Chapter 22
Bits, Characters, C Strings and structs

C++ How to Program, Late Objects Version, 7/e

OBJECTIVES

In this chapter you'll learn:

- To create and use `structs`.
- To pass `structs` by value and by reference.
- To use `typedef` to create aliases for previously defined data types and `structs`.
- To manipulate data with the bitwise operators and to create bit fields for storing data compactly.
- To use the functions of the character-handling library `<cctype>`.
- To use the string-conversion functions of the general-utilities library `<cstdlib>`.
- To use the string-processing functions of the string-handling library `<cstring>`.

22.1 Introduction
22.2 Structure Definitions
22.3 Initializing Structures
22.4 Using Structures with Functions
22.5 `typedef`
22.6 Example: Card Shuffling and Dealing Simulation
22.7 Bitwise Operators
22.8 Bit Fields
22.9 Character-Handling Library
22.10 Pointer-Based String Manipulation Functions
22.11 Pointer-Based String-Conversion Functions
22.12 Search Functions of the Pointer-Based String-Handling Library
22.13 Memory Functions of the Pointer-Based String-Handling Library
22.14 Wrap-Up

22.1 Introduction

- We now discuss structures and the manipulation of bits, characters and C-style strings.
- Many of the techniques we present here are included for the benefit of those who will work with legacy C and C++ code.
- C++'s designers evolved structures into the notion of a class.
- Like a class, C++ structures may contain **access specifiers**, **member functions**, **constructors** and **destructors**.
- The only differences between structures and classes in C++ is that structure members default to `public` access and class members default to `private` access when no access specifiers are used, and that structures default to `public` inheritance, whereas classes default to `private` inheritance.
- Classes have been covered thoroughly in the book, so there is really no need for us to discuss structures in detail.

22.1 Introduction (cont.)

- Our presentation of **structures** in this chapter focuses on their use in C, where structures contain only `public` data members.
- This use of structures is typical of the legacy C code and early C++ code you'll see in industry.
- We discuss how to **declare**, **initialize** and **pass structures** to functions.
- Then, we present a high-performance *card shuffling and dealing simulation* in which we use structure objects and C-style strings to represent the cards.

22.1 Introduction (cont.)

- We discuss the **bitwise operators** that allow you to access and manipulate the individual bits in bytes of data.
- We also present **bitfields**—special structures that can be used to specify the exact number of bits a variable occupies in memory.
- These **bit manipulation** techniques are common in C and C++ programs that interact directly with hardware devices that have limited memory.

22.1 Introduction (cont.)

- The chapter finishes with examples of many character and C string manipulation functions—some of which are designed to process blocks of memory as arrays of bytes.
- The detailed C string treatment in this chapter is mostly for reasons of legacy code support and because there are still remnants of C string use in C++, such as command-line arguments (Appendix F).
- New development should use C++ string objects rather than C strings.

22.2 Structure Definitions

- `struct` are aggregate data types—that is, they can be built using elements of several types including other `struct`s.
- Consider the following structure definition:
  - ```
    struct Card
    {
       string face;
       string suit;
    }; // end struct Card
    ```
  - Keyword struct introduces the definition for structure `Card`.
  - The identifier `Card` is the structure name and is used in C++ to declare variables of the structure type (in C, the type name of the preceding structure is `struct Card`).
    - Data (and possibly functions—just as with classes) declared within the braces of the structure definition are the structure's members.
    ```

22.2 Structure Definitions (cont.)

- **Members of the same structure must have unique names**, but two different structures may contain members of the same name without conflict.

- Each structure definition must end with a semicolon.

Common Programming Error 22.1

Forgetting the semicolon that terminates a structure definition is a syntax error.

22.2 Structure Definitions (cont.)

- Structure members can be variables of the fundamental data types (e.g., `int`, `double`, etc.) or aggregates, such as arrays, other structures and classes.
- Data members in a single structure definition can be of many data types.
- A structure cannot contain an instance of itself.
  - A pointer to a structure of the same type, however, can be included.
  - A structure containing a member that is a pointer to the same structure type is referred to as a self-referential structure.
  - We used a similar construct—self-referential classes—in Chapter 20, Data Structures, to build various kinds of linked data structures.

22.2 Structure Definitions (cont.)

- A structure definition does not reserve any space in memory; rather, it creates a new data type that is used to declare structure variables.
- Structure variables are declared like variables of other types.
- Variables of a given structure type can also be declared by placing a comma-separated list of the variable names **between the closing brace of the structure definition and the semicolon** that ends the structure definition.
  - The structure name is optional!
  - If a structure definition does not contain a structure name, variables of the structure type may **be declared only between the closing right brace of the structure definition and the semicolon** that terminates the structure definition.

```
struct Example
{
    char c;
    int i;
} sample1, sample2;
```

Software Engineering Observation 22.1

Provide a structure name when creating a structure type. The structure name is required for declaring new variables of the structure type later in the program, declaring parameters of the structure type and, if the structure is being used like a C++ class, specifying the name of the constructor and destructor.

22.2 Structure Definitions (cont.)

- The only valid built-in operations that may be performed on structure objects are
  - assigning one structure object to another of the same type,
  - taking the address (&) of a structure object,
  - accessing the members of a structure object (in the same manner as members of a class are accessed) and
  - using the `sizeof` operator to determine the size of a structure.

- As with classes, most operators **can be overloaded** to work with objects of a structure type.

22.2 Structure Definitions (cont.)

- **Structure members are NOT necessarily stored in consecutive bytes of memory!! (C++)**

- Sometimes there are "holes" in a structure, because some computers store specific data types only on certain memory boundaries for performance reasons, such as half-word, word or double-word boundaries.
  - A word is a standard memory unit used to store data in a computer—usually two bytes or four bytes and typically four bytes on today's popular 32-bit systems.

22.2 Structure Definitions (cont.)

- Consider the following structure definition in which structure objects `sample1` and `sample2` of type `Example` are declared:

```
struct Example
{
   char c;
   int i;
} sample1, sample2;
```

- A computer with two-byte words might require that each of the members of `Example` be aligned on a word boundary (i.e., at the beginning of a word—this is machine dependent).

22.2 Structure Definitions (cont.)

- Figure 22.1 shows a sample storage alignment for an object of type `Example` that has been assigned the character `'a'` and the integer `97` (the bit representations of the values are shown).
- If the members are stored beginning at word boundaries, there is a one-byte hole (byte `1` in the figure) in the storage for objects of type `Example`.
- The value in the one-byte hole is undefined.
- If the member values of `sample1` and `sample2` are **in fact equal**, the structure objects are not necessarily equal, because the undefined one-byte holes are not likely to contain identical values.

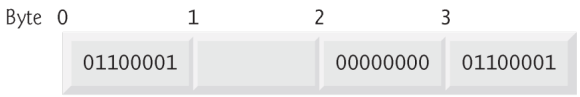

Fig. 22.1 | Possible storage alignment for a variable of type `Example`, showing an undefined area in memory.

Fig. 22.1 | Possible storage alignment for a variable of type `Example`, showing an undefined area in memory.

Common Programming Error 22.2

Comparing variables of structure types is a compilation error.

Portability Tip 22.1

Because the size of data items of a particular type is machine dependent, and because storage alignment considerations are machine dependent, so too is the representation of a structure.

22.3 Initializing Structures

- Structures can be initialized using initializer lists, like arrays.
- For example, the declaration
  - `Card oneCard = { "Three", "Hearts" };`
  - creates `Card` variable `oneCard` and initializes member `face` to `"Three"` and member `suit` to `"Hearts"`.
  - If there are fewer initializers in the list than members in the structure, the remaining members are initialized to their default values.
- Structure variables declared outside a function definition (i.e., externally) are initialized to their default values if they're not explicitly initialized in the external declaration.
- Structure variables may also be set in assignment expressions by assigning a structure variable of the same type or by assigning values to the individual data members of the structure.

22.4 Using Structures with Functions

- There are two ways to pass the information in structures to functions.
  - You can either pass the entire structure or pass the individual members of a structure.
- By default, structures are passed by value.
- Structures and their members can also be passed by reference by passing either references or pointers.
  - To pass a structure by reference, pass the address of the structure object or a reference to the structure object.
  - In Chapter 7, we stated that an array could be passed by value by using a structure.
  - To pass an array by value, create a structure (or a class) with the array as a member, then pass an object of that structure (or class) type to a function by value.
- Because structure objects are passed by value, the array member, too, is passed by value.

Performance Tip 22.1

Passing structures (and especially large structures) by reference is more efficient than passing them by value (which requires the entire structure to be copied).

22.5 typedef

- Keyword `typedef` provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter, simpler or more readable type names.
- For example, the statement
  - `typedef Card *CardPtr;`
- defines the new type name `CardPtr` as a synonym for type `Card *`.
- `typedef` simply creates a new type name that can then be used in the program as an alias for an existing type name.

Good Programming Practice 22.1

Capitalize `typedef` *names to emphasize that they are synonyms for other type names.*

Portability Tip 22.2

Synonyms for built-in data types can be created with `typedef` *to make programs more portable. For example, a program can use* `typedef` *to create alias* `Integer` *for four-byte integers.* `Integer` *can then be aliased to* `int` *on systems with four-byte integers and can be aliased to* `long int` *on systems with two-byte integers where* `long int` *values occupy four bytes. Then, you simply declare all four-byte integer variables to be of type* `Integer`.

22.6 Example: Card Shuffling and Dealing Simulation

- The card shuffling and dealing program in Figs. 22.2–22.4 is similar to the one described in Exercise 10.10.
- This program represents the deck of cards as a `vector` of structures and uses high-performance shuffling and dealing algorithms.
- The constructor (lines 12–32 of Fig. 22.3) initializes the `Card vector` in order with character strings representing Ace through King of each suit.

22.6 Example: Card Shuffling and Dealing Simulation (cont.)

- Function `shuffle` implements the high-performance shuffling algorithm.
  - The function loops through all 52 cards (subscripts 0 to 51).
  - For each card, a number between 0 and 51 is picked randomly.
  - Next, the current `Card` structure and the randomly selected `Card` structure are swapped in the `vector`.
  - A total of 52 swaps are made in a single pass of the entire `vector`, and the `vector` of `Card` structures is shuffled.
  - Because the `Card` structures were swapped in place in the `vector`, the dealing algorithm implemented in function `deal` requires only one pass of the `vector` to deal the shuffled cards.

