Introduction to Custom Templates

18

Behind that outside
pattern the dim shapes
get clearer every day.
It is always the same shape,
only very numerous.
—Charlotte Perkins Gilman

Every man of genius sees the world at a different angle from his fellows.
—Havelock Ellis

…our special individuality, as distinguished from our generic humanity.
—Oliver Wendell Holmes, Sr.

Objectives

In this chapter you'll:

- Use class templates to create groups of related classes.
- Distinguish between class templates and class-template specializations.
- Learn about nontype template parameters.
- Learn about default template arguments.
- Learn about overloading function templates.

Outline

18.1 Introduction

In Chapters 7, 15 and 16, you used many of the Standard Library's *prepackaged* templatized containers and algorithms. Function templates (which were introduced in Chapter 6) and **class templates** enable you to conveniently specify a variety of related (overloaded) functions—called **function-template specializations**—or a variety of related classes—called **class-template specializations**, respectively. This is called **generic programming**. Function templates and class templates are like *stencils* out of which we trace shapes; function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors and textures.

In this chapter, we demonstrate how to create a custom class template and a function template that manipulates objects of our class-template specializations. We focus on the template capabilities you'll need to build the custom templatized data structures that we present in Chapter 19.[1]

18.2 Class Templates

It's possible to *understand* the concept of a stack (a data structure into which we insert items *only* at the *top* and retrieve those items *only* from the *top* in *last-in, first-out order*) *independent of the type of the items* being placed in the stack. However, to *instantiate* a stack, a data type must be specified. This creates a nice opportunity for software reusability—as you already saw with the `stack` container adapter in Section 15.7.1. Here, we define a stack *generically* then use *type-specific* versions of this generic stack class.

Software Engineering Observation 18.1

Class templates encourage software reusability by enabling a variety of type-specific class-template specializations to be instantiated from a single class template.

Class templates are called **parameterized types**, because they require one or more *type parameters* to specify how to customize a generic class template to form a *class-template specialization*. To produce many specializations you write only one class-template definition (as we'll do shortly). When a particular specialization is needed, you use a concise, simple notation, and the compiler writes the specialization source code. One `Stack` class template, for example, could thus become the basis for creating many `Stack` class-template

1. Building custom templates is an advanced topic with many features that are beyond the scope of this book.

specializations (such as "Stack of doubles," "Stack of ints," "Stack of Employees," "Stack of Bills," etc.) used in a program.

Common Programming Error 18.1

To create a template specialization with a user-defined type, the user-defined type must meet the template's requirements. For example, the template might compare objects of the user-defined type with < to determine sorting order, or the template might call a specific member function on an object of the user-defined type. If the user-defined type does not overload the required operator or provide the required functions, compilation errors occur.

Creating Class Template `Stack<T>`

The `Stack` class-template definition in Fig. 18.1 looks like a conventional class definition, with a few key differences. First, it's preceded by line 7

```
template< typename T >
```

All class templates begin with keyword `template` followed by a list of template parameters enclosed in angle brackets (< and >); each template parameter that represents a type *must* be preceded by either of the *interchangeable* keywords `typename` or `class`. The type parameter `T` acts as a placeholder for the `Stack`'s element type. The names of type parameters must be *unique* inside a template definition. You need not specifically use identifier `T`—any valid identifier can be used. The element type is mentioned generically throughout the `Stack` class-template definition as `T` (lines 12, 18 and 42). The type parameter becomes associated with a specific type when you create an object using the class template—at that point, the compiler generates a copy of the class template in which all occurrences of the type parameter are replaced with the specified type. Another key difference is that we did *not* separate the class template's interface from its implementation.

Software Engineering Observation 18.2

Templates are typically defined in headers, which are then `#include`d in the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header—typically inside the class definition's body, as we do in Fig. 18.1.

```
1  // Fig. 18.1: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5  #include <deque>
6
7  template< typename T >
8  class Stack
9  {
10 public:
11    // return the top element of the Stack
12    T& top()
13    {
14       return stack.front();
15    } // end function template top
```

Fig. 18.1 | Stack class template. (Part 1 of 2.)

```
16
17      // push an element onto the Stack
18      void push( const T &pushValue )
19      {
20         stack.push_front( pushValue );
21      } // end function template push
22
23      // pop an element from the stack
24      void pop()
25      {
26         stack.pop_front();
27      } // end function template pop
28
29      // determine whether Stack is empty
30      bool isEmpty() const
31      {
32         return stack.empty();
33      } // end function template isEmpty
34
35      // return size of Stack
36      size_t size() const
37      {
38         return stack.size();
39      } // end function template size
40
41   private:
42      std::deque< T > stack; // internal representation of the Stack
43   }; // end class template Stack
44
45   #endif
```

Fig. 18.1 | Stack class template. (Part 2 of 2.)

Class Template `Stack<T>`*'s Data Representation*

Section 15.7.1 showed that the Standard Library's `stack` adapter class can use various containers to store its elements. Of course, a `stack` requires insertions and deletions *only* at its *top*. So, for example, a `vector` or a `deque` could be used to store the `stack`'s elements. A `vector` supports fast insertions and deletions at its *back*. A `deque` supports fast insertions and deletions at its *front* and its *back*. A `deque` is the default representation for the Standard Library's `stack` adapter because a `deque` grows more efficiently than a `vector`. A `vector` is maintained as a *contiguous* block of memory—when that block is full and a new element is added, the `vector` allocates a larger contiguous block of memory and *copies* the old elements into that new block. A `deque`, on the other hand, is typically implemented as list of fixed-size, built-in arrays—new fixed-size built-in arrays are added as necessary and none of the existing elements are copied when new items are added to the front or back. For these reasons, we use a `deque` (line 42) as the underlying container for our `Stack` class.

Class Template `Stack<T>`*'s Member Functions*

The member-function definitions of a class template are *function templates*, but are not preceded with the `template` keyword and template parameters in angle brackets (< and >) when they're defined within the class template's body. As you can see, however, they do

use the class template's template parameter `T` to represent the element type. Our `Stack` class template does *not* define it's own constructors—the *default constructor* provided by the compiler will invoke the `deque`'s default constructor. We also provide the following member functions in Fig. 18.1:

- `top` (lines 12–15) returns a reference to the `Stack`'s top element.
- `push` (lines 18–21) places a new element on the top of the `Stack`.
- `pop` (lines 24–27) removes the `Stack`'s top element.
- `isEmpty` (lines 30–33) returns a `bool` value—`true` if the `Stack` is empty and `false` otherwise.
- `size` (lines 36–39) returns the number if elements in the `Stack`.

Each of these member functions *delegates* its responsibility to the appropriate member function of class template `deque`.

Declaring a Class Template's Member Functions Outside the Class Template Definition

Though we did *not* do so in our `Stack` class template, member-function definitions can appear *outside* a class template definition. If you do this, each must begin with the `template` keyword followed by the *same* set of template parameters as the class template. In addition, the member functions must be qualified with the class name and scope resolution operator. For example, you can define the `pop` function outside the class-template definition as follows:

```
template< typename T >
inline void Stack<T>::pop()
{
   stack.pop_front();
} // end function template pop
```

`Stack<T>::` indicates that `pop` is in the scope of class `Stack<T>`. The Standard Library's container classes tend to define all their member functions *inside* their class definitions.

Testing Class Template `Stack<T>`

Now, let's consider the driver (Fig. 18.2) that exercises the `Stack` class template. The driver begins by instantiating object `doubleStack` (line 9). This object is declared as a `Stack<double>` (pronounced "`Stack` of `double`"). The compiler associates type `double` with type parameter `T` in the class template to produce the source code for a `Stack` class with elements of type `double` that actually stores its elements in a `deque<double>`.

Lines 16–21 invoke `push` (line 18) to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 onto `doubleStack`. Next, lines 26–30 invoke `top` and `pop` in a `while` loop to remove the five values from the stack. Notice in the output of Fig. 18.2, that the values do `pop` off in *last-in, first-out order*. When `doubleStack` is empty, the `pop` loop terminates.

```
1  // Fig. 18.2: fig18_02.cpp
2  // Stack class template test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class template definition
```

Fig. 18.2 | Stack class template `test` program. (Part 1 of 3.)

```
5  using namespace std;
6
7  int main()
8  {
9     Stack< double > doubleStack; // create a Stack of double
10    const size_t doubleStackSize = 5; // stack size
11    double doubleValue = 1.1; // first value to push
12
13    cout << "Pushing elements onto doubleStack\n";
14
15    // push 5 doubles onto doubleStack
16    for ( size_t i = 0; i < doubleStackSize; ++i )
17    {
18       doubleStack.push( doubleValue );
19       cout << doubleValue << ' ';
20       doubleValue += 1.1;
21    } // end while
22
23    cout << "\n\nPopping elements from doubleStack\n";
24
25    // pop elements from doubleStack
26    while ( !doubleStack.isEmpty() ) // loop while Stack is not empty
27    {
28       cout << doubleStack.top() << ' '; // display top element
29       doubleStack.pop(); // remove top element
30    } // end while
31
32    cout << "\nStack is empty, cannot pop.\n";
33
34    Stack< int > intStack; // create a Stack of int
35    const size_t intStackSize = 10; // stack size
36    int intValue = 1; // first value to push
37
38    cout << "\nPushing elements onto intStack\n";
39
40    // push 10 integers onto intStack
41    for ( size_t i = 0; i < intStackSize; ++i )
42    {
43       intStack.push( intValue );
44       cout << intValue++ << ' ';
45    } // end while
46
47    cout << "\n\nPopping elements from intStack\n";
48
49    // pop elements from intStack
50    while ( !intStack.isEmpty() ) // loop while Stack is not empty
51    {
52       cout << intStack.top() << ' '; // display top element
53       intStack.pop(); // remove top element
54    } // end while
55
56    cout << "\nStack is empty, cannot pop." << endl;
57 } // end main
```

Fig. 18.2 | Stack class template `test` program. (Part 2 of 3.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop
```

Fig. 18.2 | Stack class template `test` program. (Part 3 of 3.)

Line 34 instantiates `int` stack `intStack` with the declaration

```
Stack< int > intStack;
```

(pronounced "`intStack` is a `Stack` of `int`"). Lines 41–45 repeatedly invoke `push` (line 43) to place values onto `intStack`, then lines 50–54 repeatedly invoke `top` and `pop` to remove values from `intStack` until it's empty. Once again, notice in the output that the values `pop` off in last-in, first-out order.

18.3 Function Template to Manipulate a Class-Template Specialization Object

Notice that the code in function `main` of Fig. 18.2 is *almost identical* for both the `double`-`Stack` manipulations in lines 9–32 and the `intStack` manipulations in lines 34–56. This presents another opportunity to use a function template. Figure 18.3 defines function template `testStack` (lines 10–39) to perform the same tasks as `main` in Fig. 18.2—`push` a series of values onto a `Stack<T>` and `pop` the values off a `Stack<T>`.

```
1  // Fig. 18.3: fig18_03.cpp
2  // Passing a Stack template object
3  // to a function template.
4  #include <iostream>
5  #include <string>
6  #include "Stack.h" // Stack class template definition
7  using namespace std;
8
9  // function template to manipulate Stack< T >
10 template< typename T >
11 void testStack(
12    Stack< T > &theStack, // reference to Stack< T >
13    const T &value, // initial value to push
14    const T &increment, // increment for subsequent values
15    size_t size, // number of items to push
16    const string &stackName ) // name of the Stack< T > object
17 {
```

Fig. 18.3 | Passing a `Stack` template object to a function template. (Part 1 of 2.)

```
18      cout << "\nPushing elements onto " << stackName << '\n';
19      T pushValue = value;
20
21      // push element onto Stack
22      for ( size_t i = 0; i < size; ++i )
23      {
24         theStack.push( pushValue ); // push element onto Stack
25         cout << pushValue << ' ';
26         pushValue += increment;
27      } // end while
28
29      cout << "\n\nPopping elements from " << stackName << '\n';
30
31      // pop elements from Stack
32      while ( !theStack.isEmpty() ) // loop while Stack is not empty
33      {
34         cout << theStack.top() << ' ';
35         theStack.pop(); // remove top element
36      } // end while
37
38      cout << "\nStack is empty. Cannot pop." << endl;
39   } // end function template testStack
40
41   int main()
42   {
43      Stack< double > doubleStack;
44      const size_t doubleStackSize = 5;
45      testStack( doubleStack, 1.1, 1.1, doubleStackSize, "doubleStack" );
46
47      Stack< int > intStack;
48      const size_t intStackSize = 10;
49      testStack( intStack, 1, 1, intStackSize, "intStack" );
50   } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop
```

Fig. 18.3 | Passing a `Stack` template object to a function template. (Part 2 of 2.)

Function template `testStack` uses `T` (specified at line 10) to represent the data type stored in the `Stack<T>`. The function template takes five arguments (lines 12–16):

- the `Stack<T>` to manipulate

- a value of type `T` that will be the first value `pushed` onto the `Stack<T>`
- a value of type `T` used to increment the values `pushed` onto the `Stack<T>`
- the number of elements to push onto the `Stack<T>`
- a `string` that represents the name of the `Stack<T>` object for output purposes

Function `main` (lines 41–50) instantiates an object of type `Stack<double>` called `doubleStack` (line 43) and an object of type `Stack<int>` called `intStack` (line 47) and uses these objects in lines 45 and 49. The compiler infers the type of `T` for `testStack` from the type used to instantiate the function's first argument (i.e., the type used to instantiate `doubleStack` or `intStack`).

18.4 Nontype Parameters

Class template `Stack` of Section 18.2 used only a type parameter (Fig. 18.1, line 7) in its template declaration. It's also possible to use **nontype template parameters**, which can have default arguments and are treated as constants. For example, the C++ standard's `array` class template begins with the template declaration:

```
template < class T, size_t N >
```

(Recall that keywords `class` and `typename` are *interchangeable* in template declarations.) So, a declaration such as

```
array< double, 100 > salesFigures;
```

creates a 100-element `array` of `double`s class-template specialization, then uses it to instantiate the object `salesFigures`. The `array` class template encapsulates a *built-in array*. When you create an `array` class-template specialization, the `array`'s built-in array data member has the type and size specified in the declaration—in the preceding example, it would be a built-in array of `double` values with `100` elements.

18.5 Default Arguments for Template Type Parameters

In addition, a type parameter can specify a **default type argument**. For example, the C++ standard's `stack` *container adapter* class template begins with:

```
template < class T, class Container = deque< T > >
```

which specifies that a `stack` uses a `deque` *by default* to store the `stack`'s elements of type `T`. The declaration

```
stack< int > values;
```

creates a `stack` of `int`s class-template specialization (behind the scenes) and uses it to instantiate the object named `values`. The `stack`'s elements are stored in a `deque<int>`.

Default type parameters must be the *rightmost* (trailing) parameters in a template's type-parameter list. When you instantiate a template with two or more default arguments, if an omitted argument is not the rightmost, then all type parameters to the right of it also must be omitted. As of C++11, you can now use default type arguments for template type parameters in function templates.

18.6 Overloading Function Templates

Function templates and overloading are intimately related. In Section 6.19, you learned that when overloaded functions perform *identical* operations on *different* types of data, they can be expressed more compactly and conveniently using function templates. You can then write function calls with different types of arguments and let the compiler generate separate *function-template specializations* to handle each function call appropriately. The function-template specializations generated from a given function template all have the same name, so the compiler uses overload resolution to invoke the proper function.

You may also *overload* function templates. For example, you can provide other function templates that specify the *same* function name but *different* function parameters. A function template also can be overloaded by providing nontemplate functions with the same function name but different function parameters.

Matching Process for Overloaded Functions

The compiler performs a matching process to determine what function to call when a function is invoked. It looks at both existing functions and function templates to locate a function or generate a function-template specialization whose function name and argument types are consistent with those of the function call. If there are no matches, the compiler issues an error message. If there are multiple matches for the function call, the compiler attempts to determine the *best* match. If there's *more than one* best match, the call is *ambiguous* and the compiler issues an error message.[2]

18.7 Wrap-Up

This chapter discussed class templates and class-template specializations. We used a class template to create a group of related class-template specializations that each perform identical processing on different data types. We discussed nontype template parameters. We also discussed how to overload a function template to create a customized version that handles a particular data type's processing in a manner that differs from the other function-template specializations. In the next chapter, we demonstrate how to create your own custom templatized dynamic data structures, including linked lists, stacks, queues and binary trees.

2. The compiler's process for resolving function calls is complex. The complete details are discussed in Section 13.3.3 of the C++ standard.

Summary

Section 18.1 Introduction

- Templates enable us to specify a range of related (overloaded) functions—called function-template specializations (p. 766)—or a range of related classes—called class-template specializations (p. 766).

Section 18.2 Class Templates

- Class templates provide the means for describing a class generically and for instantiating classes that are type-specific versions of this generic class.

- Class templates are called parameterized types (p. 766); they require type parameters to specify how to customize a generic class template to form a specific class-template specialization.
- To use class-template specializations you write one class template. When you need a new type-specific class, the compiler writes the source code for the class-template specialization.
- A class-template definition (p. 766) looks like a conventional class definition, but it's preceded by `template<typename T>` (or `template<class T>`) to indicate this is a class-template definition. `T` is a type parameter that acts as a placeholder for the type of the class to create. The type `T` is mentioned throughout the class definition and member-function definitions as a generic type name.
- The names of template parameters must be unique inside a template definition.
- Member-function definitions outside a class template each begin with the same `template` declaration as their class. Then, each function definition resembles a conventional function definition, except that the generic data in the class always is listed generically as type parameter `T`. The binary scope-resolution operator is used with the class-template name to tie each member-function definition to the class template's scope.

Section 18.4 Nontype Parameters

- It's possible to use nontype parameters (p. 773) in a class or function template declaration.

Section 18.5 Default Arguments for Template Type Parameters

- You can specify a default type argument (p. 773) for a type parameter in the type-parameter list.

Section 18.6 Overloading Function Templates

- A function template may be overloaded in several ways. We can provide other function templates that specify the same function name but different function parameters. A function template can also be overloaded by providing other nontemplate functions with the same function name, but different function parameters. If both the template and non-template versions match a call, the non-template version will be used.

Self-Review Exercises

18.1 State which of the following are *true* and which are *false*. If *false*, explain why.

a) Keywords `typename` and `class` as used with a template type parameter specifically mean "any user-defined class type."

b) A function template can be overloaded by another function template with the same function name.

c) Template parameter names among template definitions must be unique.

d) Each member-function definition outside its corresponding class template definition must begin with `template` and the same template parameters as its class template.

18.2 Fill in the blanks in each of the following:

a) Templates enable us to specify, with a single code segment, an entire range of related functions called _____, or an entire range of related classes called _____.

b) All template definitions begin with the keyword _____, followed by a list of template parameters enclosed in _____.

c) The related functions generated from a function template all have the same name, so the compiler uses _____ resolution to invoke the proper function.

d) Class templates also are called _____ types.

e) The _____ operator is used with a class-template name to tie each member-function definition to the class template's scope.

Answers to Self-Review Exercises

18.1 a) False. Keywords `typename` and `class` in this context also allow for a type parameter of a fundamental type. b) True. c) False. Template parameter names among function templates need not be unique. d) True.

18.2 a) function-template specializations, class-template specializations. b) `template`, angle brackets (< and >). c) overload. d) parameterized. e) scope resolution.

Exercises

18.3 *(Operator Overloads in Templates)* Write a simple function template for predicate function `isEqualTo` that compares its two arguments of the same type with the equality operator (==) and returns `true` if they are equal and `false` otherwise. Use this function template in a program that calls `isEqualTo` only with a variety of fundamental types. Now write a separate version of the program that calls `isEqualTo` with a user-defined class type, but does not overload the equality operator. What happens when you attempt to run this program? Now overload the equality operator (with the operator function) `operator==`. Now what happens when you attempt to run this program?

18.4 *(`Array` Class Template)* Reimplement class `Array` from Figs. 10.10–10.11 as a class template. Demonstrate the new `Array` class template in a program.

18.5 Distinguish between the terms "function template" and "function-template specialization."

18.6 Explain which is more like a stencil—a class template or a class-template specialization?

18.7 What's the relationship between function templates and overloading?

18.8 The compiler performs a matching process to determine which function-template specialization to call when a function is invoked. Under what circumstances does an attempt to make a match result in a compile error?

18.9 Why is it appropriate to refer to a class template as a parameterized type?

18.10 Explain why a C++ program would use the statement

```
Array< Employee > workerList( 100 );
```

18.11 Review your answer to Exercise 18.10. Explain why a C++ program might use the statement

```
Array< Employee > workerList;
```

18.12 Explain the use of the following notation in a C++ program:

```
template< typename T > Array< T >::Array( int s )
```

18.13 Why might you use a nontype parameter with a class template for a container such as an array or stack?

Custom Templatized Data Structures

19

'Will you walk a little faster?' said a whiting to a snail, 'There's a porpoise close behind us, and he's treading on my tail.'
—Lewis Carroll

There is always room at the top.
—Daniel Webster

Push on—keep moving.
—Thomas Morton

I'll turn over a new leaf.
—Miguel de Cervantes

Objectives

In this chapter you'll:

- Form linked data structures using pointers, self-referential classes and recursion.
- Create and manipulate dynamic data structures such as linked lists, queues, stacks and binary trees.
- Use binary search trees for high-speed searching and sorting.
- Learn important applications of linked data structures.
- Create reusable data structures with class templates, inheritance and composition.

Outline

19.1 Introduction

We've studied *fixed-size data structures*—such as one- and two-dimensional template-based `array`s (Chapter 7) and built-in arrays (Chapter 8)—and various C++ Standard Library *dynamic data structures* (`vector`s in Chapter 7 and other template-based containers in Chapter 15) that can grow and shrink during execution.

In this chapter, we demonstrate how you can create your own custom templatized dynamic data structures. We discuss several popular and important data structures and implement programs that create and manipulate them:

- **Linked lists** are collections of data items logically "lined up in a row"—insertions and removals are made *anywhere* in a linked list.
- *Stacks* are important in compilers and operating systems: Insertions and removals are made *only* at one end of a stack—its *top*.
- *Queues* represent *waiting lines*; insertions are made at the *back* (also referred to as the *tail*) of a queue and removals are made from the *front* (also referred to as the *head*) of a queue.
- **Binary trees** facilitate searching and sorting data, *duplicate elimination* and *compiling* expressions into machine code.

Each of these data structures has many other interesting applications. We use class templates, inheritance and composition to create and package these data structures for reusability and maintainability. The programs employ extensive pointer manipulation. The exercises include a rich collection of useful applications.

Always Prefer the Standard Library's Containers, Iterators and Algorithms, if Possible

The C++ Standard Library's *containers*, *iterators* for traversing those containers and *algorithms* for processing the containers' elements meet the needs of most C++ programmers. The Standard Library code is carefully written to be correct, portable, efficient and extensible. Understanding how to build custom templatized data structures will also help you use the Standard Library containers, iterators and algorithms, more effectively.

Special Section: Building Your Own Compiler

We encourage you to attempt the optional project described in the Special Section: Building Your Own Compiler (`www.deitel.com/books/cpphtp9`). You've been using a C++ compiler to translate your programs to machine code so that you can execute these programs on your computer. In this project, you'll actually build your own compiler. It will

read a file of statements written in a simple, yet powerful, high-level language similar to early versions of BASIC. Your compiler will translate these statements into a file of Simpletron Machine Language (SML) instructions—SML is the language you learned in the Chapter 8 Special Section: Building Your Own Computer. Your Simpletron Simulator program will then execute the SML program produced by your compiler! The special section discusses the high-level language and the algorithms you'll need to convert each type of high-level language statement into machine code. We provide compiler-theory exercises and in the special section suggest enhancements to both the compiler and the Simpletron Simulator.

19.2 Self-Referential Classes

A **self-referential class** contains a member that points to a class object of the same class type. For example, the definition

```
class Node
{
public:
   explicit Node( int ); // constructor
   void setData( int ); // set data member
   int getData() const; // get data member
   void setNextPtr( Node * ); // set pointer to next Node
   Node *getNextPtr() const; // get pointer to next Node
private:
   int data; // data stored in this Node
   Node *nextPtr; // pointer to another object of same type
}; // end class Node
```

defines a type, `Node`. Type `Node` has two `private` data members—integer member `data` and pointer member `nextPtr`. Member `nextPtr` points to an object of type `Node`—an object of the *same* type as the one being declared here, hence the term *self-referential class*. Member `nextPtr` is referred to as a **link**—i.e., `nextPtr` can "tie" an object of type `Node` to another object of the *same* type. Type `Node` also has five member functions—a constructor that receives an integer to initialize member `data`, a `setData` function to set the value of member `data`, a `getData` function to return the value of member `data`, a `setNextPtr` function to set the value of member `nextPtr` and a `getNextPtr` function to return the value of member `nextPtr`.

Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees. Figure 19.1 illustrates two self-referential class objects linked together to form a list. Note that a slash—representing a null pointer (`nullptr`)—is placed in the link member of the second self-referential class object to indicate that the link does *not* point to another object. The slash is for illustration purposes only; it does *not* correspond to the backslash character in C++. A null pointer normally indicates the *end of a data structure*.

Common Programming Error 19.1

Not setting the link in the last node of a linked data structure to `nullptr` is a (possibly fatal) logic error.

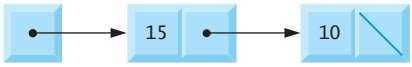

Fig. 19.1 | Two self-referential class objects linked together.

The following sections discuss lists, stacks, queues and trees. The data structures presented in this chapter are created and maintained with *dynamic memory allocation* (Section 10.9), *self-referential classes*, *class templates* (Chapter 18) and *function templates* (Section 6.19).

19.3 Linked Lists

A linked list is a *linear* collection of self-referential class objects, called **nodes**, connected by **pointer links**—hence, the term "linked" list. A linked list is accessed via a pointer to the list's first node. Each subsequent node is accessed via the *link-pointer member* stored in the previous node. By *convention*, the link pointer in the last node of a list is set to `nullptr` to mark the end of the list. Data is stored in a linked list *dynamically*—each node is created and destroyed as necessary. A node can contain data of any type, including objects of other classes. If nodes contain base-class pointers to base-class and derived-class objects related by inheritance, we can have a linked list of such nodes and process them *polymorphically* using `virtual` function calls. Stacks and queues are also **linear data structures** and, as we'll see, can be viewed as constrained versions of linked lists. Trees are **nonlinear data structures**.

Linked lists provide several advantages over `array` objects and built-in arrays. A linked list is appropriate when the number of data elements to be represented at one time is *unpredictable*. Linked lists are dynamic, so the length of a list can increase or decrease as necessary. The size of an `array` object or built-in array, however, cannot be altered, because the array size is fixed at compile time. An `array` object or built-in array can become full. Linked lists become full only when the system has insufficient memory to satisfy additional dynamic storage allocation requests.

Performance Tip 19.1

An `array` object or built-in array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime. Class template `vector` (Section 7.10) implements a dynamically resizable array-based data structure.

Linked lists can be maintained in *sorted order* by inserting each new element at the proper point in the list. Existing list elements do *not* need to be moved. Pointers merely need to be updated to point to the correct node.

Performance Tip 19.2

Insertion and deletion in a sorted `array` object or built-in array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere *in the list.*

Performance Tip 19.3

The elements of an `array` *object or built-in array are stored contiguously in memory. This allows* immediate *access to any element, because an element's address can be calculated directly based on its position relative to the beginning of the* `array` *object or built-in array. Linked lists do not afford such immediate* direct access *to their elements, so accessing individual elements can be considerably more expensive. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, if you have a pointer to the insertion location, it's typically more efficient to insert an item in a sorted linked list than a sorted* `array` *object or built-in array.*

Linked-list nodes typically are *not* stored contiguously in memory, but logically they *appear* to be contiguous. Figure 19.2 illustrates a linked list with several nodes.

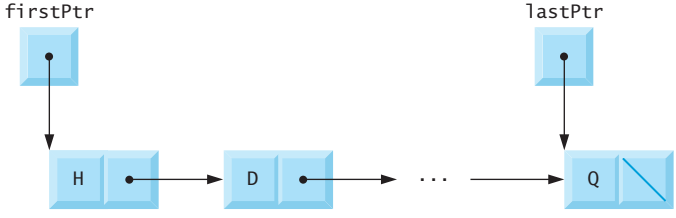

Fig. 19.2 | A graphical representation of a list.

Performance Tip 19.4

Using dynamic memory allocation for data structures that grow and shrink at execution time can save memory.

Testing Our Linked List Implementation

The program of Figs. 19.3–19.5 uses a `List` *class template* to manipulate a list of integer values and a list of floating-point values. The driver program (Fig. 19.3) has five options:

- insert a value at the beginning of the `List`
- insert a value at the end of the `List`
- delete a value from the beginning of the `List`
- delete a value from the end of the `List`
- end the `List` processing

The linked list implementation we present here does not allow insertions and deletions anywhere in the linked list. We ask you to implement these operations in Exercise 19.26. Exercise 19.20 asks you to implement a recursive function that prints a linked list backward, and Exercise 19.21 asks you to implement a recursive function that searches a linked list for a particular data item.

In Fig. 19.3, Lines 69 and 73 create `List` objects for types `int` and `double`, respectively. Lines 70 and 74 invoke the `testList` function template to manipulate objects.

```
// Fig. 19.3: fig19_03.cpp
// Manipulating a linked list.
#include <iostream>
#include <string>
#include "List.h" // List class definition
using namespace std;

// display program instructions to user
void instructions()
{
   cout << "Enter one of the following:\n"
      << "  1 to insert at beginning of list\n"
      << "  2 to insert at end of list\n"
      << "  3 to delete from beginning of list\n"
      << "  4 to delete from end of list\n"
      << "  5 to end list processing\n";
} // end function instructions

// function to test a List
template< typename T >
void testList( List< T > &listObject, const string &typeName )
{
   cout << "Testing a List of " << typeName << " values\n";
   instructions(); // display instructions

   int choice; // store user choice
   T value; // store input value

   do // perform user-selected actions
   {
      cout << "? ";
      cin >> choice;

      switch ( choice )
      {
         case 1: // insert at beginning
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtFront( value );
            listObject.print();
            break;
         case 2: // insert at end
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3: // remove from beginning
            if ( listObject.removeFromFront( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
```

Fig. 19.3 | Manipulating a linked list. (Part 1 of 3.)

```
54            case 4: // remove from end
55               if ( listObject.removeFromBack( value ) )
56                  cout << value << " removed from list\n";
57
58               listObject.print();
59               break;
60         } // end switch
61      } while ( choice < 5 ); // end do...while
62
63      cout << "End list test\n\n";
64   } // end function testList
65
66   int main()
67   {
68      // test List of int values
69      List< int > integerList;
70      testList( integerList, "integer" );
71
72      // test List of double values
73      List< double > doubleList;
74      testList( doubleList, "double" );
75   } // end main
```

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3
```

Fig. 19.3 | Manipulating a linked list. (Part 2 of 3.)

```
? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Fig. 19.3 | Manipulating a linked list. (Part 3 of 3.)

Class Template `ListNode`

Figure 19.3 uses class templates `ListNode` (Fig. 19.4) and `List` (Fig. 19.5). Encapsulated in each `List` object is a linked list of `ListNode` objects. Class template `ListNode` (Fig. 19.4) contains `private` members `data` and `nextPtr` (lines 27–28), a constructor (lines 16–20) to initialize these members and function `getData` (lines 22–25) to return the data in a node. Member `data` stores a value of type `NODETYPE`, the type parameter passed to the class tem-

plate. Member `nextPtr` stores a pointer to the next `ListNode` object in the linked list. Line 13 of the `ListNode` class template definition declares class `List<NODETYPE>` as a `friend`. This makes all member functions of a given specialization of class template `List` friends of the corresponding specialization of class template `ListNode`, so they can access the `private` members of `ListNode` objects of that type. We do this for performance and because these two classes are tightly coupled—only class template `List` manipulates objects of class template `ListNode`. Because the `ListNode` template parameter `NODETYPE` is used as the template argument for `List` in the `friend` declaration, `ListNode`s specialized with a particular type can be processed only by a `List` specialized with the *same* type (e.g., a `List` of `int` values manages `ListNode` objects that store `int` values). To use the type name `List<NODETYPE>` in line 13, the compiler needs to know that class template `List` exists. Line 8 is a so-called forward declaration of class template `List`. A **forward declaration** tells the compiler that a type exists, even if it has not yet been defined.

Error-Prevention Tip 19.1

Assign `nullptr` to the link member of a new node. Pointers must be initialized before they're used.

```
1  // Fig. 19.4: ListNode.h
2  // ListNode class-template definition.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // forward declaration of class List required to announce that class
7  // List exists so it can be used in the friend declaration at line 13
8  template< typename NODETYPE > class List;
9
10 template< typename NODETYPE >
11 class ListNode
12 {
13    friend class List< NODETYPE >; // make List a friend
14
15 public:
16    explicit ListNode( const NODETYPE &info ) // constructor
17       : data( info ), nextPtr( nullptr )
18    {
19       // empty body
20    } // end ListNode constructor
21
22    NODETYPE getData() const; // return data in node
23    {
24       return data;
25    } // end function getData
26 private:
27    NODETYPE data; // data
28    ListNode< NODETYPE > *nextPtr; // next node in list
29 }; // end class ListNode
30
31 #endif
```

Fig. 19.4 | `ListNode` class-template definition.

Class Template `List`

Lines 148–149 of the `List` class template (Fig. 19.5) declare `private` data members `firstPtr` and `lastPtr`—pointers to the `List`'s first and last `ListNode`s. The default constructor (lines 14–18) initializes both pointers to `nullptr`. The destructor (lines 21–40) destroys all of the `List`'s `ListNode` objects when the `List` is destroyed. The primary `List` functions are `insertAtFront` (lines 43–54), `insertAtBack` (lines 57–68), `removeFromFront` (lines 71–88) and `removeFromBack` (lines 91–117). We discuss each of these after Fig. 19.5.

Function `isEmpty` (lines 120–123) is called a *predicate function*—it *does not alter* the `List`; rather, it determines whether the `List` is *empty*. If so, `true` is returned; otherwise, `false` is returned. Function `print` (lines 126–145) displays the `List`'s contents. Utility function `getNewNode` (lines 152–155) returns a dynamically allocated `ListNode` object. This function is called from functions `insertAtFront` and `insertAtBack`.

```
1  // Fig. 19.5: List.h
2  // List class-template definition.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7  #include "ListNode.h" // ListNode class definition
8
9  template< typename NODETYPE >
10 class List
11 {
12 public:
13    // default constructor
14    List()
15       : firstPtr( nullptr ), lastPtr( nullptr )
16    {
17       // empty body
18    } // end List constructor
19
20    // destructor
21    ~List()
22    {
23       if ( !isEmpty() ) // List is not empty
24       {
25          std::cout << "Destroying nodes ...\n";
26
27          ListNode< NODETYPE > *currentPtr = firstPtr;
28          ListNode< NODETYPE > *tempPtr = nullptr;
29
30          while ( currentPtr != nullptr ) // delete remaining nodes
31          {
32             tempPtr = currentPtr;
33             std::cout << tempPtr->data << '\n';
34             currentPtr = currentPtr->nextPtr;
35             delete tempPtr;
36          } // end while
37       } // end if
```

Fig. 19.5 | `List` class-template definition. (Part 1 of 4.)

