```
Eight of Diamonds       Ace of Hearts
Eight of Clubs          Five of Spades
Seven of Hearts       Deuce of Diamonds
  Ace of Clubs          Ten of Diamonds
Deuce of Spades         Six of Diamonds
Seven of Spades       Deuce of Clubs
 Jack of Clubs          Ten of Spades
 King of Hearts       Jack of Diamonds
Three of Hearts       Three of Diamonds
Three of Clubs         Nine of Clubs
  Ten of Hearts       Deuce of Hearts
  Ten of Clubs        Seven of Diamonds
  Six of Clubs        Queen of Spades
  Six of Hearts       Three of Spades
 Nine of Diamonds       Ace of Diamonds
 Jack of Spades        Five of Clubs
 King of Diamonds      Seven of Clubs
 Nine of Spades         Four of Hearts
  Six of Spades       Eight of Spades
Queen of Diamonds       Five of Diamonds
  Ace of Spades        Nine of Hearts
 King of Clubs          Five of Hearts
 King of Spades         Four of Diamonds
Queen of Hearts       Eight of Hearts
 Four of Spades         Jack of Hearts
 Four of Clubs        Queen of Clubs
```

Fig. 16.3 Output for the high-performance card shuffling and dealing simulation.

## 16.7 Bitwise Operators

C++ provides extensive bit manipulation capabilities for programmers who need to get down to the so-called "bits-and-bytes" level. Operating systems, test equipment software, networking software, and many other kinds of software require that the programmer communicate "directly with the hardware." In this and the next several sections, we discuss C++'s bit manipulation capabilities. We introduce each of C++'s many bitwise operators and we discuss how to save memory by using bit fields.

All data is represented internally by computers as sequences of bits. Each bit can assume the value `0` or the value `1`. On most systems, a sequence of 8 bits forms a byte—the standard storage unit for a variable of type `char`. Other data types are stored in larger numbers of bytes. The bitwise operators are used to manipulate the bits of integral operands (`char`, `short`, `int`, and `long`; both `signed` and `unsigned`). Unsigned integers are normally used with the bitwise operators.

*Portability Tip 16.3*

*Bitwise data manipulations are machine dependent.*

Note that the bitwise operator discussions in this section show the binary representations of the integer operands. For a detailed explanation of the binary (also called base 2) number system see the appendix, "Number Systems". Also, the programs in Sections 16.7 and 16.8 were tested on a PC compatible using Borland C++. This system uses 16-bit (2-byte) integers. Because of the machine-dependent nature of bitwise manipulations, these programs may not work on your system.

The bitwise operators are: *bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>),* and *complement (~).* (Note that we have been using `&`, `<<`, and `>>` for other purposes. This is a classic example of operator overloading.) The bitwise AND, bitwise inclusive OR, and bitwise exclusive OR operators compare their two operands bit-by-bit. The bitwise AND operator sets each bit in the result to 1 if the corresponding bit in both operands is 1. The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1. The left shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The right shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. The bitwise complement operator sets all `0` bits in its operand to `1` in the result and sets all `1` bits to `0` in the result. Detailed discussions of each bitwise operator appear in the following examples. The bitwise operators are summarized in Fig. 16.4.

When using the bitwise operators, it is useful to print values in their binary representation to illustrate the precise effects of these operators. The program of Fig. 16.5 prints an **`unsigned`** integer in its binary representation in groups of eight bits each. Function **`displayBits`** uses the bitwise AND operator to combine variable **`value`** with variable **`displayMask`**. Often, the bitwise AND operator is used with an operand called a *mask*—an integer value with specific bits set to `1`. Masks are used to hide some bits in a value while selecting other bits. In **`displayBits`**, mask variable **`displayMask`** is assigned the value `1 << 15` (`10000000 00000000`). The left shift operator shifts the value `1` from the low order (rightmost) bit to the high order (leftmost) bit in **`displayMask`**, and fills in `0` bits from the right. The statement

| Operator | Name | Description |
|---|---|---|
| `&` | bitwise AND | The bits in the result are set to `1` if the corresponding bits in the two operands are both `1`. |
| `\|` | bitwise inclusive OR | The bits in the result are set to `1` if at least one of the corresponding bits in the two operands is `1`. |
| `^` | bitwise exclusive OR | The bits in the result are set to `1` if exactly one of the corresponding bits in the two operands is `1`. |
| `<<` | left shift | Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with `0` bits. |