```
1  // Fig. 22.2: DeckOfCards.h
2  // Definition of class DeckOfCards that
3  // represents a deck of playing cards.
4  #include <string>
5  #include <vector>
6  using namespace std;
7
8  // Card structure definition
9  struct Card
10 {
11    string face;
12    string suit;
13 }; // end structure Card
14
```

Fig. 22.2 | Header file for `DeckOfCards` class. (Part 1 of 2.)

```
15  // DeckOfCards class definition
16  class DeckOfCards
17  {
18  public:
19     static const int numberOfCards = 52;
20     static const int faces = 13;
21     static const int suits = 4;
22
23     DeckOfCards(); // constructor initializes deck
24     void shuffle(); // shuffles cards in deck
25     void deal() const; // deals cards in deck
26
27  private:
28     vector< Card > deck; // represents deck of cards
29  }; // end class DeckOfCards
```

Fig. 22.2 | Header file for `DeckOfCards` class. (Part 2 of 2.)

```
1  // Fig. 22.3: DeckOfCards.cpp
2  // Member-function definitions for class DeckOfCards that simulates
3  // the shuffling and dealing of a deck of playing cards.
4  #include <iostream>
5  #include <iomanip>
6  #include <cstdlib> // prototypes for rand and srand
7  #include <ctime> // prototype for time
8  #include "DeckOfCards.h" // DeckOfCards class definition
9  using namespace std;
10
11 // no-argument DeckOfCards constructor intializes deck
12 DeckOfCards::DeckOfCards()
13    : deck( numberOfCards )
14 {
15    // initialize suit array
16    static string suit[ suits ] =
17       { "Hearts", "Diamonds", "Clubs", "Spades" };
18
19    // initialize face array
20    static string face[ faces ] =
21       { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
22       "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
23
```

Fig. 22.3 | Class file for `DeckOfCards`. (Part 1 of 3.)

```
24       // set values for deck of 52 Cards
25       for ( int i = 0; i < numberOfCards; i++ )
26       {
27          deck[ i ].face = face[ i % faces ];
28          deck[ i ].suit = suit[ i / faces ];
29       } // end for
30
31       srand( time( 0 ) ); // seed random number generator
32    } // end no-argument DeckOfCards constructor
33
34    // shuffle cards in deck
35    void DeckOfCards::shuffle()
36    {
37       // shuffle cards randomly
38       for ( int i = 0; i < numberOfCards; i++ )
39       {
40          int j = rand() % numberOfCards;
41          Card temp = deck[ i ];
42          deck[ i ] = deck[ j ];
43          deck[ j ] = temp;
44       } // end for
45    } // end function shuffle
46
```

Fig. 22.3 | Class file for `DeckOfCards`. (Part 2 of 3.)

```
47  // deal cards in deck
48  void DeckOfCards::deal() const
49  {
50     // display each card's face and suit
51     for ( int i = 0; i < numberOfCards; i++ )
52        cout << right << setw( 5 ) << deck[ i ].face << " of "
53           << left << setw( 8 ) << deck[ i ].suit
54           << ( ( i + 1 ) % 2 ? '\t' : '\n' );
55  } // end function deal
```

Fig. 22.3 | Class file for `DeckOfCards`. (Part 3 of 3.)

```
1  // Fig. 22.4: fig22_04.cpp
2  // Card shuffling and dealing program.
3  #include "DeckOfCards.h" // DeckOfCards class definition
4
5  int main()
6  {
7     DeckOfCards deckOfCards; // create DeckOfCards object
8     deckOfCards.shuffle(); // shuffle the cards in the deck
9     deckOfCards.deal(); // deal the cards in the deck
10 } // end main
```

Fig. 22.4 | High-performance card shuffling and dealing simulation. (Part 1 of 2.)

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

Fig. 22.4 | High-performance card shuffling and dealing simulation. (Part 2 of 2.)

22.7 Bitwise Operators

- C++ provides extensive bit-manipulation capabilities for getting down to the so-called “bits-and-bytes” level.
- Operating systems, test-equipment software, networking software and many other kinds of software require that you communicate “directly with the hardware.”
- We introduce each of C++’s many bitwise operators, and we discuss how to save memory by using bit fields.

22.7 Bitwise Operators (cont.)

- 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.
- 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 22.3

Bitwise data manipulations are machine dependent.

22.7 Bitwise Operators (cont.)

- 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 Appendix D, Number Systems.**
- Because of the machine-dependent nature of bitwise manipulations, some of these programs might not work on your system without modification.
- The bitwise operators are: bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>) and bitwise complement (~)—also known as the one's complement.

22.7 Bitwise Operators (cont.)

- 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 either operand—but not both—is 1.

22.7 Bitwise Operators (cont.)

- 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 in its operand to 0 in the result.
- The bitwise operators are summarized in Fig. 22.5.

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 one or both 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.
>>	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.
~	bitwise complement	All 0 bits are set to 1 and all 1 bits are set to 0.

Fig. 22.5 | Bitwise operators.

22.7 Bitwise Operators (cont.)

- When using the bitwise operators, it's useful to illustrate their precise effects by printing values in their binary representation.

- The program of Fig. 22.6 prints an `unsigned` integer in its binary representation in groups of eight bits each.

```
1  // Fig. 22.6: fig22_06.cpp
2  // Printing an unsigned integer in bits.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  void displayBits( unsigned ); // prototype
8
9  int main()
10 {
11    unsigned inputValue; // integral value to print in binary
12
13    cout << "Enter an unsigned integer: ";
14    cin >> inputValue;
15    displayBits( inputValue );
16 } // end main
17
```

Fig. 22.6 | Printing an unsigned integer in bits. (Part 1 of 3.)

```
18  // display bits of an unsigned integer value
19  void displayBits( unsigned value )
20  {
21     const int SHIFT = 8 * sizeof( unsigned ) - 1;
22     const unsigned MASK = 1 << SHIFT;
23
24     cout << setw( 10 ) << value << " = ";
25
26     // display bits
27     for ( unsigned i = 1; i <= SHIFT + 1; i++ )
28     {
29        cout << ( value & MASK ? '1' : '0' );
30        value <<= 1; // shift value left by 1
31
32        if ( i % 8 == 0 ) // output a space after 8 bits
33           cout << ' ';
34     } // end for
35
36     cout << endl;
37  } // end function displayBits
```

Fig. 22.6 | Printing an unsigned integer in bits. (Part 2 of 3.)

```
Enter an unsigned integer: 65000
     65000 = 00000000 00000000 11111101 11101000
```

```
Enter an unsigned integer: 29
        29 = 00000000 00000000 00000000 00011101
```

Fig. 22.6 | Printing an unsigned integer in bits. (Part 3 of 3.)

22.7 Bitwise Operators (cont.)

- Function `displayBits` (lines 19–37) uses the bitwise AND operator to combine variable `value` with constant `MASK`.
- 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`, line 22 assigns constant `MASK` the value `1 << SHIFT`.

22.7 Bitwise Operators (cont.)

- The value of constant `SHIFT` was calculated in line 21 with the expression
  - `8 * sizeof( unsigned ) - 1`
- which multiplies the number of bytes an `unsigned` object requires in memory by `8` (the number of bits in a byte) to get the total number of bits required to store an `unsigned` object, then subtracts 1.
- The bit representation of `1 << SHIFT` on a computer that represents `unsigned` objects in four bytes of memory is
  - `10000000 00000000 00000000 00000000`
- The left-shift operator shifts the value `1` from the low-order (rightmost) bit to the high-order (leftmost) bit in `MASK`, and fills in `0` bits from the right.

22.7 Bitwise Operators (cont.)

- Line 29 prints a 1 or a 0 for the current leftmost bit of variable value.
- Assume that variable value contains 65000 (00000000 00000000 11111101 11101000).
- When value and MASK 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 & MASK evaluates to

```
00000000 00000000 11111101 11101000     (value)
10000000 00000000 00000000 00000000     (MASK)
--------------------------------
00000000 00000000 00000000 00000000     (value & MASK)
```

- which is interpreted as false, and 0 is printed.
- Then line 30 shifts variable value left by one bit with the expression value <<= 1 (i.e., value = value << 1).
- These steps are repeated for each bit variable value.

22.7 Bitwise Operators (cont.)

- Eventually, a bit with a value of 1 is shifted into the leftmost bit position, and the bit manipulation is as follows:
  -
```
11111101 11101000 00000000 00000000    (value)
10000000 00000000 00000000 00000000    (MASK)
--------------------------------------
10000000 00000000 00000000 00000000    (value & MASK)
```
- Because both left bits are 1s, the expression's result is nonzero (true) and 1 is printed.
- Figure 22.7 summarizes the results of combining two bits with the bitwise AND operator.

Common Programming Error 22.3

Using the logical AND operator (&&) for the bitwise AND operator (&) and vice versa is a logic error.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 22.7 | Results of combining two bits with the bitwise AND operator (&).

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 22.9 | 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. 22.10 | Combining two bits with the bitwise exclusive-OR operator (^).

22.7 Bitwise Operators (cont.)

- The program of Fig. 22.8 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator.
- Function `displayBits` (lines 53–71) prints the `unsigned` integer values.

```
1  // Fig. 22.8: fig22_08.cpp
2  // Bitwise AND, inclusive OR,
3  // exclusive OR and complement operators.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  void displayBits( unsigned ); // prototype
9
10 int main()
11 {
12    unsigned number1;
13    unsigned number2;
14    unsigned mask;
15    unsigned setBits;
16
17    // demonstrate bitwise &
18    number1 = 2179876355;
19    mask = 1;
20    cout << "The result of combining the following\n";
21    displayBits( number1 );
22    displayBits( mask );
```

Fig. 22.8 | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 1 of 5.)

```
23      cout << "using the bitwise AND operator & is\n";
24      displayBits( number1 & mask );
25
26      // demonstrate bitwise |
27      number1 = 15;
28      setBits = 241;
29      cout << "\nThe result of combining the following\n";
30      displayBits( number1 );
31      displayBits( setBits );
32      cout << "using the bitwise inclusive OR operator | is\n";
33      displayBits( number1 | setBits );
34
35      // demonstrate bitwise exclusive OR
36      number1 = 139;
37      number2 = 199;
38      cout << "\nThe result of combining the following\n";
39      displayBits( number1 );
40      displayBits( number2 );
41      cout << "using the bitwise exclusive OR operator ^ is\n";
42      displayBits( number1 ^ number2 );
43
```

Fig. 22.8 | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 2 of 5.)

```
44      // demonstrate bitwise complement
45      number1 = 21845;
46      cout << "\nThe one's complement of\n";
47      displayBits( number1 );
48      cout << "is" << endl;
49      displayBits( ~number1 );
50   } // end main
51
```

Fig. 22.8 | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 3 of 5.)