```
38
39          std::cout << "All nodes destroyed\n\n";
40       } // end List destructor
41
42       // insert node at front of list
43       void insertAtFront( const NODETYPE &value )
44       {
45          ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
46
47          if ( isEmpty() ) // List is empty
48             firstPtr = lastPtr = newPtr; // new list has only one node
49          else // List is not empty
50          {
51             newPtr->nextPtr = firstPtr; // point new node to old 1st node
52             firstPtr = newPtr; // aim firstPtr at new node
53          } // end else
54       } // end function insertAtFront
55
56       // insert node at back of list
57       void insertAtBack( const NODETYPE &value )
58       {
59          ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
60
61          if ( isEmpty() ) // List is empty
62             firstPtr = lastPtr = newPtr; // new list has only one node
63          else // List is not empty
64          {
65             lastPtr->nextPtr = newPtr; // update previous last node
66             lastPtr = newPtr; // new last node
67          } // end else
68       } // end function insertAtBack
69
70       // delete node from front of list
71       bool removeFromFront( NODETYPE &value )
72       {
73          if ( isEmpty() ) // List is empty
74             return false; // delete unsuccessful
75          else
76          {
77             ListNode< NODETYPE > *tempPtr = firstPtr; // hold item to delete
78
79             if ( firstPtr == lastPtr )
80                firstPtr = lastPtr = nullptr; // no nodes remain after removal
81             else
82                firstPtr = firstPtr->nextPtr; // point to previous 2nd node
83
84             value = tempPtr->data; // return data being removed
85             delete tempPtr; // reclaim previous front node
86             return true; // delete successful
87          } // end else
88       } // end function removeFromFront
89
```

Fig. 19.5 | `List` class-template definition. (Part 2 of 4.)

```
90    // delete node from back of list
91    bool removeFromBack( NODETYPE &value )
92    {
93       if ( isEmpty() ) // List is empty
94          return false; // delete unsuccessful
95       else
96       {
97          ListNode< NODETYPE > *tempPtr = lastPtr; // hold item to delete
98
99          if ( firstPtr == lastPtr ) // List has one element
100            firstPtr = lastPtr = nullptr; // no nodes remain after removal
101         else
102         {
103            ListNode< NODETYPE > *currentPtr = firstPtr;
104
105            // locate second-to-last element
106            while ( currentPtr->nextPtr != lastPtr )
107               currentPtr = currentPtr->nextPtr; // move to next node
108
109            lastPtr = currentPtr; // remove last node
110            currentPtr->nextPtr = nullptr; // this is now the last node
111         } // end else
112
113         value = tempPtr->data; // return value from old last node
114         delete tempPtr; // reclaim former last node
115         return true; // delete successful
116      } // end else
117   } // end function removeFromBack
118
119   // is List empty?
120   bool isEmpty() const
121   {
122      return firstPtr == nullptr;
123   } // end function isEmpty
124
125   // display contents of List
126   void print() const
127   {
128      if ( isEmpty() ) // List is empty
129      {
130         std::cout << "The list is empty\n\n";
131         return;
132      } // end if
133
134      ListNode< NODETYPE > *currentPtr = firstPtr;
135
136      std::cout << "The list is: ";
137
138      while ( currentPtr != nullptr ) // get element data
139      {
140         std::cout << currentPtr->data << ' ';
141         currentPtr = currentPtr->nextPtr;
142      } // end while
```

Fig. 19.5 | `List` class-template definition. (Part 3 of 4.)

```
143
144        std::cout << "\n\n";
145     } // end function print
146
147  private:
148     ListNode< NODETYPE > *firstPtr; // pointer to first node
149     ListNode< NODETYPE > *lastPtr; // pointer to last node
150
151     // utility function to allocate new node
152     ListNode< NODETYPE > *getNewNode( const NODETYPE &value )
153     {
154        return new ListNode< NODETYPE >( value );
155     } // end function getNewNode
156  }; // end class List
157
158  #endif
```

Fig. 19.5 | `List` class-template definition. (Part 4 of 4.)

Member Function `insertAtFront`

Over the next several pages, we discuss each of the member functions of class `List` in detail. Function `insertAtFront` (Fig. 19.5, lines 43–54) places a new node at the front of the list. The function consists of several steps:

1. Call function `getNewNode` (line 45), passing it `value`, which is a constant reference to the node value to be inserted.
2. Function `getNewNode` (lines 152–155) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtFront` (line 45).
3. If the list is *empty* (line 47), `firstPtr` and `lastPtr` are set to `newPtr` (line 48)—i.e., the first and last node are the same node.
4. If the list is *not empty* (line 49), then the node pointed to by `newPtr` is threaded into the list by copying `firstPtr` to `newPtr->nextPtr` (line 51), so that the new node points to what used to be the first node of the list, and copying `newPtr` to `firstPtr` (line 52), so that `firstPtr` now points to the new first node of the list.

Figure 19.6 illustrates function `insertAtFront`. Part (a) shows the list and the new node before calling `insertAtFront`. The dashed arrows in part (b) illustrate *Step 4* of the `insertAtFront` operation that enables the node containing `12` to become the new list front.

Member Function `insertAtBack`

Function `insertAtBack` (Fig. 19.5, lines 57–68) places a new node at the back of the list. The function consists of several steps:

1. Call function `getNewNode` (line 59), passing it `value`, which is a constant reference to the node value to be inserted.
2. Function `getNewNode` (lines 152–155) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtBack` (line 59).

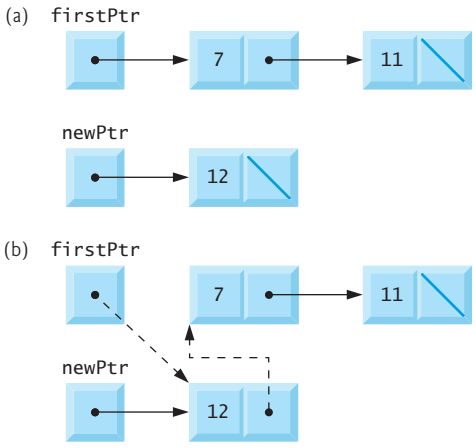

Fig. 19.6 | Operation `insertAtFront` represented graphically.

3. If the list is *empty* (line 61), then both `firstPtr` and `lastPtr` are set to `newPtr` (line 62).
4. If the list is *not empty* (line 63), then the node pointed to by `newPtr` is threaded into the list by copying `newPtr` into `lastPtr->nextPtr` (line 65), so that the new node is pointed to by what used to be the last node of the list, and copying `newPtr` to `lastPtr` (line 66), so that `lastPtr` now points to the new last node of the list.

Figure 19.7 illustrates an `insertAtBack` operation. Part (a) of the figure shows the list and the new node before the operation. The dashed arrows in part (b) illustrate *Step 4* of function `insertAtBack` that enables a new node to be added to the end of a list that's not empty.

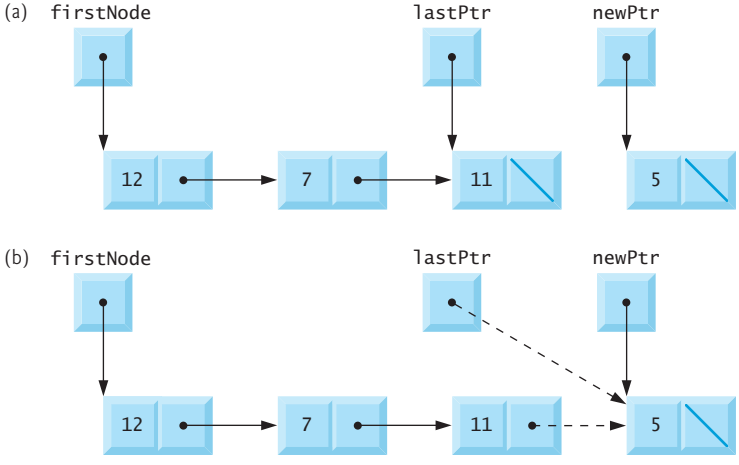

Fig. 19.7 | Operation `insertAtBack` represented graphically.

Member Function **`removeFromFront`**

Function `removeFromFront` (Fig. 19.5, lines 71–88) removes the front node of the list and copies the node value to the reference parameter. The function returns `false` if an attempt is made to remove a node from an empty list (lines 73–74) and returns `true` if the removal is successful. The function consists of several steps:

1. Assign `tempPtr` the address to which `firstPtr` points (line 77). Eventually, `tempPtr` will be used to delete the node being removed.
2. If `firstPtr` is equal to `lastPtr` (line 79), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to `nullptr` (line 80) to dethread that node from the list (leaving the list empty).
3. If the list has more than one node prior to removal, then leave `lastPtr` as is and set `firstPtr` to `firstPtr->nextPtr` (line 82); i.e., modify `firstPtr` to point to what was the second node prior to removal (and is now the new first node).
4. After all these pointer manipulations are complete, copy to reference parameter `value` the `data` member of the node being removed (line 84).
5. Now `delete` the node pointed to by `tempPtr` (line 85).
6. Return `true`, indicating successful removal (line 86).

Figure 19.8 illustrates function `removeFromFront`. Part (a) illustrates the list before the removal operation. Part (b) shows the actual pointer manipulations for removing the front node from a nonempty list.

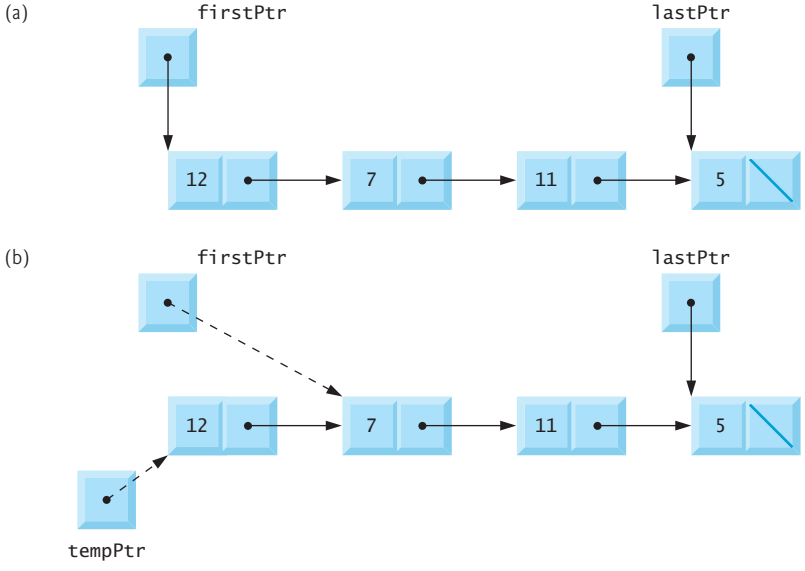

Fig. 19.8 | Operation `removeFromFront` represented graphically.

Member Function `removeFromBack`

Function `removeFromBack` (Fig. 19.5, lines 91–117) removes the back node of the list and copies the node value to the reference parameter. The function returns `false` if an attempt is made to remove a node from an empty list (lines 93–94) and returns `true` if the removal is successful. The function consists of several steps:

1. Assign to `tempPtr` the address to which `lastPtr` points (line 97). Eventually, `tempPtr` will be used to delete the node being removed.
2. If `firstPtr` is equal to `lastPtr` (line 99), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to `nullptr` (line 100) to dethread that node from the list (leaving the list empty).
3. If the list has more than one node prior to removal, then assign `currentPtr` the address to which `firstPtr` points (line 103) to prepare to "walk the list."
4. Now "walk the list" with `currentPtr` until it points to the node before the last node. This node will become the last node after the remove operation completes. This is done with a `while` loop (lines 106–107) that keeps replacing `currentPtr` by `currentPtr->nextPtr`, while `currentPtr->nextPtr` is not `lastPtr`.
5. Assign `lastPtr` to the address to which `currentPtr` points (line 109) to dethread the back node from the list.
6. Set `currentPtr->nextPtr` to `nullptr` (line 110) in the new last node of the list.
7. After all the pointer manipulations are complete, copy to reference parameter `value` the `data` member of the node being removed (line 113).
8. Now `delete` the node pointed to by `tempPtr` (line 114).
9. Return `true` (line 115), indicating successful removal.

Figure 19.9 illustrates `removeFromBack`. Part (a) of the figure illustrates the list before the removal operation. Part (b) of the figure shows the actual pointer manipulations.

Member Function `print`

Function `print` (lines 126–145) first determines whether the list is *empty* (line 128). If so, it prints `"The list is empty"` and returns (lines 130–131). Otherwise, it iterates through the list and outputs the value in each node. The function initializes `currentPtr` as a copy of `firstPtr` (line 134), then prints the string `"The list is: "` (line 136). While `currentPtr` is not `nullptr` (line 138), `currentPtr->data` is printed (line 140) and `currentPtr` is assigned the value of `currentPtr->nextPtr` (line 141). Note that if the link in the last node of the list does not have the value `nullptr`, the printing algorithm will erroneously attempt to print past the end of the list. Our printing algorithm here is identical for linked lists, stacks and queues (because we base each of these data structures on the same linked list infrastructure).

Circular Linked Lists and Double Linked Lists

The kind of linked list we've been discussing is a **singly linked list**—the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence." This list terminates with a node whose pointer member has the value `nullptr`. A singly linked list may be traversed in only *one* direction.

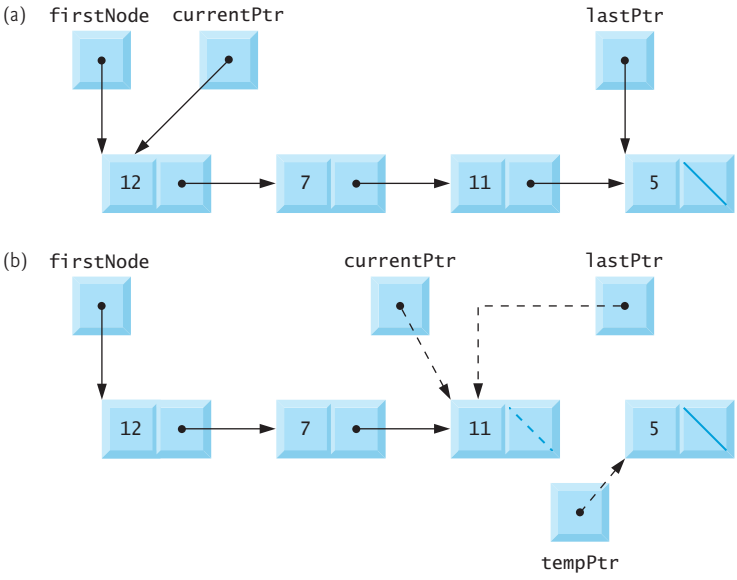

Fig. 19.9 | Operation `removeFromBack` represented graphically.

A **circular, singly linked list** (Fig. 19.10) begins with a pointer to the first node, and each node contains a pointer to the next node. The "last node" does not contain `nullptr`; rather, the pointer in the last node points back to the first node, thus closing the "circle."

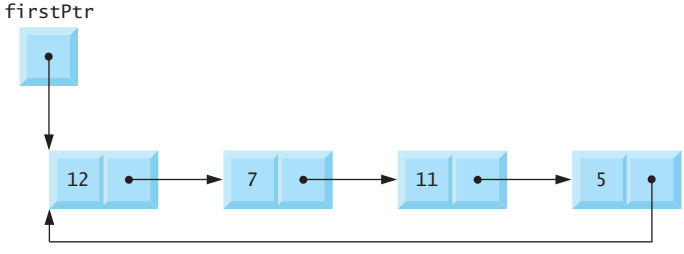

Fig. 19.10 | Circular, singly linked list.

A **doubly linked list** (Fig. 19.11)—such as the Standard Library `list` class template—allows traversals *both forward and backward*. Such a list is often implemented with two "start pointers"—one that points to the first element of the list to allow *front-to-back traversal* of the list and one that points to the last element to allow *back-to-front traversal*. Each node has *both* a *forward pointer* to the next node in the list in the forward direction *and* a *backward pointer* to the next node in the list in the backward direction. If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might best begin from the front of the list. Searching for someone whose name begins with a letter near the end of the alphabet might best begin from the back of the list.

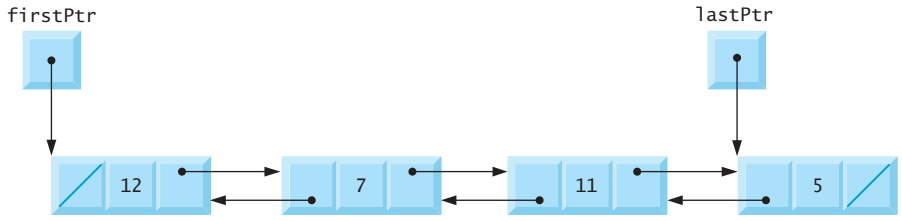

Fig. 19.11 | Doubly linked list.

In a **circular, doubly linked list** (Fig. 19.12), the *forward pointer* of the last node points to the first node, and the *backward pointer* of the first node points to the last node, thus closing the "circle."

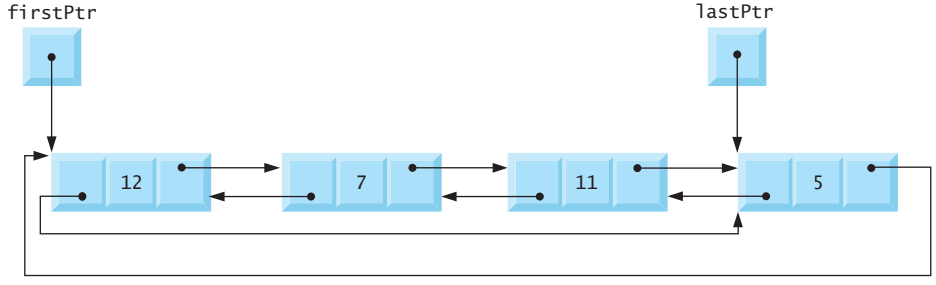

Fig. 19.12 | Circular, doubly linked list.

19.4 Stacks

You learned the notion of a stack in Section 6.12, Section 15.7.1, `stack` Adapter and Section 18.2. Recall that a nodes can be added to a stack and removed from a stack only at its *top*, so a stack is referred to as a *last-in, first-out (LIFO)* data structure. One way to implement a stack is as a *constrained version* of a linked list. In such an implementation, the link member in the last node of the stack is set to `nullptr` to indicate the *bottom* of the stack.

The primary member functions used to manipulate a stack are `push` and `pop`. Function `push` *inserts* a new node at the top of the stack. Function `pop` *removes* a node from the top of the stack, stores the popped value in a reference variable that's passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

Applications of Stacks

Stacks have many interesting applications:

- In Section 6.12, you learned that when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its

caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

- Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function. When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code. The exercises explore several applications of stacks, including using them to develop your own complete working compiler.

Taking Advantage of the Relationship Between `Stack` *and* `List`

We'll take advantage of the close relationship between lists and stacks to implement a stack class primarily by *reusing* our `List` class template. First, we'll implement the `Stack` class template via `private` *inheritance* from our `List` class template. Then we'll implement an identically performing `Stack` class template through *composition* by including a `List` object as a `private` member of a `Stack` class template.

Implementing a Class Template `Stack` *Class Based By Inheriting from* `List`

The program of Figs. 19.13–19.14 creates a `Stack` class template (Fig. 19.13) primarily through `private` *inheritance* (line 9) of the `List` class template of Fig. 19.5. We want the `Stack` to have member functions `push` (lines 13–16), `pop` (lines 19–22), `isStackEmpty` (lines 25–28) and `printStack` (lines 31–34). Note that these are essentially the `insertAtFront`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template. Of course, the `List` class template contains other member functions (i.e., `insertAtBack` and `removeFromBack`) that we would not want to make accessible through the `public` interface to the `Stack` class. So when we indicate that the `Stack` class template is to inherit from the `List` class template, we specify `private` *inheritance*. This makes all the `List` class template's member functions `private` in the `Stack` class template. When we implement the `Stack`'s member functions, we then have each of these call the appropriate member function of the `List` class—`push` calls `insertAtFront` (line 15), `pop` calls `removeFromFront` (line 21), `isStackEmpty` calls `isEmpty` (line 27) and `printStack` calls `print` (line 33)—this is referred to as **delegation**.

```
1  // Fig. 19.13: Stack.h
2  // Stack class-template definition.
3  #ifndef STACK_H
4  #define STACK_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack : private List< STACKTYPE >
10 {
```

Fig. 19.13 | Stack class-template definition. (Part 1 of 2.)

```
11   public:
12      // push calls the List function insertAtFront
13      void push( const STACKTYPE &data )
14      {
15         insertAtFront( data );
16      } // end function push
17
18      // pop calls the List function removeFromFront
19      bool pop( STACKTYPE &data )
20      {
21         return removeFromFront( data );
22      } // end function pop
23
24      // isStackEmpty calls the List function isEmpty
25      bool isStackEmpty() const
26      {
27         return this->isEmpty();
28      } // end function isStackEmpty
29
30      // printStack calls the List function print
31      void printStack() const
32      {
33         this->print();
34      } // end function print
35   }; // end class Stack
36
37   #endif
```

Fig. 19.13 | Stack class-template definition. (Part 2 of 2.)

Dependent Names in Class Templates

The *explicit use of* `this` on lines 27 and 33 is required so the compiler can properly resolve identifiers in template definitions. A **dependent name** is an identifier that depends on a template parameter. For example, the call to `removeFromFront` (line 21) depends on the argument `data` which has a type that's dependent on the template parameter `STACKTYPE`. Resolution of *dependent names* occurs when the template is instantiated. In contrast, the identifier for a function that takes no arguments like `isEmpty` or `print` in the `List` superclass is a **non-dependent name**. Such identifiers are normally resolved at the point where the template is defined. If the template has not yet been instantiated, then the code for the function with the *non-dependent name* does not yet exist and some compilers will generate compilation errors. Adding the explicit use of `this->` in lines 27 and 33 makes the calls to the base class's member functions dependent on the template parameter and ensures that the code will compile properly.

Testing the `Stack` *Class Template*

The stack class template is used in `main` (Fig. 19.14) to instantiate integer stack `intStack` of type `Stack< int >` (line 9). Integers 0 through 2 are pushed onto `intStack` (lines 14–18), then popped off `intStack` (lines 23–28). The program uses the `Stack` class template to create `doubleStack` of type `Stack< double >` (line 30). Values 1.1, 2.2 and 3.3 are pushed onto `doubleStack` (lines 36–41), then popped off `doubleStack` (lines 46–51).

```
// Fig. 19.14: fig19_14.cpp
// A simple stack program.
#include <iostream>
#include "Stack.h" // Stack class definition
using namespace std;

int main()
{
   Stack< int > intStack; // create Stack of ints

   cout << "processing an integer Stack" << endl;

   // push integers onto intStack
   for ( int i = 0; i < 3; ++i )
   {
      intStack.push( i );
      intStack.printStack();
   } // end for

   int popInteger; // store int popped from stack

   // pop integers from intStack
   while ( !intStack.isStackEmpty() )
   {
      intStack.pop( popInteger );
      cout << popInteger << " popped from stack" << endl;
      intStack.printStack();
   } // end while

   Stack< double > doubleStack; // create Stack of doubles
   double value = 1.1;

   cout << "processing a double Stack" << endl;

   // push floating-point values onto doubleStack
   for ( int j = 0; j < 3; ++j )
   {
      doubleStack.push( value );
      doubleStack.printStack();
      value += 1.1;
   } // end for

   double popDouble; // store double popped from stack

   // pop floating-point values from doubleStack
   while ( !doubleStack.isStackEmpty() )
   {
      doubleStack.pop( popDouble );
      cout << popDouble << " popped from stack" << endl;
      doubleStack.printStack();
   } // end while
} // end main
```

Fig. 19.14 | A simple stack program. (Part 1 of 2.)

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 19.14 | A simple stack program. (Part 2 of 2.)

Implementing a Class Template `Stack` *Class With Composition of a* `List` *Object*

Another way to implement a `Stack` class template is by reusing the `List` class template through *composition*. Figure 19.15 is a new implementation of the `Stack` class template that contains a `List< STACKTYPE >` object called `stackList` (line 38). This version of the `Stack` class template uses class `List` from Fig. 19.5. To test this class, use the driver program in Fig. 19.14, but include the new header—`Stackcomposition.h` in line 4 of that file. The output of the program is identical for both versions of class `Stack`.

```
1  // Fig. 19.15: Stackcomposition.h
2  // Stack class template with a composed List object.
3  #ifndef STACKCOMPOSITION_H
4  #define STACKCOMPOSITION_H
5
```

Fig. 19.15 | `Stack` class template with a composed `List` object. (Part 1 of 2.)

```
#include "List.h" // List class definition

template< typename STACKTYPE >
class Stack
{
public:
   // no constructor; List constructor does initialization

   // push calls stackList object's insertAtFront member function
   void push( const STACKTYPE &data )
   {
      stackList.insertAtFront( data );
   } // end function push

   // pop calls stackList object's removeFromFront member function
   bool pop( STACKTYPE &data )
   {
      return stackList.removeFromFront( data );
   } // end function pop

   // isStackEmpty calls stackList object's isEmpty member function
   bool isStackEmpty() const
   {
      return stackList.isEmpty();
   } // end function isStackEmpty

   // printStack calls stackList object's print member function
   void printStack() const
   {
      stackList.print();
   } // end function printStack
private:
   List< STACKTYPE > stackList; // composed List object
}; // end class Stack

#endif
```

Fig. 19.15 | `Stack` class template with a composed `List` object. (Part 2 of 2.)

19.5 Queues

Recall that queue nodes are removed only from the *head* of the queue and are inserted only at the *tail* of the queue. For this reason, a queue is referred to as a first-in, *first-out (FIFO)* data structure. The insert and remove operations are known as **enqueue** and **dequeue**.

Applications of Queues

Queues have many applications in computer systems.

- Computers that have a *single* processor can service only one user at a time. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queue as users receive service. The entry at the front of the queue is the next to receive service.

- Queues are also used to support **print spooling**. For example, a single printer might be shared by all users of a network. Many users can send print jobs to the printer, even when the printer is already busy. These print jobs are placed in a queue until the printer becomes available. A program called a **spooler** manages the queue to ensure that, as each print job completes, the next print job is sent to the printer.
- Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet's final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.
- A file server in a computer network handles file access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

Implementing a Class Template `Queue` *Class Based By Inheriting from* `List`

The program of Figs. 19.16–19.17 creates a `Queue` class template (Fig. 19.16) through *`private` inheritance* (line 9) of the `List` class template from Fig. 19.5. The `Queue` has member functions `enqueue` (Fig. 19.16, lines 13–16), `dequeue` (lines 19–22), `isQueueEmpty` (lines 25–28) and `printQueue` (lines 31–34). These are essentially the `insertAtBack`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template. Of course, the `List` class template contains other member functions that we do *not* want to make accessible through the `public` interface to the `Queue` class. So when we indicate that the `Queue` class template is to inherit the `List` class template, we specify *`private` inheritance*. This makes all the `List` class template's member functions `private` in the `Queue` class template. When we implement the `Queue`'s member functions, we have each of these call the appropriate member function of the list class—`enqueue` calls `insertAtBack` (line 15), `dequeue` calls `removeFromFront` (line 21), `isQueueEmpty` calls `isEmpty` (line 27) and `printQueue` calls `print` (line 33). As with the `Stack` example in Fig. 19.13, this *delegation* requires *explicit use of the `this` pointer* in `isQueueEmpty` and `printQueue` to avoid compilation errors.

```
1  // Fig. 19.16: Queue.h
2  // Queue class-template definition.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "List.h" // List class definition
7
8  template< typename QUEUETYPE >
9  class Queue : private List< QUEUETYPE >
10 {
11 public:
12    // enqueue calls List member function insertAtBack
13    void enqueue( const QUEUETYPE &data )
14    {
15       insertAtBack( data );
16    } // end function enqueue
17
```

Fig. 19.16 | Queue class-template definition. (Part 1 of 2.)

```
18      // dequeue calls List member function removeFromFront
19      bool dequeue( QUEUETYPE &data )
20      {
21         return removeFromFront( data );
22      } // end function dequeue
23
24      // isQueueEmpty calls List member function isEmpty
25      bool isQueueEmpty() const
26      {
27         return this->isEmpty();
28      } // end function isQueueEmpty
29
30      // printQueue calls List member function print
31      void printQueue() const
32      {
33         this->print();
34      } // end function printQueue
35   }; // end class Queue
36
37   #endif
```

Fig. 19.16 | Queue class-template definition. (Part 2 of 2.)

Testing the `Queue` *Class Template*

Figure 19.17 uses the `Queue` class template to instantiate integer queue `intQueue` of type `Queue<int>` (line 9). Integers 0 through 2 are *enqueued* to `intQueue` (lines 14–18), then *dequeued* from `intQueue` in first-in, first-out order (lines 23–28). Next, the program instantiates queue `doubleQueue` of type `Queue< double >` (line 30). Values 1.1, 2.2 and 3.3 are *enqueued* to `doubleQueue` (lines 36–41), then *dequeued* from `doubleQueue` in first-in, first-out order (lines 46–51).

```
1   // Fig. 19.17: fig19_17.cpp
2   // Queue-processing program.
3   #include <iostream>
4   #include "Queue.h" // Queue class definition
5   using namespace std;
6
7   int main()
8   {
9      Queue< int > intQueue; // create Queue of integers
10
11     cout << "processing an integer Queue" << endl;
12
13     // enqueue integers onto intQueue
14     for ( int i = 0; i < 3; ++i )
15     {
16        intQueue.enqueue( i );
17        intQueue.printQueue();
18     } // end for
19
```

Fig. 19.17 | Queue-processing program. (Part 1 of 3.)

```
20      int dequeueInteger; // store dequeued integer
21
22      // dequeue integers from intQueue
23      while ( !intQueue.isQueueEmpty() )
24      {
25         intQueue.dequeue( dequeueInteger );
26         cout << dequeueInteger << " dequeued" << endl;
27         intQueue.printQueue();
28      } // end while
29
30      Queue< double > doubleQueue; // create Queue of doubles
31      double value = 1.1;
32
33      cout << "processing a double Queue" << endl;
34
35      // enqueue floating-point values onto doubleQueue
36      for ( int j = 0; j < 3; ++j )
37      {
38         doubleQueue.enqueue( value );
39         doubleQueue.printQueue();
40         value += 1.1;
41      } // end for
42
43      double dequeueDouble; // store dequeued double
44
45      // dequeue floating-point values from doubleQueue
46      while ( !doubleQueue.isQueueEmpty() )
47      {
48         doubleQueue.dequeue( dequeueDouble );
49         cout << dequeueDouble << " dequeued" << endl;
50         doubleQueue.printQueue();
51      } // end while
52   } // end main
```

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing a double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3
```

Fig. 19.17 | Queue-processing program. (Part 2 of 3.)

```
1.1 dequeued
The list is: 2.2 3.3

2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 19.17 | Queue-processing program. (Part 3 of 3.)

19.6 Trees

Linked lists, stacks and queues are linear data structures. *A tree is a nonlinear, two-dimensional data structure.* Tree nodes contain two or more links. This section discusses **binary trees** (Fig. 19.18)—trees whose nodes all contain two links (none, one or both of which may have the value `nullptr`).

Basic Terminology

For this discussion, refer to nodes `A`, `B`, `C` and `D` in Fig. 19.18. The **root node** (node `B`) is the first node in a tree. Each link in the root node refers to a **child** (nodes `A` and `D`). The **left child** (node `A`) is the root node of the **left subtree** (which contains only node `A`), and the **right child** (node `D`) is the root node of the **right subtree** (which contains nodes `D` and `C`). The children of a given node are called **siblings** (e.g., nodes `A` and `D` are siblings). A node with no children is a **leaf node** (e.g., nodes `A` and `C` are leaf nodes). Computer scientists normally draw trees from the root node down—the opposite of how trees grow in nature.

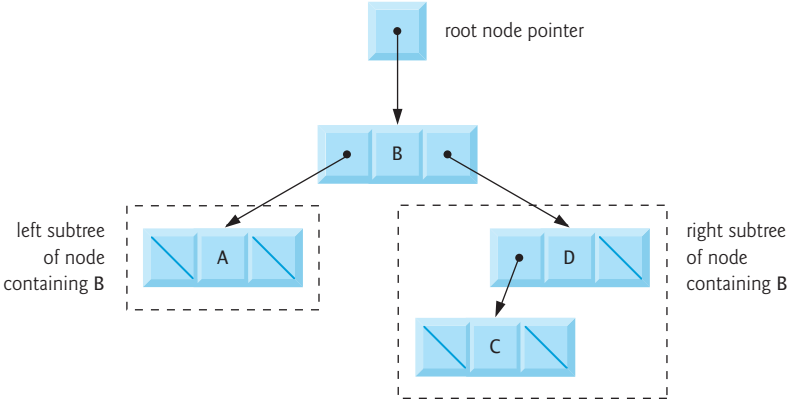

Fig. 19.18 | A graphical representation of a binary tree.

Binary Search Trees

A **binary search tree** (with *no duplicate node values*) has the characteristic that the values in any left subtree are *less than* the value in its **parent node**, and the values in any right subtree

are *greater than* the value in its parent node. Figure 19.19 illustrates a binary search tree with 9 values. Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the *order* in which the values are inserted into the tree.

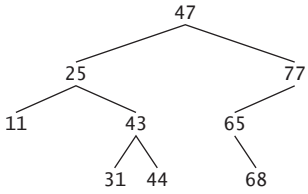

Fig. 19.19 | A binary search tree.

Implementing the Binary Search Tree Program

The program of Figs. 19.20–19.22 creates a binary search tree and traverses it (i.e., walks through all its nodes) three ways—using *recursive* **inorder**, **preorder** and **postorder traversals**. We explain these traversal algorithms shortly.

Testing the `Tree` *Class Template*

We begin our discussion with the *driver program* (Fig. 19.20), then continue with the implementations of classes `TreeNode` (Fig. 19.21) and `Tree` (Fig. 19.22). Function `main` (Fig. 19.20) begins by instantiating integer tree `intTree` of type `Tree< int >` (line 10). The program prompts for 10 integers, each of which is inserted in the binary tree by calling `insertNode` (line 19). The program then performs *preorder*, *inorder* and *postorder traversals* (these are explained shortly) of `intTree` (lines 23, 26 and 29, respectively). The program then instantiates floating-point tree `doubleTree` of type `Tree< double >` (line 31). The program prompts for 10 `double` values, each of which is inserted in the binary tree by calling `insertNode` (line 41). The program then performs preorder, inorder and postorder traversals of `doubleTree` (lines 45, 48 and 51, respectively).