Fig. 16.4 The bitwise operators (part 1 of 2).

| Operator | Name | Description |
|---|---|---|
| >> | right shift with sign extension | Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent. |
| ~ | one's complement | All 0 bits are set to 1 and all 1 bits are set to 0. |

Fig. 16.4 The bitwise operators (part 2 of 2).

```
// Fig. 16.5: fig16_05.cpp
// Printing an unsigned integer in bits
#include <iostream.h>
#include <iomanip.h>

void displayBits( unsigned );

int main()
{
   unsigned x;

   cout << "Enter an unsigned integer: ";
   cin >> x;
   displayBits( x );
   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 15;

   cout << setw( 7 ) << value << " = ";

   for ( c = 1; c <= 16; c++ ) {
      cout << ( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         cout << ' ';
   }

   cout << endl;
}
```

```
Enter an unsigned integer: 65000
  65000 = 11111101 11101000
```

Fig. 16.5 Printing an unsigned integer in bits.

```
cout << ( value & displayMask ? '1' : '0' );
```

determines whether a `1` or a `0` should be printed for the current leftmost bit of variable `value`. Assume variable `value` contains `65000 (11111101 11101000)`. When `value` and `displayMask` are combined using `&`, all the bits except the high order bit in variable `value` are "masked off" (hidden) because any bit "ANDed" with `0` yields `0`. If the leftmost bit is `1`, `value & displayMask` evaluates to `10000000 00000000` (which is interpreted as `true`, and `1` is printed—otherwise, `0` is printed. Variable `value` is then left shifted one bit by the expression `value <<= 1` (this is equivalent to the assignment `value = value << 1`). These steps are repeated for each bit variable `value`. Figure 16.6 summarizes the results of combining two bits with the bitwise AND operator.

*Common Programming Error 16.5*

*Using the logical AND operator (&&) for the bitwise AND operator (&) and vice versa.*

The program of Fig. 16.7 demonstrates the use of the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator, and the bitwise complement operator. The program uses function `displayBits` to print the `unsigned` integer values. The output is shown in Fig. 16.8.

| Bit 1 | Bit 2 | Bit 1 & Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Fig. 16.6 Results of combining two bits with the bitwise AND operator (&).

```
1  // Fig. 16.7: fig16_07.cpp
2  // Using the bitwise AND, bitwise inclusive OR, bitwise
3  // exclusive OR, and bitwise complement operators.
4  #include <iostream.h>
5  #include <iomanip.h>
6
7  void displayBits( unsigned );
8
9  int main()
10 {
11    unsigned number1, number2, mask, setBits;
12
13    number1 = 65535;
14    mask = 1;
```

Fig. 16.7 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and bitwise complement operators (part 1 of 2).

```
   cout << "The result of combining the following\n";
   displayBits( number1 );
   displayBits( mask );
   cout << "using the bitwise AND operator & is\n";
   displayBits( number1 & mask );

   number1 = 15;
   setBits = 241;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( setBits );
   cout << "using the bitwise inclusive OR operator | is\n";
   displayBits( number1 | setBits );

   number1 = 139;
   number2 = 199;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( number2 );
   cout << "using the bitwise exclusive OR operator ^ is\n";
   displayBits( number1 ^ number2 );

   number1 = 21845;
   cout << "\nThe one's complement of\n";
   displayBits( number1 );
   cout << "is" << endl;
   displayBits( ~number1 );

   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 15;

   cout << setw( 7 ) << value << " = ";

   for ( c = 1; c <= 16; c++ ) {
      cout << ( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         cout << ' ';
   }

   cout << endl;
}
```

Fig. 16.7 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and bitwise complement operators (part 2 of 2).

```
The result of combining the following
  65535 = 11111111 11111111
      1 = 00000000 00000001
using the bitwise AND operator & is
      1 = 00000000 00000001

The result of combining the following
     15 = 00000000 00001111
    241 = 00000000 11110001
using the bitwise inclusive OR operator | is
    255 = 00000000 11111111

The result of combining the following
    139 = 00000000 10001011
    199 = 00000000 11000111
using the bitwise exclusive OR operator ^ is
     76 = 00000000 01001100

The one's complement of
  21845 = 01010101 01010101
is
  43690 = 10101010 10101010
```

**Fig. 16.8** Output for the program of Fig. 16.7.

In Fig. 16.7, variable **mask** is assigned the value **1** (**00000000 00000001**), and variable **number1** is assigned value **65535** (**11111111 11111111**). When **mask** and **number1** are combined using the bitwise AND operator (**&**) in the expression **number1 & mask**, the result is **00000000 00000001**. All the bits except the low order bit in variable **number1** are "masked off" (hidden) by "ANDing" with variable **mask**.