```
52  // display bits of an unsigned integer value
53  void displayBits( unsigned value )
54  {
55     const int SHIFT = 8 * sizeof( unsigned ) - 1;
56     const unsigned MASK = 1 << SHIFT;
57
58     cout << setw( 10 ) << value << " = ";
59
60     // display bits
61     for ( unsigned i = 1; i <= SHIFT + 1; i++ )
62     {
63        cout << ( value & MASK ? '1' : '0' );
64        value <<= 1; // shift value left by 1
65
66        if ( i % 8 == 0 ) // output a space after 8 bits
67           cout << ' ';
68     } // end for
69
70     cout << endl;
71  } // end function displayBits
```

Fig. 22.8 | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 4 of 5.)

```
The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

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

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

The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

Fig. 22.8 | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 5 of 5.)

22.7 Bitwise Operators (cont.)

- In Fig. 22.8, line 18 assigns `2179876355` (`10000001 11101110 01000110 00000011`) to variable `number1`, and line 19 assigns `1` (`00000000 00000000 00000000 00000001`) to variable `mask`.
- When `mask` and `number1` are combined using the bitwise AND operator (`&`) in the expression `number1 & mask` (line 24), the result is `00000000 00000000 00000000 00000001`.
- All the bits except the low-order bit in variable `number1` are "masked off" (hidden) by "ANDing" with constant `MASK`.

22.7 Bitwise Operators (cont.)

- The bitwise inclusive-OR operator is used to set specific bits to 1 in an operand.
- In Fig. 22.8, line 27 assigns `15` (`00000000 00000000 00000000 00001111`) to variable `number1`, and line 28 assigns `241` (`00000000 00000000 00000000 11110001`) to variable `setBits`.
- When `number1` and `setBits` are combined using the bitwise OR operator in the expression `number1 | setBits` (line 33), the result is `255` (`00000000 00000000 00000000 11111111`).
- Figure 22.9 summarizes the results of combining two bits with the bitwise inclusive-OR operator.

Common Programming Error 22.4

Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is a logic error.

22.7 Bitwise Operators (cont.)

- 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. 22.8, lines 36–37 assign variables `number1` and `number2` the values `139` (`00000000 00000000 00000000 10001011`) and `199` (`00000000 00000000 00000000 11000111`), respectively.
- When these variables are combined with the exclusive-OR operator in the expression `number1 ^ number2` (line 42), the result is `00000000 00000000 00000000 01001100`.
- Figure 22.10 summarizes the results of combining two bits with the bitwise exclusive-OR operator.

22.7 Bitwise Operators (cont.)

- 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. 22.8, line 45 assigns variable `number1` the value `21845` (`00000000 00000000 01010101 01010101`).
- When the expression `~number1` evaluates, the result is (`11111111 11111111 10101010 10101010`).
- Figure 22.11 demonstrates the left-shift operator (`<<`) and the right-shift operator (`>>`).
- Function `displayBits` (lines 27–45) prints the `unsigned` integer values.

```
1  // Fig. 22.11: fig22_11.cpp
2  // Using the bitwise shift operators.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  void displayBits( unsigned ); // prototype
8
9  int main()
10 {
11    unsigned number1 = 960;
12
13    // demonstrate bitwise left shift
14    cout << "The result of left shifting\n";
15    displayBits( number1 );
16    cout << "8 bit positions using the left-shift operator is\n";
17    displayBits( number1 << 8 );
18
19    // demonstrate bitwise right shift
20    cout << "\nThe result of right shifting\n";
21    displayBits( number1 );
22    cout << "8 bit positions using the right-shift operator is\n";
23    displayBits( number1 >> 8 );
24 } // end main
```

```
The result of left shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the left-shift operator is
    245760 = 00000000 00000011 11000000 00000000

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

Fig. 22.11 | Bitwise shift operators. (Part 1 of 3.)

```
25
26 // display bits of an unsigned integer value
27 void displayBits( unsigned value )
28 {
29    const int SHIFT = 8 * sizeof( unsigned ) - 1;
30    const unsigned MASK = 1 << SHIFT;
31
32    cout << setw( 10 ) << value << " = ";
33
34    // display bits
35    for ( unsigned i = 1; i <= SHIFT + 1; i++ )
36    {
37       cout << ( value & MASK ? '1' : '0' );
38       value <<= 1; // shift value left by 1
39
40       if ( i % 8 == 0 ) // output a space after 8 bits
41          cout << ' ';
42    } // end for
43
44    cout << endl;
45 } // end function displayBits
```

Fig. 22.11 | Bitwise shift operators. (Part 2 of 3.)

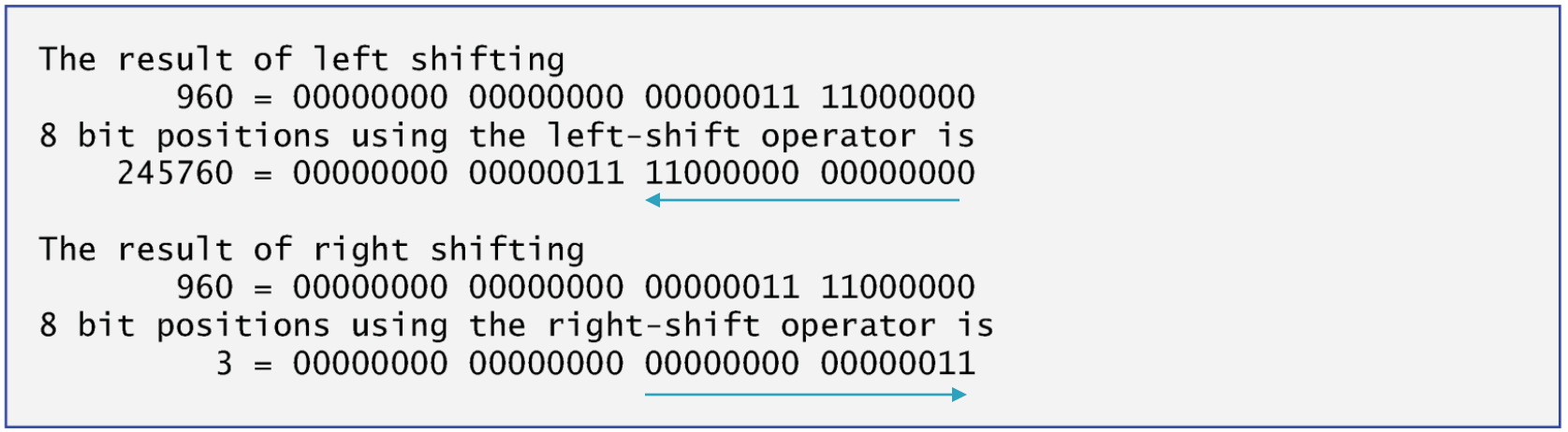

Fig. 22.11 | Bitwise shift operators. (Part 3 of 3.)

22.7 Bitwise Operators (cont.)

- 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 0s; bits shifted off the left are lost.
- In the program of Fig. 22.11, line 11 assigns variable `number1` the value `960` (`00000000 00000000 00000011 11000000`).
- The result of left-shifting variable `number1` 8 bits in the expression `number1 << 8` (line 17) is `245760` (`00000000 00000011 11000000 00000000`).

22.7 Bitwise Operators (cont.)

- 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; bits shifted off the right are lost.
- In the program of Fig. 22.11, the result of right-shifting `number1` in the expression `number1 >> 8` (line 23) is 3 (00000000 00000000 00000000 00000011).

Common Programming Error 22.5

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

Portability Tip 22.4

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

22.7 Bitwise Operators (cont.)

- Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator.
- These bitwise assignment operators are shown in Fig. 22.12; they're used in a similar manner to the arithmetic assignment operators introduced in Chapter 2.

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. 22.12 | Bitwise assignment operators.

22.7 Bitwise Operators (cont.)

- Figure 22.13 shows the precedence and associativity of the operators introduced up to this point in the text.
- They're shown top to bottom in decreasing order of precedence.

Operators	Associativity	Type
`::` (unary; right to left) `::` (binary; left to right)	left to right	highest
`()` `[]` `.` `->` `++` `--` `static_cast< type >()`	left to right	unary
`++` `--` `+` `-` `!` `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
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`	right to left	assignment
`,`	left to right	comma

Fig. 22.13 | Operator precedence and associativity.

22.8 Bit Fields

- C++ provides the ability to specify the number of bits in which an integral type or `enum` type member of a class or a structure 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 an integral or `enum` type.

Performance Tip 22.2

Bit fields help conserve storage.

22.8 Bit Fields (cont.)

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

- 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 integral type or `enum` type member with a colon (`:`) and an integer constant representing the width of the bit field (i.e., the number of bits in which the member is stored).
- The width must be an integer constant.
- The preceding structure definition indicates that member `face` is stored in 4 bits, member `suit` in 2 bits and member `color` in 1 bit.

22.8 Bit Fields (cont.)

- 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`.

22.8 Bit Fields (cont.)

- The program in Figs. 22.14–22.16 creates `vector deck` containing `BitCard` structures (line 27 of Fig. 22.14).
- The constructor inserts the 52 cards in the `deck vector`, and function `deal` prints the 52 cards.
- Notice that bit fields are accessed exactly as any other structure member is (lines 15–17 and 26–31 of Fig. 22.15).
- The member `color` is included as a means of indicating the card color.

```
1  // Fig. 22.14: DeckOfCards.h
2  // Definition of class DeckOfCards that
3  // represents a deck of playing cards.
4  #include <vector>
5  using namespace std;
6
7  // BitCard structure definition with bit fields
8  struct BitCard
9  {
10    unsigned face : 4; // 4 bits; 0-15
11    unsigned suit : 2; // 2 bits; 0-3
12    unsigned color : 1; // 1 bit; 0-1
13 }; // end struct BitCard
14
```

Fig. 22.14 | Header file for class `DeckOfCards`. (Part 1 of 2.)

```
15  // DeckOfCards class definition
16  class DeckOfCards
17  {
18  public:
19     static const int faces = 13;
20     static const int colors = 2; // black and red
21     static const int numberOfCards = 52;
22
23     DeckOfCards(); // constructor initializes deck
24     void deal(); // deals cards in deck
25
26  private:
27     vector< BitCard > deck; // represents deck of cards
28  }; // end class DeckOfCards
```

Fig. 22.14 | Header file for class `DeckOfCards`. (Part 2 of 2.)

