. Numbers expressed in base 2 are termed *binary numbers*. The number 2 plays the same role for binary numbers that the number 10 does for base 10 numbers. For example, a binary number such as 1101 mean this:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

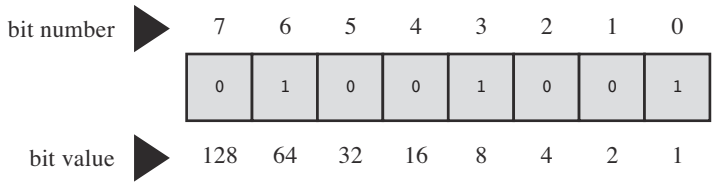
In decimal numbers, it becomes this:

```
1 x 8 + 1 x 4 + 0 x 2 + 1 x 1 = 13
```

You can use the binary system to express any integer (if you have enough bits) as a combination of 1s and 0s. This system is very convenient for digital computers, which express information in combinations of on and off states that can be interpreted as 1s and 0s. Let's see how the binary system works for a 1-byte integer.

Binary Integers

Usually, a byte contains 8 bits. C, remember, uses the term *byte* to denote the size used to hold a system's character set, so a C byte could be 8 bits, 9 bits, 16 bits, or some other value. However, the 8-bit byte is the byte used to describe memory chips and the byte used to describe data transfer rates. To keep matters simple, this chapter assumes an 8-bit byte. (For clarity, the computing world often uses the term *octet* for an 8-bit byte.) You can think of these 8 bits as being numbered from 7 to 0, left to right. Bit 7 is called the *high-order bit,* and bit 0 is the *low-order bit* in the byte. Each bit number corresponds to a particular exponent of 2. Imagine the byte as looking like Figure 15.1.



This example shows bits 6, 3, and 0 set to 1.
The value of this byte is 64 + 8 + 1 or 73.

Figure 15.1 Bit numbers and bit values.

Here, 128 is 2 to the 7th power, and so on. The largest number this byte can hold is 1, with all bits set to 1: 11111111. The value of this binary number is as follows:

```
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

The smallest binary number would be 00000000, or a simple 0. A byte can store numbers from 0 to 255, for a total of 256 possible values. Or, by interpreting the bit pattern differently, a program can use a byte to store numbers from –128 to +127, again a total of 256 values. For example, `unsigned char` typically uses a byte to represent the 0-to-255 range, whereas `signed char` typically uses a byte to represent the –128 to +127 range.

Signed Integers

The representation of signed numbers is determined by the hardware, not by C. Probably the simplest way to represent signed numbers is to reserve 1 bit, such as the high-order bit, to represent the sign. In a 1-byte value, this leaves 7 bits for the number itself. In such a *sign-magnitude* representation, 10000001 is –1 and 00000001 is 1. The total range, then, is –127 to +127.

One disadvantage of this approach is that it has two zeros: +0 and –0. This is confusing, and it also uses up two bit patterns for just one value.

The *two's-complement* method avoids that problem and is the most common system used today. We'll discuss this method as it applies to a 1-byte value. In that context, the values 0 through 127 are represented by the last 7 bits, with the high-order bit set to 0. So far, that's the same as the sign-magnitude method. Also, if the high-order bit is 1, the value is negative. The difference comes in determining the value of that negative number. Subtract the bit-pattern for a negative number from the 9-bit pattern 100000000 (256 as expressed in binary), and the result is the magnitude of the value. For example, suppose the pattern is 10000000. As an unsigned byte, it would be 128. As a signed value, it is negative (bit 7 is 1) and has a value of 100000000–10000000, or 10000000 (128). Therefore, the number is –128. (It would have been –0 in the sign-magnitude system.) Similarly, 10000001 is –127, and 11111111 is –1. The method represents numbers in the range –128 to +127.

The simplest method for reversing the sign of a two's-complement binary number is to invert each bit (convert 0s to 1s and 1s to 0s) and then add 1. Because 1 is 00000001, –1 is 11111110 + 1, or 11111111, just as you saw earlier.

The *one's-complement* method forms the negative of a number by inverting each bit in the pattern. For instance, 00000001 is 1 and 11111110 is –1. This method also has a –0: 11111111. Its range (for a 1-byte value) is –127 to +127.

Binary Floating Point

Floating-point numbers are stored in two parts: a binary fraction and a binary exponent. Let's see how this is done.

Binary Fractions

The ordinary fraction 0.527 represents

```
5/10 + 2/100 + 7/1000
```

with the denominators representing increasing powers of 10. In a binary fraction, you use powers of two for denominators, so the binary fraction .101 represents

```
1/2 + 0/4 + 1/8
```

which in decimal notation is

```
0.50 + 0.00 + 0.125
```

or 0.625.

Many fractions, such as 1/3, cannot be represented exactly in decimal notation. Similarly, many fractions cannot be represented exactly in binary notation. Indeed, the only fractions that can be represented exactly are combinations of multiples of powers of 1/2. Therefore, 3/4 and 7/8 can be represented exactly as binary fractions, but 1/3 and 2/5 cannot be.

Floating-Point Representation

To represent a floating-point number in a computer, a certain number of bits (depending on the system) are set aside to hold a binary fraction. Additional bits hold an exponent. In general terms, the actual value of the number consists of the binary fraction times 2 to the indicated exponent. Multiplying a floating-point number by, say, 4, increases the exponent by 2 and leaves the binary fraction unchanged. Multiplying by a number that is not a power of 2 changes the binary fraction and, if necessary, the exponent.

Other Number Bases

Computer workers often use number systems based on 8 and on 16. Because 8 and 16 are powers of 2, these systems are more closely related to a computer's binary system than the decimal system is.

Octal

Octal refers to a base 8 system. In this system, the different places in a number represent powers of 8. You use the digits 0 to 7. For example, the octal number 451 (written 0451 in C) represents this:

$4 \times 8^2 + 5 \times 8^1 + 1 \times 8^0 = 297$ (base 10)

A handy thing to know about octal is that each octal digit corresponds to three binary digits. Table 15.1 shows the correspondence. This correspondence makes it simple to translate between the two systems. For example, the octal number 0377 is 11111111 in binary. We replaced the 3 with 011, dropped the leading 0, and then replaced each 7 with 111. The only awkward part is that a 3-digit octal number might take up to 9 bits in binary form, so an octal value larger than 0377 requires more than a byte. Note that internal 0s are not dropped: 0173 is 01 111 011, not 01 111 11.

Table 15.1 **Binary Equivalents for Octal Digits**

Octal Digit	Binary Equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Hexadecimal

Hexadecimal (or *hex*) refers to a base 16 system. It uses powers of 16 and the digits 0 to 15, but because base 10 doesn't have single digits to represent the values 10 to 15, hexadecimal uses the letters A to F for that purpose. For instance, the hex number A3F (written 0xA3F in C) represents

$10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623$ (base 10)

because A represents 10 and F represents 15. In C, you can use either lowercase or uppercase letters for the additional hex digits. Therefore, you can also write 2623 as `0xa3f`.

Each hexadecimal digit corresponds to a 4-digit binary number, so two hexadecimal digits correspond exactly to an 8-bit byte. The first digit represents the upper 4 bits, and the second digit the last 4 bits. This makes hexadecimal a natural choice for representing byte values.

Table 15.2 shows the correspondence. For example, the hex value 0xC2 translates to 11000010. Going the other direction, the binary value 11010101 can be viewed as 1101 0101, which translates to 0xD5.

Table 15.2 **Decimal, Hexadecimal, and Binary Equivalents**

Decimal Digit	Hexadecimal Digit	Binary Equivalent	Decimal Digit	Hexadecimal Digit	Binary Equivalent
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Now that you've seen what bits and bytes are, let's examine what C can do with them. C has two facilities to help you manipulate bits. The first is a set of six bitwise operators that act on bits. The second is the *field* data form, which gives you access to bits within an `int`. The following discussion outlines these C features.

C's Bitwise Operators

C offers bitwise logical operators and shift operators. In the following examples, we will write out values in binary notation so that you can see what happens to the bits. In an actual program, you would use integer variables or constants written in the usual forms. For example, instead of `00011001`, you would use `25` or `031` or `0x19`. For our examples, we will use 8-bit numbers, with the bits numbered 7 to 0, left to right.

Bitwise Logical Operators

The four bitwise logical operators work on integer-type data, including `char`. They are called *bitwise* because they operate on each bit independently of the bit to the left or right. Don't confuse them with the regular logical operators (`&&`, `||`, and `!`), which operate on values as a whole.

One's Complement, or Bitwise Negation: ~

The unary operator ~ changes each 1 to a 0 and each 0 to a 1, as in the following example:

```
~(10011010)  // expression
 (01100101)  // resulting value
```

Suppose that `val` is an `unsigned char` assigned the value `2`. In binary, `2` is `00000010`. Then `~val` has the value `11111101`, or `253`. Note that the operator does not change the value of `val`, just as `3 * val` does not change the value of `val`; `val` is still `2`, but it does create a new value that can be used or assigned elsewhere:

```
newval = ~val;
printf("%d", ~val);
```

If you want to change the value of `val` to `~val`, use this simple assignment:

```
val = ~val;
```

Bitwise AND: &

The binary operator `&` produces a new value by making a bit-by-bit comparison between two operands. For each bit position, the resulting bit is 1 only if both corresponding bits in the operands are 1. (In terms of true/false, the result is true only if each of the two bit operands is true.) Therefore, the expression

```
(10010011) & (00111101)  // expression
```

evaluates to the following value:

```
(00010001)               // resulting value
```

The reason is that only bits 4 and 0 are `1` in both operands.

C also has a combined bitwise AND-assignment operator: `&=`. The statement

```
val &= 0377;
```

produces the same final result as the following:

```
val = val & 0377;
```

Bitwise OR: |

The binary operator | produces a new value by making a bit-by-bit comparison between two operands. For each bit position, the resulting bit is 1 if either of the corresponding bits in the operands is 1. (In terms of true/false, the result is true if one or the other bit operands are true or if both are true.) Therefore, the expression

```
(10010011) | (00111101)  // expression
```

evaluates to the following value:

```
(10111111)              // resulting value
```

The reason is that all bit positions but bit 6 have the value 1 in one or the other operand (or both).

C also has a combined bitwise OR-assignment operator: |=. The statement

```
val |= 0377;
```

produces the same final result as this:

```
val = val | 0377;
```

Bitwise EXCLUSIVE OR: ^

The binary operator ^ makes a bit-by-bit comparison between two operands. For each bit position, the resulting bit is 1 if one or the other (but not both) of the corresponding bits in the operands is 1. (In terms of true/false, the result is true if one or the other bit operands—but not both— is true.) Therefore, the expression

```
(10010011) ^ (00111101)  // expression
```

evaluates to the following:

```
(10101110)               // resulting value
```

Note that because bit position 0 has the value 1 in both operands, the resulting 0 bit has value 0.

C also has a combined bitwise OR-assignment operator: ^=. The statement

```
val ^= 0377;
```

produces the same final result as this:

```
val = val ^ 0377;
```

Usage: Masks

The bitwise AND operator is often used with a mask. A *mask* is a bit pattern with some bits set to on (1) and some bits to off (0). To see why a mask is called a mask, let's see what happens when a quantity is combined with a mask by using &. For example, suppose you define the symbolic constant MASK as 2 (that is, binary 00000010), with only bit number 1 being nonzero. Then the statement

```
flags = flags & MASK;
```

would cause all the bits of flags (except bit 1) to be set to 0 because any bit combined with 0 using the & operator yields 0. Bit number 1 will be left unchanged. (If the bit is 1, 1 & 1 is 1; if the bit is 0, 0 & 1 is 0.) This process is called "using a mask" because the zeros in the mask hide the corresponding bits in flags.

Extending the analogy, you can think of the 0s in the mask as being opaque and the 1s as being transparent. The expression `flags & MASK` is like covering the `flags` bit pattern with the mask; only the bits under `MASK`'s 1s are visible (see Figure 15.2).

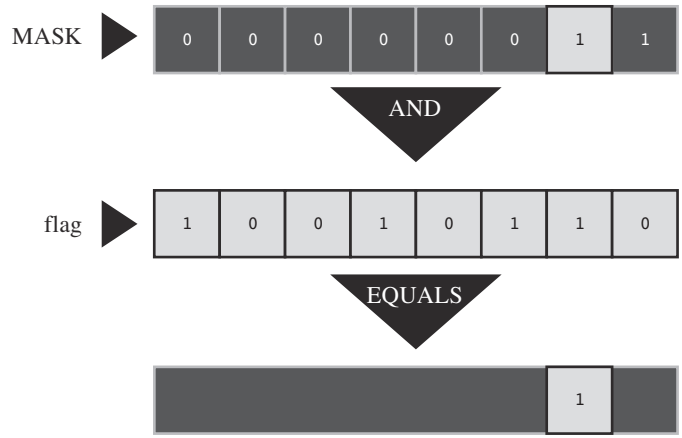


Figure 15.2 A mask.

You can shorten the code by using the AND-assignment operator, as shown here:

```
flags &= MASK;
```

One common C usage is this statement:

```
ch &= 0xff;  /* or ch &= 0377; */
```

The value `0xff`, recall, is `11111111` in binary, as is the value `0377`. This mask leaves the final 8 bits of `ch` alone and sets the rest to 0. Regardless of whether the original `ch` is 8 bits, 16 bits, or more, the final value is trimmed to something that fits into a single 8-bit byte. In this case, the mask is 8 bits wide.

Usage: Turning Bits On (Setting Bits)

Sometimes you might need to turn on particular bits in a value while leaving the remaining bits unchanged. For instance, an IBM PC controls hardware through values sent to ports. To turn on, say, the internal speaker, you might have to turn on the 1 bit while leaving the others unchanged. You can do this with the bitwise OR operator.

For example, consider the `MASK`, which has bit 1 set to 1. The statement

```
flags = flags | MASK;
```

sets bit number 1 in `flags` to 1 and leaves all the other bits unchanged. This follows because any bit combined with 0 by using the | operator is itself, and any bit combined with 1 by using the | operator is 1.

For example, suppose `flags` is `00001111` and `MASK` is `10110110`. The expression