```
1  // Fig. 19.20: fig19_20.cpp
2  // Creating and traversing a binary tree.
3  #include <iostream>
4  #include <iomanip>
5  #include "Tree.h" // Tree class definition
6  using namespace std;
7
8  int main()
9  {
10    Tree< int > intTree; // create Tree of int values
11
12    cout << "Enter 10 integer values:\n";
13
```

Fig. 19.20 | Creating and traversing a binary tree. (Part 1 of 3.)

```
14      // insert 10 integers to intTree
15      for ( int i = 0; i < 10; ++i )
16      {
17         int intValue = 0;
18         cin >> intValue;
19         intTree.insertNode( intValue );
20      } // end for
21
22      cout << "\nPreorder traversal\n";
23      intTree.preOrderTraversal();
24
25      cout << "\nInorder traversal\n";
26      intTree.inOrderTraversal();
27
28      cout << "\nPostorder traversal\n";
29      intTree.postOrderTraversal();
30
31      Tree< double > doubleTree; // create Tree of double values
32
33      cout << fixed << setprecision( 1 )
34         << "\n\n\nEnter 10 double values:\n";
35
36      // insert 10 doubles to doubleTree
37      for ( int j = 0; j < 10; ++j )
38      {
39         double doubleValue = 0.0;
40         cin >> doubleValue;
41         doubleTree.insertNode( doubleValue );
42      } // end for
43
44      cout << "\nPreorder traversal\n";
45      doubleTree.preOrderTraversal();
46
47      cout << "\nInorder traversal\n";
48      doubleTree.inOrderTraversal();
49
50      cout << "\nPostorder traversal\n";
51      doubleTree.postOrderTraversal();
52      cout << endl;
53   } // end main
```

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5
```

Fig. 19.20 | Creating and traversing a binary tree. (Part 2 of 3.)

```
Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

Fig. 19.20 | Creating and traversing a binary tree. (Part 3 of 3.)

Class Template `TreeNode`

The `TreeNode` class template (Fig. 19.21) definition declares `Tree<NODETYPE>` as its `friend` (line 13). This makes all member functions of a given specialization of class template `Tree` (Fig. 19.22) `friend`s of the corresponding specialization of class template `TreeNode`, so they can access the `private` members of `TreeNode` objects of that type. Because the `TreeNode` template parameter `NODETYPE` is used as the template argument for `Tree` in the `friend` declaration, `TreeNode`s specialized with a particular type can be processed only by a `Tree` specialized with the same type (e.g., a `Tree` of `int` values manages `TreeNode` objects that store `int` values).

Lines 30–32 declare a `TreeNode`'s `private` data—the node's `data` value, and pointers `leftPtr` (to the node's *left subtree*) and `rightPtr` (to the node's *right subtree*). The constructor (lines 16–22) sets `data` to the value supplied as a constructor argument and sets pointers `leftPtr` and `rightPtr` to `nullptr` (thus initializing this node to be a *leaf node*). Member function `getData` (lines 25–28) returns the `data` value.

```
1  // Fig. 19.21: TreeNode.h
2  // TreeNode class-template definition.
3  #ifndef TREENODE_H
4  #define TREENODE_H
5
6  // forward declaration of class Tree
7  template< typename NODETYPE > class Tree;
8
9  // TreeNode class-template definition
10 template< typename NODETYPE >
11 class TreeNode
12 {
13    friend class Tree< NODETYPE >;
14 public:
15    // constructor
16    TreeNode( const NODETYPE &d )
17       : leftPtr( nullptr ), // pointer to left subtree
18         data( d ), // tree node data
19         rightPtr( nullptr ) // pointer to right substree
20    {
21       // empty body
22    } // end TreeNode constructor
23
```

Fig. 19.21 | `TreeNode` class-template definition. (Part 1 of 2.)

```
24      // return copy of node's data
25      NODETYPE getData() const
26      {
27         return data;
28      } // end getData function
29   private:
30      TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
31      NODETYPE data;
32      TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
33   }; // end class TreeNode
34
35   #endif
```

Fig. 19.21 | `TreeNode` class-template definition. (Part 2 of 2.)

Class Template `Tree`

Class template `Tree` (Fig. 19.22) has as `private` data `rootPtr` (line 42), a pointer to the tree's root node. The `Tree` constructor (lines 14–15) initializes `rootPtr` to `nullptr` to indicate that the tree is initially empty. The class's `public` member functions are `insertNode` (lines 18–21) that inserts a new node in the tree and `preOrderTraversal` (lines 24–27), `inOrderTraversal` (lines 30–33) and `postOrderTraversal` (lines 36–39), each of which walks the tree in the designated manner. Each of these member functions calls its own recursive utility function to perform the appropriate operations on the internal representation of the tree, so the program is *not* required to access the underlying `private` data to perform these functions. Remember that the recursion requires us to pass in a pointer that represents the next subtree to process.

```
1   // Fig. 19.22: Tree.h
2   // Tree class-template definition.
3   #ifndef TREE_H
4   #define TREE_H
5
6   #include <iostream>
7   #include "TreeNode.h"
8
9   // Tree class-template definition
10  template< typename NODETYPE > class Tree
11  {
12  public:
13     // constructor
14     Tree()
15        : rootPtr( nullptr ) { /* empty body */ }
16
17     // insert node in Tree
18     void insertNode( const NODETYPE &value )
19     {
20        insertNodeHelper( &rootPtr, value );
21     } // end function insertNode
22
```

Fig. 19.22 | `Tree` class-template definition. (Part 1 of 3.)

```
23      // begin preorder traversal of Tree
24      void preOrderTraversal() const
25      {
26         preOrderHelper( rootPtr );
27      } // end function preOrderTraversal
28
29      // begin inorder traversal of Tree
30      void inOrderTraversal() const
31      {
32         inOrderHelper( rootPtr );
33      } // end function inOrderTraversal
34
35      // begin postorder traversal of Tree
36      void postOrderTraversal() const
37      {
38         postOrderHelper( rootPtr );
39      } // end function postOrderTraversal
40
41   private:
42      TreeNode< NODETYPE > *rootPtr;
43
44      // utility function called by insertNode; receives a pointer
45      // to a pointer so that the function can modify pointer's value
46      void insertNodeHelper(
47         TreeNode< NODETYPE > **ptr, const NODETYPE &value )
48      {
49         // subtree is empty; create new TreeNode containing value
50         if ( *ptr == nullptr )
51            *ptr = new TreeNode< NODETYPE >( value );
52         else // subtree is not empty
53         {
54            // data to insert is less than data in current node
55            if ( value < ( *ptr )->data )
56               insertNodeHelper( &( ( *ptr )->leftPtr ), value );
57            else
58            {
59               // data to insert is greater than data in current node
60               if ( value > ( *ptr )->data )
61                  insertNodeHelper( &( ( *ptr )->rightPtr ), value );
62               else // duplicate data value ignored
63                  cout << value << " dup" << endl;
64            } // end else
65         } // end else
66      } // end function insertNodeHelper
67
68      // utility function to perform preorder traversal of Tree
69      void preOrderHelper( TreeNode< NODETYPE > *ptr ) const
70      {
71         if ( ptr != nullptr )
72         {
73            cout << ptr->data << ' '; // process node
74            preOrderHelper( ptr->leftPtr ); // traverse left subtree
```

Fig. 19.22 | Tree class-template definition. (Part 2 of 3.)

```
75            preOrderHelper( ptr->rightPtr ); // traverse right subtree
76         } // end if
77      } // end function preOrderHelper
78
79      // utility function to perform inorder traversal of Tree
80      void inOrderHelper( TreeNode< NODETYPE > *ptr ) const
81      {
82         if ( ptr != nullptr )
83         {
84            inOrderHelper( ptr->leftPtr ); // traverse left subtree
85            cout << ptr->data << ' '; // process node
86            inOrderHelper( ptr->rightPtr ); // traverse right subtree
87         } // end if
88      } // end function inOrderHelper
89
90      // utility function to perform postorder traversal of Tree
91      void postOrderHelper( TreeNode< NODETYPE > *ptr ) const
92      {
93         if ( ptr != nullptr )
94         {
95            postOrderHelper( ptr->leftPtr ); // traverse left subtree
96            postOrderHelper( ptr->rightPtr ); // traverse right subtree
97            cout << ptr->data << ' '; // process node
98         } // end if
99      } // end function postOrderHelper
100  }; // end class Tree
101
102  #endif
```

Fig. 19.22 | `Tree` class-template definition. (Part 3 of 3.)

`Tree` Member Function `insertNodeHelper`

The `Tree` class's utility function `insertNodeHelper` (lines 46–66) is called by `insertNode` (lines 18–21) to recursively insert a node into the tree. *A node can only be inserted as a leaf node in a binary search tree.* If the tree is *empty*, a new `TreeNode` is created, initialized and inserted in the tree (lines 50–51).

If the tree is *not empty*, the program compares the value to be inserted with the `data` value in the *root node*. If the insert value is smaller (line 55), the program recursively calls `insertNodeHelper` (line 56) to insert the value in the *left subtree*. If the insert value is larger (line 60), the program recursively calls `insertNodeHelper` (line 61) to insert the value in the *right subtree*. If the value to be inserted is identical to the data value in the *root node*, the program prints the message `" dup"` (line 63) and returns *without inserting the duplicate value into the tree*. Note that `insertNode` passes the address of `rootPtr` to `insertNodeHelper` (line 20) so it can modify the value stored in `rootPtr` (i.e., the address of the *root node*). To receive a pointer to `rootPtr` (which is also a pointer), `insertNodeHelper`'s first argument is declared as a *pointer to a pointer* to a `TreeNode`.

`Tree` Traversal Functions

Member functions `preOrderTraversal` (lines 24–27), `inOrderTraversal` (lines 30–33) and `postOrderTraversal` (lines 36–39) traverse the tree and print the node values. For the purpose of the following discussion, we use the binary search tree in Fig. 19.23.

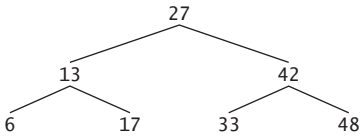

Fig. 19.23 | A binary search tree.

Inorder Traversal Algorithm

Function `inOrderTraversal` invokes utility function `inOrderHelper` (lines 80–88) to perform the inorder traversal of the binary tree. The steps for an inorder traversal are:

1. Traverse the *left subtree* with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 84.)
2. Process the value in the node—i.e., print the node value (line 85).
3. Traverse the *right subtree* with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 86.)

The value in a node is not processed until the values in its left subtree are processed, because each call to `inOrderHelper` immediately calls `inOrderHelper` again with the pointer to the *left subtree*. The inorder traversal of the tree in Fig. 19.23 is

```
6 13 17 27 33 42 48
```

The inorder traversal of a binary search tree prints the node values in *ascending* order. The process of creating a binary search tree actually *sorts* the data—thus, this process is called the **binary tree sort**.

Preorder Traversal Algorithm

Function `preOrderTraversal` invokes utility function `preOrderHelper` (lines 69–77) to perform the preorder traversal of the binary tree. The steps for a preorder traversal are:

1. Process the value in the node (line 73).
2. Traverse the *left subtree* with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 74.)
3. Traverse the *right subtree* with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 75.)

The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the *left subtree* are processed. Then the values in the *right subtree* are processed. The preorder traversal of the tree in Fig. 19.23 is

```
27 13 6 17 42 33 48
```

Postorder Traversal Algorithm

Function `postOrderTraversal` invokes utility function `postOrderHelper` (lines 91–99) to perform the postorder traversal of the binary tree. The steps for a postorder traversal are:

1. Traverse the *left subtree* with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 95.)

2. Traverse the *right subtree* with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 96.)
3. Process the value in the node (line 97).

The value in each node is not printed until the values of its children are printed. The `postOrderTraversal` of the tree in Fig. 19.23 is

```
6 17 13 33 48 42 27
```

Duplicate Elimination

The binary search tree facilitates **duplicate elimination**. As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same "go left" or "go right" decisions on each comparison as the original value did when it was inserted in the tree. Thus, the duplicate will eventually be compared with a node containing the same value. The duplicate value may be *discarded* at this point.

Searching a binary tree for a value that matches a key value is also fast. If the tree is balanced, then each branch contains about *half* the number of nodes in the tree. Each comparison of a node to the search key *eliminates half the nodes*.This is called an $O(\log n)$ algorithm (Big O notation is discussed in Chapter 20). So a binary search tree with n elements would require a maximum of $\log_2 n$ comparisons either to find a match or to determine that no match exists. This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made, because $2^{10} > 1000$. When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made, because $2^{20} > 1{,}000{,}000$.

Overview of the Binary Tree Exercises

In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a **level-order traversal** of a binary tree. The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

19.7 Wrap-Up

In this chapter, you learned that linked lists are collections of data items that are "linked up in a chain." You also learned that a program can perform insertions and deletions anywhere in a linked list (though our implementation performed insertions and deletions only at the ends of the list). We demonstrated that the stack and queue data structures are constrained versions of lists. For stacks, you saw that insertions and deletions are made only at the top. For queues, you saw that insertions are made at the tail and deletions are made from the head. We also presented the binary tree data structure. You saw a binary search tree that facilitated high-speed searching and sorting of data and efficient duplicate elimination. You learned how to create these data structures for reusability (as templates) and maintainability. In the next chapter, we study various searching and sorting techniques and implement them as function templates.

Summary

Section 19.1 Introduction

- Dynamic data structures (p. 778) grow and shrink during execution.
- Linked lists (p. 778) are collections of data items "lined up in a row"—insertions and removals are made anywhere in a linked list.
- Stacks (p. 778) are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top (p. 778).
- Queues (p. 778) represent waiting lines; insertions are made at the back (also referred to as the tail; p. 778) of a queue and removals are made from the front (also referred to as the head; p. 778).
- Binary trees (p. 778) facilitate high-speed searching and sorting of data, efficient duplicate elimination, representation of file-system directories and compilation of expressions into machine code.

Section 19.2 Self-Referential Classes

- A self-referential class (p. 779) contains a pointer that points to an object of the same class type.
- Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees.

Section 19.3 Linked Lists

- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links (p. 780)—hence, the term "linked" list.
- A linked list is accessed via a pointer to the first node of the list. Each subsequent node is accessed via the link-pointer member stored in the previous node and the last node contains a null pointer.
- Linked lists, stacks and queues are linear data structures (p. 780). Trees are nonlinear data structures (p. 780).
- A linked list is appropriate when the number of data elements to be represented is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- A singly linked list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence."
- A circular, singly linked list (p. 793) begins with a pointer to the first node, and each node contains a pointer to the next node. The "last node" does not contain a null pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."
- A doubly linked list (p. 793) allows traversals both forward and backward.
- A doubly linked list is often implemented with two "start pointers"—one that points to the first element to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal. Each node has a pointer to both the next and previous nodes.
- In a circular, doubly linked list (p. 794), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the "circle."

Section 19.4 Stacks

- A stack data structure allows nodes to be added to and removed from the stack only at the top.
- A stack is referred to as a last-in, first-out (LIFO) data structure.
- Function `push` inserts a new node at the top of the stack. Function `pop` removes a node from the top of the stack.

- A dependent name (p. 796) is an identifier that depends on the value of a template parameter. Resolution of dependent names occurs when the template is instantiated.
- Non-dependent names (p. 796) are resolved at the point where the template is defined.

Section 19.5 Queues

- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from a queue's head and are inserted only at its tail.
- A queue is referred to as a first-in, first-out (FIFO) data structure. The insert and remove operations are known as `enqueue` and `dequeue` (p. 799).

Section 19.6 Trees

- Binary trees (p. 803) are trees whose nodes all contain two links (none, one or both of which may have the value `nullptr`).
- The root node (p. 803) is the first node in a tree.
- Each link in the root node refers to a child. The left child is the root node of the left subtree (p. 803), and the right child is the root node of the right subtree (p. 803).
- The children of a single node are called siblings (p. 803). A node with no children is called a leaf node (p. 803).
- A binary search tree (p. 803) (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node (p. 803), and the values in any right subtree are greater than the value in its parent node.
- A node can only be inserted as a leaf node in a binary search tree.
- An inorder traversal (p. 804) of a binary tree traverses the left subtree, processes the value in the root node then traverses the right subtree. The value in a node is not processed until the values in its left subtree are processed. An inorder traversal of a binary search tree processes the nodes in sorted order.
- A preorder traversal (p. 804) processes the value in the root node, traverses the left subtree, then traverses the right subtree. The value in each node is processed as the node is encountered.
- A postorder traversal (p. 804) traverses the left subtree, traverses the right subtree, then processes the root node's value. The value in each node is not processed until the values in both subtrees are processed.
- The binary search tree helps eliminate duplicate data (p. 811). As the tree is being created, an attempt to insert a duplicate value will be recognized and the duplicate value may be discarded.
- The level-order traversal (p. 811) of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right.

Self-Review Exercises

19.1 Fill in the blanks in each of the following:

a) A self-_____ class is used to form dynamic data structures that can grow and shrink at execution time

b) The _____ operator is used to dynamically allocate memory and construct an object; this operator returns a pointer to the object.

c) A(n) _____ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list and node values are returned in last-in, first-out order.

d) A function that does not alter a linked list, but looks at the list to determine whether it's empty, is an example of a(n) _____ function.
e) A queue is referred to as a(n) _____ data structure, because the first nodes inserted are the first nodes removed.
f) The pointer to the next node in a linked list is referred to as a(n) _____.
g) The _____ operator is used to destroy an object and release dynamically allocated memory.
h) A(n) _____ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.
i) A(n) _____ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
j) A stack is referred to as a(n) _____ data structure, because the last node inserted is the first node removed.
k) The nodes of a(n) _____ tree contain two link members.
l) The first node of a tree is the _____ node.
m) Each link in a tree node points to a(n) _____ or _____ of that node.
n) A tree node that has no children is called a(n) _____ node.
o) The four traversal algorithms we mentioned in the text for binary search trees are _____, _____, _____ and _____.

19.2 What are the differences between a linked list and a stack?

19.3 What are the differences between a stack and a queue?

19.4 Perhaps a more appropriate title for this chapter would have been "Reusable Data Structures." Comment on how each of the following entities or concepts contributes to the reusability of data structures:

a) classes
b) class templates
c) inheritance
d) `private` inheritance
e) composition

19.5 Provide the inorder, preorder and postorder traversals of the binary search tree of Fig. 19.24.

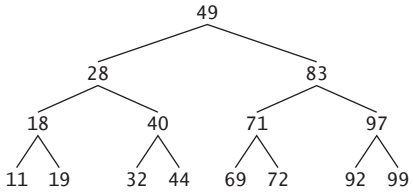

Fig. 19.24 | A 15-node binary search tree.

Answers to Self-Review Exercises

19.1 a) referential. b) `new`. c) stack. d) predicate. e) first-in, first-out (FIFO). f) link. g) `delete`. h) queue. i) tree. j.) last-in, first-out (LIFO). k) binary. l) root. m) child or subtree. n) leaf. o) inorder, preorder, postorder and level order.

19.2 It's possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may only be inserted at the top of the stack and removed from the top of a stack.

19.3 A queue data structure allows nodes to be removed only from the head of the queue and inserted only at the tail of the queue. A queue is referred to as a first-in, first-out (FIFO) data structure. A stack data structure allows nodes to be added to the stack and removed from the stack only at the top. A stack is referred to as a last-in, first-out (LIFO) data structure.

19.4 a) Classes allow us to instantiate as many data structure objects of a certain type (i.e., class) as we wish.
b) Class templates enable us to instantiate related classes, each based on different type parameters—we can then generate as many objects of each template class as we like.
c) Inheritance enables us to reuse code from a base class in a derived class, so that the derived-class data structure is also a base-class data structure (with `public` inheritance, that is).
d) Private inheritance enables us to reuse portions of the code from a base class to form a derived-class data structure; because the inheritance is `private`, all `public` base-class member functions become `private` in the derived class. This enables us to prevent clients of the derived-class data structure from accessing base-class member functions that do not apply to the derived class.
e) Composition enables us to reuse code by making a class object data structure a member of a composed class; if we make the class object a `private` member of the composed class, then the class object's `public` member functions are not available through the composed object's interface.

19.5 The inorder traversal is

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

The preorder traversal is

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

The postorder traversal is

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Exercises

19.6 ***(Concatenating Lists)*** Write a program that concatenates two linked list objects of characters. The program should include function `concatenate`, which takes references to both list objects as arguments and concatenates the second list to the first list.

19.7 ***(Merging Ordered Lists)*** Write a program that merges two ordered list objects of integers into a single ordered list object of integers. Function `merge` should receive references to each of the list objects to be merged and a reference to a list object into which the merged elements will be placed.

19.8 ***(Summing and Averaging Elements in a List)*** Write a program that inserts 25 random integers from 0 to 100 in order in a linked list object. The program should calculate the sum of the elements and the floating-point average of the elements.

19.9 ***(Copying a List in Reverse Order)*** Write a program that creates a linked list object of 10 characters and creates a second list object containing a copy of the first list, but in reverse order.

19.10 ***(Printing a Sentence in Reverse Order with a Stack)*** Write a program that inputs a line of text and uses a stack object to print the line reversed.

19.11 ***(Palindrome Testing with Stacks)*** Write a program that uses a stack object to determine if a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

19.12 ***(Infix-to-Postfix Conversion)*** Stacks are used by compilers to help in the process of evaluating expressions and generating machine language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like `3 + 4` and `7 / 9` in which the operator (`+` or `/` here) is written between its operands—this is called **infix notation**. Computers "prefer" **postfix notation** in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation and evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, and in each algorithm the stack is used for a different purpose.

In this exercise, you'll write a C++ version of the infix-to-postfix conversion algorithm. In the next exercise, you'll write a C++ version of the postfix expression evaluation algorithm. Later in the chapter, you'll discover that code you write in this exercise can help you implement a complete working compiler.

Write a program that converts an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression is

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into `string infix` and use modified versions of the stack functions implemented in this chapter to help create the postfix expression in `string postfix`. The algorithm for creating a postfix expression is as follows:

1) Push a left parenthesis `'('` onto the stack.
2) Append a right parenthesis `')'` to the end of `infix`.
3) While the stack is not empty, read `infix` from left to right and do the following:
   - If the current character in `infix` is a digit, copy it to the next element of `postfix`.
   - If the current character in `infix` is a left parenthesis, push it onto the stack.
   - If the current character in `infix` is an operator,
     - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in `postfix`.
     - Push the current character in `infix` onto the stack.
   - If the current character in `infix` is a right parenthesis
     - Pop operators from the top of the stack and insert them in `postfix` until a left parenthesis is at the top of the stack.
     - Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`%`	modulus

[*Note:* We assume left-to-right associativity for all operators for the purpose of this exercise.] The stack should be maintained with stack nodes, each containing a data member and a pointer to the next stack node.

Some of the functional capabilities you may want to provide are:

a) function `convertToPostfix` that converts the infix expression to postfix notation

b) function `isOperator` that determines whether `c` is an operator
c) function `precedence` that determines whether the precedence of `operator1` is greater than or equal to the precedence of `operator2`, and, if so, returns `true`.
d) function `push` that pushes a value onto the stack
e) function `pop` that pops a value off the stack
f) function `stackTop` that returns the top value of the stack without popping the stack
g) function `isEmpty` that determines if the stack is empty
h) function `printStack` that prints the stack

19.13 ***(Postfix Evaluation)*** Write a program that evaluates a postfix expression (assume it's valid) such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a `string`. Using modified versions of the stack functions implemented earlier in this chapter, the program should scan the expression and evaluate it. The algorithm is as follows:

1) While you have not reached the end of the `string`, read the expression from left to right.
   If the current character is a digit,
      Push its integer value onto the stack (the integer value of a digit character is its value in the computer's character set minus the value of `'0'` in the computer's character set).
   Otherwise, if the current character is an *operator*,
      Pop the two top elements of the stack into variables `x` and `y`.
      Calculate `y` *operator* `x`.
      Push the result of the calculation onto the stack.
2) When you reach the end of the `string`, pop the top value of the stack. This is the result of the postfix expression.

[Note: In *Step 2* above, if the operator is `'/'`, the top of the stack is `2` and the next element in the stack is `8`, then pop `2` into `x`, pop `8` into `y`, evaluate `8 / 2` and push the result, `4`, back onto the stack. This note also applies to operator `'-'`.] The arithmetic operations allowed in an expression are

`+` addition
`-` subtraction
`*` multiplication
`/` division
`^` exponentiation
`%` modulus

[*Note:* We assume left-to-right associativity for all operators for the purpose of this exercise.] The stack should be maintained with stack nodes that contain an `int` data member and a pointer to the next stack node. You may want to provide the following functional capabilities:

a) function `evaluatePostfixExpression` that evaluates the postfix expression
b) function `calculate` that evaluates the expression `op1 operator op2`
c) function `push` that pushes a value onto the stack
d) function `pop` that pops a value off the stack
e) function `isEmpty` that determines if the stack is empty
f) function `printStack` that prints the stack

19.14 ***(Postfix Evaluation Enhanced)*** Modify the postfix evaluator program of Exercise 19.13 so that it can process integer operands larger than 9.

19.15 ***(Supermarket Simulation)*** Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of 1–4 minutes. Also, each customer is served in random integer intervals of 1–4 minutes. Obviously,

the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with "balanced" rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes) using the following algorithm:

1) Choose a random integer from 1 to 4 to determine the minute at which the first customer arrives.
2) At the first customer's arrival time:
   Determine customer's service time (random integer from 1 to 4);
   Begin servicing the customer;
   Schedule arrival time of next customer (random integer 1 to 4 added to the current time).
3) For each minute of the day:
   If the next customer arrives,
      Say so, enqueue the customer, and schedule the arrival time of the next customer;
   If service was completed for the last customer;
      Say so, dequeue next customer to be serviced and determine customer's service completion time (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes, and answer each of the following:

a) What's the maximum number of customers in the queue at any time?
b) What's the longest wait any one customer experiences?
c) What happens if the arrival interval is changed from 1–4 minutes to 1–3 minutes?

19.16 ***(Allowing Duplicates in Binary Trees)*** Modify the program of Figs. 19.20–19.22 to allow the binary tree object to contain duplicates.

19.17 ***(Binary Tree of Strings)*** Write a program based on Figs. 19.20–19.22 that inputs a line of text, tokenizes the sentence into separate words (you may want to use the `istringstream` library class), inserts the words in a binary search tree and prints the inorder, preorder and postorder traversals of the tree. Use an OOP approach.

19.18 ***(Duplicate Elimination)*** In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you'd perform duplicate elimination using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

19.19 ***(Depth of a Binary Tree)*** Write a function `depth` that receives a binary tree and determines how many levels it has.

19.20 (***Recursively Print a List Backward***) Write a member function `printListBackward` that recursively outputs the items in a linked list object in reverse order. Write a test program that creates a sorted list of integers and prints the list in reverse order.

19.21 (***Recursively Search a List***) Write a member function `searchList` that recursively searches a linked list object for a specified value. The function should return a pointer to the value if it's found; otherwise, `nullptr` should be returned. Use your function in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

19.22 (***Binary Tree Delete***) Deleting items from binary search trees is not as straightforward as the insertion algorithm. There are three cases that are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child or the item is contained in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the pointer in the parent node is set to `nullptr`.

If the item to be deleted is contained in a node with one child, the pointer in the parent node is set to point to the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the pointer in the parent node cannot be assigned to point to one of the children of the node to be deleted. In most cases, the resulting binary search tree would not adhere to the following characteristic of binary search trees (with no duplicate values): *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node*.

Which node is used as a *replacement node* to maintain this characteristic? Either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let's consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the pointer to the right child of the current node is `nullptr`. We are now pointing to the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

1) Store the pointer to the node to be deleted in a temporary pointer variable (this pointer is used to delete the dynamically allocated memory).
2) Set the pointer in the parent of the node being deleted to point to the replacement node.
3) Set the pointer in the parent of the replacement node to `nullptr`.
4) Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.
5) Delete the node to which the temporary pointer variable points.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node's position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

1) Store the pointer to the node to be deleted in a temporary pointer variable.
2) Set the pointer in the parent of the node being deleted to point to the replacement node.
3) Set the pointer in the parent of the replacement node to point to the left child of the replacement node.
4) Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.
5) Delete the node to which the temporary pointer variable points.

Write member function `deleteNode`, which takes as its arguments a pointer to the root node of the tree object and the value to be deleted. The function should locate in the tree the node containing the value to be deleted and use the algorithms discussed here to delete the node. The function should print a message that indicates whether the value is deleted. Modify the program of Figs. 19.20–19.22 to use this function. After deleting an item, call the `inOrder`, `preOrder` and `postOrder` traversal functions to confirm that the delete operation was performed correctly.

19.23 (***Binary Tree Search***) Write member function `binaryTreeSearch`, which attempts to locate a specified value in a binary search tree object. The function should take as arguments a pointer to the binary tree's root node and a search key to locate. If the node containing the search key is found, the function should return a pointer to that node; otherwise, the function should return a `nullptr` pointer.

19.24 (***Level-Order Binary Tree Traversal***) The program of Figs. 19.20–19.22 illustrated three recursive methods of traversing a binary tree—inorder, preorder and postorder traversals. This exercise presents the *level-order traversal* of a binary tree, in which the node values are printed level by level, starting at the root node level. The nodes on each level are printed from left to right. The level-

order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

1) Insert the root node in the queue
2) While there are nodes left in the queue,
   Get the next node in the queue
   Print the node's value
   If the pointer to the left child of the node is not `nullptr`
      Insert the left child node in the queue
   If the pointer to the right child of the node is not `nullptr`
      Insert the right child node in the queue.

Write member function `levelOrder` to perform a level-order traversal of a binary tree object. Modify the program of Figs. 19.20–19.22 to use this function. [*Note:* You'll also need to modify and incorporate the queue-processing functions of Fig. 19.16 in this program.]

19.25 (***Printing Trees***) Write a recursive member function `outputTree` to display a binary tree object on the screen. The function should output the tree row by row, with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree illustrated in Fig. 19.24 is output as shown in Fig. 19.25. Note that the rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column of output starts five spaces to the right of the previous column. Function `outputTree` should receive an argument `totalSpaces` representing the number of spaces preceding the value to be output (this variable should start at zero, so the root node is output at the left of the screen). The function uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

While the pointer to the current node is not `nullptr`
   Recursively call `outputTree` with the current node's right subtree and `totalSpaces` + 5
   Use a `for` structure to count from 1 to `totalSpaces` and output spaces
   Output the value in the current node
   Set the pointer to the current node to point to the left subtree of the current node
   Increment `totalSpaces` by 5.

```
                99
          97
                92
     83
                72
          71
                69
49
                44
          40
                32
     28
                19
          18
                11
```

Fig. 19.25 | Outputting the binary tree illustrated in Fig. 19.24.

19.26 ***(Insert/Delete Anywhere in a Linked List)*** Our linked list class template allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used `private` inheritance and composition to produce a stack class template and a queue class template with a minimal amount of code by reusing the list class template. Actually,

linked lists are more general than those we provided. Modify the linked list class template we developed in this chapter to handle insertions and deletions anywhere in the list.

19.27 ***(List and Queues without Tail Pointers)*** Our implementation of a linked list (Figs. 19.4–19.5) used both a `firstPtr` and a `lastPtr`. The `lastPtr` was useful for the `insertAtBack` and `removeFromBack` member functions of the `List` class. The `insertAtBack` function corresponds to the `enqueue` member function of the `Queue` class. Rewrite the `List` class so that it does not use a `lastPtr`. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the `Queue` class (Fig. 19.16)?

19.28 ***(Performance of Binary Tree Sorting and Searching)*** One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree—for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data were inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?

19.29 ***(Indexed Lists)*** As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list searching performance is to create and maintain an index to the list. An index is a set of pointers to various key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries—one for each letter of the alphabet. A search operation for a last name beginning with `"Y"` would first search the index to determine where the `"Y"` entries begin and "jump into" the list at that point and search linearly until the desired name was found. This would be much faster than searching the linked list from the beginning. Use the `List` class of Figs. 19.4–19.5 as the basis of an `IndexedList` class. Write a program that demonstrates the operation of indexed lists. Be sure to include member functions `insertInIndexedList`, `searchIndexedList` and `deleteFromIndexedList`.

Special Section: Building Your Own Compiler

In Exercises 8.15–8.17, we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute SML programs. In Exercises 19.30–19.34, we build a compiler that converts programs written in a high-level programming language to SML. This section "ties" together the entire programming process. You'll write programs in this new high-level language, compile them on the compiler you build and run them on the simulator you built in Exercise 8.16. You should make every effort to implement your compiler in an object-oriented manner. [*Note:* Due to the size of the descriptions for Exercises 19.30–19.34, we've posted them in a PDF document located at `www.deitel.com/books/cpphtp9/`.]

20

Searching and Sorting

With sobs and tears
he sorted out
Those of the largest size …
—Lewis Carroll

Attempt the end, and never
stand to doubt;
Nothing's so hard, but search
will find it out.
—Robert Herrick

'Tis in my memory lock'd,
And you yourself shall keep the
key of it.
—William Shakespeare

Objectives

In this chapter you'll:

- Search for a given value in an `array` using linear search and binary search.
- Use Big O notation to express the efficiency of searching and sorting algorithms and to compare their performance.
- Sort an `array` using insertion sort, selection sort and the recursive merge sort algorithms.
- Understand the nature of algorithms of constant, linear and quadratic runtime.

Outline

20.1 Introduction

Searching data involves determining whether a value (referred to as the **search key**) is present in the data and, if so, finding the value's location. Two popular search algorithms are the simple *linear search* (Section 20.2.1) and the faster but more complex *binary search* (Section 20.2.2).

Sorting places data in ascending or descending order, based on one or more **sort keys**. A list of names could be sorted alphabetically, bank accounts could be sorted by account number, employee payroll records could be sorted by social security number, and so on. You'll learn about *insertion sort* (Section 20.3.1), *selection sort* (Section 20.3.2) and the more efficient, but more complex *merge sort* (Section 20.3.3). Figure 20.1 summarizes the searching and sorting algorithms discussed in the book's examples and exercises. This chapter also introduces **Big O notation**, which is used to characterize an algorithm's worst-case runtime—that is, how hard an algorithm may have to work to solve a problem.

Algorithm	Location	Algorithm	Location
Searching Algorithms		*Sorting Algorithms*	
Linear search	Section 20.2.1	Insertion sort	Section 20.3.1
Binary search	Section 20.2.2	Selection sort	Section 20.3.2
Recursive linear search	Exercise 20.8	Recursive merge sort	Section 20.3.3
Recursive binary search	Exercise 20.9	Bubble sort	Exercises 20.5–20.6
Binary tree search	Section 19.6	Bucket sort	Exercise 20.7
Linear search (linked list)	Exercise 19.21	Recursive quicksort	Exercise 20.10
`binary_search` standard library function	Section 16.3.6	Binary tree sort	Section 19.6
		`sort` standard library function	Section 16.3.6
		Heap sort	Section 16.3.12

Fig. 20.1 | Searching and sorting algorithms in this text.

A Note About This Chapter's Examples

The searching and sorting algorithms in this chapter are implemented as function templates that manipulate objects of the `array` class template. To help you visualize how certain algorithms work, some of the examples display `array`-element values throughout the searching or sorting process. These output statements slow an algorithm's performance and would *not* be included in industrial-strength code.

20.2 Searching Algorithms

Looking up a phone number, accessing a website and checking a word's definition in a dictionary all involve searching through large amounts of data. A searching algorithm finds an element that matches a given search key, if such an element does, in fact, exist. There are, however, a number of things that differentiate search algorithms from one another. The major difference is the amount of *effort* they require to complete the search. One way to describe this *effort* is with Big O notation. For searching and sorting algorithms, this is particularly dependent on the number of data elements.

In Section 20.2.1, we present the linear search algorithm then discuss the algorithm's *efficiency* as measured by Big O notation. In Section 20.2.2, we introduce the binary search algorithm, which is much more efficient but more complex to implement.

20.2.1 Linear Search

In this section, we discuss the simple **linear search** for determining whether an *unsorted* `array` (i.e., an `array` with element values that are in no particular order) contains a specified search key. Exercise 20.8 at the end of this chapter asks you to implement a recursive version of the linear search.

Function Template `linearSearch`

Function template `linearSearch` (Fig. 20.2, lines 10–18) compares each element of an `array` with a *search key* (line 14). Because the array is not in any particular order, it's just as likely that the search key will be found in the first element as the last. On average, therefore, the program must compare the search key with *half* of the `array`'s elements. To determine that a value is *not* in the `array`, the program must compare the search key to *every* `array` element. Linear search works well for *small* or *unsorted* arrays. However, for large arrays, linear searching is inefficient. If the array is *sorted* (e.g., its elements are in ascending order), you can use the high-speed binary search technique (Section 20.2.2).