```
1  // Fig. 22.15: DeckOfCards.cpp
2  // Member-function definitions for class DeckOfCards that simulates
3  // the shuffling and dealing of a deck of playing cards.
4  #include <iostream>
5  #include <iomanip>
6  #include "DeckOfCards.h" // DeckOfCards class definition
7  using namespace std;
8
9  // no-argument DeckOfCards constructor intializes deck
10 DeckOfCards::DeckOfCards()
11
12 {
13    for ( int i = 0; i < numberOfCards; i++ )
14    {
15       deck[ i ].face = i % faces; // faces in order 0,1,2...12
16       deck[ i ].suit = i / faces; // suits in order 0,1,2,3
17       deck[ i ].color = i / ( faces * colors ); // colors in order
18    } // end for
19 } // end no-argument DeckOfCards constructor
20
```

Fig. 22.15 | Class file for `DeckOfCards`. (Part 1 of 2.)

```
21  // deal cards in deck
22  void DeckOfCards::deal()
23  {
24     for ( int k1 = 0, k2 = k1 + numberOfCards / 2;
25        k1 < numberOfCards / 2 - 1; k1++, k2++ )
26        cout << "Card:" << setw( 3 ) << deck[ k1 ].face
27           << "  Suit:" << setw( 2 ) << deck[ k1 ].suit
28           << "  Color:" << setw( 2 ) << deck[ k1 ].color
29           << "    " << "Card:" << setw( 3 ) << deck[ k2 ].face
30           << "  Suit:" << setw( 2 ) << deck[ k2 ].suit
31           << "  Color:" << setw( 2 ) << deck[ k2 ].color << endl;
32  } // end function deal
```

Fig. 22.15 | Class file for `DeckOfCards`. (Part 2 of 2.)

```
1  // Fig. 22.16: fig22_16.cpp
2  // Card shuffling and dealing program.
3  #include "DeckOfCards.h" // DeckOfCards class definition
4
5  int main()
6  {
7     DeckOfCards deckOfCards; // create DeckOfCards object
8     deckOfCards.deal(); // deal the cards in the deck
9  } // end main
```

Fig. 22.16 | Bit fields used to store a deck of cards. (Part 1 of 2.)

```
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. 22.16 | Bit fields used to store a deck of cards. (Part 2 of 2.)

22.8 Bit Fields (cont.)

- It's 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 3 bits.
- Member b is stored in another storage unit.
  - ```
    struct Example
    {
       unsigned a : 13;
       unsigned   : 3; // align to next storage-
    unit boundary
       unsigned b : 4;
    }; // end struct Example
    ```
    ```

22.8 Bit Fields (cont.)

- 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; // align to next storage-
    unit boundary
       unsigned b : 4;
    }; // end struct Example
    ```
- 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 22.5

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

Common Programming Error 22.6

Attempting to access individual bits of a bit field with subscripting as if they were elements of an array is a compilation error. Bit fields are not "arrays of bits."

Common Programming Error 22.7

Attempting to take the address of a bit field (the & operator may not be used with bit fields because a pointer can designate only a particular byte in memory and bit fields can start in the middle of a byte) is a compilation error.

Performance Tip 22.3

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 space–time trade-offs that occur in computer science.

22.9 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 8.

22.9 Character-Handling Library (cont.)

- 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 that some hardware architectures do not allow negative values to be stored in `char` variables.
  - Therefore, the character-handling functions manipulate characters as integers.

22.9 Character-Handling Library (cont.)

- Figure 22.17 summarizes the functions of the character-handling library.
- When using functions from the character-handling library, include the `<cctype>` header file.

Prototype	Description
`int isdigit( int c )`	Returns 1 if c is a digit and 0 otherwise.
`int isalpha( int c )`	Returns 1 if c is a letter and 0 otherwise.
`int isalnum( int c )`	Returns 1 if c is a digit or a letter and 0 otherwise.
`int isxdigit( int c )`	Returns 1 if c is a hexadecimal digit character and 0 otherwise. (See Appendix D, Number Systems, for a detailed explanation of binary, octal, decimal and hexadecimal numbers.)
`int islower( int c )`	Returns 1 if c is a lowercase letter and 0 otherwise.
`int isupper( int c )`	Returns 1 if c is an uppercase letter; 0 otherwise.
`int tolower( int c )`	If c is an uppercase letter, `tolower` returns c as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`int toupper( int c )`	If c is a lowercase letter, `toupper` returns c as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c )`	Returns 1 if c is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and 0 otherwise.

Fig. 22.17 | Character-handling library functions. (Part 1 of 2.)

Prototype	Description
`int iscntrl( int c )`	Returns `1` if `c` is a control character, such as newline (`'\n'`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), vertical tab (`'\v'`), alert (`'\a'`), or backspace (`'\b'`)—and `0` otherwise.
`int ispunct( int c )`	Returns `1` if `c` is a printing character other than a space, a digit, or a letter and `0` otherwise.
`int isprint( int c )`	Returns `1` if `c` is a printing character including space (`' '`) and `0` otherwise.
`int isgraph( int c )`	Returns `1` if `c` is a printing character other than space (`' '`) and `0` otherwise.

Fig. 22.17 | Character-handling library functions. (Part 2 of 2.)

22.9 Character-Handling Library (cont.)

- Figure 22.18 demonstrates functions isdigit, isalpha, isalnum and isxdigit.
  - Function `isdigit` determines whether its argument is a digit (`0`–`9`).
  - Function `isalpha` determines whether its argument is an uppercase letter (`A-Z`) or a lowercase letter (`a`–`z`).
  - Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit.
  - Function `isxdigit` determines whether its argument is a hexadecimal digit (`A`–`F`, `a`–`f`, `0`–`9`).

```
1  // Fig. 22.18: fig22_18.cpp
2  // Character-handling functions isdigit, isalpha, isalnum and isxdigit.
3  #include <iostream>
4  #include <cctype> // character-handling function prototypes
5  using namespace std;
6
7  int main()
8  {
9     cout << "According to isdigit:\n"
10       << ( isdigit( '8' ) ? "8 is a" : "8 is not a" ) << " digit\n"
11       << ( isdigit( '#' ) ? "# is a" : "# is not a" ) << " digit\n";
12
13    cout << "\nAccording to isalpha:\n"
14       << ( isalpha( 'A' ) ? "A is a" : "A is not a" ) << " letter\n"
15       << ( isalpha( 'b' ) ? "b is a" : "b is not a" ) << " letter\n"
16       << ( isalpha( '&' ) ? "& is a" : "& is not a" ) << " letter\n"
17       << ( isalpha( '4' ) ? "4 is a" : "4 is not a" ) << " letter\n";
18
```

Fig. 22.18 | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 3.)

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter
```

```
19      cout << "\nAccording to isalnum:\n"
20         << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
21         << " digit or a letter\n"
22         << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
23         << " digit or a letter\n"
24         << ( isalnum( '#' ) ? "# is a" : "# is not a" )
25         << " digit or a letter\n";
26
27      cout << "\nAccording to isxdigit:\n"
28         << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
29         << " hexadecimal digit\n"
30         << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
31         << " hexadecimal digit\n"
32         << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
33         << " hexadecimal digit\n"
34         << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
35         << " hexadecimal digit\n"
36         << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
37         << " hexadecimal digit" << endl;
38   } // end main
```

```
According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 22.18 | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 2 of 3.)

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 22.18 | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 3 of 3.)

22.9 Character-Handling Library (cont.)

- Figure 22.18 uses the conditional operator (`?:`) with each function to determine whether the string `" is a "` or the string `" is not a "` should be printed in the output for each character tested.
- For example, line 10 indicates that if `'8'` is a digit—i.e., if `isdigit` returns a true (nonzero) value—the string `"8 is a "` is printed.
- If `'8'` is not a digit (i.e., if `isdigit` returns `0`), the string `"8 is not a "` is printed.
- Figure 22.19 demonstrates functions islower, isupper, tolower and toupper.

22.9 Character-Handling Library (cont.)

- Function `islower` determines whether its argument is a lowercase letter (`a`–`z`).
- Function `isupper` determines whether its argument is an uppercase letter (`A`–`Z`).
- Function `tolower` converts an uppercase letter to lowercase and returns the lowercase letter—if the argument is not an uppercase letter, `tolower` returns the argument value unchanged.
- Function `toupper` converts a lowercase letter to uppercase and returns the uppercase letter—if the argument is not a lowercase letter, `toupper` returns the argument value unchanged.

```
1  // Fig. 22.19: fig22_19.cpp
2  // Character-handling functions islower, isupper, tolower and toupper.
3  #include <iostream>
4  #include <cctype> // character-handling function prototypes
5  using namespace std;
6
7  int main()
8  {
9     cout << "According to islower:\n"
10       << ( islower( 'p' ) ? "p is a" : "p is not a" )
11       << " lowercase letter\n"
12       << ( islower( 'P' ) ? "P is a" : "P is not a" )
13       << " lowercase letter\n"
14       << ( islower( '5' ) ? "5 is a" : "5 is not a" )
15       << " lowercase letter\n"
16       << ( islower( '!' ) ? "! is a" : "! is not a" )
17       << " lowercase letter\n";
18
```

Fig. 22.19 | Character-handling functions `islower`, `isupper`, `tolower` and `toupper`. (Part 1 of 3.)

```
19      cout << "\nAccording to isupper:\n"
20         << ( isupper( 'D' ) ? "D is an" : "D is not an" )
21         << " uppercase letter\n"
22         << ( isupper( 'd' ) ? "d is an" : "d is not an" )
23         << " uppercase letter\n"
24         << ( isupper( '8' ) ? "8 is an" : "8 is not an" )
25         << " uppercase letter\n"
26         << ( isupper( '$' ) ? "$ is an" : "$ is not an" )
27         << " uppercase letter\n";
28
29      cout << "\nu converted to uppercase is "
30         << static_cast< char >( toupper( 'u' ) )
31         << "\n7 converted to uppercase is "
32         << static_cast< char >( toupper( '7' ) )
33         << "\n$ converted to uppercase is "
34         << static_cast< char >( toupper( '$' ) )
35         << "\nL converted to lowercase is "
36         << static_cast< char >( tolower( 'L' ) ) << endl;
37   } // end main
```

Fig. 22.19 | Character-handling functions `islower`, `isupper`, `tolower` and `toupper`. (Part 2 of 3.)

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Fig. 22.19 | Character-handling functions `islower`, `isupper`, `tolower` and `toupper`. (Part 3 of 3.)

22.9 Character-Handling Library (cont.)

- Figure 22.20 demonstrates functions isspace, iscntrl, ispunct, isprint and isgraph.
  - Function `isspace` determines whether its argument is a white-space character, such as space (`' '`), form feed (`'\f'`), newline (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`).
  - Function `iscntrl` determines whether its argument is a control character such as horizontal tab (`'\t'`), vertical tab (`'\v'`), form feed (`'\f'`), alert (`'\a'`), backspace (`'\b'`), carriage return (`'\r'`) or newline (`'\n'`).
  - Function `ispunct` determines whether its argument is a printing character other than a space, digit or letter, such as `$`, `#`, `(`, `)`, `[`, `]`, `{`, `}`, `;`, `:` or `%`.
  - Function `isprint` determines whether its argument is a character that can be displayed on the screen (including the space character).
  - Function `isgraph` tests for the same characters as `isprint`, but the space character is not included.