```
flags | MASK
```

becomes

```
(00001111) | (10110110)  // expression
```

and evaluates to the following:

```
(10111111)               // resulting value
```

All the bits that are set to `1` in `MASK` are also set to `1` in the result. All the bits in `flags` that corresponded to `0` bits in `MASK` are left unchanged.

For short, you can use the bitwise OR-assignment operator:

```
flags |= MASK;
```

This, too, sets to 1 those bits in `flags` that are also on in `MASK`, leaving the other bits unchanged.

Usage: Turning Bits Off (Clearing Bits)

Just as it's useful to be able to turn on particular bits without disturbing the other bits, it's useful to be able to turn them off. Suppose you want to turn off bit 1 in the variable `flags`. Once again, `MASK` has only the 1 bit turned on. You can do this:

```
flags = flags & ~MASK;
```

Because `MASK` is all 0s except for bit 1, `~MASK` is all 1s except for bit 1. A 1 combined with any bit using `&` is that bit, so the statement leaves all the bits other than bit 1 unchanged. Also, a 0 combined with any bit using `&` is 0, so bit 1 is set to 0 regardless of its original value.

For example, suppose `flags` is `00001111` and `MASK` is `10110110`. The expression

```
flags & ~MASK
```

becomes

```
(00001111) &^ (10110110)  // expression
```

and evaluates to the following:

```
(00001001)               // resulting value
```

All the bits that are set to `1` in `MASK` are set to `0` (cleared) in the result. All the bits in `flags` that corresponded to `0` bits in `MASK` are left unchanged.

You can use this short form instead:

```
flags &= ~MASK;
```

Usage: Toggling Bits

Toggling a bit means turning it off if it is on, and turning it on if it is off. You can use the bitwise EXCLUSIVE OR operator to toggle a bit. The idea is that if `b` is a bit setting (1 or 0), then `1 ^ b` is `0` if `b` is `1` and is `1` if `b` is `0`. Also `0 ^ b` is `b`, regardless of its value. Therefore, if you combine a value with a mask by using `^`, values corresponding to 1s in the mask are toggled, and values corresponding to 0s in the mask are unaltered. To toggle bit 1 in `flags`, you can do either of the following:

```
flags = flags ^ MASK;
flags ^= MASK;
```

For example, suppose `flags` is `00001111` and `MASK` is `10110110`. The expression

```
flags ^ MASK
```

becomes

```
(00001111) ^ (10110110)  // expression
```

and evaluates to the following:

```
(10111001)               // resulting value
```

All the bits that are set to `1` in `MASK` result in the corresponding bits of `flags` being toggled. All the bits in `flags` that corresponded to `0` bits in `MASK` are left unchanged.

Usage: Checking the Value of a Bit

You've seen how to change the values of bits. Suppose, instead, that you want to check the value of a bit. For example, does `flags` have bit 1 set to 1? You shouldn't simply compare `flags` to `MASK`:

```
if (flags == MASK)
  puts("Wow!");    /* doesn't work right */
```

Even if bit 1 in `flags` is set to 1, the other bit setting in `flags` can make the comparison untrue. Instead, you must first mask the other bits in `flags` so that you compare only bit 1 of `flags` with `MASK`:

```
if ((flags & MASK) == MASK)
  puts("Wow!");
```

The bitwise operators have lower precedence than `==`, so the parentheses around `flags & MASK` are needed.

To avoid information peeking around the edges, a bit mask should be at least as wide as the value it's masking.

Bitwise Shift Operators

Now let's look at C's shift operators. The bitwise shift operators shift bits to the left or right. Again, we will write binary numbers explicitly to show the mechanics.

Left Shift: <<

The left shift operator (<<) shifts the bits of the value of the left operand to the left by the number of places given by the right operand. The vacated positions are filled with 0s, and bits moved past the end of the left operand are lost. In the following example, then, each bit is moved two places to the left:

```
(10001010) << 2  // expression
(00101000)       // resulting value
```

This operation produces a new bit value, but it doesn't change its operands. For example, suppose `stonk` is `1`. Then `stonk<<2` is `4`, but `stonk` is still `1`. You can use the left-shift assignment operator (<<=) to actually change a variable's value. This operator shifts the bit in the variable to its left by the number of places given by the right-hand value. Here's an example:

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2;   /* assigns 4 to onkoo */
stonk <<= 2;          /* changes stonk to 4 */
```

Right Shift: >>

The right-shift operator (>>) shifts the bits of the value of the left operand to the right by the number of places given by the right operand. Bits moved past the right end of the left operand are lost. For `unsigned` types, the places vacated at the left end are replaced by 0s. For signed types, the result is machine dependent. The vacated places may be filled with 0s, or they may be filled with copies of the sign (leftmost) bit:

```
(10001010) >> 2  // expression, signed value
(00100010)       // resulting value, some systems
(10001010) >> 2  // expression, signed value
(11100010)       // resulting value, other systems
```

For an unsigned value, you have the following:

```
(10001010) >> 2  // expression, unsigned value
(00100010)       // resulting value, all system
```

Each bit is moved two places to the right, and the vacated places are filled with 0s.

The right-shift assignment operator (>>=) shifts the bits in the left-hand variable to the right by the indicated number of places, as shown here:

```
int sweet = 16;
int ooosw;

ooosw = sweet >> 3;  // ooosw = 2, sweet still 16
sweet >>=3;          // sweet changed to 2
```

Usage: Bitwise Shift Operators

The bitwise shift operators can provide swift, efficient (depending on the hardware) multiplication and division by powers of 2:

`number << n`	Multiplies `number` by 2 to the `n`th power
`number >> n`	Divides `number` by 2 to the `n`th power if `number` is not negative

These shift operations are analogous to the decimal system procedure of shifting the decimal point to multiply or divide by 10.

The shift operators can also be used to extract groups of bits from larger units. Suppose, for example, you use an `unsigned long` value to represent color values, with the low-order byte holding the red intensity, the next byte holding the green intensity, and the third byte holding the blue intensity. Supposed you then wanted to store the intensity of each color in its own `unsigned char` variable. Then you could do something like this:

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

The code uses the right-shift operator to move the 8-bit color value to the low-order byte, and then uses the mask technique to assign the low-order byte to the desired variable.

Programming Example

In Chapter 9, "Functions," we used recursion to write a program to convert numbers to a binary representation. Now we'll solve the same problem by using the bitwise operators. The program in Listing 15.1 reads an integer from the keyboard and passes it and a string address to a function called `itobs()` (for *integer-to-binary string*, of course). This function then uses the bitwise operators to figure out the correct pattern of 1s and 0s to put into the string.