```
1  // Fig. 20.2: LinearSearch.cpp
2  // Linear search of an array.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  // compare key to every element of array until location is
8  // found or until end of array is reached; return location of
9  // element if key is found or -1 if key is not found
10 template < typename T, size_t size >
11 int linearSearch( const array< T, size > &items, const T& key )
12 {
13    for ( size_t i = 0; i < items.size(); ++i )
14       if ( key == items[ i ] ) // if found,
15          return i; // return location of key
16
17    return -1; // key not found
18 } // end function linearSearch
```

Fig. 20.2 | Linear search of an array. (Part 1 of 2.)

```
19
20  int main()
21  {
22     const size_t arraySize = 100; // size of array
23     array< int, arraySize > arrayToSearch; // create array
24
25     for ( size_t i = 0; i < arrayToSearch.size(); ++i )
26        arrayToSearch[ i ] = 2 * i; // create some data
27
28     cout << "Enter integer search key: ";
29     int searchKey; // value to locate
30     cin >> searchKey;
31
32     // attempt to locate searchKey in arrayToSearch
33     int element = linearSearch( arrayToSearch, searchKey );
34
35     // display results
36     if ( element != -1 )
37        cout << "Found value in element " << element << endl;
38     else
39        cout << "Value not found" << endl;
40  } // end main
```

```
Enter integer search key: 36
Found value in element 18
```

```
Enter integer search key: 37
Value not found
```

Fig. 20.2 | Linear search of an array. (Part 2 of 2.)

Big O: Constant Runtime

Suppose an algorithm simply tests whether the first element of an `array` is equal to the second element. If the `array` has 10 elements, this algorithm requires only *one* comparison. If the `array` has 1000 elements, the algorithm still requires only *one* comparison. In fact, the algorithm is *independent* of the number of `array` elements. This algorithm is said to have a **constant runtime**, which is represented in Big O notation as $O(1)$. An algorithm that's $O(1)$ does not necessarily require only one comparison. $O(1)$ just means that the number of comparisons is *constant*—it does *not* grow as the size of the `array` increases. An algorithm that tests whether the first element of an `array` is equal to any of the next three elements will always require three comparisons, but in Big O notation it's still considered $O(1)$. $O(1)$ is often pronounced "on the order of 1" or more simply "**order 1**."

Big O: Linear Runtime

An algorithm that tests whether the first element of an `array` is equal to *any* of the other elements of the `array` requires at most $n - 1$ comparisons, where n is the number of elements in the `array`. If the `array` has 10 elements, the algorithm requires up to nine comparisons. If the `array` has 1000 elements, the algorithm requires up to 999 comparisons. As n grows larger, the n part of the expression $n - 1$ "dominates," and subtracting one be-

comes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. For this reason, an algorithm that requires a total of $n - 1$ comparisons (such as the one we described in this paragraph) is said to be $O(n)$ and is referred to as having a **linear runtime**. $O(n)$ is often pronounced "on the order of n" or more simply "**order n**."

Big O: Quadratic Runtime

Now suppose you have an algorithm that tests whether *any* element of an `array` is duplicated elsewhere in the `array`. The first element must be compared with *all the other elements*. The second element must be compared with all the other elements except the first (it was already compared to the first). The third element then must be compared with all the other elements except the first two. In the end, this algorithm will end up making $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term *dominates* and the n term becomes inconsequential. Again, Big O notation highlights the n^2 term, leaving $n^2/2$. As we'll soon see, even *constant factors*, such as the 1/2 here, are omitted in Big O notation.

Big O is concerned with how an algorithm's runtime grows in relation to the *number of items processed*. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons. With this algorithm, *doubling* the number of elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons. Again, doubling the number of elements quadruples the number of comparisons. Both of these algorithms *grow as the square of n*, so Big O ignores the constant, and both algorithms are considered to be $O(n^2)$, which is referred to as **quadratic runtime** and pronounced "on the order of n-squared" or more simply "**order n-squared**."

$O(n^2)$ Performance

When n is small, $O(n^2)$ algorithms (running on today's billions-of-operations-per-second personal computers) will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation. An $O(n^2)$ algorithm running on a million-element `array` would require a trillion "operations" (where each could actually require several machine instructions to execute). This could require hours to execute. A billion-element `array` would require a quintillion operations, a number so large that the algorithm could take decades! Unfortunately, $O(n^2)$ algorithms tend to be easy to write. In this chapter, you'll see algorithms with more favorable Big O measures. Such efficient algorithms often take a bit more cleverness and effort to create, but their superior performance can be worth the extra effort, especially as n gets large.

Linear Search's Runtime

The *linear search* algorithm runs in $O(n)$ time. The worst case in this algorithm is that *every* element must be checked to determine whether the search key is in the `array`. If the `array`'s size *doubles*, the number of comparisons that the algorithm must perform also *doubles*. Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the `array`. But we seek algorithms that perform well, on average, across *all* searches, including those where the element matching the search key is near the end of the `array`. If a program needs to perform many searches

on large arrays, it may be better to implement a different, more efficient algorithm, such as the *binary search* which we consider in the next section.

Performance Tip 20.1

Sometimes the simplest algorithms perform poorly. Their virtue is that they're easy to program, test and debug. Sometimes more complex algorithms are required to maximize performance.

20.2.2 Binary Search

The **binary search algorithm** is more efficient than the linear search algorithm, but it requires that the array first be *sorted*. This is only worthwhile when the array, once sorted, will be searched a great many times—or when the searching application has *stringent* performance requirements. The first iteration of this algorithm tests the *middle* array element. If this matches the search key, the algorithm ends. Assuming the array is sorted in *ascending* order, then if the search key is *less* than the middle element, the search key cannot match any element in the array's second half so the algorithm continues with only the first *half* (i.e., the first element up to, but *not* including, the middle element). If the search key is *greater* than the middle element, the search key cannot match any element in the array's first half so the algorithm continues with only the second *half* (i.e., the element *after* the middle element through the last element). Each iteration tests the *middle value* of the array's remaining elements. If the element does not match the search key, the algorithm eliminates half of the remaining elements. The algorithm ends either by finding an element that matches the search key or by reducing the sub-array to zero size.

Binary Search of 15 Integer Values

As an example, consider the sorted 15-element array

```
   2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

and the search key 65. A binary search first checks whether the *middle* element (51) is the search key. The search key (65) is larger than 51, so 51 is eliminated from consideration along with the first half of the array (all elements smaller than 51.) Next, the algorithm checks whether 81 (the middle element of the remaining elements) matches the search key. The search key (65) is smaller than 81, so 81 is eliminated from consideration along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of elements to check to three (56, 65 and 77). The algorithm then checks 65 (which matches the search key), and returns the element's index (9). In this case, the algorithm required just *three* comparisons to determine whether the array contained the search key. Using a *linear search* algorithm would have required 10 comparisons. [*Note:* In this example, we've chosen to use an array with 15 elements, so that there will always be an obvious middle element in the array. With an even number of elements, the middle of the array lies between two elements. We implement the algorithm to choose the element with the higher index number.]

Binary Search Example

Figure 20.3 implements and demonstrates the binary-search algorithm. Throughout the program's execution, we use function template `displayElements` (lines 11–22) to display the portion of the array that's currently being searched.

```
// Fig 20.3: BinarySearch.cpp
// Binary search of an array.
#include <algorithm>
#include <array>
#include <ctime>
#include <iostream>
#include <random>
using namespace std;

// display array elements from index low through index high
template < typename T, size_t size >
void displayElements( const array< T, size > &items,
   size_t low, size_t high )
{
   for ( size_t i = 0; i < items.size() && i < low; ++i )
      cout << "   "; // display spaces for alignment

   for ( size_t i = low; i < items.size() && i <= high; ++i )
      cout << items[ i ] << " "; // display element

   cout << endl;
} // end function displayElements

// perform a binary search on the data
template < typename T, size_t size >
int binarySearch( const array< T, size > &items, const T& key)
{
   int low = 0; // low index of elements to search
   int high = items.size() - 1; // high index of elements to search
   int middle = ( low + high + 1 ) / 2; // middle element
   int location = -1; // key's index; -1 if not found

   do // loop to search for element
   {
      // display remaining elements of array to be searched
      displayElements( items, low, high );

      // output spaces for alignment
      for ( int i = 0; i < middle; ++i )
        cout << "   ";

      cout << " * " << endl; // indicate current middle

      // if the element is found at the middle
      if ( key == items[ middle ] )
         location = middle; // location is the current middle
      else if ( key < items[ middle ] ) // middle is too high
         high = middle - 1; // eliminate the higher half
      else // middle element is too low
         low = middle + 1; // eliminate the lower half

      middle = ( low + high + 1 ) / 2; // recalculate the middle
   } while ( ( low <= high ) && ( location == -1 ) );
```

Fig. 20.3 | Binary search of an array. (Part 1 of 3.)

```
54
55      return location; // return location of key
56   } // end function binarySearch
57
58   int main()
59   {
60      // use the default random-number generation engine to produce
61      // uniformly distributed pseudorandom int values from 10 to 99
62      default_random_engine engine(
63         static_cast<unsigned int>( time( nullptr ) ) );
64      uniform_int_distribution<unsigned int> randomInt( 10, 99 );
65
66      const size_t arraySize = 15; // size of array
67      array< int, arraySize > arrayToSearch; // create array
68
69      // fill arrayToSearch with random values
70      for ( int &item : arrayToSearch )
71         item = randomInt( engine );
72
73      sort( arrayToSearch.begin(), arrayToSearch.end() ); // sort the array
74
75      // display arrayToSearch's values
76      displayElements( arrayToSearch, 0, arrayToSearch.size() - 1 );
77
78      // get input from user
79      cout << "\nPlease enter an integer value (-1 to quit): ";
80      int searchKey; // value to locate
81      cin >> searchKey; // read an int from user
82      cout << endl;
83
84      // repeatedly input an integer; -1 terminates the program
85      while ( searchKey != -1 )
86      {
87         // use binary search to try to find integer
88         int position = binarySearch( arrayToSearch, searchKey );
89
90         // return value of -1 indicates integer was not found
91         if ( position == -1 )
92            cout << "The integer " << searchKey << " was not found.\n";
93         else
94            cout << "The integer " << searchKey
95               << " was found in position " << position << ".\n";
96
97         // get input from user
98         cout << "\n\nPlease enter an integer value (-1 to quit): ";
99         cin >> searchKey; // read an int from user
100        cout << endl;
101     } // end while
102  } // end main
```

Fig. 20.3 | Binary search of an array. (Part 2 of 3.)

```
10 23 27 48 52 55 58 60 62 63 68 72 75 92 97

Please enter an integer value (-1 to quit): 48

10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
                     *
10 23 27 48 52 55 58
          *
The integer 48 was found in position 3.

Please enter an integer value (-1 to quit): 92

10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
                     *
                        62 63 68 72 75 92 97
                                 *
                                    75 92 97
                                       *
The integer 92 was found in position 13.

Please enter an integer value (-1 to quit): 22

10 23 27 48 52 55 58 60 62 63 68 72 75 92 97
                     *
10 23 27 48 52 55 58
          *
10 23 27
    *
10
 *
The integer 22 was not found.

Please enter an integer value (-1 to quit): -1
```

Fig. 20.3 | Binary search of an array. (Part 3 of 3.)

Function Template `binarySearch`

Lines 25–56 define function template `binarySearch`, which has two parameters—a reference to the `array` to search and a reference to the search key. Lines 28–30 calculate the `low` end index, `high` end index and `middle` index of the portion of the `array` that the algorithm is currently searching. When `binarySearch` is first called, `low` is `0`, `high` is the `array`'s size minus `1` and `middle` is the average of these two values. Line 31 initializes `location` to `-1`—the value that `binarySearch` returns if the search key is *not* found. Lines 33–53 loop until `low` is greater than `high` (indicating that the element was not found) or `location` does not equal `-1` (indicating that the search key was found). Line 45 tests whether the value in the `middle` element is equal to `key`. If so, line 46 assigns the `middle` index to `location`. Then the loop terminates and `location` is returned to the caller. Each iteration of the loop that does not find the search key tests a single value (line 45) and eliminates half of the remaining values in the `array` (line 48 or 50).

Function `main`

Lines 62–64 set up a random-number generator for `int` values from `10`–`99`. Lines 66–71 create an `array` and fill it with random `int`s. Recall that the binary search algorithm requires a *sorted* `array`, so line 73 calls the Standard Library function `sort` to sort `arrayToSearch`'s elements into ascending order. Line 76 displays `arrayToSearch`'s sorted contents.

Lines 85–101 loop until the user enters the value `-1`. For each search key the user enters, the program performs a binary search of `arrayToSearch` to determine whether it contains the search key. The first line of output from this program shows `arrayToSearch`'s contents in ascending order. When the user instructs the program to search for `48`, the program first tests the middle element, which is `60` (as indicated by `*`). The search key is less than `60`, so the program eliminates the second half of the `array` and tests the middle element from the first half of the `array`. The search key equals `48`, so the program returns the index `3` after performing just *two* comparisons. The output also shows the results of searching for the values `92` and `22`.

Efficiency of Binary Search

In the worst-case scenario, searching a sorted `array` of 1023 elements will take only 10 comparisons when using a binary search. Repeatedly dividing 1023 by 2 (because, after each comparison, we can eliminate from consideration *half* of the remaining elements) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0. The number 1023 ($2^{10} - 1$) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, an `array` of 1,048,575 ($2^{20} - 1$) elements takes a maximum of 20 comparisons to find the key, and an `array` of approximately one billion elements takes a maximum of 30 comparisons to find the key. This is a *tremendous* performance improvement over the linear search. For a one-billion-element `array`, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted `array` is the exponent of the first power of 2 greater than the number of elements in the `array`, which is represented as $\log_2 n$. *All logarithms grow at roughly the same rate*, so in Big O notation the base can be omitted. This results in a Big O of $O(\log n)$ for a binary search, which is also known as **logarithmic runtime** and pronounced "on the order of log n" or more simply "**order log n**."

20.3 Sorting Algorithms

Sorting data (i.e., placing the data into some particular order, such as *ascending* or *descending*) is one of the most important computing applications. A bank sorts all of its checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, further, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and often, massive amounts of it. Sorting data is an intriguing, computer-intensive problem that has attracted intense research efforts.

An important point to understand about sorting is that the end result—the sorted `array`*—will be the same no matter which algorithm you use to sort the* `array`*. Your algorithm choice*

affects only the algorithm's runtime and memory use. The next two sections, introduce the *selection sort* and *insertion sort*—simple algorithms to implement, but inefficient. In each case, we examine the efficiency of the algorithms using Big O notation. We then present the merge sort algorithm, which is much faster but is more difficult to implement.

20.3.1 Insertion Sort

Figure 20.4 uses **insertion sort**—a simple, but inefficient, sorting algorithm—to sort a 10-element `array`'s values into ascending order. Function template `insertionSort` (lines 9–28) implements the algorithm.

```
1  // Fig. 20.4: InsertionSort.cpp
2  // Sorting an array into ascending order with insertion sort.
3  #include <array>
4  #include <iomanip>
5  #include <iostream>
6  using namespace std;
7
8  // sort an array into ascending order
9  template < typename T, size_t size >
10 void insertionSort( array< T, size > &items )
11 {
12    // loop over the elements of the array
13    for ( size_t next = 1; next < items.size(); ++next )
14    {
15       T insert = items[ next ]; // save value of next item to insert
16       size_t moveIndex = next; // initialize location to place element
17
18       // search for the location in which to put the current element
19       while ( ( moveIndex > 0 ) && ( items[ moveIndex - 1 ] > insert ) )
20       {
21          // shift element one slot to the right
22          items[ moveIndex ] = items[ moveIndex - 1 ];
23          --moveIndex;
24       } // end while
25
26       items[ moveIndex ] = insert; // place insert item back into array
27    } // end for
28 } // end function insertionSort
29
30 int main()
31 {
32    const size_t arraySize = 10; // size of array
33    array < int, arraySize > data =
34       { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
35
36    cout << "Unsorted array:\n";
37
38    // output original array
39    for ( size_t i = 0; i < arraySize; ++i )
40       cout << setw( 4 ) << data[ i ];
```

Fig. 20.4 | Sorting an array into ascending order with insertion sort. (Part 1 of 2.)

```
41
42      insertionSort( data ); // sort the array
43
44      cout << "\nSorted array:\n";
45
46      // output sorted array
47      for ( size_t i = 0; i < arraySize; ++i )
48         cout << setw( 4 ) << data[ i ];
49
50      cout << endl;
51   } // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Fig. 20.4 | Sorting an array into ascending order with insertion sort. (Part 2 of 2.)

Insertion Sort Algorithm

The algorithm's first iteration takes the `array`'s second element and, if it's less than the first element, swaps it with the first element (i.e., the algorithm *inserts* the second element in front of the first element). The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the i^{th} iteration of this algorithm, the first i elements in the original array will be sorted.

First Iteration

Lines 33–34 declare and initialize the `array` named `data` with the following values:

```
      34   56   4   10   77   51   93   30   5   52
```

Line 42 passes the `array` to the `insertionSort` function, which receives the `array` in parameter `items`. The function first looks at `items[0]` and `items[1]`, whose values are `34` and `56`, respectively. These two elements are already in order, so the algorithm continues—if they were out of order, the algorithm would swap them.

Second Iteration

In the second iteration, the algorithm looks at the value of `items[2]` (that is, `4`). This value is less than `56`, so the algorithm stores `4` in a temporary variable and moves `56` one element to the right. The algorithm then determines that `4` is less than `34`, so it moves `34` one element to the right. At this point, the algorithm has reached the beginning of the `array`, so it places `4` in `items[0]`. The array now is

```
      4   34   56   10   77   51   93   30   5   52
```

Third Iteration and Beyond

In the third iteration, the algorithm places the value of `items[3]` (that is, `10`) in the correct location with respect to the first four `array` elements. The algorithm compares `10` to `56` and moves `56` one element to the right because it's larger than `10`. Next, the algorithm compares `10` to `34`, moving `34` right one element. When the algorithm compares `10` to `4`, it observes that `10` is larger than `4` and places `10` in `items[1]`. The array now is

```
     4   10   34   56   77   51   93   30   5   52
```

Using this algorithm, after the i^{th} iteration, the first $i + 1$ `array` elements are sorted. They may not be in their *final* locations, however, because the algorithm might encounter smaller values later in the `array`.

Function Template `insertionSort`

Function template `insertionSort` performs the sorting in lines 13–27, which iterates over the `array`'s elements. In each iteration, line 15 temporarily stores in variable `insert` the value of the element that will be inserted into the `array`'s sorted portion. Line 16 declares and initializes the variable `moveIndex`, which keeps track of where to insert the element. Lines 19–24 loop to locate the correct position where the element should be inserted. The loop terminates either when the program reaches the `array`'s first element or when it reaches an element that's less than the value to insert. Line 22 moves an element to the right, and line 23 decrements the position at which to insert the next element. After the `while` loop ends, line 26 inserts the element into place. When the `for` statement in lines 13–27 terminates, the `array`'s elements are sorted.

Big O: Efficiency of Insertion Sort

Insertion sort is *simple*, but *inefficient*, sorting algorithm. This becomes apparent when sorting *large* arrays. Insertion sort iterates $n - 1$ times, inserting an element into the appropriate position in the elements sorted so far. For each iteration, determining where to insert the element can require comparing the element to each of the preceding elements—$n - 1$ comparisons in the worst case. Each individual repetition statement runs in $O(n)$ time. To determine Big O notation, *nested* statements mean that you must *multiply* the number of comparisons. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each $O(n)$ iteration of the outer loop, there will be $O(n)$ iterations of the inner loop, resulting in a Big O of $O(n * n)$ or $O(n^2)$.

20.3.2 Selection Sort

Figure 20.5 uses the **selection sort** algorithm—another easy-to-implement, but inefficient, sorting algorithm—to sort a 10-element `array`'s values into ascending order. Function template `selectionSort` (lines 9–27) implements the algorithm.

```
1  // Fig. 20.5: fig08_13.cpp
2  // Sorting an array into ascending order with selection sort.
3  #include <array>
4  #include <iomanip>
5  #include <iostream>
6  using namespace std;
7
8  // sort an array into ascending order
9  template < typename T, size_t size >
10 void selectionSort( array< T, size > &items )
11 {
```

Fig. 20.5 | Sorting an array into ascending order with selection sort. (Part 1 of 2.)

```
12    // loop over size - 1 elements
13    for ( size_t i = 0; i < items.size() - 1; ++i )
14    {
15       size_t indexOfSmallest = i; // will hold index of smallest element
16
17       // loop to find index of smallest element
18       for ( size_t index = i + 1; index < items.size(); ++index )
19          if ( items[ index ] < items[ indexOfSmallest ] )
20             indexOfSmallest = index;
21
22       // swap the elements at positions i and indexOfSmallest
23       T hold = items[ i ];
24       items[ i ] = items[ indexOfSmallest ];
25       items[ indexOfSmallest ] = hold;
26    } // end for
27 } // end function insertionSort
28
29 int main()
30 {
31    const size_t arraySize = 10;
32    array < int, arraySize > data =
33       { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
34
35    cout << "Unsorted array:\n";
36
37    // output original array
38    for ( size_t i = 0; i < arraySize; ++i )
39       cout << setw( 4 ) << data[ i ];
40
41    selectionSort( data ); // sort the array
42
43    cout << "\nSorted array:\n";
44
45    // output sorted array
46    for ( size_t i = 0; i < arraySize; ++i )
47       cout << setw( 4 ) << data[ i ];
48
49    cout << endl;
50 } // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Fig. 20.5 | Sorting an array into ascending order with selection sort. (Part 2 of 2.)

Selection Sort Algorithm

The algorithm's first iteration selects the smallest element value and swaps it with the first element's value. The second iteration selects the second-smallest element value (which is the smallest of the remaining elements) and swaps it with the second element's value. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element's value, leaving the largest value in the last element. After

the i^{th} iteration, the smallest i values will be sorted into increasing order in the first i `array` elements.

First Iteration

Lines 32–33 declare and initialize the `array` named `data` with the following values:

```
34   56   4   10   77   51   93   30   5   52
```

The selection sort first determines the smallest value (4) in the array, which is in element 2. The algorithm swaps 4 with the value in element 0 (34), resulting in

```
4   56   34   10   77   51   93   30   5   52
```

Second Iteration

The algorithm then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8. The program swaps the 5 with the 56 in element 1, resulting in

```
4   5   34   10   77   51   93   30   56   52
```

Third Iteration

On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

```
4   5   10   34   77   51   93   30   56   52
```

The process continues until the array is fully sorted.

```
4   5   10   30   34   51   52   56   77   93
```

After the first iteration, the *smallest* element is in the *first* position; after the second iteration, the *two smallest* elements are *in order* in the *first two* positions and so on.

Function Template `selectionSort`

Function template `selectionSort` performs the sorting in lines 13–26. The loop iterates `size - 1` times. Line 15 declares and initializes the variable `indexOfSmallest`, which stores the index of the smallest element in the unsorted portion of the `array`. Lines 18–20 iterate over the remaining array elements. For each element, line 19 compares the current element's value to the value at `indexOfSmallest`. If the current element is smaller, line 20 assigns the current element's index to `indexOfSmallest`. When this loop finishes, `indexOfSmallest` contains the index of the smallest element remaining in the `array`. Lines 23–25 then swap the elements at positions `i` and `indexOfSmallest`, using the temporary variable `hold` to store `items[i]`'s value while that element is assigned `items[indexOfSmallest]`.

Efficiency of Selection Sort

The selection sort algorithm iterates $n - 1$ times, each time swapping the smallest remaining element into its sorted position. Locating the smallest remaining element requires $n - 1$ comparisons during the first iteration, $n - 2$ during the second iteration, then $n - 3, \ldots, 3, 2, 1$. This results in a total of $n(n - 1)/2$ or $(n^2 - n)/2$ comparisons. In Big O notation, smaller terms drop out and constants are ignored, leaving a Big O of $O(n^2)$. Can we develop sorting algorithms that perform *better* than $O(n^2)$?

20.3.3 Merge Sort (A Recursive Implementation)

Merge sort is an *efficient* sorting algorithm but is conceptually *more complex* than insertion sort and selection sort. The merge sort algorithm sorts an `array` by splitting it into two equal-sized sub-`array`s, sorting each sub-`array` then *merging* them into one larger `array`. With an odd number of elements, the algorithm creates the two sub-`array`s such that one has one more element than the other.

Merge sort performs the merge by looking at each sub-`array`'s first element, which is also the smallest element in that sub-`array`. Merge sort takes the smallest of these and places it in the first element of merged sorted `array`. If there are still elements in the sub-`array`, merge sort looks at the second element in that sub-`array` (which is now the smallest element remaining) and compares it to the first element in the other sub-`array`. Merge sort continues this process until the merged `array` is filled. Once a sub-`array` has no more elements, the merge copies the other `array`'s remaining elements into the merged array.

Sample Merge

Suppose the algorithm has already merged smaller `array`s to create sorted `array`s A:

```
4  10  34  56  77
```

and B:

```
5  30  51  52  93
```

Merge sort merges these `array`s into a sorted `array`. The smallest value in A is 4 (located in the zeroth element of A). The smallest value in B is 5 (located in the zeroth element of B). In order to determine the smallest element in the larger `array`, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the value of the first element in the merged `array`. The algorithm continues by comparing 10 (the value of the second element in A) to 5 (the value of the first element in B). The value from B is smaller, so 5 becomes the value of the second element in the larger `array`. The algorithm continues by comparing 10 to 30, with 10 becoming the value of the third element in the `array`, and so on.

Recursive Implementation

Our merge sort implementation is *recursive*. The *base case* is an `array` with one element. Such an `array` is, of course, sorted, so merge sort immediately returns when it's called with a one-element `array`. The *recursion step* splits an `array` of two or more elements into two equal-sized sub-`array`s, recursively sorts each sub-`array`, then merges them into one larger, sorted `array`. [Again, if there is an odd number of elements, one sub-`array` is one element larger than the other.]

Demonstrating Merge Sort

Figure 20.6 implements and demonstrates the merge sort algorithm. Throughout the program's execution, we use function template `displayElements` (lines 10–21) to display the portions of the `array` that are currently being split and merged. Function templates `mergeSort` (lines 24–49) and `merge` (lines 52–98) implement the merge sort algorithm. Function `main` (lines 100–125) creates an `array`, populates it with random integers, executes the algorithm (line 120) and displays the sorted array. The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm.

```
// Fig 20.6: Fig20_06.cpp
// Sorting an array into ascending order with merge sort.
#include <array>
#include <ctime>
#include <iostream>
#include <random>
using namespace std;

// display array elements from index low through index high
template < typename T, size_t size >
void displayElements( const array< T, size > &items,
   size_t low, size_t high )
{
   for ( size_t i = 0; i < items.size() && i < low; ++i )
      cout << "   "; // display spaces for alignment

   for ( size_t i = low; i < items.size() && i <= high; ++i )
      cout << items[ i ] << " "; // display element

   cout << endl;
} // end function displayElements

// split array, sort subarrays and merge subarrays into sorted array
template < typename T, size_t size >
void mergeSort( array< T, size > &items, size_t low, size_t high )
{
   // test base case; size of array equals 1
   if ( ( high - low ) >= 1 ) // if not base case
   {
      int middle1 = ( low + high ) / 2; // calculate middle of array
      int middle2 = middle1 + 1; // calculate next element over

      // output split step
      cout << "split:   ";
      displayElements( items, low, high );
      cout << "         ";
      displayElements( items, low, middle1 );
      cout << "         ";
      displayElements( items, middle2, high );
      cout << endl;

      // split array in half; sort each half (recursive calls)
      mergeSort( items, low, middle1 ); // first half of array
      mergeSort( items, middle2, high ); // second half of array

      // merge two sorted arrays after split calls return
      merge( items, low, middle1, middle2, high );
   } // end if
} // end function mergeSort
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 1 of 4.)

```
51  // merge two sorted subarrays into one sorted subarray
52  template < typename T, size_t size >
53  void merge( array< T, size > &items,
54     size_t left, size_t middle1, size_t middle2, size_t right )
55  {
56     size_t leftIndex = left; // index into left subarray
57     size_t rightIndex = middle2; // index into right subarray
58     size_t combinedIndex = left; // index into temporary working array
59     array< T, size > combined; // working array
60
61     // output two subarrays before merging
62     cout << "merge:    ";
63     displayElements( items, left, middle1 );
64     cout << "          ";
65     displayElements( items, middle2, right );
66     cout << endl;
67
68     // merge arrays until reaching end of either
69     while ( leftIndex <= middle1 && rightIndex <= right )
70     {
71        // place smaller of two current elements into result
72        // and move to next space in array
73        if ( items[ leftIndex ] <= items[ rightIndex ] )
74           combined[ combinedIndex++ ] = items[ leftIndex++ ];
75        else
76           combined[ combinedIndex++ ] = items[ rightIndex++ ];
77     } // end while
78
79     if ( leftIndex == middle2 ) // if at end of left array
80     {
81        while ( rightIndex <= right ) // copy in rest of right array
82           combined[ combinedIndex++ ] = items[ rightIndex++ ];
83     } // end if
84     else // at end of right array
85     {
86        while ( leftIndex <= middle1 ) // copy in rest of left array
87           combined[ combinedIndex++ ] = items[ leftIndex++ ];
88     } // end else
89
90     // copy values back into original array
91     for ( size_t i = left; i <= right; ++i )
92        items[ i ] = combined[ i ];
93
94     // output merged array
95     cout << "          ";
96     displayElements( items, left, right );
97     cout << endl;
98  } // end function merge
99
100 int main()
101 {
102    // use the default random-number generation engine to produce
103    // uniformly distributed pseudorandom int values from 10 to 99
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 2 of 4.)

```
104    default_random_engine engine(
105       static_cast<unsigned int>( time( nullptr ) ) );
106    uniform_int_distribution<unsigned int> randomInt( 10, 99 );
107
108    const size_t arraySize = 10; // size of array
109    array< int, arraySize > data; // create array
110
111    // fill data with random values
112    for ( int &item : data )
113       item = randomInt( engine );
114
115    // display data's values before mergeSort
116    cout << "Unsorted array:" << endl;
117    displayElements( data, 0, data.size() - 1 );
118    cout << endl;
119
120    mergeSort( data, 0, data.size() - 1 ); // sort the array data
121
122    // display data's values after mergeSort
123    cout << "Sorted array:" << endl;
124    displayElements( data, 0, data.size() - 1 );
125 } // end main
```

```
Unsorted array:
 30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                         18 60 78 26 54

split:    30 47 22 67 79
          30 47 22
                   67 79

split:    30 47 22
          30 47
                22

split:    30 47
          30
             47

merge:    30
             47
          30 47

merge:    30 47
                22
          22 30 47

split:              67 79
                    67
                       79
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 3 of 4.)

```
merge:              67
                       79
                    67 79

merge:    22 30 47
                    67 79
          22 30 47 67 79

split:                      18 60 78 26 54
                            18 60 78
                                     26 54

split:                      18 60 78
                            18 60
                                  78

split:                      18 60
                            18
                               60

merge:                      18
                               60
                            18 60

merge:                      18 60
                                  78
                            18 60 78

split:                                26 54
                                      26
                                         54

merge:                                26
                                         54
                                      26 54

merge:                      18 60 78
                                      26 54
                            18 26 54 60 78

merge:    22 30 47 67 79
                            18 26 54 60 78
          18 22 26 30 47 54 60 67 78 79

Sorted array:
 18 22 26 30 47 54 60 67 78 79
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 4 of 4.)

***Function* `mergeSort`**

Recursive function `mergeSort` (lines 24–49) receives as parameters the `array` to sort and the `low` and `high` indices of the range of elements to sort. Line 28 tests the base case. If the `high` index minus the `low` index is `0` (i.e., a one-element sub-`array`), the function simply returns. If the difference between the indices is greater than or equal to `1`, the function splits the `array` in two—lines 30–31 determine the split point. Next, line 43 recursively calls function `mergeSort` on the `array`'s first half, and line 44 recursively calls function `mergeSort` on the `array`'s second half. When these two function calls return, each half is

sorted. Line 47 calls function `merge` (lines 52–98) on the two halves to combine the two sorted `array`s into one larger sorted `array`.

Function `merge`

Lines 69–77 in function `merge` loop until the program reaches the end of either sub-`array`. Line 73 tests which element at the beginning of the two sub-`array`s is smaller. If the element in the left sub-`array` is smaller or both are equal, line 74 places it in position in the combined `array`. If the element in the right sub-`array` is smaller, line 76 places it in position in the combined `array`. When the `while` loop completes, one entire sub-`array` is in the combined `array`, but the other sub-`array` still contains data. Line 79 tests whether the left sub-`array` has reached the end. If so, lines 81–82 fill the combined `array` with the elements of the right sub-`array`. If the left sub-`array` has not reached the end, then the right sub-`array` must have reached the end, and lines 86–87 fill the combined `array` with the elements of the left sub-`array`. Finally, lines 91–92 copy the combined `array` into the original `array`.

Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort—although that may be difficult to believe when looking at the busy output in Fig. 20.6. Consider the first (nonrecursive) call to function `mergeSort` (line 120). This results in two recursive calls to function `mergeSort` with sub-`array`s that are each approximately half the original `array`'s size, and a single call to function `merge`. The call to `merge` requires, at worst, $n - 1$ comparisons to fill the original `array`, which is $O(n)$. (Recall that each `array` element is chosen by comparing one element from each of the sub-`array`s.) The two calls to function `mergeSort` result in four more recursive calls to function `mergeSort`—each with a sub-`array` approximately one-quarter the size of the original `array`—and two calls to function `merge`. These two calls to function `merge` each require, at worst, $n/2 - 1$ comparisons, for a total number of comparisons of $O(n)$. This process continues, each call to `mergeSort` generating two additional calls to `mergeSort` and a call to `merge`, until the algorithm has split the `array` into one-element sub-`array`s. At each level, $O(n)$ comparisons are required to merge the sub-`array`s. Each level splits the size of the `array`s in half, so doubling the size of the `array` requires one more level. Quadrupling the size of the `array` requires two more levels. This pattern is logarithmic and results in $\log_2 n$ levels. This results in a total efficiency of $O(n \log n)$.

Summary of Searching and Sorting Algorithm Efficiencies

Figure 20.7 summarizes the searching and sorting algorithms we cover in this chapter and lists the Big O for each. Figure 20.8 lists the Big O categories we've covered in this chapter along with a number of values for n to highlight the differences in the growth rates.

Algorithm	Location	Big O
Searching Algorithms		
Linear search	Section 20.2.1	$O(n)$
Binary search	Section 20.2.2	$O(\log n)$

Fig. 20.7 | Searching and sorting algorithms with Big O values. (Part 1 of 2.)

Algorithm	Location	Big O
Recursive linear search	Exercise 20.8	$O(n)$
Recursive binary search	Exercise 20.9	$O(\log n)$
Sorting Algorithms		
Insertion sort	Section 20.3.1	$O(n^2)$
Selection sort	Section 20.3.2	$O(n^2)$
Merge sort	Section 20.3.3	$O(n \log n)$
Bubble sort	Exercises 20.5–20.6	$O(n^2)$
Quicksort	Exercise 20.10	Worst case: $O(n^2)$ Average case: $O(n \log n)$

Fig. 20.7 | Searching and sorting algorithms with Big O values. (Part 2 of 2.)

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$2^{10} \cdot 10$	2^{20}
2^{20}	1,000,000	20	2^{20}	$2^{20} \cdot 20$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$2^{30} \cdot 30$	2^{60}

Fig. 20.8 | Approximate number of comparisons for common Big O notations.

20.4 Wrap-Up

This chapter discussed searching and sorting data. We began by discussing searching. We first presented the simple, but inefficient linear search algorithm. Then, we presented the binary search algorithm, which is faster but more complex than linear search. Next, we discussed sorting data. You learned two simple, but inneficient sorting techniques—insertion sort and selection sort. Then, we presented the merge sort algorithm, which is more efficient than either the insertion sort or the selection sort. Throughout the chapter we also introduced Big O notation, which helps you express the efficiency of an algorithm by measuring the worst-case runtime of an algorithm. Big O is useful for comparing algorithms so that you can choose the most efficient one. In the next chapter, we discuss typical string-manipulation operations provided by class template `basic_string`. We also introduce string stream-processing capabilities that allow strings to be input from and output to memory.

Summary

Section 20.1 Introduction

- Searching data involves determining whether a search key (p. 823) is present in the data and, if so, returning its location.
- Sorting (p. 823) involves arranging data into order.

- One way to describe the efficiency of an algorithm is with Big O notation (p. 823), which indicates how much work an algorithm must do to solve a problem.

Section 20.2 Searching Algorithms

- A key difference among searching algorithms is the amount of effort they require to return a result.

Section 20.2.1 Linear Search

- The linear search (p. 824) compares each array element with a search key. Because the `array` is not in any particular order, it's just as likely that the value will be found in the first element as the last. On average, the algorithm must compare the search key with half the `array` elements. To determine that a value is not in the `array`, the algorithm must compare the search key to every element in the `array`.
- Big O describes how an algorithm's effort varies depending on the number of elements in the data.
- An algorithm that's $O(1)$ has a constant runtime (p. 825)—the number of comparisons does not grow as the size of the `array` increases.
- An $O(n)$ algorithm is referred to as having a linear runtime (p. 826).
- Big O highlights dominant factors and ignores terms that are unimportant with high values of n.
- Big O notation represents the growth rate of algorithm runtimes, so constants are ignored.
- The linear search algorithm runs in $O(n)$ time.
- In the worst case for linear search every element must be checked to determine whether the search element exists. This occurs if the search key is the last element in the `array` or is not present.

Section 20.2.2 Binary Search

- Binary search (p. 827) is more efficient than linear search, but it requires that the `array` first be sorted. This is worthwhile only when the `array`, once sorted, will be searched many times.
- The first iteration of binary search tests the middle element. If this is the search key, the algorithm returns its location. If the search key is less than the middle element, binary search continues with the first half of the `array`. If the search key is greater than the middle element, binary search continues with the second half. Each iteration tests the middle value of the remaining `array` and, if the element is not found, eliminates from consideration half of the remaining elements.
- Binary search is more efficient than linear search, because with each comparison it eliminates from consideration half of the elements in the `array`.
- Binary search runs in $O(\log n)$ (p. 831) time.
- If the size of the `array` is doubled, binary search requires only one extra comparison to complete.

Section 20.3.1 Insertion Sort

- The first iteration of an insertion sort (p. 832) takes the second element and, if it's less than the first element, swaps it with the first element (i.e., the algorithm *inserts* the second element in front of the first element). The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the i^{th} iteration of this algorithm, the first i elements in the original array will be sorted. For small arrays, the insertion sort is acceptable, but for larger arrays it's inefficient compared to other more sophisticated sorting algorithms.
- The insertion sort algorithm runs in $O(n^2)$ time.

Section 20.3.2 Selection Sort

- The first iteration of selection sort (p. 834) selects the smallest element and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest remain-

ing element) and swaps it with the second element. This continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. At the i^{th} iteration, the smallest i elements are sorted into the first i elements.

- The selection sort algorithm runs in $O(n^2)$ time (p. 826).

Section 20.3.3 Merge Sort (A Recursive Implementation)

- Merge sort (p. 837) is faster, but more complex to implement, than insertion sort and selection sort.
- The merge sort algorithm sorts an `array` by splitting the `array` into two equal-sized sub-`arrays`, sorting each sub-`array` and merging the sub-`arrays` into one larger `array`.
- Merge sort's base case is an `array` with one element, which is already sorted. The merge part of merge sort takes two sorted `arrays` (these could be one-element `arrays`) and combines them into one larger sorted `array`.
- Merge sort performs the merge by looking at the first element in each `array`, which is also the smallest element in each. Merge sort takes the smallest of these and places it in the first element of the larger, sorted `array`. If there are still elements in the sub-`array`, merge sort looks at the second element in that sub-`array` (which is now the smallest element remaining) and compares it to the first element in the other sub-`array`. Merge sort continues this process until the larger `array` is filled.
- In the worst case, the first call to merge sort has to make $O(n)$ comparisons to fill the n slots in the final `array`.
- The merging portion of the merge sort algorithm is performed on two sub-`arrays`, each of approximately size $n/2$. Creating each of these sub-`arrays` requires $n/2 - 1$ comparisons for each sub-`array`, or $O(n)$ comparisons total. This pattern continues, as each level works on twice as many `arrays`, but each is half the size of the previous `array`.
- Similar to binary search, this halving results in log n levels, each level requiring $O(n)$ comparisons, for a total efficiency of $O(n \log n)$ (p. 842).

Self-Review Exercises

20.1 Fill in the blanks in each of the following statements:

a) A selection sort application would take approximately _____ times as long to run on a 128-element `array` as on a 32-element `array`.

b) The efficiency of merge sort is _____.

20.2 What key aspect of both the binary search and the merge sort accounts for the logarithmic portion of their respective Big Os?

20.3 In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

20.4 In the text, we say that after the merge sort splits the `array` into two sub-`arrays`, it then sorts these two sub-`arrays` and merges them. Why might someone be puzzled by our statement that "it then sorts these two sub-`arrays`"?

Answers to Self-Review Exercises

20.1 a) 16, because an $O(n^2)$ algorithm takes 16 times as long to sort four times as much information. b) $O(n \log n)$.

20.2 Both of these algorithms incorporate "halving"—somehow reducing something by half. The binary search eliminates from consideration half of the `array` after each comparison. The merge sort splits the `array` in half each time it's called.

20.3 The insertion sort is easier to understand and to implement than the merge sort. The merge sort is far more efficient ($O(n \log n)$) than the insertion sort ($O(n^2)$).

20.4 In a sense, it does not really sort these two sub-arrays. It simply keeps splitting the original `array` in half until it provides a one-element sub-`array`, which is, of course, sorted. It then builds up the original two sub-arrays by merging these one-element `arrays` to form larger sub-arrays, which are then merged, and so on.

Exercises

20.5 ***(Bubble Sort)*** Implement the bubble sort algorithm—another simple yet inefficient sorting technique. It's called bubble sort or sinking sort because smaller values gradually "bubble" their way to the top of the `array` (i.e., toward the first element) like air bubbles rising in water, while the larger values sink to the bottom (end) of the `array`. The technique uses nested loops to make several passes through the `array`. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the `array`.

The first pass compares the first two element values of the `array` and swaps them if necessary. It then compares the second and third element values in the `array`. The end of this pass compares the last two element values in the `array` and swaps them if necessary. After one pass, the largest value will be in the last element. After two passes, the largest two values will be in the last two elements. Explain why bubble sort is an $O(n^2)$ algorithm.

20.6 ***(Enhanced Bubble Sort)*** Make the following simple modifications to improve the performance of the bubble sort you developed in Exercise 20.5:

a) After the first pass, the largest value is guaranteed to be in the highest-numbered element of the `array`; after the second pass, the two highest values are "in place"; and so on. Instead of making nine comparisons (for a 10-element `array`) on every pass, modify the bubble sort to make only the eight necessary comparisons on the second pass, seven on the third pass, and so on.

b) The data in the `array` may already be in the proper order or near-proper order, so why make nine passes (of a 10-element `array`) if fewer will suffice? Modify the sort to check at the end of each pass whether any swaps have been made. If none have been made, the data must already be in the proper order, so the program should terminate. If swaps have been made, at least one more pass is needed.

20.7 (***Bucket Sort***) A bucket sort begins with a one-dimensional `array` of positive integers to be sorted and a two-dimensional `array` of integers with rows indexed from 0 to 9 and columns indexed from 0 to $n - 1$, where n is the number of values to be sorted. Each row of the two-dimensional `array` is referred to as a *bucket*. Write a class named `BucketSort` containing a function called `sort` that operates as follows:

a) Place each value of the one-dimensional `array` into a row of the bucket `array`, based on the value's "ones" (rightmost) digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This procedure is called a *distribution pass*.

b) Loop through the bucket `array` row by row, and copy the values back to the original `array`. This procedure is called a *gathering pass*. The new order of the preceding values in the one-dimensional `array` is 100, 3 and 97.

c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second (tens digit) pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional `array` is 100, 3 and 97. On the third (hundreds digit) pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After this last gathering pass, the original `array` is in sorted order.

Note that the two-dimensional array of buckets is 10 times the length of the integer array being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory—the bubble sort requires space for only one additional element of data. This comparison is an example of the space–time trade-off: The bucket sort uses more memory than the bubble sort, but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second two-dimensional bucket array and repeatedly swap the data between the two bucket arrays.

20.8 (***Recursive Linear Search)*** Modify Fig. 20.2 to use recursive function recursiveLinearSearch to perform a linear search of the array. The function should receive the array, the search key and starting index as arguments. If the search key is found, return its index in the array; otherwise, return -1. Each call to the recursive function should check one element value in the array.

20.9 (***Recursive Binary Search***) Modify Fig. 20.3 to use recursive function recursiveBinarySearch to perform a binary search of the array. The function should receive the array, the search key, starting index and ending index as arguments. If the search key is found, return its index in the array. If the search key is not found, return -1.

20.10 ***(Quicksort***) The recursive sorting technique called quicksort uses the following basic algorithm for a one-dimensional array of values:

a) *Partitioning Step*: Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element's value, and all values to the right of the element in the array are greater than the element's value—we show how to do this below). We now have one value in its proper location and two unsorted sub-arrays.

b) *Recursion Step*: Perform the *Partitioning Step* on each unsorted sub-array.

Each time *Step 1* is performed on a sub-array, another element is placed in its final location of the sorted array, and two unsorted sub-arrays are created. When a sub-array consists of one element, that sub-array must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each sub-array? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

Starting from the rightmost element of the array, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The values now reside in the array as follows:

12 2 6 4 89 8 10 **37** 68 45

Element 12 is in italics to indicate that it was just swapped with **37**.

Starting from the left of the array, but beginning with the element after 12, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The values now reside in the array as follows:

12 2 6 4 **37** 8 10 *89* 68 45

Starting from the right, but beginning with the element before 89, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The values now reside in the array as follows:

12 2 6 4 *10* 8 **37** 89 68 45

Starting from the left, but beginning with the element after 10, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. There are no more

elements greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the array, there are two unsorted sub-arrays. The sub-array with values less than 37 contains 12, 2, 6, 4, 10 and 8. The sub-array with values greater than 37 contains 89, 68 and 45. The sort continues with both sub-arrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive function `quickSort` to sort a single-subscripted integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function `partition` should be called by `quickSort` to perform the partitioning step.

Class `string` and String Stream Processing: A Deeper Look

21

Suit the action to the word, the word to the action; with this special observance, that you o'erstep not the modesty of nature.
—William Shakespeare

The difference between the almost-right word and the right word is really a large matter—it's the difference between the lightning bug and the lightning.
—Mark Twain

Mum's the word.
—Miguel de Cervantes

I have made this letter longer than usual, because I lack the time to make it short.
—Blaise Pascal

Objectives

In this chapter you'll:

- Manipulate `string` objects.
- Determine `string` characteristics.
- Find, replace and insert characters in `string`s.
- Convert `string` objects to pointer-based strings and vice versa.
- Use `string` iterators.
- Perform input from and output to `string`s in memory.
- Use C++11 numeric conversion functions.

Outline

21.1 Introduction

The class template **basic_string** provides typical string-manipulation operations such as copying, searching, etc. The template definition and all support facilities are defined in namespace std; these include the typedef statement

```
typedef basic_string< char > string;
```

that creates the alias type string for **basic_string<char>**. A typedef is also provided for the **wchar_t** type (wstring). Type wchar_t[1] stores characters (e.g., two-byte characters, four-byte characters, etc.) for supporting other character sets. We use string exclusively throughout this chapter. To use strings, include header <string>.

Initializing a **string** *Object*

A string object can be initialized with a constructor argument as in

```
string text( "Hello" ); // creates a string from a const char *
```

which creates a string containing the characters in "Hello", or with two constructor arguments as in

```
string name( 8, 'x' ); // string of 8 'x' characters
```

which creates a string containing eight 'x' characters. Class string also provides a *default constructor* (which creates an *empty* string) and a *copy constructor*. A string also can be initialized in its definition as in

```
string month = "March"; // same as: string month( "March" );
```

Remember that = in the preceding declaration is *not* an assignment; rather it's an *implicit call to the* string *class constructor*, which does the conversion.

1. Type wchar_t commonly is used to represent Unicode®, but wchar_t's size is not specified by the standard. C++11 also has types char16_t and char32_t for Unicode support. The Unicode Standard outlines a specification to produce consistent encoding of the world's characters and *symbols*. To learn more about the Unicode Standard, visit www.unicode.org.

***strings* Are Not Necessarily Null Terminated**
Unlike pointer-based `char *` strings, `string` objects are not necessarily null terminated. [*Note:* The C++ standard document provides only a description of the capabilities of class `string`—implementation is platform dependent.]

Length of a string
The length of a `string` can be retrieved with member function `size` and with member function `length`. The subscript operator, `[]` (which does not perform bounds checking), can be used with `string`s to access and modify individual characters. A `string` object has a first subscript of `0` and a last subscript of `size() - 1`.

Processing strings
Most `string` member functions take as arguments a *starting subscript location* and the number of characters on which to operate.

string I/O
The stream extraction operator (>>) is overloaded to support `string`s. The statements

```
string stringObject;
cin >> stringObject;
```

declare a `string` object and read a `string` from `cin`. Input is delimited by whitespace characters. When a delimiter is encountered, the input operation is terminated. Function `getline` also is overloaded for `string`s. Assuming `string1` is a `string`, the statement

```
getline( cin, string1 );
```

reads a `string` from the keyboard into `string1`. Input is delimited by a newline (`'\n'`), so `getLine` can read a line of text into a `string` object. You can specify an *alternate delimiter* as the optional third argument to `getline`.

Validating Input
In earlier chapters, we mentioned the importance of validating user input in industrial-strength code. The capabilities presented in this chapter—and the regular-expression capabilities shown in Section 24.5—are frequently used to perform validation.

21.2 string Assignment and Concatenation

Figure 21.1 demonstrates *`string` assignment* and *concatenation*. Line 4 includes header `<string>` for class `string`. The `string`s `string1`, `string2` and `string3` are created in lines 9–11. Line 13 assigns the value of `string1` to `string2`. After the assignment takes place, `string2` is a copy of `string1`. Line 14 uses member function `assign` to copy `string1` into `string3`. A separate copy is made (i.e., `string1` and `string3` are independent objects). Class `string` also provides an overloaded version of member function `assign` that copies a specified number of characters, as in

```
targetString.assign( sourceString, start, numberOfCharacters );
```

where `sourceString` is the `string` to be copied, `start` is the starting subscript and `numberOfCharacters` is the number of characters to copy.

Line 19 uses the subscript operator to assign `'r'` to `string3[ 2 ]` (forming `"car"`) and to assign `'r'` to `string2[ 0 ]` (forming `"rat"`). The `string`s are then output.

```
// Fig. 21.1: Fig21_01.cpp
// Demonstrating string assignment and concatenation.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string string1( "cat" );
   string string2; // initialized to the empty string
   string string3; // initialized to the empty string

   string2 = string1; // assign string1 to string2
   string3.assign( string1 ); // assign string1 to string3
   cout << "string1: " << string1 << "\nstring2: " << string2
      << "\nstring3: " << string3 << "\n\n";

   // modify string2 and string3
   string2[ 0 ] = string3[ 2 ] = 'r';

   cout << "After modification of string2 and string3:\n" << "string1: "
      << string1 << "\nstring2: " << string2 << "\nstring3: ";

   // demonstrating member function at
   for ( size_t i = 0; i < string3.size(); ++i )
      cout << string3.at( i ); // can throw out_of_range exception

   // declare string4 and string5
   string string4( string1 + "apult" ); // concatenation
   string string5; // initialized to the empty string

   // overloaded +=
   string3 += "pet"; // create "carpet"
   string1.append( "acomb" ); // create "catacomb"

   // append subscript locations 4 through end of string1 to
   // create string "comb" (string5 was initially empty)
   string5.append( string1, 4, string1.size() - 4 );

   cout << "\n\nAfter concatenation:\nstring1: " << string1
      << "\nstring2: " << string2 << "\nstring3: " << string3
      << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
} // end main
```

```
string1: cat
string2: cat
string3: cat

After modification of string2 and string3:
string1: cat
string2: rat
string3: car
```

Fig. 21.1 | Demonstrating `string` assignment and concatenation. (Part 1 of 2.)

```
After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb
```

Fig. 21.1 | Demonstrating `string` assignment and concatenation. (Part 2 of 2.)

Lines 25–26 output the contents of `string3` one character at a time using member function **`at`**. Member function `at` provides **checked access** (or **range checking**); i.e., going past the end of the `string` throws an `out_of_range` exception. *The subscript operator, `[]`, does not provide checked access.* This is consistent with its use on arrays. Note that you can also iterate through the characters in a string using C++11's range-based `for` as in

```
for ( char c : string3 )
   cout << c;
```

which ensures that you do not access any elements outside the `string`'s bounds.

Common Programming Error 21.1

Accessing an element beyond the size of the `string` using the subscript operator is an unreported logic error.

String `string4` is declared (line 29) and initialized to the result of concatenating `string1` and `"apult"` using the *overloaded + operator*, which for class `string` denotes *concatenation*. Line 33 uses the *overloaded addition assignment operator, +=*, to concatenate `string3` and `"pet"`. Line 34 uses member function **`append`** to concatenate `string1` and `"acomb"`.

Line 38 appends the string `"comb"` to empty `string` `string5`. This member function is passed the `string` (`string1`) to retrieve characters from, the starting subscript in the `string` (4) and the number of characters to append (the value returned by `string1.size() - 4`).

21.3 Comparing `strings`

Class `string` provides member functions for comparing `strings`. Figure 21.2 demonstrates class `string`'s comparison capabilities.

```
1  // Fig. 21.2: Fig21_02.cpp
2  // Comparing strings.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "Testing the comparison functions." );
```

Fig. 21.2 | Comparing `strings`. (Part 1 of 3.)

```
10    string string2( "Hello" );
11    string string3( "stinger" );
12    string string4( string2 ); // "Hello"
13
14    cout << "string1: " << string1 << "\nstring2: " << string2
15       << "\nstring3: " << string3 << "\nstring4: " << string4 << "\n\n";
16
17    // comparing string1 and string4
18    if ( string1 == string4 )
19       cout << "string1 == string4\n";
20    else if ( string1 > string4 )
21          cout << "string1 > string4\n";
22    else // string1 < string4
23       cout << "string1 < string4\n";
24
25    // comparing string1 and string2
26    int result = string1.compare( string2 );
27
28    if ( result == 0 )
29       cout << "string1.compare( string2 ) == 0\n";
30    else if ( result > 0 )
31       cout << "string1.compare( string2 ) > 0\n";
32    else // result < 0
33       cout << "string1.compare( string2 ) < 0\n";
34
35    // comparing string1 (elements 2-5) and string3 (elements 0-5)
36    result = string1.compare( 2, 5, string3, 0, 5 );
37
38    if ( result == 0 )
39       cout << "string1.compare( 2, 5, string3, 0, 5 ) == 0\n";
40    else if ( result > 0 )
41       cout << "string1.compare( 2, 5, string3, 0, 5 ) > 0\n";
42    else // result < 0
43       cout << "string1.compare( 2, 5, string3, 0, 5 ) < 0\n";
44
45    // comparing string2 and string4
46    result = string4.compare( 0, string2.size(), string2 );
47
48    if ( result == 0 )
49       cout << "string4.compare( 0, string2.size(), "
50          << "string2 ) == 0" << endl;
51    else if ( result > 0 )
52       cout << "string4.compare( 0, string2.size(), "
53          << "string2 ) > 0" << endl;
54    else // result < 0
55       cout << "string4.compare( 0, string2.size(), "
56             << "string2 ) < 0" << endl;
57
58    // comparing string2 and string4
59    result = string2.compare( 0, 3, string4 );
60
61    if ( result == 0 )
62       cout << "string2.compare( 0, 3, string4 ) == 0" << endl;
```

Fig. 21.2 | Comparing strings. (Part 2 of 3.)

```
63      else if ( result > 0 )
64         cout << "string2.compare( 0, 3, string4 ) > 0" << endl;
65      else // result < 0
66         cout << "string2.compare( 0, 3, string4 ) < 0" << endl;
67   } // end main
```

```
string1: Testing the comparison functions.
string2: Hello
string3: stinger
string4: Hello

string1 > string4
string1.compare( string2 ) > 0
string1.compare( 2, 5, string3, 0, 5 ) == 0
string4.compare( 0, string2.size(), string2 ) == 0
string2.compare( 0, 3, string4 ) < 0
```

Fig. 21.2 | Comparing `string`s. (Part 3 of 3.)

The program declares four `string`s (lines 9–12) and outputs each (lines 14–15). Line 18 tests `string1` against `string4` for equality using the *overloaded equality operator*. If the condition is `true`, `"string1 == string4"` is output. If the condition is `false`, the condition in line 20 is tested. All the `string` class overloaded relational and equality operator functions return `bool` values.

Line 26 uses `string` member function **`compare`** to compare `string1` to `string2`. Variable `result` is assigned `0` if the `string`s are equivalent, a *positive number* if `string1` is **lexicographically** greater than `string2` or a *negative number* if `string1` is *lexicographically* less than `string2`. When we say that a `string` is *lexicographically* less than another, we mean that the `compare` method uses the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string` to determine that the first `string` is less than the second. Because a `string` starting with `'T'` is considered lexicographically greater than a string starting with `'H'`, `result` is assigned a value greater than `0`, as confirmed by the output. A lexicon is a dictionary.

Line 36 compares portions of `string1` and `string3` using an *overloaded* version of member function `compare`. The first two arguments (`2` and `5`) specify the *starting subscript* and *length* of the portion of `string1` (`"sting"`) to compare with `string3`. The third argument is the comparison `string`. The last two arguments (`0` and `5`) are the *starting subscript* and *length* of the portion of the comparison `string` being compared (also `"sting"`). The value assigned to `result` is `0` for equality, a positive number if `string1` is *lexicographically* greater than `string3` or a negative number if `string1` is *lexicographically* less than `string3`. The two pieces being compared here are identical, so `result` is assigned `0`.

Line 46 uses another *overloaded* version of function `compare` to compare `string4` and `string2`. The first two arguments are the same—the *starting subscript* and *length*. The last argument is the comparison `string`. The value returned is also the same—`0` for equality, a positive number if `string4` is *lexicographically* greater than `string2` or a negative number if `string4` is *lexicographically* less than `string2`. Because the two pieces of `string`s being compared here are identical, `result` is assigned `0`.

Line 59 calls member function `compare` to compare the first `3` characters in `string2` to `string4`. Because `"Hel"` is less than `"Hello"`, a value less than zero is returned.

21.4 Substrings

Class `string` provides member function **`substr`** for retrieving a substring from a `string`. The result is a new `string` object that's copied from the source `string`. Figure 21.3 demonstrates `substr`. The program declares and initializes a `string` at line 9. Line 13 uses member function `substr` to retrieve a substring from `string1`. The first argument specifies the *beginning subscript* of the desired substring; the second argument specifies the substring's *length*.

```
1  // Fig. 21.3: Fig21_03.cpp
2  // Demonstrating string member function substr.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "The airplane landed on time." );
10
11    // retrieve substring "plane" which
12    // begins at subscript 7 and consists of 5 characters
13    cout << string1.substr( 7, 5 ) << endl;
14 } // end main
```

```
plane
```

Fig. 21.3 | Demonstrating `string` member function `substr`.

21.5 Swapping strings

Class `string` provides member function **`swap`** for swapping `strings`. Figure 21.4 swaps two `strings`. Lines 9–10 declare and initialize `strings` `first` and `second`. Each `string` is then output. Line 15 uses `string` member function `swap` to swap the values of `first` and `second`. The two `strings` are printed again to confirm that they were indeed swapped. The `string` member function `swap` is useful for implementing programs that sort strings.

```
1  // Fig. 21.4: Fig21_04.cpp
2  // Using the swap function to swap two strings.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string first( "one" );
10    string second( "two" );
11
12    // output strings
13    cout << "Before swap:\n first: " << first << "\nsecond: " << second;
```

Fig. 21.4 | Using the `swap` function to swap two `strings`. (Part 1 of 2.)

```
14
15      first.swap( second ); // swap strings
16
17      cout << "\n\nAfter swap:\n first: " << first
18         << "\nsecond: " << second << endl;
19   } // end main
```

```
Before swap:
 first: one
second: two

After swap:
 first: two
second: one
```

Fig. 21.4 | Using the swap function to swap two strings. (Part 2 of 2.)

21.6 string Characteristics

Class string provides member functions for gathering information about a string's *size*, *length*, *capacity*, *maximum length* and other characteristics. A string's *size* or *length* is the number of characters currently stored in the string. A string's **capacity** is the number of characters that can be stored in the string without allocating more memory. The capacity of a string must be at least equal to the current size of the string, though it can be greater. The exact capacity of a string depends on the implementation. The **maximum size** is the largest possible size a string can have. If this value is exceeded, a length_error exception is thrown. Figure 21.5 demonstrates string class member functions for determining various characteristics of strings.

```
1   // Fig. 21.5: Fig21_05.cpp
2   // Printing string characteristics.
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   void printStatistics( const string & );
8
9   int main()
10  {
11     string string1; // empty string
12
13     cout << "Statistics before input:\n" << boolalpha;
14     printStatistics( string1 );
15
16     // read in only "tomato" from "tomato soup"
17     cout << "\n\nEnter a string: ";
18     cin >> string1; // delimited by whitespace
19     cout << "The string entered was: " << string1;
20
```

Fig. 21.5 | Printing string characteristics. (Part 1 of 3.)

```
21      cout << "\nStatistics after input:\n";
22      printStatistics( string1 );
23
24      // read in "soup"
25      cin >> string1; // delimited by whitespace
26      cout << "\n\nThe remaining string is: " << string1 << endl;
27      printStatistics( string1 );
28
29      // append 46 characters to string1
30      string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
31      cout << "\n\nstring1 is now: " << string1 << endl;
32      printStatistics( string1 );
33
34      // add 10 elements to string1
35      string1.resize( string1.size() + 10 );
36      cout << "\n\nStats after resizing by (length + 10):\n";
37      printStatistics( string1 );
38      cout << endl;
39   } // end main
40
41   // display string statistics
42   void printStatistics( const string &stringRef )
43   {
44      cout << "capacity: " << stringRef.capacity() << "\nmax size: "
45         << stringRef.max_size() << "\nsize: " << stringRef.size()
46         << "\nlength: " << stringRef.size()
47         << "\nempty: " << stringRef.empty();
48   } // end printStatistics
```

```
Statistics before input:
capacity: 15
max size: 4294967294
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 4294967294
size: 6
length: 6
empty: false

The remaining string is: soup
capacity: 15
max size: 4294967294
size: 4
length: 4
empty: false

string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
capacity: 63
max size: 4294967294
```

Fig. 21.5 | Printing `string` characteristics. (Part 2 of 3.)

```
size: 50
length: 50
empty: false

Stats after resizing by (length + 10):
capacity: 63
max size: 4294967294
size: 60
length: 60
empty: false
```

Fig. 21.5 | Printing `string` characteristics. (Part 3 of 3.)

The program declares empty `string` `string1` (line 11) and passes it to function `printStatistics` (line 14). Function `printStatistics` (lines 42–48) takes a reference to a `const` `string` as an argument and outputs the capacity (using member function `capacity`), maximum size (using member function `max_size`), size (using member function `size`), length (using member function `size`) and whether the `string` is empty (using member function `empty`). The initial call to `printStatistics` indicates that the initial values for the size and length of `string1` are `0`.

The size and length of `0` indicate that there are no characters stored in `string`. Recall that the *size* and *length* are always identical. In this implementation, the maximum size is 4,294,967,294. Object `string1` is an empty `string`, so function `empty` returns `true`.

Line 18 inputs a string. In this example, `"tomato soup"` is input. Because a space character is a delimiter, only `"tomato"` is stored in `string1`; however, `"soup"` remains in the input buffer. Line 22 calls function `printStatistics` to output statistics for `string1`. Notice in the output that the length is `6` and the capacity is `15`.

Line 25 reads `"soup"` from the input buffer and stores it in `string1`, thereby replacing `"tomato"`. Line 27 passes `string1` to `printStatistics`.

Line 30 uses the *overloaded* `+=` operator to concatenate a 46-character-long string to `string1`. Line 32 passes `string1` to `printStatistics`. The capacity has increased to `63` elements and the length is now `50`.

Line 35 uses member function `resize` to increase the length of `string1` by `10` characters. The additional elements are set to null characters. The output shows that the capacity has *not* changed and the length is now `60`.

21.7 Finding Substrings and Characters in a `string`

Class `string` provides `const` member functions for finding substrings and characters in a `string`. Figure 21.6 demonstrates the `find` functions.

```
1  // Fig. 21.6: Fig21_06.cpp
2  // Demonstrating the string find member functions.
3  #include <iostream>
4  #include <string>
```

Fig. 21.6 | Demonstrating the `string` `find` member functions. (Part 1 of 2.)

```
5  using namespace std;
6
7  int main()
8  {
9     string string1( "noon is 12 pm; midnight is not." );
10    int location;
11
12    // find "is" at location 5 and 24
13    cout << "Original string:\n" << string1
14       << "\n\n(find) \"is\" was found at: " << string1.find( "is" )
15       << "\n(rfind) \"is\" was found at: " << string1.rfind( "is" );
16
17    // find 'o' at location 1
18    location = string1.find_first_of( "misop" );
19    cout << "\n\n(find_first_of) found '" << string1[ location ]
20       << "' from the group \"misop\" at: " << location;
21
22    // find 'o' at location 28
23    location = string1.find_last_of( "misop" );
24    cout << "\n\n(find_last_of) found '" << string1[ location ]
25       << "' from the group \"misop\" at: " << location;
26
27    // find '1' at location 8
28    location = string1.find_first_not_of( "noi spm" );
29    cout << "\n\n(find_first_not_of) '" << string1[ location ]
30       << "' is not contained in \"noi spm\" and was found at: "
31       << location;
32
33    // find '.' at location 13
34    location = string1.find_first_not_of( "12noi spm" );
35    cout << "\n\n(find_first_not_of) '" << string1[ location ]
36       << "' is not contained in \"12noi spm\" and was "
37       << "found at: " << location << endl;
38
39    // search for characters not in string1
40    location = string1.find_first_not_of(
41       "noon is 12 pm; midnight is not." );
42    cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
43       << " returned: " << location << endl;
44 } // end main
```

```
Original string:
noon is 12 pm; midnight is not.

(find) "is" was found at: 5
(rfind) "is" was found at: 24

(find_first_of) found 'o' from the group "misop" at: 1

(find_last_of) found 'o' from the group "misop" at: 28

(find_first_not_of) '1' is not contained in "noi spm" and was found at: 8

(find_first_not_of) '.' is not contained in "12noi spm" and was found at: 13

find_first_not_of("noon is 12 pm; midnight is not.") returned: -1
```

Fig. 21.6 | Demonstrating the `string` `find` member functions. (Part 2 of 2.)

String string1 is declared and initialized in line 9. Line 14 attempts to find "is" in string1 using function find. If "is" is found, the subscript of the starting location of that string is returned. If the string is not found, the value string::npos (a public static constant defined in class string) is returned. This value is returned by the string find-related functions to indicate that a substring or character was not found in the string.