```
1  // Fig. 22.20: fig22_20.cpp
2  // Using functions isspace, iscntrl, ispunct, isprint, isgraph.
3  #include <iostream>
4  #include <cctype> // character-handling function prototypes
5  using namespace std;
6
7  int main()
8  {
9     cout << "According to isspace:\nNewline "
10       << ( isspace( '\n' ) ? "is a" : "is not a" )
11       << " whitespace character\nHorizontal tab "
12       << ( isspace( '\t' ) ? "is a" : "is not a" )
13       << " whitespace character\n"
14       << ( isspace( '%' ) ? "% is a" : "% is not a" )
15       << " whitespace character\n";
16
17    cout << "\nAccording to iscntrl:\nNewline "
18       << ( iscntrl( '\n' ) ? "is a" : "is not a" )
19       << " control character\n"
20       << ( iscntrl( '$' ) ? "$ is a" : "$ is not a" )
21       << " control character\n";
22
```

Fig. 22.20 | Character-handling functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 1 of 3.)

```
23      cout << "\nAccording to ispunct:\n"
24         << ( ispunct( ';' ) ? "; is a" : "; is not a" )
25         << " punctuation character\n"
26         << ( ispunct( 'Y' ) ? "Y is a" : "Y is not a" )
27         << " punctuation character\n"
28         << ( ispunct( '#' ) ? "# is a" : "# is not a" )
29         << " punctuation character\n";
30
31      cout << "\nAccording to isprint:\n"
32         << ( isprint( '$' ) ? "$ is a" : "$ is not a" )
33         << " printing character\nAlert "
34         << ( isprint( '\a' ) ? "is a" : "is not a" )
35         << " printing character\nSpace "
36         << ( isprint( ' ' ) ? "is a" : "is not a" )
37         << " printing character\n";
38
39      cout << "\nAccording to isgraph:\n"
40         << ( isgraph( 'Q' ) ? "Q is a" : "Q is not a" )
41         << " printing character other than a space\nSpace "
42         << ( isgraph( ' ' ) ? "is a" : "is not a" )
43         << " printing character other than a space" << endl;
44   } // end main
```

Fig. 22.20 | Character-handling functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 2 of 3.)

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character
Space is a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Fig. 22.20 | Character-handling functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 3 of 3.)

22.10 Pointer-Based String Manipulation Functions

- The string-handling library provides many useful functions for manipulating string data, comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces such as the separate words in a sentence) and determining the length of strings.
- This section presents some common string-manipulation functions of the string-handling library (from the C++ standard library).
- The functions are summarized in Fig. 22.21; then each is used in a live-code example.
- The prototypes for these functions are located in header file `<cstring>`.

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 );`	Copies the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string `s2` to `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.

Fig. 22.21 | String-manipulation functions of the string-handling library. (Part 1 of 3.)

Function prototype	Function description
`int strcmp( const char *s1, const char *s2 );`	Compares the string `s1` with the string `s2`. The function returns a value of zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if the `n`-character portion of `s1` is equal to, less than or greater than the corresponding `n`-character portion of `s2`, respectively.

Fig. 22.21 | String-manipulation functions of the string-handling library. (Part 2 of 3.)

Function prototype	Function description
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text. The string is broken up based on the characters contained in string `s2`. For instance, if we were to break the string `"this:is:a:string"` into tokens based on the character `':'`, the resulting tokens would be `"this"`, `"is"`, `"a"` and `"string"`. Function `strtok` returns only one token at a time—the first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned.
`size_t strlen( const char *s );`	Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

Fig. 22.21 | String-manipulation functions of the string-handling library. (Part 3 of 3.)

22.10 Pointer-Based String Manipulation Functions (cont.)

- Several functions in Fig. 22.21 contain parameters with data type `size_t`.
- This type is defined in the header file `<cstring>` to be an unsigned integral type such as `unsigned int` or `unsigned long`.

Common Programming Error 22.8

Forgetting to include the `<cstring>` *header file when using functions from the string-handling library causes compilation errors.*

22.10 Pointer-Based String Manipulation Functions (cont.)

- Function strcpy copies its second argument—a string—into its first argument—a character array that must be large enough to store the string and its terminating null character, (which is also copied).
- Function strncpy is much like `strcpy`, except that `strncpy` specifies the number of characters to be copied from the string into the array.
- Function `strncpy` does not necessarily copy the terminating null character of its second argument—a terminating null character is written only if the number of characters to be copied is at least one more than the length of the string.

Common Programming Error 22.9

When using `strncpy`*, the terminating null character of the second argument (a* `char *` *string) will not be copied if the number of characters specified by* `strncpy`*'s third argument is not greater than the second argument's length. In that case, a fatal error may occur if you do not manually terminate the resulting* `char *` *string with a null character.*

22.10 Pointer-Based String Manipulation Functions (cont.)

- Figure 22.22 uses `strcpy` (line 13) to copy the entire string in array `x` into array `y` and uses `strncpy` (line 19) to copy the first `14` characters of array `x` into array `z`.
- Line 20 appends a null character (`'\0'`) to array `z`, because the call to `strncpy` in the program does not write a terminating null character.
- (The third argument is less than the string length of the second argument plus one.)

```
1  // Fig. 22.22: fig22_22.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4  #include <cstring> // prototypes for strcpy and strncpy
5  using namespace std;
6
7  int main()
8  {
9     char x[] = "Happy Birthday to You"; // string length 21
10    char y[ 25 ];
11    char z[ 15 ];
12
13    strcpy( y, x ); // copy contents of x into y
14
15    cout << "The string in array x is: " << x
16       << "\nThe string in array y is: " << y << '\n';
17
18    // copy first 14 characters of x into z
19    strncpy( z, x, 14 ); // does not copy null character
20    z[ 14 ] = '\0'; // append '\0' to z's contents
21
22    cout << "The string in array z is: " << z << endl;
23 } // end main
```

Fig. 22.22 | `strcpy` and `strncpy`. (Part 1 of 2.)

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Fig. 22.22 | `strcpy` and `strncpy`. (Part 2 of 2.)

22.10 Pointer-Based String Manipulation Functions (cont.)

- Function strcat appends its second argument (a string) to its first argument (a character array containing a string).
- The first character of the second argument replaces the null character (`'\0'`) that terminates the string in the first argument.
- You must ensure that the array used to store the first string is large enough to store the combination of the first string, the second string and the terminating null character (copied from the second string).
- Function strncat appends a specified number of characters from the second string to the first string and appends a terminating null character to the result.
- The program of Fig. 22.23 demonstrates function `strcat` (lines 15 and 25) and function `strncat` (line 20).

22.10 Pointer-Based String Manipulation Functions (cont.)

- Function strcat appends its second argument (a string) to its first argument (a character array containing a string).
- The first character of the second argument replaces the null character (`'\0'`) that terminates the string in the first argument.
- You must ensure that the array used to store the first string is large enough to store the combination of the first string, the second string and the terminating null character (copied from the second string).
- Function strncat appends a specified number of characters from the second string to the first string and appends a terminating null character to the result.
- The program of Fig. 22.23 demonstrates function `strcat` (lines 15 and 25) and function `strncat` (line 20).

```
1  // Fig. 22.23: fig23_23.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4  #include <cstring> // prototypes for strcat and strncat
5  using namespace std;
6
7  int main()
8  {
9     char s1[ 20 ] = "Happy "; // length 6
10    char s2[] = "New Year "; // length 9
11    char s3[ 40 ] = "";
12
13    cout << "s1 = " << s1 << "\ns2 = " << s2;
14
15    strcat( s1, s2 ); // concatenate s2 to s1 (length 15)
16
17    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
18
19    // concatenate first 6 characters of s1 to s3
20    strncat( s3, s1, 6 ); // places '\0' after last character
21
```

Fig. 22.23 | `strcat` and `strncat`. (Part 1 of 2.)

```
22      cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
23         << "\ns3 = " << s3;
24
25      strcat( s3, s1 ); // concatenate s1 to s3
26      cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
27         << "\ns3 = " << s3 << endl;
28   } // end main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

Fig. 22.23 | `strcat` and `strncat`. (Part 2 of 2.)

22.10 Pointer-Based String Manipulation Functions (cont.)

- Figure 22.24 compares three strings using `strcmp` (lines 15–17) and `strncmp` (lines 20–22).
- Function `strcmp` compares its first string argument with its second string argument character by character.
- The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string.
- Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares up to a specified number of characters.
- Function `strncmp` stops comparing characters if it reaches the null character in one of its string arguments.
- The program prints the integer value returned by each function call.

Common Programming Error 22.10

Assuming that `strcmp` *and* `strncmp` *return one (a true value) when their arguments are equal is a logic error. Both functions return zero (C++'s false value) for equality. Therefore, when testing two strings for equality, the result of the* `strcmp` *or* `strncmp` *function should be compared with zero to determine whether the strings are equal.*

```
1  // Fig. 22.24: fig22_24.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstring> // prototypes for strcmp and strncmp
6  using namespace std;
7
8  int main()
9  {
10    char *s1 = "Happy New Year";
11    char *s2 = "Happy New Year";
12    char *s3 = "Happy Holidays";
13
14    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
15       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
16       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
17       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
18
19    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
20       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
21       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
22       << strncmp( s3, s1, 7 ) << endl;
23 } // end main
```

Fig. 22.24 | strcmp and strncmp. (Part 1 of 2.)

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Fig. 22.24 | `strcmp` and `strncmp`. (Part 2 of 2.)