The bitwise inclusive OR operator is used to set specific bits to 1 in an operand. In Fig. 16.7, variable **number1** is assigned **15** (**00000000 00001111**), and variable **setBits** is assigned **241** (**00000000 11110001**). When **number1** and **setBits** are combined using the bitwise OR operator in the expression **number1 | setBits**, the result is **255** (**00000000 11111111**). Figure 16.9 summarizes the results of combining two bits with the bitwise inclusive OR operator.

***Common Programming Error 16.6***

*Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa.*

The bitwise exclusive OR operator (**^**) sets each bit in the result to 1 if *exactly* one of the corresponding bits in its two operands is 1. In Fig. 16.7, variables **number1** and **number2** are assigned the values **139** (**00000000 10001011**) and **199** (**00000000 11000111**). When these variables are combined with the exclusive OR operator in the expression **number1 ^ number2**, the result is **00000000 01001100**. Figure 16.10 summarizes the results of combining two bits with the bitwise exclusive OR operator.

| Bit 1 | Bit 2 | Bit 1 \| Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Fig. 16.9 Results of combining two bits with the bitwise inclusive OR operator (|).

| Bit 1 | Bit 2 | Bit 1 ^ Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Fig. 16.10 Results of combining two bits with the bitwise exclusive OR operator (^).

The *bitwise* complement operator (~) sets all **1** bits in its operand to **0** in the result and sets all **0** bits to **1** in the result—otherwise referred to as "taking the *one's complement* of the value." In Fig. 16.7, variable **number1** is assigned the value **21845 (01010101 01010101)**. When the expression **~number1** is evaluated, the result is (**10101010 10101010**).

The program of Fig. 16.11 demonstrates the left shift operator (<<) and the right shift operator (>>). Function **displayBits** is used to print the **unsigned** integer values.

```
1  // Fig. 16.11: fig16_11.cpp
2  // Using the bitwise shift operators
3  #include <iostream.h>
4  #include <iomanip.h>
5
6  void displayBits( unsigned );
7
8  int main()
9  {
10    unsigned number1 = 960;
11
12    cout << "The result of left shifting\n";
13    displayBits( number1 );
14    cout << "8 bit positions using the left "
15         << "shift operator is\n";
```

Fig. 16.11 Using the bitwise shift operators (part 1 of 2).

```
   displayBits( number1 << 8 );
   cout << "\nThe result of right shifting\n";
   displayBits( number1 );
   cout << "8 bit positions using the right "
        << "shift operator is\n";
   displayBits( number1 >> 8 );
   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 15;

   cout << setw( 7 ) << value << " = ";

   for ( c = 1; c <= 16; c++ ) {
      cout << ( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         cout << ' ';
   }

   cout << endl;
}
```

```
The result of left shifting
    960 = 00000011 11000000
8 bit positions using the left shift operator << is
  49152 = 11000000 00000000

The result of right shifting
    960 = 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000011
```

Fig. 16.11 Using the bitwise shift operators (part 2 of 2).

The left shift operator (**<<**) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with **0**s; **1**s shifted off the left are lost. In the program of Fig. 16.11, variable **number1** is assigned the value **960** (**00000011 11000000**). The result of left shifting variable **number1** 8 bits in the expression **number1 << 8** is **49152** (**11000000 00000000**).

The right shift operator (**>>**) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an **unsigned** integer causes the vacated bits at the left to be replaced by 0s; **1**s shifted off the right are lost. In the program of Fig. 16.11, the result of right shifting **number1** in the expression **number1 >> 8** is **3** (**00000000 00000011**).