Line 15 uses member function rfind to search string1 *backward* (i.e., *right-to-left*). If "is" is found, the subscript location is returned. If the string is not found, string::npos is returned. [*Note:* The rest of the find functions presented in this section return the same type unless otherwise noted.]

Line 18 uses member function find_first_of to locate the *first* occurrence in string1 of any character in "misop". The searching is done from the beginning of string1. The character 'o' is found in element 1.

Line 23 uses member function find_last_of to find the *last* occurrence in string1 of any character in "misop". The searching is done from the end of string1. The character 'o' is found in element 28.

Line 28 uses member function find_first_not_of to find the *first* character in string1 *not* contained in "noi spm". The character '1' is found in element 8. Searching is done from the beginning of string1.

Line 34 uses member function find_first_not_of to find the *first* character *not* contained in "12noi spm". The character '.' is found in element 13. Searching is done from the beginning of string1.

Lines 40–41 use member function find_first_not_of to find the *first* character *not* contained in "noon is 12 pm; midnight is not.". In this case, the string being searched contains every character specified in the string argument. Because a character was not found, string::npos (which has the value -1 in this case) is returned.

21.8 Replacing Characters in a string

Figure 21.7 demonstrates string member functions for *replacing* and *erasing* characters. Lines 10–14 declare and initialize string string1. Line 20 uses string member function erase to erase everything from (and including) the character in position 62 to the end of string1. [*Note:* Each newline character occupies one character in the string.]

```
1  // Fig. 21.7: Fig21_07.cpp
2  // Demonstrating string member functions erase and replace.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     // compiler concatenates all parts into one string
10    string string1( "The values in any left subtree"
11       "\nare less than the value in the"
12       "\nparent node and the values in"
13       "\nany right subtree are greater"
14       "\nthan the value in the parent node" );
```

Fig. 21.7 | Demonstrating string member functions erase and replace. (Part 1 of 2.)

```
15
16      cout << "Original string:\n" << string1 << endl << endl;
17
18      // remove all characters from (and including) location 62
19      // through the end of string1
20      string1.erase( 62 );
21
22      // output new string
23      cout << "Original string after erase:\n" << string1
24         << "\nAfter first replacement:\n";
25
26      size_t position = string1.find( " " ); // find first space
27
28      // replace all spaces with period
29      while ( position != string::npos )
30      {
31         string1.replace( position, 1, "." );
32         position = string1.find( " ", position + 1 );
33      } // end while
34
35      cout << string1 << "\nAfter second replacement:\n";
36
37      position = string1.find( "." ); // find first period
38
39      // replace all periods with two semicolons
40      // NOTE: this will overwrite characters
41      while ( position != string::npos )
42      {
43         string1.replace( position, 2, "xxxxx;;yyy", 5, 2 );
44         position = string1.find( ".", position + 1 );
45      } // end while
46
47      cout << string1 << endl;
48   } // end main
```

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

Original string after erase:
The values in any left subtree
are less than the value in the

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

Fig. 21.7 | Demonstrating `string` member functions `erase` and `replace`. (Part 2 of 2.)

Lines 26–33 use `find` to locate each occurrence of the space character. Each space is then *replaced* with a period by a call to `string` member function **`replace`**. Function `replace` takes three arguments: the *subscript* of the character in the `string` at which replacement should *begin*, the *number of characters to replace* and the *replacement string*. Member function `find` returns `string::npos` when the search character is *not found*. In line 32, `1` is added to `position` to continue searching at the location of the *next* character.

Lines 37–45 use function `find` to find every period and another overloaded function `replace` to replace every period and its following character with two semicolons. The arguments passed to this version of `replace` are the subscript of the element where the replace operation begins, the number of characters to replace, a replacement character string from which a *substring* is selected to use as replacement characters, the element in the character string where the replacement substring begins and the number of characters in the replacement character string to use.

21.9 Inserting Characters into a `string`

Class `string` provides member functions for *inserting* characters into a `string`. Figure 21.8 demonstrates the `string` `insert` capabilities.

The program declares, initializes then outputs `string`s `string1`, `string2`, `string3` and `string4`. Line 19 uses `string` member function **`insert`** to insert `string2`'s content before element 10 of `string1`.

Line 22 uses `insert` to insert `string4` before `string3`'s element 3. The last two arguments specify the *starting* and *last* element of `string4` that should be inserted. Using `string::npos` causes the *entire* `string` to be inserted.

```
1  // Fig. 21.8: Fig21_08.cpp
2  // Demonstrating class string insert member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "beginning end" );
10    string string2( "middle " );
11    string string3( "12345678" );
12    string string4( "xx" );
13
14    cout << "Initial strings:\nstring1: " << string1
15       << "\nstring2: " << string2 << "\nstring3: " << string3
16       << "\nstring4: " << string4 << "\n\n";
17
18    // insert "middle" at location 10 in string1
19    string1.insert( 10, string2 );
20
21    // insert "xx" at location 3 in string3
22    string3.insert( 3, string4, 0, string::npos );
23
```

Fig. 21.8 | Demonstrating class `string` `insert` member functions. (Part 1 of 2.)

```
24      cout << "Strings after insert:\nstring1: " << string1
25         << "\nstring2: " << string2 << "\nstring3: " << string3
26         << "\nstring4: " << string4 << endl;
27   } // end main
```

```
Initial strings:
string1: beginning end
string2: middle
string3: 12345678
string4: xx

Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

Fig. 21.8 | Demonstrating class `string` `insert` member functions. (Part 2 of 2.)

21.10 Conversion to Pointer-Based char * Strings

You can convert `string` class objects to pointer-based strings. As mentioned earlier, unlike pointer-based strings, `string`s are *not necessarily null terminated*. These conversion functions are useful when a given function takes a pointer-based string as an argument. Figure 21.9 demonstrates conversion of `string`s to pointer-based strings.

```
1   // Fig. 21.9: Fig21_09.cpp
2   // Converting strings to pointer-based strings and character arrays.
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int main()
8   {
9      string string1( "STRINGS" ); // string constructor with char * arg
10     const char *ptr1 = nullptr; // initialize *ptr1
11     size_t length = string1.size();
12     char *ptr2 = new char[ length + 1 ]; // including null
13
14     // copy characters from string1 into allocated memory
15     string1.copy( ptr2, length, 0 ); // copy string1 to ptr2 char *
16     ptr2[ length ] = '\0'; // add null terminator
17
18     cout << "string string1 is " << string1
19        << "\nstring1 converted to a pointer-based string is "
20        << string1.c_str()   << "\nptr1 is ";
21
22     // Assign to pointer ptr1 the const char * returned by
23     // function data(). NOTE: this is a potentially dangerous
```

Fig. 21.9 | Converting `string`s to pointer-based strings and character arrays. (Part 1 of 2.)

```
24      // assignment. If string1 is modified, pointer ptr1 can
25      // become invalid.
26      ptr1 = string1.data(); // non-null terminated char array
27
28      // output each character using pointer
29      for ( size_t i = 0; i < length; ++i )
30         cout << *( ptr1 + i ); // use pointer arithmetic
31
32      cout << "\nptr2 is " << ptr2 << endl;
33      delete [] ptr2; // reclaim dynamically allocated memory
34   } // end main
```

```
string string1 is STRINGS
string1 converted to a pointer-based string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

Fig. 21.9 | Converting `strings` to pointer-based strings and character arrays. (Part 2 of 2.)

The program declares a `string`, a `size_t` and two `char` pointers (lines 9–12). The `string` `string1` is initialized to `"STRINGS"`, `ptr1` is initialized to `nullptr` and `length` is initialized to the length of `string1`. Memory of sufficient size to hold a pointer-based string equivalent of `string` `string1` is allocated dynamically and attached to `char` pointer `ptr2`.

Line 15 uses `string` member function **`copy`** to copy object `string1` into the `char` array pointed to by `ptr2`. Line 16 places a terminating null character in the array pointed to by `ptr2`.

Line 20 uses function **`c_str`** to obtain a `const char *` that points to a null terminated pointer-based string with the same content as `string1`. The pointer is passed to the stream insertion operator for output.

Line 26 assigns the `const char * ptr1` a pointer returned by class `string` member function **`data`**. This member function returns a *non-null-terminated* built-in character array. We do *not* modify `string` `string1` in this example. If `string1` were to be modified (e.g., the `string`'s dynamic memory changes its address due to a member function call such as `string1.insert( 0, "abcd" );`), `ptr1` could become invalid—which could lead to unpredictable results.

Lines 29–30 use pointer arithmetic to output the character array pointed to by `ptr1`. In lines 32–33, the pointer-based string `ptr2` is output and the memory allocated for `ptr2` is `deleted` to avoid a memory leak.

Common Programming Error 21.2

Not terminating the character array returned by `data` *with a null character can lead to execution-time errors.*

21.11 Iterators

Class `string` provides *iterators* (introduced in Chapter 15) for forward and backward *traversal* of `strings`. Iterators provide access to individual characters with a syntax that's similar to pointer operations. *Iterators are not range checked.* Figure 21.10 demonstrates iterators.

```
1  // Fig. 21.10: Fig21_10.cpp
2  // Using an iterator to output a string.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "Testing iterators" );
10    string::const_iterator iterator1 = string1.begin();
11
12    cout << "string1 = " << string1
13       << "\n(Using iterator iterator1) string1 is: ";
14
15    // iterate through string
16    while ( iterator1 != string1.end() )
17    {
18       cout << *iterator1; // dereference iterator to get char
19       ++iterator1; // advance iterator to next char
20    } // end while
21
22    cout << endl;
23 } // end main
```

```
string1 = Testing iterators
(Using iterator iterator1) string1 is: Testing iterators
```

Fig. 21.10 | Using an iterator to output a `string`.

Lines 9–10 declare `string string1` and **`string::const_iterator`** `iterator1`. Recall that a `const_iterator` *cannot* be used to modify the data that you're iterating through—in this case the `string`. Iterator `iterator1` is initialized to the beginning of `string1` with the `string` class member function **`begin`**. Two versions of `begin` exist—one that returns an `iterator` for iterating through a non-`const` `string` and a `const` version that returns a `const_iterator` for iterating through a `const` `string`. Line 12 outputs `string1`.

Lines 16–20 use iterator `iterator1` to "walk through" `string1`. Class `string` member function **`end`** returns an `iterator` (or a `const_iterator`) for the position past the last element of `string1`. Each element is printed by *dereferencing the iterator* much as you'd dereference a pointer, and the iterator is advanced one position using operator `++`. In C++11, lines 10 and 16–20 can be replaced with a range-based `for`, as in

```
for ( char c : string1 )
   cout << c;
```

Class `string` provides member functions **`rend`** and **`rbegin`** for accessing individual `string` characters in reverse from the end of a `string` toward the beginning. Member functions `rend` and `rbegin` return **`reverse_iterators`** or **`const_reverse_iterators`** (based on whether the `string` is non-`const` or `const`). Exercise 21.8 asks you to write a program that demonstrates these capabilities.

Good Programming Practice 21.1

When the operations involving the iterator should not modify the data being processed, use a `const_iterator`*. This is another example of employing the principle of least privilege.*

21.12 String Stream Processing

In addition to standard stream I/O and file stream I/O, C++ stream I/O includes capabilities for inputting from, and outputting to, `string`s in memory. These capabilities often are referred to as **in-memory I/O** or **string stream processing**.

Input from a `string` is supported by class **`istringstream`**. Output to a `string` is supported by class **`ostringstream`**. The class names `istringstream` and `ostringstream` are actually *aliases* defined by the `typedef`s

```
typedef basic_istringstream< char > istringstream;
typedef basic_ostringstream< char > ostringstream;
```

Class templates `basic_istringstream` and `basic_ostringstream` provide the same functionality as classes `istream` and `ostream` plus other member functions specific to *in-memory formatting*. Programs that use in-memory formatting must include the **`<sstream>`** and `<iostream>` headers.

Error-Prevention Tip 21.1

One application of these techniques is data validation. *A program can read an entire line at a time from the input stream into a* `string`*. Next, a validation routine can scrutinize the contents of the* `string` *and correct (or repair) the data, if necessary. Then the program can proceed to input from the* `string`*, knowing that the input data is in the proper format.*

Error-Prevention Tip 21.2

To assist with data validation, C++11 provides powerful regular-expression *capabilities. For example, if a program requires a user to enter a U.S. format telephone number (e.g.,* `(800) 555-1212`*), you can use a regular-expression pattern to confirm that the user's input matches the expected format. Many websites provide regular expressions for validating email addresses, URLs, phone numbers, addresses and other popular kinds of data. We introduce regular expressions and provide several examples in Chapter 24.*

Software Engineering Observation 21.1

Outputting to a `string` *is a nice way to take advantage of the powerful output formatting capabilities of C++ streams. Data can be prepared in a* `string` *to mimic the edited screen format. That* `string` *could be written to a disk file to preserve the screen image.*

An `ostringstream` object uses a `string` object to store the output data. The **`str`** member function of class `ostringstream` returns a copy of that `string`.

Demonstrating `ostringstream`

Figure 21.11 demonstrates an `ostringstream` object. The program creates `ostringstream` object `outputString` (line 10) and uses the stream insertion operator to output a series of `string`s and numerical values to the object.

```
1  // Fig. 21.11: Fig21_11.cpp
2  // Using an ostringstream object.
3  #include <iostream>
4  #include <string>
5  #include <sstream> // header for string stream processing
6  using namespace std;
7
8  int main()
9  {
10    ostringstream outputString; // create ostringstream instance
11
12    string string1( "Output of several data types " );
13    string string2( "to an ostringstream object:" );
14    string string3( "\n        double: " );
15    string string4( "\n           int: " );
16    string string5( "\naddress of int: " );
17
18    double double1 = 123.4567;
19    int integer = 22;
20
21    // output strings, double and int to ostringstream outputString
22    outputString << string1 << string2 << string3 << double1
23       << string4 << integer << string5 << &integer;
24
25    // call str to obtain string contents of the ostringstream
26    cout << "outputString contains:\n" << outputString.str();
27
28    // add additional characters and call str to output string
29    outputString << "\nmore characters added";
30    cout << "\n\nafter additional stream insertions,\n"
31       << "outputString contains:\n" << outputString.str() << endl;
32 } // end main
```

```
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540

after additional stream insertions,
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540
more characters added
```

Fig. 21.11 | Using an `ostringstream` object.

Lines 22–23 output `string string1`, `string string2`, `string string3`, `double double1`, `string string4`, `int integer`, `string string5` and the address of `int integer`—all to `outputString` in memory. Line 26 uses the stream insertion operator and the call `outputString.str()` to display a copy of the `string` created in lines 22–23. Line 29 demonstrates that more data can be *appended* to the `string` in memory by simply

issuing another stream insertion operation to `outputString`. Lines 30–31 display `string outputString` after appending additional characters.

An `istringstream` object inputs data from a `string` in memory to program variables. Data is stored in an `istringstream` object as characters. Input from the `istringstream` object works identically to input from any file. The end of the `string` is interpreted by the `istringstream` object as *end-of-file*.

Demonstrating `istringstream`

Figure 21.12 demonstrates input from an `istringstream` object. Lines 10–11 create `string input` containing the data and `istringstream` object `inputString` constructed to contain the data in `string input`. The `string input` contains the data

```
Input test 123 4.7 A
```

which, when read as input to the program, consist of two strings (`"Input"` and `"test"`), an `int` (`123`), a `double` (`4.7`) and a `char` (`'A'`). These characters are extracted to variables `string1`, `string2`, `integer`, `double1` and `character` in line 18.

```
1  // Fig. 21.12: Fig21_12.cpp
2  // Demonstrating input from an istringstream object.
3  #include <iostream>
4  #include <string>
5  #include <sstream>
6  using namespace std;
7
8  int main()
9  {
10    string input( "Input test 123 4.7 A" );
11    istringstream inputString( input );
12    string string1;
13    string string2;
14    int integer;
15    double double1;
16    char character;
17
18    inputString >> string1 >> string2 >> integer >> double1 >> character;
19
20    cout << "The following items were extracted\n"
21       << "from the istringstream object:" << "\nstring: " << string1
22       << "\nstring: " << string2 << "\n   int: " << integer
23       << "\ndouble: " << double1 << "\n  char: " << character;
24
25    // attempt to read from empty stream
26    long value;
27    inputString >> value;
28
29    // test stream results
30    if ( inputString.good() )
31       cout << "\n\nlong value is: " << value << endl;
32    else
33       cout << "\n\ninputString is empty" << endl;
34 } // end main
```

Fig. 21.12 | Demonstrating input from an `istringstream` object. (Part 1 of 2.)

```
The following items were extracted
from the istringstream object:
string: Input
string: test
   int: 123
double: 4.7
  char: A

inputString is empty
```

Fig. 21.12 | Demonstrating input from an `istringstream` object. (Part 2 of 2.)

The data is then output in lines 20–23. The program attempts to read from `inputString` again in line 27. The `if` condition in line 30 uses function `good` (Section 13.8) to test if any data remains. Because no data remains, the function returns `false` and the `else` part of the `if...else` statement is executed.

21.13 C++11 Numeric Conversion Functions

C++11 now contains functions for converting from numeric values to `string`s and from `string`s to numeric values. Though you could previously perform such conversions using other techniques, the functions presented in this section were added for convenience.

Converting Numeric Values to `string` *Objects*

C++11's `to_string` function (from the `<string>` header) returns the `string` representation of its numeric argument. The function is overloaded for types `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double` and `long double`.

Converting `string` *Objects to Numeric Values*

C++11 provides eight functions (Fig. 21.13; from the `<string>` header) for converting `string` objects to numeric values. Each function attempts to convert the *beginning* of its `string` argument to a numeric value. If no conversion can be performed, each function throws an `invalid_argument` exception. If the result of the conversion is out of range for the function's return type, each function throws an `out_of_range` exception.

Function	Return type	Function	Return type
Functions that convert to integral types		*Functions that convert to floating-point types*	
`stoi`	`int`	`stof`	`float`
`stol`	`long`	`stod`	`double`
`stoul`	`unsigned long`	`stold`	`long double`
`stoll`	`long long`		
`stoull`	`unsigned long long`		

Fig. 21.13 | C++11 functions that convert from `string`s to numeric types.

Functions That Convert `string`*s to Integral Types*

Consider an example of converting a `string` to an integral value. Assuming the `string`:

```
string s( "100hello" );
```

the following statement converts the beginning of the string to the `int` value `100` and stores that value in `convertedInt`:

```
int convertedInt = stoi( s );
```

Each function that converts a `string` to an integral type actually receives *three* parameters—the last two have default arguments. The parameters are:

- A `string` containing the characters to convert.
- A pointer to a `size_t` variable. The function uses this pointer to store the index of the first character that was *not* converted. The default argument is a null pointer, in which case the function does *not* store the index.
- An `int` from 2 to 36 representing the number's base—the default is base 10.

So, the preceding statement is equivalent to

```
int convertedInt = stoi( s, nullptr, 10 );
```

Given a `size_t` variable named `index`, the statement:

```
int convertedInt = stoi( s, &index, 2 );
```

converts the binary number `"100"` (base 2) to an `int` (100 in binary is the `int` value 4) and stores in `index` the location of the string's letter `"h"` (the first character that was not converted).

Functions That Convert `string`*s to Floating-Point Types*

The functions that convert `string`s to floating-point types each receive two parameters:

- A `string` containing the characters to convert.
- A pointer to a `size_t` variable where the function stores the index of the first character that was *not* converted. The default argument is a null pointer, in which case the function does *not* store the index.

Consider an example of converting a `string` to an floating-point value. Assuming the `string`:

```
string s( "123.45hello" );
```

the following statement converts the beginning of the `string` to the `double` value `123.45` and stores that value in `convertedDouble`:

```
double convertedDouble = stod( s );
```

Again, the second argument is a null pointer by default.

21.14 Wrap-Up

This chapter discussed the details of C++ Standard Library class `string`. We discussed assigning, concatenating, comparing, searching and swapping strings. We also introduced a

number of methods to determine string characteristics, to find, replace and insert characters in a string, and to convert strings to pointer-based strings and vice versa. You learned about string iterators and performing input from and output to strings in memory. Finally, we introduced C++11's new functions for converting numeric values to strings and for converting strings to numeric values. In the next chapter, we introduce structs, which are similar to classes, and discuss the manipulation of bits, characters and C strings.

Summary

Section 21.1 Introduction

- Class template basic_string provides typical string-manipulation operations.
- The typedef statement

```
typedef basic_string< char > string;
```

creates the alias type string for basic_string<char> (p. 850). A typedef also is provided for the wchar_t type (wstring).

- To use strings, include C++ Standard Library header <string>.
- Assigning a single character to a string object is permitted in an assignment statement.
- strings are not necessarily null terminated.
- Most string member functions take as arguments a starting subscript location and the number of characters on which to operate.

Section 21.2 `string` Assignment and Concatenation

- Class string provides overloaded operator= and member function assign (p. 851) for assignments.
- The subscript operator, [], provides read/write access to any element of a string.
- string member function at (p. 853) provides checked access (p. 853)—going past either end of the string throws an out_of_range exception. The subscript operator, [], does not provide checked access.
- The overloaded + and += operators and member function append (p. 853) perform string concatenation.

Section 21.3 Comparing `string`s

- Class string provides overloaded ==, !=, <, >, <= and >= operators for string comparisons.
- string member function compare (p. 855) compares two strings (or substrings) and returns 0 if the strings are equal, a positive number if the first string is lexicographically (p. 855) greater than the second or a negative number if the first string is lexicographically less than the second.

Section 21.4 Substrings

- string member function substr (p. 856) retrieves a substring from a string.

Section 21.5 Swapping `string`s

- string member function swap (p. 856) swaps the contents of two strings.

Section 21.6 `string` Characteristics

- `string` member functions `size` and `length` (p. 851) return the number of characters currently stored in a `string`.
- `string` member function `capacity` (p. 857) returns the total number of characters that can be stored in a `string` without increasing the amount of memory allocated to the `string`.
- `string` member function `max_size` (p. 859) returns the maximum size a `string` can have.
- `string` member function `resize` (p. 859) changes the length of a `string`.
- `string` member function `empty` returns `true` if a `string` is empty.

Section 21.7 Finding Substrings and Characters in a `string`

- Class `string` find functions (p. 861) `find`, `rfind`, `find_first_of`, `find_last_of` and `find_first_not_of` locate substrings or characters in a `string`.

Section 21.8 Replacing Characters in a `string`

- `string` member function `erase` (p. 861) deletes elements of a `string`.
- `string` member function `replace` (p. 863) replaces characters in a `string`.

Section 21.9 Inserting Characters into a `string`

- `string` member function `insert` (p. 863) inserts characters in a `string`.

Section 21.10 Conversion to Pointer-Based `char *` Strings

- `string` member function `c_str` (p. 865) returns a `const char *` pointing to a null-terminated pointer-based string that contains all the characters in a `string`.
- `string` member function `data` (p. 865) returns a `const char *` pointing to a non-null-terminated built-in character array that contains all the characters in a `string`.

Section 21.11 Iterators

- Class `string` provides member functions `begin` and `end` (p. 866) to iterate through individual elements.
- Class `string` provides member functions `rend` and `rbegin` (p. 866) for accessing individual `string` characters in reverse from the end of a `string` toward the beginning.

Section 21.12 String Stream Processing

- Input from a `string` is supported by type `istringstream` (p. 867). Output to a `string` is supported by type `ostringstream` (p. 867).
- `ostringstream` member function `str` (p. 867) returns the `string` from the stream.

Section 21.13 C++11 Numeric Conversion Functions

- C++11's `<string>` header now contains functions for converting from numeric values to string objects and from string objects to numeric values.
- The `to_string` function (p. 870) returns the string representation of its numeric argument and is overloaded for types `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double` and `long double`.
- C++11 provides eight functions for converting string objects to numeric values. Each function attempts to convert the beginning of its string argument to a numeric value. If no conversion can be performed, an `invalid_argument` exception occurs. If the result of the convertion is out of range for the function's return type, an `out_of_range` exception occurs.
- Each function that converts a `string` to an integral type receives three parameters—a `string` containing the characters to convert, a pointer to a `size_t` variable where the function stores the

index of the first character that was not converted (a null pointer, by default) and an `int` from 2 to 36 representing the number's base (base 10, by default).

- The functions that convert strings to floating-point types each receive two parameters—a `string` containing the characters to convert and a pointer to a `size_t` variable where the function stores the index of the first character that was not converted (a null pointer, by default).

Self-Review Exercises

21.1 Fill in the blanks in each of the following:

a) Header _____ must be included for class `string`.
b) Class `string` belongs to the _____ `namespace`.
c) Function _____ deletes characters from a `string`.
d) Function _____ finds the first occurrence of one of several characters from a `string`.

21.2 State which of the following statements are *true* and which are *false*. If a statement is *false*, explain why.

a) Concatenation of `string` objects can be performed with the addition assignment operator, `+=`.
b) Characters within a `string` begin at index `0`.
c) The assignment operator, `=`, copies a `string`.
d) A pointer-based string is a `string` object.

21.3 Find the error(s) in each of the following, and explain how to correct it (them):

a)
```
string string1( 28 ); // construct string1
string string2( 'z' ); // construct string2
```

b)
```
// assume std namespace is known
const char *ptr = name.data(); // name is "joe bob"
ptr[ 3 ] = '-';
cout << ptr << endl;
```

Answers to Self-Review Exercises

21.1 a) `<string>`. b) `std`. c) `erase`. d) `find_first_of`.

21.2 a) True.
b) True.
c) True.
d) False. A `string` is an object that provides many different services. A pointer-based string does not provide any services. Pointer-based strings are null terminated; `string`s are not necessarily null terminated. Pointer-based strings are pointers and `string`s are objects.

21.3 a) Constructors for class `string` do not exist for integer and character arguments. Other valid constructors should be used—converting the arguments to `string`s if need be.
b) Function `data` does not add a null terminator. Also, the code attempts to modify a `const char`. Replace all of the lines with the code:

```
cout << name.substr( 0, 3 ) + "-" + name.substr( 4 ) << endl;
```

Exercises

21.4 ***(Fill in the Blanks)*** Fill in the blanks in each of the following:

a) Class `string` member function _____ converts a `string` to a pointer-based string.
b) Class `string` member function _____ is used for assignment.
c) _____ is the return type of function `rbegin`.
d) Class `string` member function _____ is used to retrieve a substring.

21.5 *(True or False)* State which of the following statements are *true* and which are *false*. If a statement is *false*, explain why.

a) `string`s are always null terminated.
b) Class `string` member function `max_size` returns the maximum size for a `string`.
c) Class `string` member function `at` can throw an `out_of_range` exception.
d) Class `string` member function `begin` returns an `iterator`.

21.6 *(Find Code Errors)* Find any errors in the following and explain how to correct them:

a)
```
std::cout << s.data() << std::endl; // s is "hello"
```
b)
```
erase( s.rfind( "x" ), 1 ); // s is "xenon"
```
c)
```
string& foo()
{
   string s( "Hello" );
   ...   // other statements
   return;
} // end function foo
```

21.7 (*Simple Encryption*) Some information on the Internet may be encrypted with a simple algorithm known as "rot13," which rotates each character by 13 positions in the alphabet. Thus, `'a'` corresponds to `'n'`, and `'x'` corresponds to `'k'`. rot13 is an example of symmetric key encryption. With symmetric key encryption, both the encrypter and decrypter use the same key.

a) Write a program that encrypts a message using rot13.
b) Write a program that decrypts the scrambled message using 13 as the key.
c) After writing the programs of part (a) and part (b), briefly answer the following question: If you did not know the key for part (b), how difficult do you think it would be to break the code? What if you had access to substantial computing power (e.g., supercomputers)? In Exercise 21.24 we ask you to write a program to accomplish this.

21.8 *(Using `string` Iterators)* Write a program using iterators that demonstrates the use of functions `rbegin` and `rend`.

21.9 *("Words Ending in "r" or "ay")* Write a program that reads in several `string`s and prints only those ending in "`r`" or "ay". Only lowercase letters should be considered.

21.10 *(`string` Concatenation)* Write a program that separately inputs a first name and a last name and concatenates the two into a new `string`. Show two techniques for accomplishing this task.

21.11 *(Hangman Game)* Write a program that plays the game of Hangman. The program should pick a word (which is either coded directly into the program or read from a text file) and display the following:

```
Guess the word:    XXXXXX
```

Each `X` represents a letter. The user tries to guess the letters in the word. The appropriate response `yes` or `no` should be displayed after each guess. After each incorrect guess, display the diagram with another body part filled. After seven incorrect guesses, the user should be hanged. The display should look as follows:

```
 O
/|\
 |
/ \
```

After each guess, display all user guesses. If the user guesses the word correctly, display

```
Congratulations!!! You guessed my word. Play again? yes/no
```

21.12 *(Printing a `string` Backward)* Write a program that inputs a `string` and prints the `string` backward. Convert all uppercase characters to lowercase and all lowercase characters to uppercase.

21.13 *(Alphabetizing Animal Names)* Write a program that uses the comparison capabilities introduced in this chapter to alphabetize a series of animal names. Only uppercase letters should be used for the comparisons.

21.14 *(Cryptograms)* Write a program that creates a cryptogram out of a `string`. A cryptogram is a message or word in which each letter is replaced with another letter. For example the `string`

```
The bird was named squawk
```

might be scrambled to form

```
cin vrjs otz ethns zxqtop
```

Spaces are not scrambled. In this particular case, `'T'` was replaced with `'x'`, each `'a'` was replaced with `'h'`, etc. Uppercase letters become lowercase letters in the cryptogram. Use techniques similar to those in Exercise 21.7.

21.15 *(Solving Cryptograms)* Modify Exercise 21.14 to allow the user to solve the cryptogram. The user should input two characters at a time: The first character specifies a letter in the cryptogram, and the second letter specifies the replacement letter. If the replacement letter is correct, replace the letter in the cryptogram with the replacement letter in uppercase.

21.16 *(Counting Palindromes)* Write a program that inputs a sentence and counts the number of palindromes in it. A palindrome is a word that reads the same backward and forward. For example, `"tree"` is not a palindrome, but `"noon"` is.

21.17 *(Counting Vowels)* Write a program that counts the total number of vowels in a sentence. Output the frequency of each vowel.

21.18 *(String Insertion)* Write a program that inserts the characters `"******"` in the exact middle of a `string`.

21.19 *(Erasing Characters from a `string`)* Write a program that erases the sequences `"by"` and `"BY"` from a `string`.

21.20 *(Reversing a `string` with Iterators)* Write a program that inputs a line of text and prints the text backward. Use iterators in your solution.

21.21 *(Reversing a `string` with Iterators using Recursion)* Write a recursive version of Exercise 21.20.

21.22 *(Using the `erase` Functions with Iterator Arguments)* Write a program that demonstrates the use of the `erase` functions that take `iterator` arguments.

21.23 *(Letter Pyramid)* Write a program that generates the following from the `string` `"abcdefghijklmnopqrstuvwxyz"`:

```
                         a
                        bcb
                       cdedc
                      defgfed
                     efghihgfe
                    fghijkjihgf
                   ghijklmlkjihg
                  hijklmnonmlkjih
                 ijklmnopqponmlkji
                jklmnopqrsrqponmlkj
               klmnopqrstutsrqponmlk
              lmnopqrstuvwvutsrqponml
             mnopqrstuvwxyxwvutsrqponm
            nopqrstuvwxyz{zyxwvutsrqpon
```

21.24 ***(Simple Decryption)*** In Exercise 21.7, we asked you to write a simple encryption algorithm. Write a program that will attempt to decrypt a "rot13" message using simple frequency substitution. (Assume that you do not know the key.) The most frequent letters in the encrypted phrase should be replaced with the most commonly used English letters (a, e, i, o, u, s, t, r, etc.). Write the possibilities to a file. What made the code breaking easy? How can the encryption mechanism be improved?

21.25 ***(Enhanced `Employee` Class)*** Modify class `Employee` in Figs. 12.9–12.10 by adding a `private` utility function called `isValidSocialSecurityNumber`. This member function should validate the format of a social security number (e.g., `###-##-####`, where `#` is a digit). If the format is valid, return `true`; otherwise return `false`.

Making a Difference

21.26 ***(Cooking with Healthier Ingredients)*** Obesity in the United States is increasing at an alarming rate. Check the map from the Centers for Disease Control and Prevention (CDC) at `www.cdc.gov/nccdphp/dnpa/Obesity/trend/maps/index.htm`, which shows obesity trends in the United States over the last 20 years. As obesity increases, so do occurrences of related problems (e.g., heart disease, high blood pressure, high cholesterol, type 2 diabetes). Write a program that helps users choose healthier ingredients when cooking, and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The program should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, your program should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in Fig. 21.14. Your program should display a warning such as, "Always consult your physician before making significant changes to your diet."

Ingredient	Substitution
1 cup sour cream	1 cup yogurt
1 cup milk	1/2 cup evaporated milk and 1/2 cup water
1 teaspoon lemon juice	1/2 teaspoon vinegar
1 cup sugar	1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar
1 cup butter	1 cup yogurt
1 cup flour	1 cup rye or rice flour
1 cup mayonnaise	1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt
1 egg	2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed)
1 cup milk	1 cup soy milk
1/4 cup oil	1/4 cup applesauce
white bread	whole-grain bread

Fig. 21.14 | Common ingredient substitutions.

Your program should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Conversion data for measurements and substitutes can be obtained at websites such as:

```
chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
www.pioneerthinking.com/eggsub.html
www.gourmetsleuth.com/conversions.htm
```

Your program should consider the user's health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the program should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.

21.27 ***(Spam Scanner)*** Spam (or junk e-mail) costs U.S. organizations billions of dollars a year in spam-prevention software, equipment, network resources, bandwidth, and lost productivity. Research online some of the most common spam e-mail messages and words, and check your own junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write an application in which the user enters an e-mail message. Then, scan the message for each of the 30 keywords or phrases. For each occurrence of one of these within the message, add a point to the message's "spam score." Next, rate the likelihood that the message is spam, based on the number of points it received.

21.28 ***(SMS Language)*** Short Message Service (SMS) is a communications service that allows sending text messages of 160 or fewer characters between mobile phones. With the proliferation of mobile phone use worldwide, SMS is being used in many developing nations for political purposes (e.g., voicing opinions and opposition), reporting news about natural disasters, and so on. For example, check out `comunica.org/radio2.0/archives/87`. Since the length of SMS messages is limited, SMS Language—abbreviations of common words and phrases in mobile text messages, e-mails, instant messages, etc.—is often used. For example, "in my opinion" is "IMO" in SMS Language. Research SMS Language online. Write a program in which the user can enter a message using SMS Language; the program should translate it into English (or your own language). Also provide a mechanism to translate text written in English (or your own language) into SMS Language. One potential problem is that one SMS abbreviation could expand into a variety of phrases. For example, IMO (as used above) could also stand for "International Maritime Organization," "in memory of," etc.