22.10 Pointer-Based String Manipulation Functions (cont.)

- To understand what it means for one string to be "greater than" or "less than" another, consider the process of alphabetizing last names. (?)
- You'd, no doubt, place "Jones" before "Smith," because the first letter of "Jones" comes before the first letter of "Smith" in the alphabet.
- But the alphabet is more than just a list of 26 letters—it's an ordered list of characters.
- Each letter occurs in a specific position within the list.
- "Z" is more than just a letter of the alphabet; "Z" is specifically the 26th letter of the alphabet.

22.10 Pointer-Based String Manipulation Functions (cont.)

- How does the computer know that one letter comes before another?
- All characters are represented inside the computer as numeric codes; when the computer compares two strings, it actually **compares the numeric codes of the characters in the strings**.
- In an effort to standardize character representations, most computer manufacturers have designed their machines to utilize one of two popular coding schemes—ASCII or EBCDIC.
- Recall that ASCII stands for "American Standard Code for Information Interchange."
- EBCDIC stands for "Extended Binary Coded Decimal Interchange Code."
- ASCII and EBCDIC are called character codes, or character sets.

22.10 Pointer-Based String Manipulation Functions (cont.)

Most readers of this book will be using desktop or notebook computers that use the ASCII character set.

IBM mainframe computers use the EBCDIC character set.

As Internet and World Wide Web usage becomes pervasive, the newer Unicode character set is growing in popularity (`www.unicode.org`).

String and character manipulations actually involve the manipulation of the appropriate numeric codes and not the characters themselves.

This explains the interchangeability of characters and small integers in C++.

Since it's meaningful to say that one numeric code is greater than, less than or equal to another numeric code, it becomes possible to relate various characters or strings to one another by referring to the character codes.

Portability Tip 22.6

The internal numeric codes used to represent characters may be different on different computers that use different character sets.

Portability Tip 22.7

Do not explicitly test for ASCII codes, as in `if (rating == 65)`*; rather, use the corresponding character constant, as in* `if (rating == 'A')`.

22.10 Pointer-Based String Manipulation Functions (cont.)

- Function strtok breaks a string into a series of tokens.
- A token is a sequence of characters separated by delimiting characters (usually spaces or punctuation marks).
- For example, in a line of text, each word can be considered a token, and the spaces separating the words can be considered delimiters.
- Multiple calls to `strtok` are required to break a string into tokens (assuming that the string contains more than one token).

22.10 Pointer-Based String Manipulation Functions (cont.)

- The first call to `strtok` contains two arguments, a string to be tokenized and a string containing characters that separate the tokens (i.e., delimiters).
- Line 16 in Fig. 22.25 assigns to `tokenPtr` a pointer to the first token in `sentence`.
- The second argument, `" "`, indicates that tokens in `sentence` are separated by spaces.
- Function `strtok` searches for the first character in `sentence` that is not a delimiting character (space).
- This begins the first token.
- The function then finds the next delimiting character in the string and replaces it with a null (`'\0'`) character.
- This terminates the current token.

22.10 Pointer-Based String Manipulation Functions (cont.)

- Function `strtok` saves (in a `static` variable) a pointer to the next character following the token in `sentence` and returns a pointer to the current token.
- Subsequent calls to `strtok` to continue tokenizing `sentence` contain `NULL` as the first argument (line 22).
- The `NULL` argument indicates that the call to `strtok` should continue tokenizing from the location in `sentence` saved by the last call to `strtok`.
- Function `strtok` maintains this saved information in a manner that is not visible to you.
- If no tokens remain when `strtok` is called, `strtok` returns `NULL`.

```
1  // Fig. 22.25: fig22_25.cpp
2  // Using strtok to tokenize a string.
3  #include <iostream>
4  #include <cstring> // prototype for strtok
5  using namespace std;
6
7  int main()
8  {
9     char sentence[] = "This is a sentence with 7 tokens";
10    char *tokenPtr;
11
12    cout << "The string to be tokenized is:\n" << sentence
13       << "\n\nThe tokens are:\n\n";
14
15    // begin tokenization of sentence
16    tokenPtr = strtok( sentence, " " );
17
18    // continue tokenizing sentence until tokenPtr becomes NULL
19    while ( tokenPtr != NULL )
20    {
21       cout << tokenPtr << '\n';
22       tokenPtr = strtok( NULL, " " ); // get next token
23    } // end while
24
```

Fig. 22.25 | Using `strtok` to tokenize a string. (Part 1 of 2.)

```
25      cout << "\nAfter strtok, sentence = " << sentence << endl;
26   } // end main
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

Fig. 22.25 | Using `strtok` to tokenize a string. (Part 2 of 2.)

Common Programming Error 22.11

Not realizing that `strtok` *modifies the string being tokenized, then attempting to use that string as if it were the original unmodified string is a logic error.*

22.10 Pointer-Based String Manipulation Functions (cont.)

- Function strlen takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length.
- The length is also the index of the null character.
- The program of Fig. 22.26 demonstrates function `strlen`.

```
1  // Fig. 22.26: fig22_26.cpp
2  // Using strlen.
3  #include <iostream>
4  #include <cstring> // prototype for strlen
5  using namespace std;
6
7  int main()
8  {
9     char *string1 = "abcdefghijklmnopqrstuvwxyz";
10    char *string2 = "four";
11    char *string3 = "Boston";
12
13    cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
14       << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
15       << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
16       << endl;
17 } // end main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Fig. 22.26 | `strlen` returns the length of a `char *` string.

22.11 Pointer-Based String-Conversion Functions

- In Section 22.10, we discussed several of C++'s most popular pointer-based string-manipulation functions.
- In the next several sections, we cover the remaining functions, including functions for converting strings to numeric values, functions for searching strings and functions for manipulating, comparing and searching blocks of memory.
- This section presents the pointer-based string-conversion functions from the general-utilities library <cstdlib>.
- These functions convert pointer-based strings of characters to integer and floating-point values.
- In new code development, C++ programmers typically use the string stream processing capabilities introduced in Chapter 18 to perform such conversions.
- Figure 22.27 summarizes the pointer-based string-conversion functions.
- When using functions from the general-utilities library, include the `<cstdlib>` header file.

Prototype	Description
`double atof( const char *nPtr )`	Converts the string `nPtr` to `double`. If the string cannot be converted, `0` is returned.
`int atoi( const char *nPtr )`	Converts the string `nPtr` to `int`. If the string cannot be converted, `0` is returned.
`long atol( const char *nPtr )`	Converts the string `nPtr` to `long int`. If the string cannot be converted, `0` is returned.
`double strtod( const char *nPtr, char **endPtr )`	Converts the string `nPtr` to `double`. `endPtr` is the address of a pointer to the rest of the string after the `double`. If the string cannot be converted, `0` is returned.

Fig. 22.27 | Pointer-based string-conversion functions of the general-utilities library. (Part 1 of 2.)

Prototype	Description
`long strtol( const char *nPtr, char **endPtr, int base )`	Converts the string `nPtr` to `long`. `endPtr` is the address of a pointer to the rest of the string after the `long`. If the string cannot be converted, `0` is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base )`	Converts the string `nPtr` to `unsigned long`. `endPtr` is the address of a pointer to the rest of the string after the `unsigned long`. If the string cannot be converted, `0` is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal.

Fig. 22.27 | Pointer-based string-conversion functions of the general-utilities library. (Part 2 of 2.)

22.11 Pointer-Based String-Conversion Functions (cont.)

- Function atof (Fig. 22.28, line 9) converts its argument—a string that represents a floating-point number—to a `double` value.
- The function returns the `double` value.
- If the string cannot be converted—for example, if the first character of the string is not a digit—function `atof` returns zero.

```
1  // Fig. 22.28: fig22_28.cpp
2  // Using atof.
3  #include <iostream>
4  #include <cstdlib> // atof prototype
5  using namespace std;
6
7  int main()
8  {
9     double d = atof( "99.0" ); // convert string to double
10
11    cout << "The string \"99.0\" converted to double is " << d
12       << "\nThe converted value divided by 2 is " << d / 2.0 << endl;
13 } // end main
```

```
The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5
```

Fig. 22.28 | String-conversion function `atof`.

22.11 Pointer-Based String-Conversion Functions (cont.)

- Function atoi (Fig. 22.29, line 9) converts its argument—a string of digits that represents an integer—to an `int` value.
- The function returns the `int` value.
- If the string cannot be converted, function `atoi` returns zero.

```
1  // Fig. 22.29: Fig22_29.cpp
2  // Using atoi.
3  #include <iostream>
4  #include <cstdlib> // atoi prototype
5  using namespace std;
6
7  int main()
8  {
9     int i = atoi( "2593" ); // convert string to int
10
11    cout << "The string \"2593\" converted to int is " << i
12       << "\nThe converted value minus 593 is " << i - 593 << endl;
13 } // end main
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Fig. 22.29 | String-conversion function `atoi`.

22.11 Pointer-Based String-Conversion Functions (cont.)

- Function atol (Fig. 22.30, line 9) converts its argument—a string of digits representing a long integer—to a `long` value.
- The function returns the `long` value.
- If the string cannot be converted, function `atol` returns zero.
- If `int` and `long` are both stored in four bytes, function `atoi` and function `atol` work identically.

```
1  // Fig. 22.30: fig22_30.cpp
2  // Using atol.
3  #include <iostream>
4  #include <cstdlib> // atol prototype
5  using namespace std;
6
7  int main()
8  {
9     long x = atol( "1000000" ); // convert string to long
10
11    cout << "The string \"1000000\" converted to long is " << x
12       << "\nThe converted value divided by 2 is " << x / 2 << endl;
13 } // end main
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Fig. 22.30 | String-conversion function `atol`.

22.11 Pointer-Based String-Conversion Functions (cont.)

- Function strtod (Fig. 22.31) converts a sequence of characters representing a floating-point value to `double`.
- Function `strtod` receives two arguments—a string (`char *`) and the address of a `char *` pointer (i.e., a `char **`).
- The string contains the character sequence to be converted to `double`.
- The second argument enables `strtod` to modify a `char *` pointer in the calling function, such that the pointer points to the location of the first character after the converted portion of the string.
- Line 13 indicates that `d` is assigned the `double` value converted from `string` and that `stringPtr` is assigned the location of the first character after the converted value (`51.2`) in `string`.

```
1  // Fig. 22.31: fig22_31.cpp
2  // Using strtod.
3  #include <iostream>
4  #include <cstdlib> // strtod prototype
5  using namespace std;
6
7  int main()
8  {
9     double d;
10    const char *string1 = "51.2% are admitted";
11    char *stringPtr;
12
13    d = strtod( string1, &stringPtr ); // convert characters to double
14
15    cout << "The string \"" << string1
16       << "\" is converted to the\ndouble value " << d
17       << " and the string \"" << stringPtr << "\"" << endl;
18 } // end main
```

```
The string "51.2% are admitted" is converted to the
double value 51.2 and the string "% are admitted"
```

Fig. 22.31 | String-conversion function `strtod`.