***Common Programming Error 16.7***

*The result of shifting a value is undefined if the right operand is negative or if the right operand is larger than the number of bits in which the left operand is stored.*

***Portability Tip 16.4***

*The result of right shifting a signed value is machine dependent. Some machines fill with zero and others use the sign bit.*

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These *bitwise assignment operators* are shown in Fig. 16.12 and are used in a similar manner to the arithmetic assignment operators introduced in Chapter 2.

Figure 16.13 shows the precedence and associativity of the various operators introduced to this point in the text. They are shown top to bottom in decreasing order of precedence.

| Bitwise assignment operators | |
|---|---|
| `&=` | Bitwise AND assignment operator. |
| `\|=` | Bitwise inclusive OR assignment operator. |
| `^=` | Bitwise exclusive OR assignment operator. |
| `<<=` | Left shift assignment operator. |
| `>>=` | Right shift with sign extension assignment operator. |

Fig. 16.12 The bitwise assignment operators.

| Operators | Associativity | Type |
|---|---|---|
| `::` (unary; right to left) `::` (binary; left to right) | left to right | highest |
| `()` `[]` `.` `->` | left to right | highest |
| `++` `--` `+` `-` `!` `delete` `sizeof` `*` `&` `new` | right to left | unary |
| `*` `/` `%` | left to right | multiplicative |
| `+` `-` | left to right | additive |
| `<<` `>>` | left to right | shifting |
| `<` `<=` `>` `>=` | left to right | relational |
| `==` `!=` | left to right | equality |
| `&` | left to right | bitwise AND |
| `^` | left to right | bitwise XOR |
| `\|` | left to right | bitwise OR |

Fig. 16.13 Operator precedence and associativity (part 1 of 2).

| Operators | Associativity | Type |
|---|---|---|
| `&&` | left to right | logical AND |
| `\|\|` | left to right | logical OR |
| `?:` | right to left | conditional |
| `= += -= *= /= %= &= \|= ^= <<= >>=` | right to left | assignment |
| `,` | left to right | comma |

Fig. 16.13 Operator precedence and associativity (part 2 of 2).

## 16.8 Bit Fields

C++ provides the ability to specify the number of bits in which an **unsigned** or **int** member of a class or a structure (or a union—see Chapter 18, "Other Topics") is stored. Such a member is referred to as a *bit field*. Bit fields enable better memory utilization by storing data in the minimum number of bits required. Bit field members *must* be declared as **int** or **unsigned**.

*Performance Tip 16.2*

*Bit fields help conserve storage.*

Consider the following structure definition:

```
struct BitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

The definition contains three **unsigned** bit fields—**face**, **suit**, and **color**—used to represent a card from a deck of 52 cards. A bit field is declared by following an **unsigned** or **int** member with a colon (**:**) and an integer constant representing the *width* of the field (i.e., the number of bits in which the member is stored). The width must be an integer constant between 0 and the total number of bits used to store an **int** on your system. Our examples were tested on a computer with 2-byte (16 bit) integers.

The preceding structure definition indicates that member **face** is stored in 4 bits, member **suit** is stored in 2 bits, and member **color** is stored in 1 bit. The number of bits is based on the desired range of values for each structure member. Member **face** stores values between **0** (Ace) and **12** (King)—4 bits can store a value between 0 and 15. Member suit stores values between **0** and **3** (**0** = Diamonds, **1** = Hearts, **2** = Clubs, **3** = Spades)—2 bits can store a value between **0** and **3**. Finally, member **color** stores either **0** (Red) or **1** (Black)—1 bit can store either **0** or **1**.

The program in Fig. 16.14 (output shown in Fig. 16.15) creates array **deck** containing 52 **struct bitCard** structures. Function **fillDeck** inserts the 52 cards in the **deck** array, and function **deal** prints the 52 cards. Notice that bit field members of structures

are accessed exactly as any other structure member. The member **color** is included as a means of indicating the card color on a system that allows color displays.