Bits, Characters, C Strings and `structs`

22

The same old charitable lie
Repeated as the years scoot by
Perpetually makes a hit—
"You really haven't changed a bit!"
—Margaret Fishback

The chief defect of Henry King
Was chewing little bits of string.
—Hilaire Belloc

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences.
—William Strunk, Jr.

Objectives

In this chapter you'll learn:

- To create and use `structs` and to understand their near equivalence with classes.
- To use `typedef` to create aliases for data types.
- 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>`.

Outline

22.1 Introduction

We now discuss structures, their near equivalence with classes, and the manipulation of bits, characters and C strings. Many of the techniques we present here are included for the benefit of those who will work with legacy C and C++ code.

Like classes, C++ structures may contain access specifiers, member functions, constructors and destructors. In fact, *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.* Our presentation of structures here is typical of the legacy C code and early C++ code you'll see in industry.

We present a high-performance card shuffling and dealing simulation in which we use structure objects containing C++ `string` objects to represent the cards. 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 programs that interact directly with hardware devices that have limited memory. 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

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`). `Card`'s definition contains two `string` members—`face` and `suit`.

The following declarations

```
Card oneCard;
Card deck[ 52 ];
Card *cardPtr;
```

declare `oneCard` to be a structure variable of type `Card`, `deck` to be an array with 52 elements of type `Card` and `cardPtr` to be a pointer to a `Card` structure. 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. For example, the preceding declarations could have been incorporated into the `Card` structure definition as follows:

```
struct Card
{
   string face;
   string suit;
} oneCard, deck[ 52 ], *cardPtr;
```

As with classes, structure members are *not* necessarily stored in *consecutive* bytes of memory. 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, four or eight bytes and typically eight bytes on today's popular 64-bit systems. 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*). Figure 22.1 shows a sample storage alignment for an object of type `Example` that's 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 values in `sample1` and `sample2` are in fact equal, the structure objects might *not* be equal, because the *undefined* one-byte holes are not likely to contain identical values.

Common Programming Error 22.1

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.

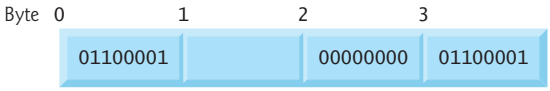

Fig. 22.1 | Possible storage alignment for an `Example` object, showing an undefined byte.

22.3 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 more readable type names. For example, the statement

```
typedef Card *CardPtr;
```

defines the new type name `CardPtr` as a synonym for type `Card *`.

Creating a new name with `typedef` does *not* create a new type; `typedef` simply creates a *new type name* that can then be used in the program as an alias for an existing type name.

22.4 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 9.23. This program represents the deck of cards as an `array` of structures.

```
1  // Fig. 22.2: DeckOfCards.h
2  // Definition of class DeckOfCards that
3  // represents a deck of playing cards.
4  #include <string>
5  #include <array>
6
7  // Card structure definition
8  struct Card
9  {
10    std::string face;
11    std::string suit;
12 }; // end structure Card
13
14 // DeckOfCards class definition
15 class DeckOfCards
16 {
17 public:
18    static const int numberOfCards = 52;
19    static const int faces = 13;
20    static const int suits = 4;
21
22    DeckOfCards(); // constructor initializes deck
23    void shuffle(); // shuffles cards in deck
24    void deal() const; // deals cards in deck
25
```

Fig. 22.2 | Definition of class `DeckOfCards` that represents a deck of playing cards. (Part 1 of 2.)

```
26   private:
27      std::array< Card, numberOfCards > deck; // represents deck of cards
28   }; // end class DeckOfCards
```

Fig. 22.2 | Definition of class `DeckOfCards` that represents a deck of playing cards. (Part 2 of 2.)

The constructor (lines 12–31 of Fig. 22.3) initializes the `array` in order with character strings representing Ace through King of each suit. Function `shuffle` implements the 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` and the randomly selected `Card` are swapped in the `array`. A total of 52 swaps are made in a single pass of the entire `array`, and the `array` is shuffled. Because the `Card` structures were swapped in place in the `array`, the dealing algorithm implemented in function `deal` requires only one pass of the `array` to deal the shuffled cards.

```
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  {
14     // initialize suit array
15     static string suit[ suits ] =
16        { "Hearts", "Diamonds", "Clubs", "Spades" };
17
18     // initialize face array
19     static string face[ faces ] =
20        { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
21        "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
22
23     // set values for deck of 52 Cards
24     for ( size_t i = 0; i < deck.size(); ++i )
25     {
26        deck[ i ].face = face[ i % faces ];
27        deck[ i ].suit = suit[ i / faces ];
28     } // end for
29
30     srand( static_cast< size_t >( time( nullptr ) ) ); // seed
31  } // end no-argument DeckOfCards constructor
32
33  // shuffle cards in deck
34  void DeckOfCards::shuffle()
35  {
```

Fig. 22.3 | Member-function definitions for class `DeckOfCards`. (Part 1 of 2.)

```
36      // shuffle cards randomly
37      for ( size_t i = 0; i < deck.size(); ++i )
38      {
39         int j = rand() % numberOfCards;
40         Card temp = deck[ i ];
41         deck[ i ] = deck[ j ];
42         deck[ j ] = temp;
43      } // end for
44   } // end function shuffle
45
46   // deal cards in deck
47   void DeckOfCards::deal() const
48   {
49      // display each card's face and suit
50      for ( size_t i = 0; i < deck.size(); ++i )
51         cout << right << setw( 5 ) << deck[ i ].face << " of "
52            << left << setw( 8 ) << deck[ i ].suit
53            << ( ( i + 1 ) % 2 ? '\t' : '\n' );
54   } // end function deal
```

Fig. 22.3 | Member-function definitions for class `DeckOfCards`. (Part 2 of 2.)

```
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
```

```
 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
```

Fig. 22.4 | Card shuffling and dealing program. (Part 1 of 2.)

```
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 | Card shuffling and dealing program. (Part 2 of 2.)

22.5 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 the *bitwise operators*, and we discuss how to save memory by using *bit fields*.

All data is represented internally by computers as sequences of bits. Each bit can assume the value `0` or the value `1`. On most systems, a sequence of eight bits, each of which 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`). Normally the bitwise operators are used with unsigned integers.

Portability Tip 22.2

Bitwise data manipulations are machine dependent.

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. 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**. We've been using `&`, `<<` and `>>` for other purposes—this is a classic example of *operator overloading*. The *bitwise AND*, *bitwise inclusive OR* and *bitwise exclusive OR* operators compare their two operands bit by bit. The *bitwise AND* operator sets each bit in the result to 1 if the corresponding bit in *both* operands is 1. The *bitwise inclusive OR* operator sets each bit in the result to 1 if the corresponding bit in *either (or both)* operand(s) is 1. The *bitwise exclusive OR* operator sets each bit in the result to 1 if the corresponding bit in *either* operand—*but not both*—is 1. The *left-shift* operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The *right-shift* operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. The *bitwise complement* operator sets all `0` bits in its operand to `1` in the result and sets all `1` bits in its operand to `0` in the result. Detailed discussions of each bitwise operator appear in the following examples. 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.

Printing a Binary Representation of an Integral Value

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 = 0; // integral value to print in binary
12
13    cout << "Enter an unsigned integer: ";
14    cin >> inputValue;
15    displayBits( inputValue );
16 } // end main
17
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 << " = ";
```

Fig. 22.6 | Printing an unsigned integer in bits. (Part 1 of 2.)

```
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
```

```
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 2 of 2.)

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`. 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. 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`. 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 `1`s, 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.

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 (`&`).

Common Programming Error 22.2

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

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 48–66) 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    // demonstrate bitwise &
13    unsigned number1 = 2179876355;
14    unsigned mask = 1;
15    cout << "The result of combining the following\n";
16    displayBits( number1 );
17    displayBits( mask );
18    cout << "using the bitwise AND operator & is\n";
19    displayBits( number1 & mask );
```

Fig. 22.8 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 1 of 3.)

```
20
21      // demonstrate bitwise |
22      number1 = 15;
23      unsigned setBits = 241;
24      cout << "\nThe result of combining the following\n";
25      displayBits( number1 );
26      displayBits( setBits );
27      cout << "using the bitwise inclusive OR operator | is\n";
28      displayBits( number1 | setBits );
29
30      // demonstrate bitwise exclusive OR
31      number1 = 139;
32      unsigned number2 = 199;
33      cout << "\nThe result of combining the following\n";
34      displayBits( number1 );
35      displayBits( number2 );
36      cout << "using the bitwise exclusive OR operator ^ is\n";
37      displayBits( number1 ^ number2 );
38
39      // demonstrate bitwise complement
40      number1 = 21845;
41      cout << "\nThe one's complement of\n";
42      displayBits( number1 );
43      cout << "is" << endl;
44      displayBits( ~number1 );
45   } // end main
46
47   // display bits of an unsigned integer value
48   void displayBits( unsigned value )
49   {
50      const int SHIFT = 8 * sizeof( unsigned ) - 1;
51      const unsigned MASK = 1 << SHIFT;
52
53      cout << setw( 10 ) << value << " = ";
54
55      // display bits
56      for ( unsigned i = 1; i <= SHIFT + 1; ++i )
57      {
58         cout << ( value & MASK ? '1' : '0' );
59         value <<= 1; // shift value left by 1
60
61         if ( i % 8 == 0 ) // output a space after 8 bits
62            cout << ' ';
63      } // end for
64
65      cout << endl;
66   } // end function displayBits
```

```
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
```

Fig. 22.8 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 2 of 3.)

```
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 3 of 3.)

Bitwise AND Operator (&)

In Fig. 22.8, line 13 assigns `2179876355` (`10000001 11101110 01000110 00000011`) to variable `number1`, and line 14 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 19), 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`.

Bitwise Inclusive OR Operator (|)

The *bitwise inclusive OR operator* is used to set specific bits to 1 in an operand. In Fig. 22.8, line 22 assigns `15` (`00000000 00000000 00000000 00001111`) to variable `number1`, and line 23 assigns `241` (`00000000 00000000 00000000 11110001`) to variable `setBits`. When `number1` and `setBits` are combined using the *bitwise inclusive OR operator* in the expression `number1 | setBits` (line 28), 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 22.3

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

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 (|).

Bitwise Exclusive OR (^)

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 31–32 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 *bitwise exclusive OR operator* in the expression `number1 ^ number2` (line 37), the result is `00000000 00000000 00000000 01001100`. Figure 22.10 summarizes the results of combining two bits with the *bitwise exclusive OR operator*.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

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

Bitwise Complement (~)

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 40 assigns variable `number1` the value `21845` (`00000000 00000000 01010101 01010101`). When the expression `~number1` evaluates, the result is (`11111111 11111111 10101010 10101010`).

Bitwise Shift Operators

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";
```

Fig. 22.11 | Bitwise shift operators. (Part 1 of 2.)

```
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
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
```

```
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 2 of 2.)

Left-Shift Operator

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 Fig. 22.11, line 11 assigns variable `number1` the value `960` (`00000000 00000000 00000011 11000000`). The result of left-shifting variable `number1` eight bits in the expression `number1 << 8` (line 17) is `245760` (`00000000 00000011 11000000 00000000`).

Right-Shift Operator

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 22.4

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 22.3

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

Bitwise Assignment Operators

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 4.

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.

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) `()` (grouping parentheses)	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`()  []  .  ->  ++  --  static_cast<type>()`	left to right	postfix
`++  --  +  -  !  delete  sizeof` `*  ~  &  new`	right to left	prefix
`*  /  %`	left to right	multiplicative
`+  -`	left to right	additive
`<<  >>`	left to right	shifting
`<  <=  >  >=`	left to right	relational
`==  !=`	left to right	equality

Fig. 22.13 | Operator precedence and associativity. (Part 1 of 2.)

Operators	Associativity	Type
&	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. (Part 2 of 2.)

22.6 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 22.1
Bit fields help conserve storage.

Consider the following structure definition:

```
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 four bits, member `suit` in 2 bits and member `color` in one bit. The number of bits is based on the desired range of values for each structure member. Member `face` stores values between `0` (Ace) and `12` (King)—four 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)—two bits can store a value between `0` and `3`. Finally, member `color` stores either `0` (Red) or `1` (Black)—one bit can store either `0` or `1`.

The program in Figs. 22.14–22.16 creates `array deck` containing `BitCard` structures (line 25 of Fig. 22.14). The constructor inserts the 52 cards in the `deck array`, and function `deal` prints the 52 cards. Notice that bit fields are accessed exactly as any other struc-

ture member is (lines 14–16 and 25–30 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 <array>
5
6  // BitCard structure definition with bit fields
7  struct BitCard
8  {
9     unsigned face : 4; // 4 bits; 0-15
10    unsigned suit : 2; // 2 bits; 0-3
11    unsigned color : 1; // 1 bit; 0-1
12 }; // end struct BitCard
13
14 // DeckOfCards class definition
15 class DeckOfCards
16 {
17 public:
18    static const int faces = 13;
19    static const int colors = 2; // black and red
20    static const int numberOfCards = 52;
21
22    DeckOfCards(); // constructor initializes deck
23    void deal() const; // deals cards in deck
24 private:
25    std::array< BitCard, numberOfCards > deck; // represents deck of cards
26 }; // end class DeckOfCards
```

Fig. 22.14 | Definition of class `DeckOfCards` that represents a deck of playing cards.

```
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    for ( size_t i = 0; i < deck.size(); ++i )
13    {
14       deck[ i ].face = i % faces; // faces in order
15       deck[ i ].suit = i / faces; // suits in order
16       deck[ i ].color = i / ( faces * colors ); // colors in order
17    } // end for
18 } // end no-argument DeckOfCards constructor
```

Fig. 22.15 | Member-function definitions for class `DeckOfCards`. (Part 1 of 2.)

```
19
20  // deal cards in deck
21  void DeckOfCards::deal() const
22  {
23     for ( size_t k1 = 0, k2 = k1 + deck.size() / 2;
24        k1 < deck.size() / 2 - 1; ++k1, ++k2 )
25        cout << "Card:" << setw( 3 ) << deck[ k1 ].face
26           << "  Suit:" << setw( 2 ) << deck[ k1 ].suit
27           << "  Color:" << setw( 2 ) << deck[ k1 ].color
28           << "    " << "Card:" << setw( 3 ) << deck[ k2 ].face
29           << "  Suit:" << setw( 2 ) << deck[ k2 ].suit
30           << "  Color:" << setw( 2 ) << deck[ k2 ].color << endl;
31  } // end function deal
```

Fig. 22.15 | Member-function definitions for class `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
```

```
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.

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 three-bit field as padding—nothing can be stored in those three 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
```

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 22.4

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

Common Programming Error 22.5

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 22.6

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 22.2

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.*

22.7 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.

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

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 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 *whitespace character*—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and `0` otherwise.
`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.

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`).

```
// Fig. 22.18: fig22_18.cpp
// Character-handling functions isdigit, isalpha, isalnum and isxdigit.
#include <iostream>
#include <cctype> // character-handling function prototypes
using namespace std;

int main()
{
   cout << "According to isdigit:\n"
      << ( isdigit( '8' ) ? "8 is a" : "8 is not a" ) << " digit\n"
      << ( isdigit( '#' ) ? "# is a" : "# is not a" ) << " digit\n";

   cout << "\nAccording to isalpha:\n"
      << ( isalpha( 'A' ) ? "A is a" : "A is not a" ) << " letter\n"
      << ( isalpha( 'b' ) ? "b is a" : "b is not a" ) << " letter\n"
      << ( isalpha( '&' ) ? "& is a" : "& is not a" ) << " letter\n"
      << ( isalpha( '4' ) ? "4 is a" : "4 is not a" ) << " letter\n";

   cout << "\nAccording to isalnum:\n"
      << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
      << " digit or a letter\n"
      << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
      << " digit or a letter\n"
      << ( isalnum( '#' ) ? "# is a" : "# is not a" )
      << " digit or a letter\n";

   cout << "\nAccording to isxdigit:\n"
      << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
      << " hexadecimal digit\n"
      << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
      << " hexadecimal digit\n"
      << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
      << " hexadecimal digit\n"
      << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
      << " hexadecimal digit\n"
      << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
      << " hexadecimal digit" << endl;
} // end main
```

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

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
```

Fig. 22.18 | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 2.)

```
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 2.)

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`**. 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
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" )
```

Fig. 22.19 | Character-handling functions `islower`, `isupper`, `tolower` and `toupper`. (Part 1 of 2.)

```
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
```

```
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 2 of 2.)

Figure 22.20 demonstrates functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. Function `isspace` determines whether its argument is a *whitespace 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 and isgraph.
3  #include <iostream>
```

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

```
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
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
```

```
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
```

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

```
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.)

22.8 C String-Manipulation Functions

The string-handling library provides any 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 `<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.
`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.

Fig. 22.21 | String-manipulation functions of the string-handling library. (Part 1 of 2.)

Function prototype	Function description
`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.
`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 2 of 2.)

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

Common Programming Error 22.7

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

Copying Strings with `strcpy` *and* `strncpy`

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. For example, if `"test"` is the second argument, a terminating null character is written *only* if the third argument to `strncpy` is at least `5` (four characters in `"test"` plus one terminating null character). If the third argument is larger than `5`, null characters are appended to the array until the total number of characters specified by the third argument is written.

Common Programming Error 22.8

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.*

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
```

```
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`.

Concatenating Strings with `strcat` *and* `strncat`

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
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.

Comparing Strings with `strcmp` *and* `strncmp`

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 char-

acters 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 22.9

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    const char *s1 = "Happy New Year";
11    const char *s2 = "Happy New Year";
12    const 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
```

```
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`.

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.

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.

[*Note:* With some compilers, functions `strcmp` and `strncmp` always return `-1`, `0` or `1`, as in the sample output of Fig. 22.24. With other compilers, these functions return `0` or the difference between the numeric codes of the first characters that differ in the strings being compared. For example, when `s1` and `s3` are compared, the first characters that differ between them are the first character of the second word in each string—`N` (numeric code 78) in `s1` and `H` (numeric code 72) in `s3`, respectively. In this case, the return value will be `6` (or `-6` if `s3` is compared to `s1`).]

Tokenizing a String with `strtok`

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). 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 15 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's 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. 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.

```
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
11    cout << "The string to be tokenized is:\n" << sentence
12       << "\n\nThe tokens are:\n\n";
13
14    // begin tokenization of sentence
15    char *tokenPtr = strtok( sentence, " " );
16
17    // continue tokenizing sentence until tokenPtr becomes NULL
18    while ( tokenPtr != NULL )
19    {
20       cout << tokenPtr << '\n';
21       tokenPtr = strtok( NULL, " " ); // get next token
22    } // end while
```

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

```
23
24     cout << "\nAfter strtok, sentence = " << sentence << endl;
25  } // 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.)

Subsequent calls to `strtok` to continue tokenizing `sentence` contain `NULL` as the first argument (line 21). 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's not visible to you. If no tokens remain when `strtok` is called, `strtok` returns `NULL`. The program of Fig. 22.25 uses `strtok` to tokenize the string `"This is a sentence with 7 tokens"`. The program prints each token on a separate line. Line 24 outputs `sentence` after tokenization. Note that *`strtok` modifies the input string*; therefore, a copy of the string should be made if the program requires the original after the calls to `strtok`. When `sentence` is output after tokenization, only the word "`This`" prints, because `strtok` replaced each blank in `sentence` with a null character (`'\0'`) during the tokenization process.

Common Programming Error 22.10

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.*

Determining String Lengths

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
```

Fig. 22.26 | `strlen` returns the length of a `char *` string. (Part 1 of 2.)

```
7  int main()
8  {
9     const char *string1 = "abcdefghijklmnopqrstuvwxyz";
10    const char *string2 = "four";
11    const 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. (Part 2 of 2.)

22.9 C String-Conversion Functions

In Section 22.8, we discussed several of C++'s most popular C 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 C **string-conversion functions** from the **general-utilities library <cstdlib>**. These functions convert C strings to integer and floating-point values. In new code, C++ programmers typically use the string stream processing capabilities (Chapter 21) to perform such conversions. Figure 22.27 summarizes the C string-conversion functions. When using functions from the general-utilities library, include the <cstdlib> header.

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 | C 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 | C string-conversion functions of the general-utilities library. (Part 2 of 2.)

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`.

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`.

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`.

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 12 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     const char *string1 = "51.2% are admitted";
10    char *stringPtr = nullptr;
11
12    double d = strtod( string1, &stringPtr ); // convert to double
13
14    cout << "The string \"" << string1
15       << "\" is converted to the\ndouble value " << d
16       << " and the string \"" << stringPtr << "\"" << endl;
17 } // 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`.

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. Line 12 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` to `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     const char *string1 = "-1234567abc";
10    char *remainderPtr = nullptr;
```

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

```
11
12     long x = strtol( string1, &remainderPtr, 0 ); // convert to long
13
14     cout << "The original string is \"" << string1
15        << "\"\nThe converted value is " << x
16        << "\nThe remainder of the original string is \"" << remainderPtr
17        << "\"\nThe converted value plus 567 is " << x + 567 << endl;
18  } // end main
```

```
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.)

In a call to function `strtol`, the base can be specified as zero or as any value between 2 and 36. (See Appendix D for a detailed explanation of the octal, decimal, hexadecimal and binary number systems.) Numeric representations of integers from base 11 to base 36 use the characters A–Z to represent the values 10 to 35. For example, hexadecimal values can consist of the digits 0–9 and the characters A–F. A base-11 integer can consist of the digits 0–9 and the character A. A base-24 integer can consist of the digits 0–9 and the characters A–N. A base-36 integer can consist of the digits 0–9 and the characters A–Z. [*Note:* The case of the letter used is ignored.]

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 13 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     const char *string1 = "1234567abc";
10    char *remainderPtr = nullptr;
11
12    // convert a sequence of characters to unsigned long
13    unsigned long x = strtoul( string1, &remainderPtr, 0 );
14
15    cout << "The original string is \"" << string1
16       << "\"\nThe converted value is " << x
17       << "\nThe remainder of the original string is \"" << remainderPtr
```

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

```
18          << "\"\nThe converted value minus 567 is " << x - 567 << endl;
19    } // end main
```

```
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.)

22.10 Search Functions of the C 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`.

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"`.

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.
`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 C string-handling library.

```
// Fig. 22.35: fig22_35.cpp
// Using strchr.
#include <iostream>
#include <cstring> // strchr prototype
using namespace std;

int main()
{
   const char *string1 = "This is a test";
   char character1 = 'a';
   char character2 = 'z';

   // search for character1 in string1
   if ( strchr( string1, character1 ) != NULL )
      cout << '\'' << character1 << "' was found in \""
         << string1 << "\".\n";
   else
      cout << '\'' << character1 << "' was not found in \""
         << string1 << "\".\n";

   // search for character2 in string1
   if ( strchr( string1, character2 ) != NULL )
      cout << '\'' << character2 << "' was found in \""
         << string1 << "\".\n";
   else
      cout << '\'' << character2 << "' was not found in \""
         << string1 << "\"." << endl;
} // 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`.

Function **`strcspn`** (Fig. 22.36, line 15) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.

```
// Fig. 22.36: fig22_36.cpp
// Using strcspn.
#include <iostream>
#include <cstring> // strcspn prototype
using namespace std;

int main()
{
   const char *string1 = "The value is 3.14159";
   const char *string2 = "1234567890";

   cout << "string1 = " << string1 << "\nstring2 = " << string2
      << "\n\nThe length of the initial segment of string1"
```

Fig. 22.36 | String-search function `strcspn`. (Part 1 of 2.)

```
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`. (Part 2 of 2.)

Function **`strpbrk`** searches for the first occurrence in its first string argument of any character in its second string argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character in the first 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
```

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

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

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 0. 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
```

Fig. 22.38 | String-search function `strrchr`. (Part 1 of 2.)

```
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`. (Part 2 of 2.)

Function **`strspn`** (Fig. 22.39, line 15) determines the length of the initial part of the string in its first argument that contains only characters from the string in its second 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`.

Function **`strstr`** searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location

of the string in the first argument is returned; otherwise, it returns 0. 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`.

22.11 Memory Functions of the C 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. [*Note:* The string-processing functions in prior sections operate on null-terminated strings. The functions in this section operate on arrays of bytes. The null-character value (i.e., a byte containing 0) has *no* significance with the functions in this section.]

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.

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

Prototype	Description
`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.
`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.)

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. Remember that *a pointer of type* `void *` *cannot be assigned directly to a pointer of any other 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).

Function `memcpy` copies a specified number of characters (bytes) from the object pointed to by its second argument into the object pointed to by its first 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_42.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 ] = {};
```

Fig. 22.42 | Memory-handling function `memcpy`. (Part 1 of 2.)

```
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`. (Part 2 of 2.)

Function **`memmove`**, like `memcpy`, copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument to a temporary array of characters, then copied from the temporary array to the first argument. This allows characters from one part of a string to be copied into another part of the same string.

Common Programming Error 22.11

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

The program in Fig. 22.43 uses `memmove` (line 13) to copy the last `10` bytes of array `x` into the first `10` bytes of array `x`.

```
1  // Fig. 22.43: fig22_43.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`.

Function **`memcmp`** (Fig. 22.44, lines 14–16) compares the specified number of characters of its first argument with the corresponding characters of its second argument. The

function returns a value greater than zero if the first argument is greater than the second argument, zero if the arguments are equal, and a value less than zero if the first argument is less than the second argument. [*Note:* With some compilers, function `memcmp` returns -1, 0 or 1, as in the sample output of Fig. 22.44. With other compilers, this function returns 0 or the difference between the numeric codes of the first characters that differ in the strings being compared. For example, when `s1` and `s2` are compared, the first character that differs between them is the fifth character of each string—`E` (numeric code 69) for `s1` and `X` (numeric code 72) for `s2`. In this case, the return value will be `19` (or `-19` when `s2` is compared to `s1`).]

```
1  // Fig. 22.44: fig22_44.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
```

```
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`.

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"`.

```
1  // Fig. 22.45: fig22_45.cpp
2  // Using memchr.
3  #include <iostream>
4  #include <cstring> // memchr prototype
5  using namespace std;
```

Fig. 22.45 | Memory-handling function `memchr`. (Part 1 of 2.)

```
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`. (Part 2 of 2.)

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

```
1  // Fig. 22.46: fig22_46.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
```

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

Fig. 22.46 | Memory-handling function `memset`.

22.12 Wrap-Up

This chapter introduced `struct` definitions, initializing `structs` and using them with functions. We discussed `typedef`, using it to create aliases to help promote portability. We also introduced bitwise operators to manipulate data and bit fields for storing data compactly. You learned about the string-conversion functions in `<cstlib>` and the string-processing functions in `<cstring>`. In the next chapter, we discuss additional C++ topics.

Summary

Section 22.2 Structure Definitions

- Keyword `struct` (p. 880) begins every structure definition. Between the braces of the structure definition are the structure member declarations.
- A structure definition creates a new data type (p. 880) that can be used to declare variables.

Section 22.3 `typedef`

- Creating a new type name with `typedef` (p. 882) does not create a new type; it creates a name that's synonymous with a type defined previously.

Section 22.5 Bitwise Operators

- The bitwise AND operator (`&`; p. 885) takes two integral operands. A bit in the result is set to one if the corresponding bits in each of the operands are one.
- Masks (p. 887) are used with bitwise AND to hide some bits while preserving others.
- The bitwise inclusive OR operator (`|`; p. 885) takes two operands. A bit in the result is set to one if the corresponding bit in either operand is set to one.
- Each of the bitwise operators (except complement) has a corresponding assignment operator.
- The bitwise exclusive OR operator (`^`; p. 885) takes two operands. A bit in the result is set to one if exactly one of the corresponding bits in the two operands is set to one.
- The left-shift operator (`<<`; p. 885) shifts the bits of its left operand left by the number of bits specified by its right operand. Bits vacated to the right are replaced with `zeros`.
- The right-shift operator (`>>`; p. 885) shifts the bits of its left operand right by the number of bits specified in its right operand. Right shifting an unsigned integer causes bits vacated at the left to be replaced by zeros. Vacated bits in signed integers can be replaced with zeros or ones.
- The bitwise complement operator (`~`; p. 885) takes one operand and inverts its bits—this produces the one's complement of the operand.

Section 22.6 Bit Fields

- Bit fields (p. 894) reduce storage use by storing data in the minimum number of bits required. Bit-field members must be declared as `int` or `unsigned`.
- A bit field is declared by following an `unsigned` or `int` member name with a colon and the width of the bit field.
- The bit-field width must be an integer constant.
- If a bit field is specified without a name, the field is used as padding (p. 897) in the structure.
- An unnamed bit field with width `0` (p. 897) aligns the next bit field on a new machine-word boundary.

Section 22.7 Character-Handling Library

- Function `islower` (p. 900) determines if its argument is a lowercase letter (`a–z`). Function `isupper` (p. 900) determines whether its argument is an uppercase letter (`A–Z`).
- Function `isdigit` (p. 898) determines if its argument is a digit (`0–9`).
- Function `isalpha` (p. 898) determines if its argument is an uppercase (`A–Z`) or lowercase letter (`a–z`).
- Function `isalnum` (p. 898) determines if its argument is an uppercase letter (`A–Z`), a lowercase letter (`a–z`), or a digit (`0–9`).
- Function `isxdigit` (p. 898) determines if its argument is a hexadecimal digit (`A–F`, `a–f`, `0–9`).

- Function `toupper` (p. 900) converts a lowercase letter to an uppercase letter. Function `tolower` (p. 900) converts an uppercase letter to a lowercase letter.
- Function `isspace` (p. 901) determines if its argument is one of the following whitespace characters: `' '` (space), `'\f'`, `'\n'`, `'\r'`, `'\t'` or `'\v'`.
- Function `iscntrl` (p. 901) determines if its argument is a control character, such as `'\t'`, `'\v'`, `'\f'`, `'\a'`, `'\b'`, `'\r'` or `'\n'`.
- Function `ispunct` (p. 901) determines if its argument is a printing character other than a space, a digit or a letter.
- Function `isprint` (p. 901) determines if its argument is any printing character, including space.
- Function `isgraph` (p. 901) determines if its argument is a printing character other than space.

Section 22.8 C String-Manipulation Functions

- Function `strcpy` (p. 904) copies its second argument into its first argument. You must ensure that the target array is large enough to store the string and its terminating null character.
- Function `strncpy` (p. 904) is equivalent to `strcpy`, but it specifies the number of characters to be copied from the string into the array. The terminating null character will be copied only if the number of characters to be copied is at least one more than the length of the string.
- Function `strcat` (p. 905) appends its second string argument—including the terminating null character—to its first string argument. The first character of the second string replaces the null (`'\0'`) character of the first string. You must ensure that the target array used to store the first string is large enough to store both the first string and the second string.
- Function `strncat` (p. 905) is equivalent to `strcat`, but it appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.
- 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`, but it compares a specified number of characters. If the number of characters in one of the strings is less than the number of characters specified, `strncmp` compares characters until the null character in the shorter string is encountered.
- A sequence of calls to `strtok` (p. 908) breaks a string into tokens that are separated by characters contained in a second string argument. The first call specifies the string to be tokenized as the first argument, and subsequent calls to continue tokenizing the same string specify `NULL` as the first argument. The function returns a pointer to the current token from each call. If there are no more tokens when `strtok` is called, `NULL` is returned.
- Function `strlen` (p. 909) 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 of the string.

Section 22.9 C String-Conversion Functions

- Function `atof` (p. 911) converts its argument—a string beginning with a series of digits that represents a floating-point number—to a `double` value.
- Function `atoi` (p. 911) converts its argument—a string beginning with a series of digits that represents an integer—to an `int` value.
- Function `atol` (p. 912) converts its argument—a string beginning with a series of digits that represents a long integer—to a `long` value.
- Function `strtod` (p. 912) converts a sequence of characters representing a floating-point value to `double`. The function receives two arguments—a string (`char *`) and the address of a `char *`

pointer. The string contains the character sequence to be converted, and the pointer to `char *` is assigned the remainder of the string after the conversion.

- Function `strtol` (p. 913) converts a sequence of characters representing an integer to `long`. It receives a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to be converted, the pointer to `char *` is assigned the location of the first character after the converted value and the integer specifies the base of the value being converted.
- Function `strtoul` (p. 914) converts a sequence of characters representing an integer to `unsigned long`. It receives a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to be converted, the pointer to `char *` is assigned the location of the first character after the converted value and the integer specifies the base of the value being converted.

Section 22.10 Search Functions of the C String-Handling Library

- Function `strchr` (p. 915) searches for the first occurrence of a character in a string. If found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns a null pointer.
- Function `strcspn` (p. 916) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.
- Function `strpbrk` (p. 917) searches for the first occurrence in its first argument of any character that appears in its second argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character; otherwise, `strpbrk` returns a null pointer.
- Function `strrchr` (p. 917) searches for the last occurrence of a character in a string. If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, it returns a null pointer.
- Function `strspn` (p. 918) determines the length of the initial part of its first argument that contains only characters from the string in its second argument and returns the length of the segment.
- Function `strstr` (p. 918) searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned; otherwise it returns 0.

Section 22.11 Memory Functions of the C String-Handling Library

- Function `memcpy` (p. 920) copies a specified number of characters from the object to which its second argument points into the object to which its first argument points. The function can receive a pointer to any object. The pointers are received as `void` pointers and converted to `char` pointers for use in the function. Function `memcpy` manipulates the bytes of its argument as characters.
- Function `memmove` (p. 921) copies a specified number of bytes from the object pointed to by its second argument to the object pointed to by its first argument. Copying is accomplished as if the bytes were copied from the second argument to a temporary character array, then copied from the temporary array to the first argument.
- Function `memcmp` (p. 921) compares the specified number of characters of its first and second arguments.
- Function `memchr` (p. 922) 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, a pointer to it is returned; otherwise, a null pointer is returned.
- Function `memset` (p. 923) copies its second argument, treated as an `unsigned char`, to a specified number of bytes of the object pointed to by the first argument.

Self-Review Exercises

22.1 Fill in the blanks in each of the following:

a) The bits in the result of an expression using the _____ operator are set to one if the corresponding bits in each operand are set to one. Otherwise, the bits are set to zero.

b) The bits in the result of an expression using the _____ operator are set to one if at least one of the corresponding bits in either operand is set to one. Otherwise, the bits are set to zero.

c) Keyword _____ introduces a structure declaration.

d) Keyword _____ is used to create a synonym for a previously defined data type.

e) Each bit in the result of an expression using the _____ operator is set to one if exactly one of the corresponding bits in either operand is set to one.

f) The bitwise AND operator `&` is often used to _____ bits (i.e., to select certain bits from a bit string while zeroing others).

g) The _____ and _____ operators are used to shift the bits of a value to the left or to the right, respectively.

22.2 Write a single statement or a set of statements to accomplish each of the following:

a) Define a structure called `Part` containing `int` variable `partNumber` and `char` array `partName`, whose values may be as long as 25 characters.

b) Define `PartPtr` to be a synonym for the type `Part *`.

c) Use separate statements to declare variable `a` to be of type `Part`, array `b[ 10 ]` to be of type `Part` and variable `ptr` to be of type pointer to `Part`.

d) Read a part number and a part name from the keyboard into the members of variable `a`.

e) Assign the member values of variable `a` to element three of array `b`.

f) Assign the address of array `b` to the pointer variable `ptr`.

g) Print the member values of element three of array `b`, using the variable `ptr` and the structure pointer operator to refer to the members.

22.3 Write a single statement to accomplish each of the following. Assume that variables `c` (which stores a character), `x`, `y` and `z` are of type `int`; variables `d`, `e` and `f` are of type `double`; variable `ptr` is of type `char *` and arrays `s1[ 100 ]` and `s2[ 100 ]` are of type `char`.

a) Convert the character stored in `c` to an uppercase letter. Assign the result to variable `c`.

b) Determine if the value of variable `c` is a digit. Use the conditional operator as shown in Figs. 22.18–22.20 to print `" is a "` or `" is not a "` when the result is displayed.

c) Convert the string `"1234567"` to `long`, and print the value.

d) Determine whether the value of variable `c` is a control character. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.

e) Assign to `ptr` the location of the last occurrence of `c` in `s1`.

f) Convert the string `"8.63582"` to `double`, and print the value.

g) Determine whether the value of `c` is a letter. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.

h) Assign to `ptr` the location of the first occurrence of `s2` in `s1`.

i) Determine whether the value of variable `c` is a printing character. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.

j) Assign to `ptr` the location of the first occurrence in `s1` of any character from `s2`.

k) Assign to `ptr` the location of the first occurrence of `c` in `s1`.

l) Convert the string `"-21"` to `int`, and print the value.

Answers to Self-Review Exercises

22.1 a) bitwise AND (`&`). b) bitwise inclusive OR (`|`). c) `struct`. d) `typedef`. e) bitwise exclusive OR (`^`). f) mask. g) left-shift operator (`<<`), right-shift operator (`>>`).

22.2 a)
```
struct Part
{
   int partNumber;
   char partName[ 26 ];
};
```
b)
```
typedef Part * PartPtr;
```
c)
```
Part a;
Part b[ 10 ];
Part *ptr;
```
d)
```
cin >> a.partNumber >> a.partName;
```
e)
```
b[ 3 ] = a;
```
f)
```
ptr = b;
```
g)
```
cout << ( ptr + 3 )->partNumber << ' '
     << ( ptr + 3 )->partName << endl;
```

22.3 a)
```
c = toupper( c );
```
b)
```
cout << '\'' << c << "\' "
     << ( isdigit( c ) ? "is a" : "is not a" )
     << " digit" << endl;
```
c)
```
cout << atol( "1234567" ) << endl;
```
d)
```
cout << '\'' << c << "\' "
     << ( iscntrl( c ) ? "is a" : "is not a" )
     << " control character" << endl;
```
e)
```
ptr = strrchr( s1, c );
```
f)
```
out << atof( "8.63582" ) << endl;
```
g)
```
cout << '\'' << c << "\' "
     << ( isalpha( c ) ? "is a" : "is not a" )
     << " letter" << endl;
```
h)
```
ptr = strstr( s1, s2 );
```
i)
```
cout << '\'' << c << "\' "
     << ( isprint( c ) ? "is a" : "is not a" )
     << " printing character" << endl;
```
j)
```
ptr = strpbrk( s1, s2 );
```
k)
```
ptr = strchr( s1, c );
```
l)
```
cout << atoi( "-21" ) << endl;
```

Exercises

22.4 *(Defining Structures)* Provide the definition for each of the following structures:

a) Structure `Inventory`, containing character array `partName[ 30 ]`, integer `partNumber`, floating-point `price`, integer `stock` and integer `reorder`.

b) A structure called `Address` that contains character arrays `streetAddress[25]`, `city[20]`, `state[3]` and `zipCode[6]`.

c) Structure `Student`, containing arrays `firstName[ 15 ]` and `lastName[ 15 ]` and variable `homeAddress` of type `struct Address` from part (b).

d) Structure `Test`, containing 16 bit fields with widths of 1 bit. The names of the bit fields are the letters a to p.

22.5 *(Card Shuffling and Dealing)* Modify Fig. 22.14 to shuffle the cards using the shuffle algorithm in Fig. 22.3. Print the resulting deck in two-column format. Precede each card with its color.

22.6 *(Shifting and Printing an Integer)* Write a program that right-shifts an integer variable four bits. The program should print the integer in bits before and after the shift operation. Does your system place zeros or ones in the vacated bits?

22.7 *(Multiplication Via Bit Shifting)* Left-shifting an `unsigned` integer by one bit is equivalent to multiplying the value by 2. Write function `power2` that takes two integer arguments, `number` and `pow`, and calculates

$$\texttt{number} * 2^{\texttt{pow}}$$

Use a shift operator to calculate the result. The program should print the values as integers and as bits.

22.8 *(Packing Characters into Unsigned Integers)* The left-shift operator can be used to pack four character values into a four-byte unsigned integer variable. Write a program that inputs four characters from the keyboard and passes them to function `packCharacters`. To pack four characters into an `unsigned` integer variable, assign the first character to the `unsigned` variable, shift the `unsigned` variable left by eight bit positions and combine the `unsigned` variable with the second character using the bitwise inclusive-OR operator, etc. The program should output the characters in their bit format before and after they're packed into the `unsigned` integer to prove that they're in fact packed correctly in the `unsigned` variable.

22.9 *(Unpacking Characters from Unsigned Integers)* Using the right-shift operator, the bitwise AND operator and a mask, write function `unpackCharacters` that takes the `unsigned` integer from Exercise 22.8 and unpacks it into four characters. To unpack characters from an `unsigned` four-byte integer, combine the unsigned integer with a mask and right-shift the result. To create the masks t you'll need to unpack the four characters, left-shift the value 255 in the mask variable by eight bits 0, 1, 2 or 3 times (depending on the byte you are unpacking). Then take the combined result each time and right shift it by eight bits the same number of times. Assign each resulting value to a `char` variable. The program should print the `unsigned` integer in bits before it's unpacked, then print the characters in bits to confirm that they were unpacked correctly.

22.10 *(Reversing Bits)* Write a program that reverses the order of the bits in an `unsigned` integer value. The program should input the value from the user and call function `reverseBits` to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly.

22.11 *(Testing Characters with the <cctype> Functions)* Write a program that inputs a character from the keyboard and tests the character with each function in the character-handling library. Print the value returned by each function.

22.12 *(Determine the Value)* The following program uses function `multiple` to determine whether the integer entered from the keyboard is a multiple of some integer `X`. Examine function `multiple`, then determine the value of `X`.