22.11 Pointer-Based String-Conversion Functions (cont.)

- Function strtol (Fig. 22.32) converts to `long` a sequence of characters representing an integer.
- The function receives a string (`char *`), the address of a `char *` pointer and an integer.
- The string contains the character sequence to convert.
- The second argument is assigned the location of the first character after the converted portion of the string.
- The integer specifies the base of the value being converted.

22.11 Pointer-Based String-Conversion Functions (cont.)

- Line 13 indicates that `x` is assigned the `long` value converted from `string` and that `remainderPtr` is assigned the location of the first character after the converted value (`-1234567`) in `string1`.
- Using a null pointer for the second argument causes the remainder of the string to be ignored.
- The third argument, `0`, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16).
- This is determined by the initial characters in the string—`0` indicates an octal number, `0x` indicates hexadecimal and a number from `1`–`9` indicates decimal.

```
1  // Fig. 22.32: fig22_32.cpp
2  // Using strtol.
3  #include <iostream>
4  #include <cstdlib> // strtol prototype
5  using namespace std;
6
7  int main()
8  {
9     long x;
10    const char *string1 = "-1234567abc";
11    char *remainderPtr;
12
13    x = strtol( string1, &remainderPtr, 0 ); // convert characters to long
14
15    cout << "The original string is \"" << string1
16       << "\"\nThe converted value is " << x
17       << "\nThe remainder of the original string is \"" << remainderPtr
18       << "\"\nThe converted value plus 567 is " << x + 567 << endl;
19 } // end main
```

Fig. 22.32 | String-conversion function `strtol`. (Part 1 of 2.)

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Fig. 22.32 | String-conversion function `strtol`. (Part 2 of 2.)

22.11 Pointer-Based String-Conversion Functions (cont.)

- Function strtoul (Fig. 22.33) converts to `unsigned long` a sequence of characters representing an `unsigned long` integer.
- The function works identically to `strtol`.
- Line 14 indicates that `x` is assigned the `unsigned long` value converted from `string` and that `remainderPtr` is assigned the location of the first character after the converted value (`1234567`) in `string1`.
- The third argument, `0`, indicates that the value to be converted can be in octal, decimal or hexadecimal format, depending on the initial characters.

```
1  // Fig. 22.33: fig22_33.cpp
2  // Using strtoul.
3  #include <iostream>
4  #include <cstdlib> // strtoul prototype
5  using namespace std;
6
7  int main()
8  {
9     unsigned long x;
10    const char *string1 = "1234567abc";
11    char *remainderPtr;
12
13    // convert a sequence of characters to unsigned long
14    x = strtoul( string1, &remainderPtr, 0 );
15
16    cout << "The original string is \"" << string1
17       << "\"\nThe converted value is " << x
18       << "\nThe remainder of the original string is \"" << remainderPtr
19       << "\"\nThe converted value minus 567 is " << x - 567 << endl;
20 } // end main
```

Fig. 22.33 | String-conversion function `strtoul`. (Part 1 of 2.)

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Fig. 22.33 | String-conversion function `strtoul`. (Part 2 of 2.)

22.12 Search Functions of the Pointer-Based String-Handling Library

- This section presents the functions of the string-handling library used to search strings for characters and other strings.
- The functions are summarized in Fig. 22.34.
- Functions `strcspn` and `strspn` specify return type `size_t`.
- Type `size_t` is a type defined by the standard as the integral type of the value returned by operator `sizeof`.

Character to be located. It is passed as its ***int*** promotion, but it is internally converted back to ***char***.

Prototype	Description
`char *strchr( const char *s, int c )`	Locates the first occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in `s` is returned. Otherwise, a null pointer is returned.
`char *strrchr( const char *s, int c )`	Searches from the end of string `s` and locates the last occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in string `s` is returned. Otherwise, a null pointer is returned.
`size_t strspn( const char *s1, const char *s2 )`	Determines and returns the length of the initial segment of string `s1` consisting only of characters contained in string `s2`.
`char *strpbrk( const char *s1, const char *s2 )`	Locates the first occurrence in string `s1` of any character in string `s2`. If a character from string `s2` is found, a pointer to the character in string `s1` is returned. Otherwise, a null pointer is returned.

Fig. 22.34 | Search functions of the pointer-based string-handling library. (Part 1 of 2.)

Prototype	Description
`size_t strcspn( const char *s1, const char *s2 )`	Determines and returns the length of the initial segment of string `s1` consisting of characters not contained in string `s2`.
`char *strstr( const char *s1, const char *s2 )`	Locates the first occurrence in string `s1` of string `s2`. If the string is found, a pointer to the string in `s1` is returned. Otherwise, a null pointer is returned.

Fig. 22.34 | Search functions of the pointer-based string-handling library. (Part 2 of 2.)

macro

NULL

<cstddef> <cstdlib> <cstring> <cwchar> <ctime> <clocale> <cstdio>

Null pointer

This macro expands to a *null pointer constant*.

C | C++98 | C++11

A *null-pointer constant* is an integral constant expression that evaluates to zero (such as `0` or `0L`).

C | C++98 | C++11

A *null-pointer constant* is either an integral constant expression that evaluates to zero (such as `0` or `0L`), or a value of type nullptr_t (such as `nullptr`).

A null pointer constant can be converted to any *pointer type* (or *pointer-to-member type*), which acquires a *null pointer value*. This is a special value that indicates that the pointer is not pointing to any object.

type

nullptr_t

<cstddef>

```
typedef decltype(nullptr) nullptr_t;
```

Null pointer type (C++)

Type of the null pointer constant `nullptr`.

This type can only take one value: `nullptr`, which when converted to a pointer type takes the proper *null pointer value*.

Even though `nullptr_t` it is not a keyword, it identifies a distinct **fundamental type**: the type of `nullptr`. As such, it participates in overload resolution as a different type.

This type is only defined for C++ (since C++11).

22.12 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function strchr searches for the first occurrence of a character in a string.
- If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns a null pointer.
- The program of Fig. 22.35 uses `strchr` (lines 14 and 22) to search for the first occurrences of `'a'` and `'z'` in the string `"This is a test"`.

```
1  // Fig. 22.35: fig22_35.cpp
2  // Using strchr.
3  #include <iostream>
4  #include <cstring> // strchr prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "This is a test";
10    char character1 = 'a';
11    char character2 = 'z';
12
13    // search for character1 in string1
14    if ( strchr( string1, character1 ) != NULL )
15       cout << '\'' << character1 << "' was found in \""
16          << string1 << "\".\n";
17    else
18       cout << '\'' << character1 << "' was not found in \""
19          << string1 << "\".\n";
20
```

Fig. 22.35 | String-search function `strchr`. (Part 1 of 2.)

```
21      // search for character2 in string1
22      if ( strchr( string1, character2 ) != NULL )
23         cout << '\'' << character2 << "' was found in \""
24            << string1 << "\".\n";
25      else
26         cout << '\'' << character2 << "' was not found in \""
27            << string1 << "\"." << endl;
28   } // end main
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Fig. 22.35 | String-search function `strchr`. (Part 2 of 2.)

C++ shell

```
1  /* strchr example */
2  #include <iostream>
3  #include <cstring>
4
5  int main ()
6  {
7    char str[] = "This is a sample string";
8    char * pch;
9    std::cout << "Looking for the 's' character in \"" << str << "\"...\n";
10   pch=strchr(str,'s');
11   while (pch!=NULL)
12   {
13     std::cout << "found at " << (pch-str+1) << std::endl;
14     pch=strchr(pch+1,'s');
15   }
16   return 0;
17 }
```

How to compute the position of it?

Get URL

options | compilation | execution

```
Looking for the 's' character in "This is a sample string"...
found at 4
found at 7
found at 11
found at 18
```

22.12 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function strcspn (Fig. 22.36, line 15) determines the length of the initial part of the string in its 1st argument that does **NOT** contain any characters from the string in its 2nd argument.
- The function returns the **length** of the segment.

```
1  // Fig. 22.36: fig22_36.cpp
2  // Using strcspn.
3  #include <iostream>
4  #include <cstring> // strcspn prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "The value is 3.14159";
10    const char *string2 = "1234567890";
11
12    cout << "string1 = " << string1 << "\nstring2 = " << string2
13       << "\n\nThe length of the initial segment of string1"
14       << "\ncontaining no characters from string2 = "
15       << strcspn( string1, string2 ) << endl;
16 } // end main
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

Fig. 22.36 | String-search function `strcspn`.

22.12 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function strpbrk searches for the first occurrence in its 1st string argument of any character in its 2nd string argument.
- If a character from the 2nd argument is found, `strpbrk` returns a pointer to the character in the 1st argument; otherwise, `strpbrk` returns a null pointer.
- Line 13 of Fig. 22.37 locates the first occurrence in `string1` of any character from `string2`.

```
1  // Fig. 22.37: fig22_37.cpp
2  // Using strpbrk.
3  #include <iostream>
4  #include <cstring> // strpbrk prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "This is a test";
10    const char *string2 = "beware";
11
12    cout << "Of the characters in \"" << string2 << "\"\n'"
13       << *strpbrk( string1, string2 ) << "\' is the first character "
14       << "to appear in\n\"" << string1 << '\"' << endl;
15 } // end main
```

single character

```
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
```

Fig. 22.37 | String-search function `strpbrk`.

22.12 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function strrchr searches for the **last** occurrence of the specified character in a string.
- If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, `strrchr` returns a null pointer.
- Line 15 of Fig. 22.38 searches for the last occurrence of the character `'z'` in the string `"A zoo has many animals including zebras"`.

```
1  // Fig. 22.38: fig22_38.cpp
2  // Using strrchr.
3  #include <iostream>
4  #include <cstring> // strrchr prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "A zoo has many animals including zebras";
10    char c = 'z';
11
12    cout << "string1 = " << string1 << "\n" << endl;
13    cout << "The remainder of string1 beginning with the\n"
14       << "last occurrence of character '"
15       << c << "' is: \"" << strrchr( string1, c ) << '\"' << endl;
16 } // end main
```

```
string1 = A zoo has many animals including zebras

The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Fig. 22.38 | String-search function `strrchr`.

22.12 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function strspn (Fig. 22.39, line 15) determines the length of the initial part of the string in its 1st argument that contains **only** characters from the string in its 2nd argument.
- The function returns the **length** of the segment.