Listing 15.1 **The `binbit.c` Program**

```
/* binbit.c -- using bit operations to display binary */
#include <stdio.h>
#include <limits.h>  // for CHAR_BIT, # of bits per char
char * itobs(int, char *);
void show_bstr(const char *);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;

    puts("Enter integers and see them in binary.");
    puts("Non-numeric input terminates program.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number,bin_str);
        printf("%d is ", number);
        show_bstr(bin_str);
        putchar('\n');
    }
    puts("Bye!");

    return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0'; // assume ASCII or similar
    ps[size] = '\0';

    return ps;
}

/* show binary string in blocks of 4 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i])  /* not the null character */
    {
        putchar(str[i]);
```

```
        if(++i % 4 == 0 && str[i])
            putchar(' ');
    }
}
```

Listing 15.1 uses the `CHAR_BIT` macro from `limits.h`. This macro represents the number of bits in `char`. The `sizeof` operator returns the size in terms of `char`, so the expression `CHAR_BIT * sizeof(int)` is the number of bits in an `int`. The `bin_str` array has that many elements plus 1 to allow for the terminating null character.

The `itobs()` function returns the same address passed to it, so you can use the function as, say, an argument to `printf()`. The first time through the `for` loop, the function evaluates the quantity `01 & n`. The term `01` is the octal representation of a mask with all but the zero bit set to 0. Therefore, `01 & n` is just the value of the final bit in `n`. This value is `0` or `1`, but for the array, you need the *character* `'0'` or the *character* `'1'`. Adding the code for `'0'` accomplishes that conversion. (This assumes the digits are coded sequentially, as in ASCII.) The result is placed in the next-to-last element of the array. (The last element is reserved for the null character.)

By the way, you can just as well use `1 & n` as `01 & n`. Using octal 1 instead of decimal 1 just makes the mood a bit more computeresque. Perhaps `0x1 & n` is even better from that perspective.

Then the loop executes the statements `i--` and `n >>= 1`. The first statement moves to one element earlier in the array, and the second shifts the bits in `n` over one position to the right. The next time through the loop, then, the code finds the value of the new rightmost bit. The corresponding digit character is then placed in the element preceding the final digit. In this fashion, the function fills the array from right to left.

You can use `printf()` or `puts()` to display the resulting string, but Listing 15.1 defines the `show_bstr()` function, which breaks up the bits into groups of four to make the string easier to read.

Here is a sample run:

```
Enter integers and see them in binary.
Non-numeric input terminates program.
7
7 is 0000 0000 0000 0000 0000 0000 0000 0111
2013
2013 is 0000 0000 0000 0000 0000 0111 1101 1101
-1
-1 is 1111 1111 1111 1111 1111 1111 1111 1111
32123
32123 is 0000 0000 0000 0000 0111 1101 0111 1011
q
Bye!
```

Another Example

Let's work through one more example. The goal this time is to write a function that inverts the last `n` bits in a value, with both `n` and the value being function arguments.

The ~ operator inverts bits, but it inverts all the bits in a byte, not just a select few. However, the ^ operator (EXCLUSIVE OR), as you have seen, can be used to toggle individual bits. Suppose you create a mask with the last `n` bits set to 1 and the remaining bits set to 0. Then applying ^ to that mask and a value toggles, or *inverts*, the last `n` bits, leaving the other bits unchanged. That's the approach used here:

```
int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }
    return num ^ mask;
}
```

The `while` loop creates the mask. Initially, `mask` has all its bits set to `0`. The first pass through the loop sets bit 0 to `1` and then increases the value of `bitval` to `2`; that is, it sets bit 0 to `0` and bit 1 to `1`. The next pass through then sets bit 1 of `mask` to `1`, and so on. Finally, the `num ^ mask` operation produces the desired result.

To test the function, you can slip it into the preceding program, as shown in Listing 15.2.

Listing 15.2 **The `invert4.c` Program**

```
/* invert4.c -- using bit operations to display binary */
#include <stdio.h>
#include <limits.h>
char * itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);

int main(void)
{
   char bin_str[CHAR_BIT * sizeof(int) + 1];

    int number;

    puts("Enter integers and see them in binary.");
    puts("Non-numeric input terminates program.");
```

```
    while (scanf("%d", &number) == 1)
    {
        itobs(number,bin_str);
        printf("%d is\n", number);
        show_bstr(bin_str);
        putchar('\n');
        number = invert_end(number, 4);
        printf("Inverting the last 4 bits gives\n");
        show_bstr(itobs(number,bin_str));
        putchar('\n');
    }
    puts("Bye!");

    return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}

/* show binary string in blocks of 4 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i])  /* not the null character */
    {
        putchar(str[i]);
        if(++i % 4 == 0 && str[i])
            putchar(' ');
    }
}

int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits-- > 0)
```

```
    {
        mask |= bitval;
        bitval <<= 1;
    }

    return num ^ mask;
}
```

Here's a sample run:

```
Enter integers and see them in binary.
Non-numeric input terminates program.
7
7 is
0000 0000 0000 0000 0000 0000 0000 0111
Inverting the last 4 bits gives
0000 0000 0000 0000 0000 0000 0000 1000
12541
12541 is
0000 0000 0000 0000 0011 0000 1111 1101
Inverting the last 4 bits gives
0000 0000 0000 0000 0011 0000 1111 0010
q
Bye!
```

Bit Fields

The second method of manipulating bits is to use a *bit field*, which is just a set of neighboring bits within a `signed int` or an `unsigned int`. (C99 and C11 additionally allow type `_Bool` bit fields.) A bit field is set up with a structure declaration that labels each field and determines its width. For example, the following declaration sets up four 1-bit fields:

```
struct   {
    unsigned int autfd   : 1;
    unsigned int bldfc   : 1;
    unsigned int undln   : 1;
    unsigned int itals   : 1;
} prnt;
```

This definition causes `prnt` to contain four 1-bit fields. Now you can use the usual structure membership operator to assign values to individual fields:

```
prnt.itals = 0;
prnt.undln = 1;
```

Because each of these particular fields is just 1 bit, `1` and `0` are the only values you can use for assignment. The variable `prnt` is stored in an `int`-sized memory cell, but only 4 bits are used in this example.

Structures with bit fields provide a handy way to keep track of settings. Many settings, such as boldface and italics for fonts, are simply a matter specifying one of two choices, such as on or off, yes or no, or true or false. There's no need to use a whole variable when all you need is a single bit. A structure with bit fields allows you to store several settings in a single unit.

Sometimes there are more than two choices for a setting, so you need more than a single bit to represent all the choices. That's not a problem because fields aren't limited to 1-bit sizes. You can also do this:

```
struct {
    unsigned int code1 : 2;
    unsigned int code2 : 2;
    unsigned int code3 : 8;
} prcode;
```

This code creates two 2-bit fields and one 8-bit field. You can now make assignments such as the following:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

Just make sure the value doesn't exceed the capacity of the field.

What if the total number of bits you declare exceeds the size of an `unsigned int`? Then the next `unsigned int` storage location is used. A single field is not allowed to overlap the boundary between two `unsigned int`s. The compiler automatically shifts an overlapping field definition so that the field is aligned with the `unsigned int` boundary. When this occurs, it leaves an unnamed hole in the first `unsigned int`.