```
// Fig. 16.14: fig16_14.cpp
// Example using a bit field
#include <iostream.h>
#include <iomanip.h>

struct BitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};

void fillDeck( BitCard * );
void deal( BitCard * );

int main()
{
   BitCard deck[ 52 ];

   fillDeck( deck );
   deal( deck );
   return 0;
}

void fillDeck( BitCard *wDeck )
{
   for ( int i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = i % 13;
      wDeck[ i ].suit = i / 13;
      wDeck[ i ].color = i / 26;
   }
}

// Output cards in two column format. Cards 0-25 subscripted
// with k1 (column 1). Cards 26-51 subscripted k2 in (column 2.)
void deal( BitCard *wDeck )
{
   for ( int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      cout << "Card:" << setw( 3 ) << wDeck[ k1 ].face
           << "  Suit:" << setw( 2 ) << wDeck[ k1 ].suit
           << "  Color:" << setw( 2 ) << wDeck[ k1 ].color
           << "    " << "Card:" << setw( 3 ) << wDeck[ k2 ].face
           << "  Suit:" << setw( 2 ) << wDeck[ k2 ].suit
           << "  Color:" << setw( 2 ) << wDeck[ k2 ].color
           << endl;
   }
}
```

Fig. 16.14 Using bit fields to store a deck of cards.

```
Card:  0  Suit: 0  Color: 0   Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0   Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0   Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0   Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0   Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0   Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0   Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0   Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0   Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0   Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0   Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0   Card: 11  Suit: 2  Color: 1
Card: 12  Suit: 0  Color: 0   Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0   Card:  0  Suit: 3  Color: 1
Card:  1  Suit: 1  Color: 0   Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0   Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0   Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0   Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0   Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0   Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0   Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0   Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0   Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0   Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0   Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0   Card: 12  Suit: 3  Color: 1
```

Fig. 16.15 Output of the program in Fig. 16.14.

It is possible to specify an *unnamed bit field* in which case the field is used as *padding* in the structure. For example, the structure definition uses an unnamed 3-bit field as padding—nothing can be stored in those three bits. Member **b** (on our 2-byte word computer) is stored in another storage unit.

```
struct Example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
};
```

An *unnamed bit field with a zero width* is used to align the next bit field on a new storage unit boundary. For example, the structure definition

```
struct Example {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
};
```

uses an unnamed **0**-bit field to skip the remaining bits (as many as there are) of the storage unit in which **a** is stored, and align **b** on the next storage unit boundary.

***Portability Tip 16.5***

*Bit field manipulations are machine dependent. For example, some computers allow bit fields to cross word boundaries, whereas others do not.*

***Common Programming Error 16.8***

*Attempting to access individual bits of a bit field as if they were elements of an array. Bit fields are not "arrays of bits."*

***Common Programming Error 16.9***

*Attempting to take the address of a bit field (the & operator may not be used with bit fields because they do not have addresses).*

***Performance Tip 16.3***

*Although bit fields save space, using them can cause the compiler to generate slower-executing machine language code. This occurs because it takes extra machine language operations to access only portions of an addressable storage unit. This is one of many examples of the kinds of space-time trade-offs that occur in computer science.*

## 16.9 Character Handling Library

Most data is entered into computers as characters—including letters, digits, and various special symbols. In this section, we discuss C++'s capabilities for examining and manipulating individual characters. In the remainder of the chapter, we continue the discussion of character string manipulation that we began in Chapter 5.

The character handling library includes several functions that perform useful tests and manipulations of character data. Each function receives a character—represented as an `int`—or `EOF` as an argument. Characters are often manipulated as integers. Remember that `EOF` normally has the value –1 and some hardware architectures do not allow negative values to be stored in `char` variables. Therefore, the character handling functions manipulate characters as integers. Figure 16.16 summarizes the functions of the character handling library. When using functions from the character handling library, be sure to include the `<ctype.h>` header file.

| Prototype | Description |
|---|---|
| `int isdigit( int c )` | Returns `true` if `c` is a digit, and `false` otherwise. |
| `int isalpha( int c )` | Returns `true` if `c` is a letter, and `false` otherwise. |
| `int isalnum( int c )` | Returns `true` if `c` is a digit or a letter, and `false` otherwise. |
| `int isxdigit( int c )` | Returns `true` if `c` is a hexadecimal digit character, and `false` otherwise. (See Appendix E, "Number Systems," for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.) |
| `int islower( int c )` | Returns `true` if `c` is a lowercase letter, and `false` otherwise. |
| `int isupper( int c )` | Returns `true` if `c` is an uppercase letter; `false` otherwise. |

**Fig. 16.16** Summary of the character handling library functions (part 1 of 2).