```
1  // Fig. 22.39: fig22_39.cpp
2  // Using strspn.
3  #include <iostream>
4  #include <cstring> // strspn prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "The value is 3.14159";
10    const char *string2 = "aehils Tuv";
11
12    cout << "string1 = " << string1 << "\nstring2 = " << string2
13       << "\n\nThe length of the initial segment of string1\n"
14       << "containing only characters from string2 = "
15       << strspn( string1, string2 ) << endl;
16 } // end main
```

```
string1 = The value is 3.14159
string2 = aehils Tuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

Fig. 22.39 | String-search function `strspn`.

22.12 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function strstr searches for the 1st occurrence of its 2nd string argument in its first string argument.
- If the 2nd string is found in the 1st string, a pointer to the location of the string in the 1st argument is returned; otherwise, it returns a null pointer.
- Line 15 of Fig. 22.40 uses `strstr` to find the string `"def"` in the string `"abcdefabcdef"`.

```
1  // Fig. 22.40: fig22_40.cpp
2  // Using strstr.
3  #include <iostream>
4  #include <cstring> // strstr prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "abcdefabcdef";
10    const char *string2 = "def";
11
12    cout << "string1 = " << string1 << "\nstring2 = " << string2
13       << "\n\nThe remainder of string1 beginning with the\n"
14       << "first occurrence of string2 is: "
15       << strstr( string1, string2 ) << endl;
16 } // end main
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Fig. 22.40 | String-search function `strstr`.

C++ shell

```
1  /* strrchr example */
2  #include <stdio.h>
3  #include <string.h>
4
5  int main ()
6  {
7    char str[] = "This is a sample string";
8    char * pch;
9    pch=strrchr(str,'z');
10   printf ("Last occurence of 's' found at %d \n",pch-str+1);
11   return 0;
12 }
```

Get URL

options | compilation | execution

```
Last occurence of 's' found at -1218588367
```

Undefined result (not found)
Don't forget to check this "pch !=NULL"

http://cpp.sh/8btw6

22.13 Memory Functions of the Pointer-Based String-Handling Library

- The string-handling library functions presented in this section facilitate manipulating, comparing and searching blocks of memory.
- The functions treat blocks of memory as arrays of bytes.
- These functions can manipulate any block of data.
- Figure 22.41 summarizes the memory functions of the string-handling library.
- In the function discussions, "object" refers to a block of data.

Prototype	Description
`void *memcpy( void *s1, const void *s2, size_t n )`	Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. A pointer to the resulting object is returned. The area from which characters are copied is not allowed to overlap the area to which characters are copied.
`void *memmove( void *s1, const void *s2, size_t n )`	Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The copy is performed as if the characters were first copied from the object pointed to by `s2` into a temporary array, then copied from the temporary array into the object pointed to by `s1`. A pointer to the resulting object is returned. The area from which characters are copied is allowed to overlap the area to which characters are copied.
`int memcmp( const void *s1, const void *s2, size_t n )`	Compares the first `n` characters of the objects pointed to by `s1` and `s2`. The function returns `0`, less than `0`, or greater than `0` if `s1` is equal to, less than or greater than `s2`, respectively.

Fig. 22.41 | Memory functions of the string-handling library. (Part 1 of 2.)

Prototype	Description
`void *memchr( const void *s, int c, size_t n )`	Locates the first occurrence of `c` (converted to `unsigned char`) in the first `n` characters of the object pointed to by `s`. If `c` is found, a pointer to `c` in the object is returned. Otherwise, `0` is returned.
`void *memset( void *s, int c, size_t n )`	Copies `c` (converted to `unsigned char`) into the first `n` characters of the object pointed to by `s`. A pointer to the result is returned.

Fig. 22.41 | Memory functions of the string-handling library. (Part 2 of 2.)

22.13 Search Functions of the Pointer-Based String-Handling Library (cont.)

- The pointer parameters to these functions are declared `void *`.
- In Chapter 8, we saw that a pointer to any data type can be assigned directly to a pointer of type `void *`.
- For this reason, these functions can receive pointers to any data type.
- Because a `void *` pointer cannot be dereferenced, each function receives a size argument that specifies the number of characters (bytes) the function will process.
- For simplicity, the examples in this section manipulate character arrays (blocks of characters).

22.13 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function memcpy **copies** a specified number of characters (bytes) from the object pointed to by its 2nd argument into the object pointed to by its 1st argument.
- The function can receive a pointer to any type of object.
- The result of this function is **undefined** if the two objects overlap in memory (i.e., are parts of the same object).
- The program of Fig. 22.42 uses `memcpy` (line 14) to copy the string in array `s2` to array `s1`.

```
1  // Fig. 22.42: fig22_36.cpp
2  // Using memcpy.
3  #include <iostream>
4  #include <cstring> // memcpy prototype
5  using namespace std;
6
7  int main()
8  {
9     char s1[ 17 ];
10
11    // 17 total characters (includes terminating null)
12    char s2[] = "Copy this string";
13
14    memcpy( s1, s2, 17 ); // copy 17 characters from s2 to s1
15
16    cout << "After s2 is copied into s1 with memcpy,\n"
17       << "s1 contains \"" << s1 << '\"' << endl;
18 } // end main
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Fig. 22.42 | Memory-handling function `memcpy`.

22.13 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function memmove, like `memcpy`, cuts a specified number (3rd argument) of bytes from the object pointed to by its 2nd argument into the object pointed to by its 1st argument.
- Copying is performed as if the bytes were copied from the 2nd argument to a temporary array of characters, then copied from the temporary array to the 1st argument.
- This allows characters from one part of a string to be copied into another part of the same string.
- Fig. 22.43 uses `memmove` (line 13) to copy the last `10` bytes of array `x` into the first `10` bytes of array `x`.

Common Programming Error 22.12

String-manipulation functions other than `memmove` *that copy characters have undefined results when copying takes place between parts of the same string.*

```
1  // Fig. 22.43: fig22_37.cpp
2  // Using memmove.
3  #include <iostream>
4  #include <cstring> // memmove prototype
5  using namespace std;
6
7  int main()
8  {
9     char x[] = "Home Sweet Home";
10
11    cout << "The string in array x before memmove is: " << x;
12    cout << "\nThe string in array x after memmove is:  "
13       << static_cast< char * >( memmove( x, &x[ 5 ], 10 ) ) << endl;
14 } // end main
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is:  Sweet Home Home
```

Fig. 22.43 | Memory-handling function `memmove`.

22.13 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function memcmp (Fig. 22.44, lines 14–16) compares the specified number of characters of its 1st argument with the corresponding characters of its 2nd argument.
- The function returns a value greater than zero if the 1st argument is greater than the 2nd argument (?),
  - **zero** if the arguments are equal
  - **less than zero** if the 1st argument is less than the 2nd argument.

Return Value

Returns an integral value indicating the relationship between the content of the memory blocks:

return value	indicates
<0	the first byte that does not match in both memory blocks has a lower value in *ptr1* than in *ptr2* (if evaluated as *unsigned char* values)
0	the contents of both memory blocks are equal
>0	the first byte that does not match in both memory blocks has a greater value in *ptr1* than in *ptr2* (if evaluated as *unsigned char* values)

```
1  // Fig. 22.44: fig22_38.cpp
2  // Using memcmp.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstring> // memcmp prototype
6  using namespace std;
7
8  int main()
9  {
10    char s1[] = "ABCDEFG";
11    char s2[] = "ABCDXYZ";
12
13    cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
14       << "\nmemcmp(s1, s2, 4) = " << setw( 3 ) << memcmp( s1, s2, 4 )
15       << "\nmemcmp(s1, s2, 7) = " << setw( 3 ) << memcmp( s1, s2, 7 )
16       << "\nmemcmp(s2, s1, 7) = " << setw( 3 ) << memcmp( s2, s1, 7 )
17       << endl;
18 } // end main
```

Fig. 22.44 | Memory-handling function **memcmp**. (Part 1 of 2.)

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =   0
memcmp(s1, s2, 7) =  -1
memcmp(s2, s1, 7) =   1
```

Fig. 22.44 | Memory-handling function **memcmp**. (Part 2 of 2.)

22.13 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function memchr searches for the first occurrence of a byte, represented as **`unsigned char`**, in the specified number of bytes of an object.
- If the byte is found in the object, a pointer to it is returned; otherwise, the function returns a null pointer.
- Line 13 of Fig. 22.45 searches for the character (byte) `'r'` in the string `"This is a string"`.

function

memchr <cstring>

```
const void * memchr ( const void * ptr, int value, size_t num );
      void * memchr (       void * ptr, int value, size_t num );
```

Locate character in block of memory

Searches within the first *num* bytes of the block of memory pointed by *ptr* for the first occurrence of *value* (interpreted as an `unsigned char`), and returns a pointer to it.

Both *value* and each of the bytes checked on the the *ptr* array are interpreted as `unsigned char` for the comparison.

```
1  // Fig. 22.45: fig22_39.cpp
2  // Using memchr.
3  #include <iostream>
4  #include <cstring> // memchr prototype
5  using namespace std;
6
7  int main()
8  {
9     char s[] = "This is a string";
10
11    cout << "s = " << s << "\n" << endl;
12    cout << "The remainder of s after character 'r' is found is \""
13       << static_cast< char * >( memchr( s, 'r', 16 ) ) << '\"' << endl;
14 } // end main
```

```
s = This is a string

The remainder of s after character 'r' is found is "ring"
```

Fig. 22.45 | Memory-handling function `memchr`.

22.13 Search Functions of the Pointer-Based String-Handling Library (cont.)

- Function memset copies the value of the byte in its 2nd argument into a specified number of bytes of the object pointed to by its 1st argument.
- Line 13 in Fig. 22.46 uses `memset` to copy `'b'` into the first 7 bytes of `string1`.

```
1  // Fig. 22.46: fig22_40.cpp
2  // Using memset.
3  #include <iostream>
4  #include <cstring> // memset prototype
5  using namespace std;
6
7  int main()
8  {
9     char string1[ 15 ] = "BBBBBBBBBBBBBB";
10
11    cout << "string1 = " << string1 << endl;
12    cout << "string1 after memset = "
13       << static_cast< char * >( memset( string1, 'b', 7 ) ) << endl;
14 } // end main
```

Fig. 22.46 | Memory-handling function `memset`. (Part 1 of 2.)

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

Fig. 22.46 | Memory-handling function `memset`. (Part 2 of 2.)

SUGGESTION!

OR

OBJECTION?

TAKE A BREAK