You can "pad" a field structure with unnamed holes by using unnamed field widths. Using an unnamed field width of 0 forces the next field to align with the next integer:

```
struct {
    unsigned int field1 : 1;
    unsigned int        : 2;
    unsigned int field2 : 1;
    unsigned int        : 0;
    unsigned int field3 : 1;
} stuff;
```

Here, there is a 2-bit gap between `stuff.field1` and `stuff.field2`, and `stuff.field3` is stored in the next `int`.

One important machine dependency is the order in which fields are placed into an `int`. On some machines, the order is left to right; on others, it is right to left. Also, machines differ in the location of boundaries between fields. For these reasons, bit fields tend not to be very portable. Typically, however, they are used for nonportable purposes, such as putting data in the exact form used by a particular hardware device.

Bit-Field Example

Often bit fields are used as a more compact way of storing data. Suppose, for example, you decided to represent the properties of an onscreen box. Let's keep the graphics simple and suppose the box has the following properties:

- The box is opaque or transparent.
- The fill color is selected from the following palette of colors: black, red, green, yellow, blue, magenta, cyan, or white.
- The border can be shown or hidden.
- The border color is selected from the same palette used for the fill color.
- The border can use one of three line styles—solid, dotted, or dashed.

You could use a separate variable or a full-sized structure member for each property, but that is a bit wasteful of bits. For example, you need only a single bit to indicate whether the box is opaque or transparent, and you need only a single bit to indicate if the border is shown or hidden. The eight possible color values can be represented by the eight possible values of a 3-bit unit, and a 2-bit unit is more than enough to represent the three possible border styles. A total of 10 bits, then, is enough to represent the possible settings for all five properties.

One possible representation of the information is to use padding to place the fill-related information in one byte and the border-related information in a second byte. The `struct box_props` declaration does this:

```
struct box_props {
    bool opaque                 : 1;
    unsigned int fill_color     : 3;
    unsigned int                : 4;
    bool show_border            : 1;
    unsigned int border_color   : 3;
    unsigned int border_style   : 2;
    unsigned int                : 2;
 };
```

The padding brings the structure up to 16 bits. Without padding, the structure would be 10 bits. Keep in mind, however, that C uses `unsigned int` as the basic layout unit for structures with bit fields. So even if the sole member of a structure is a single 1-bit field, the structure will have the same size as an `unsigned int`, which is 32 bits on our system. Also, this coding assumes that the C99 `_Bool` type is available and is aliased as `bool` in `stdbool.h`.

You can use a value of `1` for the `opaque` member to indicate that the box is opaque and a `0` value to indicate transparency. You can do the same for the `show_border` member. For colors, you can use a simple RGB (red-green-blue) representation. These are the primary colors for mixing light. A monitor blends red, green, and blue pixels to reproduce different colors. In the early days of computer color, each pixel could be either on or off, so you could use one bit to represent the intensity of each of the three binary colors. The usual order is for the left bit to represent blue intensity, the middle bit green intensity, and the right bit red intensity. Table 15.3 shows the eight possible combinations. They can be used as values for the `fill_color` and `border_color` members. Finally, you can choose to let 0, 1, and 2 represent the solid, dotted, and dashed styles; they can be used as values for the `border_style` member.

Table 15.3 **Simple Color Representation**

Bit Pattern	Decimal	Color
000	0	Black
001	1	Red
010	2	Green
011	3	Yellow
100	4	Blue
101	5	Magenta
110	6	Cyan
111	7	White

Listing 15.3 uses the `box_props` structure in a simple example. It uses `#define` to create symbolic constants for the possible member values. Note that the primary colors are represented by a single bit being on. The other colors can be represented by combinations of the primary colors. For example, magenta consists of the blue bit and the red bit being on, so it can be represented by the combination `BLUE | RED`.

Listing 15.3 **The `fields.c` Program**

```
/* fields.c -- define and use fields */
#include <stdio.h>
#include <stdbool.h>   //C99, defines bool, true, false

/* line styles     */
#define SOLID   0
#define DOTTED  1
#define DASHED  2
/* primary colors  */
#define BLUE    4
```

```
#define GREEN   2
#define RED     1
/* mixed colors    */
#define BLACK   0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

const char * colors[8] = {"black", "red", "green", "yellow",
    "blue", "magenta", "cyan", "white"};

struct box_props {
    bool opaque                 : 1;  // or unsigned int (pre C99)
    unsigned int fill_color     : 3;
    unsigned int                : 4;
    bool show_border            : 1;  // or unsigned int (pre C99)
    unsigned int border_color   : 3;
    unsigned int border_style   : 2;
    unsigned int                : 2;
};

void show_settings(const struct box_props * pb);

int main(void)
{
    /* create and initialize box_props structure */
    struct box_props box = {true, YELLOW , true, GREEN, DASHED};

    printf("Original box settings:\n");
    show_settings(&box);

    box.opaque = false;
    box.fill_color = WHITE;
    box.border_color = MAGENTA;
    box.border_style = SOLID;
    printf("\nModified box settings:\n");
    show_settings(&box);

    return 0;
}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
           pb->opaque == true ? "opaque": "transparent");
    printf("The fill color is %s.\n", colors[pb->fill_color]);
```

```
    printf("Border %s.\n",
           pb->show_border == true ? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf ("The border style is ");
    switch(pb->border_style)
    {
        case SOLID  : printf("solid.\n"); break;
        case DOTTED : printf("dotted.\n"); break;
        case DASHED : printf("dashed.\n"); break;
        default     : printf("unknown type.\n");
    }
}
```

Here is the output:

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Modified box settings:
Box is transparent.
The fill color is white.
Border shown.
The border color is magenta.
The border style is solid.
```

There are some points to note. First, you can initialize a bit-field structure by using the same syntax regular structures use:

```
struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};
```

Similarly, you can assign to bit-field members:

```
box.fill_color = WHITE;
```

Also, you can use a bit-field member as the value expression for a `switch` statement. You can even use a bit-field member as an array index:

```
printf("The fill color is %s.\n", colors[pb->fill_color]);
```

Notice that the `colors` array was defined so that each index value corresponds to a string representing the name of the color having the index value as its numeric color value. For example, an index of `1` corresponds to the string `"red"`, and the enumeration constant `red` has the value of `1`.

Bit Fields and Bitwise Operators

Bit fields and bitwise operators are two alternative approaches to the same type of programming problem. That is, often you could use either approach. For instance, the previous example used a structure the same size as `unsigned int` to hold information about a graphics box. Instead, you could use an `unsigned int` variable to hold the same information. Then, instead of using structure member notation to access different parts, you could use the bitwise operators for that purpose. Typically, this is a bit more awkward to do. Let's look at an example that takes both approaches. (The reason for taking both approaches is to illustrate the differences, not to suggest that taking both approaches simultaneously is a good idea!)

You can use a union as a means of combining the structure approach with the bitwise approach. Given the existing declaration of the `struct box_props` type, you can declare the following union:

```
union Views     /* look at data as struct or as unsigned short */
{
    struct box_props st_view;
    unsigned short us_view;
};
```

On some systems, an `unsigned int` and a `box_props` structure both occupy 16 bits of memory. On others, such as ours, `unsigned int` and `box_props` are 32 bits. In either case, with this union, you can use the `st_view` member to look at that memory as a structure or use the `us_view` member to look at the same block of memory as an `unsigned short`. Which bit fields of the structure correspond to which bits in the `unsigned short`? That depends on the implementation and the hardware. The following example assumes that structures are loaded into memory from the low-bit end to the high-bit end of a byte. That is, the first bit field in the structure goes into bit 0 of the word. (For simplicity, Figure 15.3 illustrates this idea with a 16-bit unit.)

Listing 15.4 uses the `Views` union to let you compare the bit field and bitwise approaches. In it, `box` is a `Views` union, so `box.st_view` is a `box_props` structure using bit fields, and `box.us_view` is the same data viewed as an `unsigned short`. Recall that a union can have its first member initialized, so the initialization values match the structure view. The program displays box properties using a function based on the structure view and also with a function based on the `unsigned short` view. Either approach lets you access the data, but the techniques differ. The program also uses the `itobs()` function defined earlier in this chapter to display the data as a binary string so that you can see which bits are on and which are off.

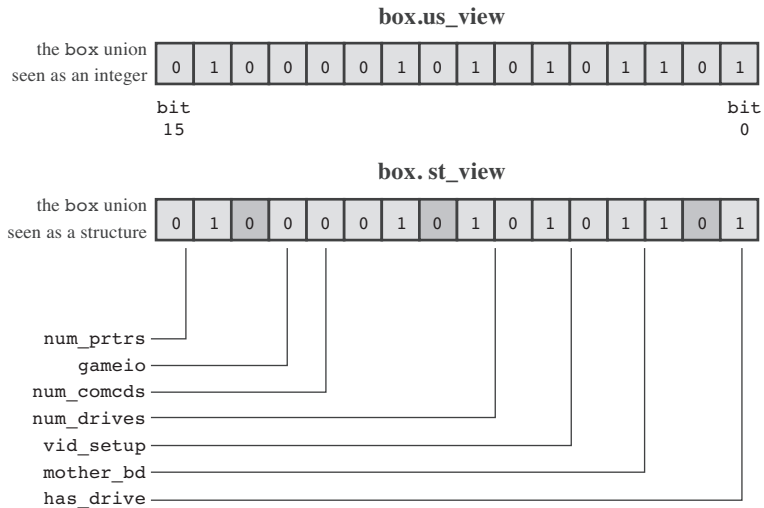


Figure 15.3 A union as an integer and as a structure.

Listing 15.4 **The `dualview.c` Program**

```
/* dualview.c -- bit fields and bitwise operators */
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
/* BIT-FIELD CONSTANTS */
/* line styles     */
#define SOLID   0
#define DOTTED  1
#define DASHED  2
/* primary colors  */
#define BLUE    4
#define GREEN   2
#define RED     1
/* mixed colors    */
#define BLACK   0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

/* BITWISE CONSTANTS   */
#define OPAQUE          0x1
#define FILL_BLUE       0x8
#define FILL_GREEN      0x4
#define FILL_RED        0x2
```

```
#define FILL_MASK       0xE
#define BORDER          0x100
#define BORDER_BLUE     0x800
#define BORDER_GREEN    0x400
#define BORDER_RED      0x200
#define BORDER_MASK     0xE00
#define B_SOLID         0
#define B_DOTTED        0x1000
#define B_DASHED        0x2000
#define STYLE_MASK      0x3000

const char * colors[8] = {"black", "red", "green", "yellow",
    "blue", "magenta", "cyan", "white"};
struct box_props {
    bool opaque                 : 1;
    unsigned int fill_color     : 3;
    unsigned int                : 4;
    bool show_border            : 1;
    unsigned int border_color   : 3;
    unsigned int border_style   : 2;
    unsigned int                : 2;
};

union Views     /* look at data as struct or as unsigned short */
{
    struct box_props st_view;
    unsigned short us_view;
};

void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(int n, char * ps);

int main(void)
{
    /* create Views object, initialize struct box view */
    union Views box = {{true, YELLOW , true, GREEN, DASHED}};
    char bin_str[8 * sizeof(unsigned int) + 1];

    printf("Original box settings:\n");
    show_settings(&box.st_view);
    printf("\nBox settings using unsigned int view:\n");
    show_settings1(box.us_view);

    printf("bits are %s\n",
           itobs(box.us_view,bin_str));
    box.us_view &= ~FILL_MASK;          /* clear fill bits */
```

```
    box.us_view |= (FILL_BLUE | FILL_GREEN); /* reset fill */
    box.us_view ^= OPAQUE;               /* toggle opacity */
    box.us_view |= BORDER_RED;           /* wrong approach */
    box.us_view &= ~STYLE_MASK;        /* clear style bits */
    box.us_view |= B_DOTTED;        /* set style to dotted */
    printf("\nModified box settings:\n");
    show_settings(&box.st_view);
    printf("\nBox settings using unsigned int view:\n");
    show_settings1(box.us_view);
    printf("bits are %s\n",
           itobs(box.us_view,bin_str));

    return 0;
}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
           pb->opaque == true ? "opaque": "transparent");
    printf("The fill color is %s.\n", colors[pb->fill_color]);
    printf("Border %s.\n",
           pb->show_border == true ? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf ("The border style is ");
    switch(pb->border_style)
    {
        case SOLID  : printf("solid.\n"); break;
        case DOTTED : printf("dotted.\n"); break;
        case DASHED : printf("dashed.\n"); break;
        default     : printf("unknown type.\n");
    }
}

void show_settings1(unsigned short us)
{
    printf("box is %s.\n",
           (us & OPAQUE) == OPAQUE? "opaque": "transparent");
    printf("The fill color is %s.\n",
           colors[(us >> 1) & 07]);
    printf("Border %s.\n",
           (us & BORDER) == BORDER? "shown" : "not shown");
    printf ("The border style is ");
    switch(us & STYLE_MASK)
    {
        case B_SOLID  : printf("solid.\n"); break;
        case B_DOTTED : printf("dotted.\n"); break;
        case B_DASHED : printf("dashed.\n"); break;
```

```
        default       : printf("unknown type.\n");
    }
    printf("The border color is %s.\n",
           colors[(us >> 9) & 07]);

}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}
```

Here is the output:

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is opaque.
The fill color is yellow.
Border shown.
The border style is dashed.
The border color is green.
bits are 00000000000000000010010100000111

Modified box settings:
Box is transparent.
The fill color is cyan.
Border shown.
The border color is yellow.
The border style is dotted.

Box settings using unsigned int view:
box is transparent.
The fill color is cyan.
Border not shown.
```

```
The border style is dotted.
The border color is yellow.
bits are 00000000000000000001011100001100
```

There are several points to discuss. One difference between the bit-field and bitwise views is that the bitwise view needs positional information. For example, we've used `BLUE` to represent the color blue. This constant has the numerical value of `4`. But, because of the way the data is arranged in the structure, the actual bit holding the blue setting for the fill color is bit 3 (remember, numbering starts at 0—refer to Figure 15.1), and the bit holding the blue setting for the border color is bit 11. Therefore, the program defines some new constants:

```
#define FILL_BLUE       0x8
#define BORDER_BLUE     0x800
```

Here, `0x8` is the value if just bit 3 is set to 1, and `0x800` is the value if just bit 11 is set to 1. You can use the first constant to set the blue bit for the fill color and the second constant to set the blue bit for the border color. Using hexadecimal notation makes it easier to see which bits are involved. Recall that each hexadecimal digit represents four bits. Thus, `0x800` is the same bit pattern as `0x8`, but with eight 0-bits tagged on. This relationship is much less obvious with 2048 and 8, the base 10 equivalents.

If the values are powers of two, you can use the left-shift operator to supply values. For example, you could replace the last `#define` statements with these:

```
#define FILL_BLUE       1<<3
#define BORDER_BLUE     1<<11
```

Here, the second operand is the power to be used with 2. That is, `0x8` is 2^3 and `0x800` is 2^{11}. Equivalently, the expression `1<<n` is the value of an integer with just the `n`th bit set to `1`. Expressions such as `1<<11` are constant expressions and are evaluated at compile time.

You can use an enumeration instead of `#define` to create symbolic constants. For example, you can do this:

```
enum { OPAQUE = 0x1, FILL_BLUE = 0x8, FILL_GREEN = 0x4, FILL_RED = 0x2,
       FILL_MASK = 0xE, BORDER = 0x100, BORDER_BLUE = 0x800,
       BORDER_GREEN = 0x400, BORDER_RED = 0x200, BORDER_MASK = 0xE00,
       B_DOTTED = 0x1000, B_DASHED = 0x2000, STYLE_MASK = 0x3000};
```

If you don't intend to create enumerated variables, you don't need to use a tag in the declaration.

Note that using bitwise operators to change settings is more complicated. For example, consider setting the fill color to cyan. It is not enough just to turn the blue bit and the green bit on:

```
box.us_view |= (FILL_BLUE | FILL_GREEN); /* reset fill */
```

The problem is that the color also depends on the red bit setting. If that bit is already set (as it is for the color yellow), this code leaves the red bit set and sets the blue and green bits,

resulting in the color white. The simplest way around this problem is to turn all the color bits off first, before setting the new values. That is why the program uses the following code:

```
box.us_view &= ~FILL_MASK;          /* clear fill bits */
box.us_view |= (FILL_BLUE | FILL_GREEN); /* reset fill */
```

To show what can happen if you don't first clear the relevant bits, the program also does this:

```
box.us_view |= BORDER_RED;           /* wrong approach */
```

Because the `BORDER_GREEN` bit already was set, the resulting color is `BORDER_GREEN | BORDER_RED`, which translates to yellow.

In cases like this, the bit-field versions are simpler:

```
box.st_view.fill_color = CYAN;  /*bit-field equivalent */
```

You don't need to clear the bits first. Also, with the bit-field members, you can use the same color values for the border as for the fill, but you need to use different values (values reflecting the actual bit positions) for the bitwise operator approach.

Next, compare the following two print statements:

```
printf("The border color is %s.\n", colors[pb->border_color]);
printf("The border color is %s.\n", colors[(us >> 9) & 07]);
```

In the first statement, the expression `pb->border_color` has a value in the range 0–7, so it can be used as an index for the `colors` array. Getting the same information with bitwise operators is more complex. One approach is to use `ui >> 9` to right-shift the border-color bits to the rightmost position in the value (bits 0–2) and then combine this value with a mask of `07` so that all bits but the rightmost three are turned off. Then what is left is in the range 0–7 and can be used as an index for the `colors` array.

> **Caution**
>
> The correspondence between bit fields and bit positions is implementation dependent. For example, running Listing 15.4 on an old Macintosh PowerPC produced the following output:
>
> ```
> Original box settings:
> Box is opaque.
> The fill color is yellow.
> Border shown.
> The border color is green.
> The border style is dashed.
>
> Box settings using unsigned int view:
> box is transparent.
> The fill color is black.
> Border not shown.
> The border style is solid.
> The border color is black.
> ```

```
bits are 10110000101010000000000000000000

Modified box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is opaque.
The fill color is cyan.
Border shown.
The border style is dotted.
The border color is red.
bits are 10110000101010000001001000001101
```

The code changed the same bits as before, but the Macintosh PowerPC loads the structure into memory differently. In particular, it loads the first bit field into the highest-order bit instead of the lowest-order bit. So the structure representation winds up in the first 16 bits (and in different order from the PC version) whereas the `unsigned int` representation winds up in the last 16 bits. Therefore, the assumptions that Listing 15.4 makes about the location of bits is incorrect for the Macintosh, and using bitwise operators to change the opacity and fill color settings alters the wrong bits.

Alignment Features (C11)

C11's alignment features are more in the nature of byte fiddling than bit fiddling, but they also represent C's capability to relate to hardware matters. Alignment, in this context, refers to how objects are positioned in memory. For example, for maximum efficiency, a system might require a type `double` value to be stored at a memory address divisible by four but allow a `char` to stored at any address. For most programmers most of the time, alignment isn't a concern. But some situations may benefit from alignment control, for example, transferring data from one hardware location to another or invoking instructions that operate upon multiple data items simultaneously.

The `_Alignof` operator yields the alignment requirement of a type. It's used by following the keyword `_Alignof` with the parenthesized type:

```
size_t d_align = _Alignof(float);
```

A value of, say, `4` for `d_align` means `float` objects have an alignment requirement of 4. That means that 4 is the number of bytes between consecutive addresses for storing values of that type. In general, alignment values should be a non-negative integer power of two. Bigger

alignment values are termed *stricter* or *stronger* than smaller ones, while smaller ones are termed *weaker*.

You can use the `_Alignas` specifier to request a specific alignment for a variable or type. But you shouldn't request an alignment weaker than the fundamental alignment for the type. For instance, if the alignment requirement for `float` is 4, don't ask for an alignment value of 1 or 2. This specifier is used as part of a declaration, and it's followed by parentheses containing either an alignment value or a type:

```
_Alignas(double) char c1;
_Alignas(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

> **Note**
>
> At the time of writing, Clang (version 3.2) required the `_Alignas(`*`type`*`)` specifier to follow the type specifier, as in the third line in the preceding example. But GCC 4.7.3 recognizes both orderings, as does the subsequent version (3.3) of Clang.

Listing 15.5 provides a short example of `_Alignas` and `_Alignof`.

Listing 15.5 **The `align.c` Program**

```
//  align.c -- using _Alignof and _Alignas  (C11)

#include <stdio.h>
int main(void)
{
    double dx;
    char ca;
    char cx;
    double dz;
    char cb;
    char _Alignas(double) cz;

    printf("char alignment:   %zd\n", _Alignof(char));
    printf("double alignment: %zd\n", _Alignof(double));
    printf("&dx: %p\n", &dx);
    printf("&ca: %p\n", &ca);
    printf("&cx: %p\n", &cx);
    printf("&dz: %p\n", &dz);
    printf("&cb: %p\n", &cb);
    printf("&cz: %p\n", &cz);

    return 0;
}
```

Here is a sample output:

```
char alignment:   1
double alignment: 8
&dx: 0x7fff5fbff660
&ca: 0x7fff5fbff65f
&cx: 0x7fff5fbff65e
&dz: 0x7fff5fbff650
&cb: 0x7fff5fbff64f
&cz: 0x7fff5fbff648
```

On our system, the alignment value of 8 for `double` implies that type aligns with addresses divisible by 8. Hexadecimal addresses ending in 0 or 8 are divisible by 8, and those were the sort of addresses used for the two `double` variables and the `char` variable `cz`, which was given the `double` alignment value. Because the alignment value for `char` was 1, the compiler could use any address for the regular `char` variables.

Including the `stdalign.h` header file allows you to use `alignas` and `alignof` for `_Alignas` and `_Alignof`. This matches the C++ keywords.

C11 also brings alignment capability for allocated memory by adding a new memory allocation function to the `stdlib.h` library. It has this prototype:

```
void *aligned_alloc(size_t alignment, size_t size);
```

The first parameter specifies the alignment required, and the second parameter requests the number of bytes required; it should be a multiple of the first parameter. As with the other memory allocation functions, use `free()` to release the memory once you are done with it.

Key Concepts

One of the features that sets C apart from most high-level languages is its ability to access individual bits in an integer. This often is the key to interfacing with hardware devices and with operating systems.

C has two main facilities for accessing bits. One is the family of bitwise operators, and the other is the ability to create bit fields in a structure.

C11 adds the capability to inspect the memory alignment requirement and to request stricter requirements.

Typically, but not always, programs using these features are tied to particular hardware platforms or operating systems and aren't intended to be portable.

Summary

Computing hardware is closely tied to the binary number system because the 1s and 0s of binary numbers can be used to represent the on and off states of bits in computer memory and registers. Although C does not allow you to write integers in binary form, it does recognize the related octal and hexadecimal notations. Just as each binary digit represents 1 bit, each octal digit represents 3 bits, and each hexadecimal digit represents 4 bits. This relationship makes it relatively simple to convert binary numbers to octal or hexadecimal form.

C features several bitwise operators, so called because they operate independently on each bit within a value. The bitwise negation operator (~) inverts each bit in its operand, converting 1s to 0s, and vice versa. The bitwise AND operator (`&`) forms a value from two operands. Each bit in the value is set to 1 if both corresponding bits in the operands are 1. Otherwise, the bit is set to 0. The bitwise OR operator (|) also forms a value from two operands. Each bit in the value is set to 1 if either or both corresponding bits in the operands are 1; otherwise, the bit is set to 0. The bitwise EXCLUSIVE OR operator (^) acts similarly, except that the resulting bit is set to 1 only if one or the other, but not both, of the corresponding bits in the operands is 1.

C also has left-shift (<<) and right-shift (>>) operators. Each produces a value formed by shifting the bits in a pattern the indicated number of bits to the left or right. For the left-shift operator, the vacated bits are set to 0. For the right-shift operator, the vacated bits are set to 0 if the value is `unsigned`. The behavior of the right-shift operator is implementation dependent for `signed` values.

You can use bit fields in a structure to address individual bits or groups of bits in a value. The details are implementation independent.

You can use `_Alignas` to impose alignment requirements on data storage.

These bit tools help C programs deal with hardware matters, so they most often appear in implementation-dependent contexts.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. Convert the following decimal values to binary:
   a. 3
   b. 13
   c. 59
   d. 119

2. Convert the following binary values to decimal, octal, and hexadecimal:
   a. 00010101
   b. 01010101
   c. 01001100
   d. 10011101

3. Evaluate the following expressions; assume each value is 8 bits:
   a. `~3`
   b. `3 & 6`
   c. `3 | 6`
   d. `1 | 6`
   e. `3 ^ 6`
   f. `7 >> 1`
   g. `7 << 2`

4. Evaluate the following expressions; assume each value is 8 bits:
   a. `~0`
   b. `!0`
   c. `2 & 4`
   d. `2 && 4`
   e. `2 | 4`
   f. `2 || 4`
   g. `5 << 3`

5. Because the ASCII code uses only the final 7 bits, sometimes it is desirable to mask off the other bits. What's the appropriate mask in binary? In decimal? In octal? In hexadecimal?

6. In Listing 15.2, you can replace

```
while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }
```

with

```
while (bits-- > 0)
    {
        mask += bitval;
        bitval *= 2;
    }
```

and the program still works. Does this mean the operation `*= 2` is equivalent to `<<= 1`? What about `|=` and `+=`?

7. **a.** The Tinkerbell computer has a hardware byte that can be read into a program. This byte contains the following information:

Bit(s)	Meaning
0–1	Number of 1.4MB floppy drives
2	Not used
3–4	Number of CD-ROM drives
5	Not used
6–7	Number of hard drives

Like the IBM PC, the Tinkerbell fills in structure bit fields from right to left. Create a bit-field template suitable for holding the information.

b. The Klinkerbell, a near Tinkerbell clone, fills in structures from left to right. Create the corresponding bit-field template for the Klinkerbell.

Programming Exercises

1. Write a function that converts a binary string to a numeric value. That is, if you have

```
char * pbin = "01001001";
```

you can pass `pbin` as an argument to the function and have the function return an `int` value of `25`.

2. Write a program that reads two binary strings as command-line arguments and prints the results of applying the `~` operator to each number and the results of applying the `&`, `|`, and `^` operators to the pair. Show the results as binary strings. (If you don't have a command-line environment available, have the program read the strings interactively.)

3. Write a function that takes an `int` argument and returns the number of "on" bits in the argument. Test the function in a program.

4. Write a function that takes two `int` arguments: a value and a bit position. Have the function return 1 if that particular bit position is 1, and have it return 0 otherwise. Test the function in a program.

5. Write a function that rotates the bits of an `unsigned int` by a specified number of bits to the left. For instance, `rotate_l(x,4)` would move the bits in `x` four places to the left, and the bits lost from the left end would reappear at the right end. That is, the bit moved out of the high-order position is placed in the low-order position. Test the function in a program.

6. Design a bit-field structure that holds the following information:

   Font ID: A number in the range 0–255

   Font Size: A number in the range 0–127

   Alignment: A number in the range 0–2 represented the choices Left, Center, and Right

   Bold: Off (0) or on (1)

   Italic: Off (0) or on (1)

   Underline: Off (0) or on (1)

   Use this structure in a program that displays the font parameters and uses a looped menu to let the user change parameters. For example, a sample run might look like this:

```
  ID SIZE ALIGNMENT   B   I   U
   1   12   left     off off off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
s
Enter font size (0-127): 36

  ID SIZE ALIGNMENT   B   I   U
   1   36   left     off off off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
a
Select alignment:
l)left   c)center   r)right
r

  ID SIZE ALIGNMENT   B   I   U
   1   36  right     off off off
```

```
f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
i

  ID SIZE ALIGNMENT   B   I   U
   1   36  right     off  on off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
q
Bye!
```

The program should use the `&` operator and suitable masks to ensure that the ID and size entries are converted to the specified range.

7. Write a program with the same behavior as described in exercise 6, but use an `unsigned long` variable to hold the font information and use the bitwise operators instead of bit members to manage the information.