```
1  // Exercise 22.12: ex22_12.cpp
2  // This program determines if a value is a multiple of X.
3  #include <iostream>
4  using namespace std;
5
6  bool multiple( int );
7
8  int main()
9  {
10    int y = 0;
11
12    cout << "Enter an integer between 1 and 32000: ";
13    cin >> y;
```

```
14
15    if ( multiple( y ) )
16       cout << y << " is a multiple of X" << endl;
17    else
18       cout << y << " is not a multiple of X" << endl;
19 } // end main
20
21 // determine if num is a multiple of X
22 bool multiple( int num )
23 {
24    bool mult = true;
25
26    for ( int i = 0, mask = 1; i < 10; ++i, mask <<= 1 )
27       if ( ( num & mask ) != 0 )
28       {
29          mult = false;
30          break;
31       } // end if
32
33    return mult;
34 } // end function multiple
```

22.13 What does the following program do?

```
1  // Exercise 22.13: ex22_13.cpp
2  #include <iostream>
3  using namespace std;
4
5  bool mystery( unsigned );
6
7  int main()
8  {
9     unsigned x;
10
11    cout << "Enter an integer: ";
12    cin >> x;
13    cout << boolalpha
14         << "The result is " << mystery( x ) << endl;
15 } // end main
16
17 // What does this function do?
18 bool mystery( unsigned bits )
19 {
20    const int SHIFT = 8 * sizeof( unsigned ) - 1;
21    const unsigned MASK = 1 << SHIFT;
22    unsigned total = 0;
23
24    for ( int i = 0; i < SHIFT + 1; ++i, bits <<= 1 )
25       if ( ( bits & MASK ) == MASK )
26          ++total;
27
28    return !( total % 2 );
29 } // end function mystery
```

22.14 Write a program that inputs a line of text with `istream` member function `getline` (as in Chapter 13) into character array `s[100]`. Output the line in uppercase letters and lowercase letters.

22.15 *(Converting Strings to Integers)* Write a program that inputs four strings that represent integers, converts the strings to integers, sums the values and prints the total of the four values. Use only the C string-processing techniques shown in this chapter.

22.16 *(Converting Strings to Floating-Point Numbers)* Write a program that inputs four strings that represent floating-point values, converts the strings to double values, sums the values and prints the total of the four values. Use only the C string-processing techniques shown in this chapter.

22.17 *(Searching for Substrings)* Write a program that inputs a line of text and a search string from the keyboard. Using function `strstr`, locate the first occurrence of the search string in the line of text, and assign the location to variable `searchPtr` of type `char *`. If the search string is found, print the remainder of the line of text beginning with the search string. Then use `strstr` again to locate the next occurrence of the search string in the line of text. If a second occurrence is found, print the remainder of the line of text beginning with the second occurrence. [*Hint:* The second call to `strstr` should contain the expression `searchPtr + 1` as its first argument.]

22.18 *(Searching for Substrings)* Write a program based on the program of Exercise 22.17 that inputs several lines of text and a search string, then uses function `strstr` to determine the total number of occurrences of the string in the lines of text. Print the result.

22.19 *(Searching for Characters)* Write a program that inputs several lines of text and a search character and uses function `strchr` to determine the total number of occurrences of the character in the lines of text.

22.20 *(Searching for Characters)* Write a program based on the program of Exercise 22.19 that inputs several lines of text and uses function `strchr` to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array, and print the values in tabular format after the totals have been determined.

22.21 *(ASCII Character Set)* The chart in Appendix B shows the numeric code representations for the characters in the ASCII character set. Study this chart, then state whether each of the following is *true* or *false*:

a) The letter "`A`" comes before the letter "`B`."
b) The digit "`9`" comes before the digit "`0`."
c) The commonly used symbols for addition, subtraction, multiplication and division all come before any of the digits.
d) The digits come before the letters.
e) If a sort program sorts strings into ascending sequence, then the program will place the symbol for a right parenthesis before the symbol for a left parenthesis.

22.22 *(Strings Beginning with `b`)* Write a program that reads a series of strings and prints only those strings beginning with the letter "`b`."

22.23 *(Strings Ending with `ED`)* Write a program that reads a series of strings and prints only those strings that end with the letters "`ED`."

22.24 *(Displaying Characters for Given ASCII Codes)* Write a program that inputs an ASCII code and prints the corresponding character. Modify this program so that it generates all possible three-digit codes in the range 000–255 and attempts to print the corresponding characters. What happens when this program is run?

22.25 *(Write Your Own Character Handling Functions)* Using the ASCII character chart in Appendix B as a guide, write your own versions of the character-handling functions in Fig. 22.17.

22.26 *(Write Your Own String Conversion Functions)* Write your own versions of the functions in Fig. 22.27 for converting strings to numbers.

22.27 *(Write Your Own String Searching Functions)* Write your own versions of the functions in Fig. 22.34 for searching strings.

22.28 *(Write Your Own Memory Handling Functions)* Write your own versions of the functions in Fig. 22.41 for manipulating blocks of memory.

22.29 *(What Does the Program Do?)* What does this program do?

```
// Ex. 22.29: ex22_29.cpp
// What does this program do?
#include <iostream>
using namespace std;

bool mystery3( const char *, const char * ); // prototype

int main()
{
   char string1[ 80 ], string2[ 80 ];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   cout << "The result is " << mystery3( string1, string2 ) << endl;
} // end main

// What does this function do?
bool mystery3( const char *s1, const char *s2 )
{
   for ( ; *s1 != '\0' && *s2 != '\0'; ++s1, ++s2 )

      if ( *s1 != *s2 )
         return false;

   return true;
} // end function mystery3
```

22.30 *(Comparing Strings)* Write a program that uses function `strcmp` to compare two strings input by the user. The program should state whether the first string is less than, equal to or greater than the second string.

22.31 *(Comparing Strings)* Write a program that uses function `strncmp` to compare two strings input by the user. The program should input the number of characters to compare. The program should state whether the first string is less than, equal to or greater than the second string.

22.32 *(Randomly Creating Sentences)* Write a program that uses random number generation to create sentences. The program should use four arrays of pointers to `char` called `article`, `noun`, `verb` and `preposition`. The program should create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, it should be concatenated to the previous words in a character array that's large enough to hold the entire sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 such sentences.

The arrays should be filled as follows: The `article` array should contain the articles `"the"`, `"a"`, `"one"`, `"some"` and `"any"`; the `noun` array should contain the nouns `"boy"`, `"girl"`, `"dog"`, `"town"` and `"car"`; the `verb` array should contain the verbs `"drove"`, `"jumped"`, `"ran"`, `"walked"` and `"skipped"`; the `preposition` array should contain the prepositions `"to"`, `"from"`, `"over"`, `"under"` and `"on"`.

After completing the program, modify it to produce a short story consisting of several of these sentences. (How about a random term-paper writer!)

22.33 ***(Limericks)*** A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 22.32, write a C++ program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

22.34 ***(Pig Latin)*** Write a program that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm: To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function `strtok`. To translate each English word into a pig-Latin word, place the first letter of the English word at the end of the English word and add the letters "`ay`." Thus, the word "`jump`" becomes "`umpjay`," the word "`the`" becomes "`hetay`" and the word "`computer`" becomes "`omputercay`." Blanks between words remain as blanks. Assume that the English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function `printLatinWord` should display each word. [*Hint:* Each time a token is found in a call to `strtok`, pass the token pointer to function `printLatinWord` and print the pig-Latin word.]

22.35 ***(Tokenizing Phone Numbers)*** Write a program that inputs a telephone number as a string in the form `(555) 555-5555`. The program should use function `strtok` to extract the area code as a token, the first three digits of the phone number as a token, and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. Both the area code and the phone number should be printed.

22.36 ***(Tokenizing and Reversing a Sentence)*** Write a program that inputs a line of text, tokenizes the line with function `strtok` and outputs the tokens in reverse order.

22.37 ***(Alphabetizing Strings)*** Use the string-comparison functions discussed in Section 22.8 and the techniques for sorting `arrays` developed in Chapter 7 to write a program that alphabetizes a list of strings. Use the names of 10 towns in your area as data for your program.

22.38 ***(Write Your Own String Copy and Concatenation Functions)*** Write two versions of each string-copy and string-concatenation function in Fig. 22.21. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

22.39 ***(Write Your Own String Comparison Functions)*** Write two versions of each string-comparison function in Fig. 22.21. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

22.40 ***(Write Your Own String Length Function)*** Write two versions of function `strlen` in Fig. 22.21. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

Special Section: Advanced String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test your understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. You should find these problems challenging, yet enjoyable. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

22.41 ***(Text Analysis)*** The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much atten-

tion has been focused on whether William Shakespeare ever lived. Some scholars believe there is substantial evidence that Francis Bacon, Christopher Marlowe or other authors actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these authors. This exercise examines three methods for analyzing texts with a computer. Thousands of texts, including Shakespeare, are available online at `www.gutenberg.org`.

a) Write a program that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

```
To be, or not to be: that is the question:
```

contains one "a," two "b's," no "c's," etc.

b) Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

```
Whether 'tis nobler in the mind to suffer
```

contains the following word lengths and occurrences:

Word length	Occurrences
1	0
2	2
3	1
4	2 (including `'tis`)
5	0
6	2
7	1

c) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the word "to" three times, the word "be" two times, the word "or" once, etc. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

22.42 ***(Word Processing)*** One important function in word-processing systems is *type justification*—the alignment of words to both the left and right margins of a page. This generates a professional-looking document that gives the appearance of being set in type rather than prepared on a typewriter. Type justification can be accomplished on computer systems by inserting blank characters between the words in a line so that the rightmost word aligns with the right margin.

Write a program that reads several lines of text and prints this text in type-justified format. Assume that the text is to be printed on paper 8-1/2 inches wide and that one-inch margins are to be allowed on both the left and right sides. Assume that the computer prints 10 characters to the horizontal inch. Therefore, your program should print 6-1/2 inches of text, or 65 characters per line.

22.43 ***(Printing Dates in Various Formats)*** Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

```
07/21/1955
July 21, 1955
```

Write a program that reads a date in the first format and prints that date in the second format.

22.44 ***(Check Protection)*** Computers are frequently employed in check-writing systems such as payroll and accounts-payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Weird amounts are printed by computerized check-writing systems, because of human error or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection.*

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose that a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example,

```
1,230.60  (check amount)
--------
12345678  (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces would ordinarily be left blank. For example,

```
   99.87
--------
12345678
```

contains three blank spaces. If a check is printed with blank spaces, it's easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
***99.87
--------
12345678
```

Write a program that inputs a dollar amount to be printed on a check then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

22.45 ***(Writing the Word Equivalent of a Check Amount)*** Continuing the discussion of the previous example, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be both written in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it's extremely difficult to change the amount in words.

Write a program that inputs a numeric check amount and writes the word equivalent of the amount. Your program should be able to handle check amounts as large as $99.99. For example, the amount 112.43 should be written as

```
ONE HUNDRED TWELVE and 43/100
```

22.46 ***(Morse Code)*** Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound, and the

dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 22.47.

Write a program that reads an English-language phrase and encodes it in Morse code. Also write a program that reads a phrase in Morse code and converts it into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

Character	Code	Character	Code	Character	Code
A	.-	N	-.	*Digits*	
B	-...	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	-..	Q	--.-	3	...--
E	.	R	.-.	4	....-
F	..-.	S	...	5	.....
G	--.	T	-	6	-....
H	....	U	..-	7	--...
I	..	V	...-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-	0	-----
L	.-..	Y	-.--		
M	--	Z	--..		

Fig. 22.47 | Letters and digits as expressed in international Morse code.

22.47 ***(Metric Conversion Program)*** Write a program that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, etc., for the metric system and inches, quarts, pounds, etc., for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the question

```
"How many feet are in 5 kilograms?"
```

is not meaningful, because `"feet"` are units of length, while `"kilograms"` are units of weight.

Challenging String-Manipulation Projects

22.48 ***(Crossword Puzzle Generator)*** Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It's suggested here as a string-manipulation project requiring substantial sophistication and effort. There are many issues that you must resolve to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should

one use a series of strings, or should two-dimensional arrays be used? You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the "clues" portion of the puzzle, in which the brief hints for each "across" word and each "down" word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

22.49 ***(Spelling Checker)*** Many popular word-processing software packages have built-in spell checkers. We used spell-checking capabilities in preparing this book and discovered that, no matter how careful we thought we were in writing a chapter, the software was always able to find a few more spelling errors than we were able to catch manually.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. You might find it helpful to use a computerized dictionary as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it's because we simply do not know the correct spelling, so we make a "best guess." In some cases, it's because we transpose two letters (e.g., "defualt" instead of "default"). Sometimes we double-type a letter accidentally (e.g., "hanndy" instead of "handy"). Sometimes we type a nearby key instead of the one we intended (e.g., "biryhday" instead of "birthday"). And so on.

Design and implement a spell-checker program. Your program maintains an array `wordList` of character strings. You can either enter these strings or obtain them from a computerized dictionary.

Your program asks a user to enter a word. The program then looks up that word in the `wordList` array. If the word is present in the array, your program should print "`Word is spelled correctly.`"

If the word is not present in the array, your program should print "`Word is not spelled correctly.`" Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word "default" is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as "edfault," "dfeault," "deafult," "defalut" and "defautl." When you find a new word that matches one in `wordList`, print that word in a message such as "`Did you mean "default?"`."

Implement other tests, such as the replacing of each double letter with a single letter and any other tests you can develop to improve the value of your spell checker.

23

Other Topics

What's in a name? that which we call a rose
By any other name would smell as sweet.
—William Shakespeare

O Diamond! Diamond! thou little knowest the mischief done!
—Sir Isaac Newton

Objectives

In this chapter you'll learn:

- To use `const_cast` to temporarily treat a `const` object as a non-`const` object.
- To use `namespace`s.
- To use operator keywords.
- To use `mutable` members in `const` objects.
- To use class-member pointer operators `.*` and `->*`.
- To use multiple inheritance.
- The role of `virtual` base classes in multiple inheritance.

Outline

23.1 Introduction

We now consider additional C++ features. First, we discuss the `const_cast` operator, which allows you to add or remove the `const` qualification of a variable. Next, we discuss `namespaces`, which can be used to ensure that every identifier in a program has a *unique* name and can help resolve naming conflicts caused by using libraries that have the same variable, function or class names. We then present several *operator keywords* that are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. We continue our discussion with the `mutable` storage-class specifier, which enables you to indicate that a data member should *always be modifiable*, even when it appears in an object that's currently being treated as a `const` object by the program. Next we introduce two special operators that you can use with pointers to class members to access a data member or member function *without knowing its name* in advance. Finally, we introduce *multiple inheritance*, which enables a derived class to inherit the members of *several* base classes. As part of this introduction, we discuss potential problems with multiple inheritance and how *`virtual` inheritance* can be used to solve them.

23.2 `const_cast` Operator

C++ provides the **`const_cast`** operator for casting away `const` or `volatile` qualification. You declare a variable with the **`volatile`** qualifier when you expect the variable to be modified by hardware or other programs not known to the compiler. Declaring a variable `volatile` indicates that the compiler should *not optimize* the use of that variable because doing so could affect the ability of those other programs to access and modify the `volatile` variable.

In general, it's dangerous to use the `const_cast` operator, because it allows a program to modify a variable that was declared `const`. There are cases in which it's desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions that have non-`const` parameters and that do not modify their parameters—if you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report error messages.

Similarly, you could pass non-`const` data to a function that treats the data as if it were constant, then returns that data as a constant. In such cases, you might need to cast away the `const`-ness of the returned data, as we demonstrate in Fig. 23.1.

```
1  // Fig. 23.1: fig23_01.cpp
2  // Demonstrating const_cast.
3  #include <iostream>
4  #include <cstring> // contains prototypes for functions strcmp and strlen
5  #include <cctype> // contains prototype for function toupper
6  using namespace std;
7
8  // returns the larger of two C strings
9  const char *maximum( const char *first, const char *second )
10 {
11    return ( strcmp( first, second ) >= 0 ? first : second );
12 } // end function maximum
13
14 int main()
15 {
16    char s1[] = "hello"; // modifiable array of characters
17    char s2[] = "goodbye"; // modifiable array of characters
18
19    // const_cast required to allow the const char * returned by maximum
20    // to be assigned to the char * variable maxPtr
21    char *maxPtr = const_cast< char * >( maximum( s1, s2 ) );
22
23    cout << "The larger string is: " << maxPtr << endl;
24
25    for ( size_t i = 0; i < strlen( maxPtr ); ++i )
26       maxPtr[ i ] = toupper( maxPtr[ i ] );
27
28    cout << "The larger string capitalized is: " << maxPtr << endl;
29 } // end main
```

```
The larger string is: hello
The larger string capitalized is: HELLO
```

Fig. 23.1 | Demonstrating operator `const_cast`.

In this program, function `maximum` (lines 9–12) receives two C strings as `const char *` parameters and returns a `const char *` that points to the larger of the two strings. Function `main` declares the two C strings as non-`const char` arrays (lines 16–17); thus, these arrays are modifiable. In `main`, we wish to output the larger of the two C strings, then modify that C string by converting it to uppercase letters.

Function `maximum`'s two parameters are of type `const char *`, so the function's return type also must be declared as `const char *`. If the return type is specified as only `char *`, the compiler issues an error message indicating that the value being returned *cannot* be converted from `const char *` to `char *`—a dangerous conversion, because it attempts to treat data that the function believes to be `const` as if it were non-`const` data.

Even though function `maximum` *believes* the data to be constant, we know that the original arrays in `main` do *not* contain constant data. Therefore, `main` *should* be able to modify the contents of those arrays as necessary. Since we know these arrays *are* modifiable, we use `const_cast` (line 21) to *cast away the `const`-ness* of the pointer returned by `maximum`, so we can then modify the data in the array representing the larger of the two C strings. We can

then use the pointer as the name of a character array in the `for` statement (lines 25–26) to convert the contents of the larger string to uppercase letters. Without the `const_cast` in line 21, this program will *not* compile, because you are *not* allowed to assign a pointer of type `const char *` to a pointer of type `char *`.

Error-Prevention Tip 23.1

In general, a `const_cast` should be used only when it is known in advance that the original data is not constant. Otherwise, unexpected results may occur.

23.3 mutable Class Members

In Section 23.2, we introduced the `const_cast` operator, which allowed us to remove the "`const`-ness" of a type. A `const_cast` operation can also be applied to a data member of a `const` object from the body of a `const` member function of that object's class. This enables the `const` member function to modify the data member, even though the object is considered to be `const` in the body of that function. Such an operation might be performed when most of an object's data members should be considered `const`, but a particular data member still needs to be modified.

As an example, consider a linked list that maintains its contents in sorted order. Searching through the linked list does not require modifications to the data of the linked list, so the search function could be a `const` member function of the linked-list class. However, it's conceivable that a linked-list object, in an effort to make future searches more efficient, might keep track of the location of the last successful match. If the next search operation attempts to locate an item that appears later in the list, the search could begin from the location of the last successful match, rather than from the beginning of the list. To do this, the `const` member function that performs the search must be able to modify the data member that keeps track of the last successful search.

If a data member such as the one described above should *always* be modifiable, C++ provides the storage-class specifier **`mutable`** as an alternative to `const_cast`. A `mutable` data member is always modifiable, even in a `const` member function or `const` object.

Portability Tip 23.1

The effect of attempting to modify an object that was defined as constant, regardless of whether that modification was made possible by a `const_cast` or C-style cast, varies among compilers.

`mutable` and `const_cast` are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` *must* be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable. Operations involving `const_cast` are typically *hidden* in a member function's implementation. The user of a class might not be aware that a member is being modified.

Software Engineering Observation 23.1

`mutable` members are useful in classes that have "secret" implementation details that do not contribute to a client's use of an object of the class.

Mechanical Demonstration of a `mutable` *Data Member*

Figure 23.2 demonstrates using a `mutable` member. The program defines class `TestMutable` (lines 7–21), which contains a constructor, function `getValue` and a `private` data member `value` that's declared `mutable`. Lines 15–18 define function `getValue` as a `const` member function that returns a copy of `value`. Notice that the function increments `mutable` data member `value` in the `return` statement. Normally, a `const` member function *cannot* modify data members unless the object on which the function operates—i.e., the one to which `this` points—is *cast* (using `const_cast`) to a non-`const` type. Because `value` is `mutable`, this `const` function *can* modify the data.

```
1  // Fig. 23.2: fig23_02.cpp
2  // Demonstrating storage-class specifier mutable.
3  #include <iostream>
4  using namespace std;
5
6  // class TestMutable definition
7  class TestMutable
8  {
9  public:
10    TestMutable( int v = 0 )
11    {
12       value = v;
13    } // end TestMutable constructor
14
15    int getValue() const
16    {
17       return ++value; // increments value
18    } // end function getValue
19 private:
20    mutable int value; // mutable member
21 }; // end class TestMutable
22
23 int main()
24 {
25    const TestMutable test( 99 );
26
27    cout << "Initial value: " << test.getValue();
28    cout << "\nModified value: " << test.getValue() << endl;
29 } // end main
```

```
Initial value: 99
Modified value: 100
```

Fig. 23.2 | Demonstrating a `mutable` data member.

Line 25 declares `const TestMutable` object `test` and initializes it to `99`. Line 27 calls the `const` member function `getValue`, which adds one to `value` and returns its previous contents. Notice that the compiler *allows* the call to member function `getValue` on the object `test` because it's a `const` object and `getValue` is a `const` member function. However, `getValue` *modifies* variable `value`. Thus, when line 28 invokes `getValue` again, the new `value` (100) is output to prove that the `mutable` data member was indeed *modified*.

23.4 namespaces

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will "overlap" (i.e., collide) with a variable of the *same* name in a *different* scope, possibly creating a *naming conflict*. Such overlapping can occur at many levels. Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers (such as functions). This can cause compilation errors.

C++ solves this problem with **namespaces**. Each `namespace` defines a scope in which identifiers and variables are placed. To use a **namespace member**, either the member's name must be qualified with the `namespace` name and the *scope resolution operator (`::`)*, as in

```
MyNameSpace::member
```

or a `using` directive must appear *before* the name is used in the program. Typically, such `using` statements are placed at the beginning of the file in which members of the `namespace` are used. For example, placing the following `using` directive at the beginning of a source-code file

```
using namespace MyNameSpace;
```

specifies that members of `namespace` *MyNameSpace* can be used in the file without preceding each member with *MyNameSpace* and the scope resolution operator (`::`).

A `using` directive of the form

```
using std::cout;
```

brings *one* name into the scope where the directive appears. A `using` directive of the form

```
using namespace std;
```

brings *all* the names from the specified `namespace` (`std`) into the scope where the directive appears.

Error-Prevention Tip 23.2

Precede a member with its `namespace` name and the scope resolution operator (`::`) if the possibility exists of a naming conflict.

Not all `namespace`s are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their `namespace` names. Figure 23.3 demonstrates the use of `namespaces`.

```
1  // Fig. 23.3: fig23_03.cpp
2  // Demonstrating namespaces.
3  #include <iostream>
4  using namespace std;
5
6  int integer1 = 98; // global variable
7
8  // create namespace Example
9  namespace Example
10 {
```

Fig. 23.3 | Demonstrating the use of `namespaces`. (Part 1 of 3.)

```
   // declare two constants and one variable
   const double PI = 3.14159;
   const double E = 2.71828;
   int integer1 = 8;

   void printValues(); // prototype

   // nested namespace
   namespace Inner
   {
      // define enumeration
      enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
   } // end Inner namespace
} // end Example namespace

// create unnamed namespace
namespace
{
   double doubleInUnnamed = 88.22; // declare variable
} // end unnamed namespace

int main()
{
   // output value doubleInUnnamed of unnamed namespace
   cout << "doubleInUnnamed = " << doubleInUnnamed;

   // output global variable
   cout << "\n(global) integer1 = " << integer1;

   // output values of Example namespace
   cout << "\nPI = " << Example::PI << "\nE = " << Example::E
      << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
      << Example::Inner::FISCAL3 << endl;

   Example::printValues(); // invoke printValues function
} // end main

// display variable and constant values
void Example::printValues()
{
   cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
      << PI << "\nE = " << E << "\ndoubleInUnnamed = "
      << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
      << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
} // end printValues
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992
```

Fig. 23.3 | Demonstrating the use of `namespaces`. (Part 2 of 3.)

```
In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

Fig. 23.3 | Demonstrating the use of namespaces. (Part 3 of 3.)

Defining namespaces

Lines 9–24 use the keyword `namespace` to define `namespace Example`. The body of a `namespace` is delimited by braces (`{}`). The `namespace Example`'s members consist of two constants (`PI` and `E` in lines 12–13), an `int` (`integer1` in line 14), a function (`printValues` in line 16) and a **nested namespace** (`Inner` in lines 19–23). Notice that member `integer1` has the same name as global variable `integer1` (line 6). *Variables that have the same name must have different scopes*—otherwise compilation errors occur. A `namespace` can contain constants, data, classes, nested `namespaces`, functions, etc. Definitions of `namespaces` must occupy the *global scope* or be *nested* within other `namespaces`. Unlike classes, different `namespace` members can be defined in separate `namespace` blocks—each standard library header has a `namespace` block placing its contents in `namespace std`.

Lines 27–30 create an **unnamed namespace** containing the member `doubleInUnnamed`. Variables, classes and functions in an *unnamed* `namespace` are accessible only in the current **translation unit** (a `.cpp` file and the files it `includes`). However, unlike variables, classes or functions with `static` linkage, those in the *unnamed* `namespace` may be used as template arguments. The unnamed `namespace` has an implicit `using` directive, so its members appear to occupy the **global namespace**, are accessible directly and *do not have to be qualified with a* `namespace` *name*. Global variables are also part of the global `namespace` and are accessible in all scopes following the declaration in the file.

Software Engineering Observation 23.2

Each separate compilation unit has its own unique unnamed `namespace`*; i.e., the unnamed* `namespace` *replaces the* `static` *linkage specifier.*

Accessing namespace *Members with Qualified Names*

Line 35 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the *unnamed* `namespace`. Line 38 outputs the value of global variable `integer1`. For both of these variables, the compiler first attempts to locate a *local* declaration of the variables in `main`. Since there are no local declarations, the compiler assumes those variables are in the global `namespace`.

Lines 41–43 output the values of `PI`, `E`, `integer1` and `FISCAL3` from `namespace Example`. Notice that each must be *qualified* with `Example::` because the program does not provide any `using` directive or declarations indicating that it will use members of `namespace Example`. In addition, member `integer1` must be qualified, because a global variable has the same name. Otherwise, the global variable's value is output. `FISCAL3` is a member of nested `namespace Inner`, so it must be *qualified* with `Example::Inner::`.

Function `printValues` (defined in lines 49–55) is a member of `Example`, so it can access other members of the `Example` `namespace` directly *without using a* `namespace` *qualifier*. The output statement in lines 51–54 outputs `integer1`, `PI`, `E`, `doubleInUnnamed`, global variable `integer1` and `FISCAL3`. Notice that `PI` and `E` are *not qualified* with `Example`. Variable `doubleInUnnamed` is still *accessible*, because it's in the *unnamed* `namespace` and the variable name does *not conflict* with any other members of `namespace` `Example`. The global version of `integer1` must be *qualified* with the scope resolution operator (`::`), because its name *conflicts* with a member of `namespace` `Example`. Also, `FISCAL3` must be *qualified* with `Inner::`. When accessing members of a *nested* `namespace`, the members must be qualified with the `namespace` name (unless the member is being used inside the nested `namespace`).

Common Programming Error 23.1

Placing `main` *in a* `namespace` *is a compilation error.*

`using` *Directives Should* Not *Be Placed in Headers*

`namespaces` are particularly useful in large-scale applications that use many class libraries. In such cases, there's a higher likelihood of naming conflicts. When working on such projects, there should *never* be a `using` directive in a header. Having one brings the corresponding names into any file that includes the header. This could result in name collisions and subtle, hard-to-find errors. Instead, use only fully qualified names in headers (for example, `std::cout` or `std::string`).

Aliases for `namespace` *Names*

`namespaces` can be *aliased*. For example the statement

```
namespace CPPHTP = CPlusPlusHowToProgram;
```

creates the namespace alias `CPPHTP` for `CPlusPlusHowToProgram`.

23.5 Operator Keywords

The C++ standard provides operator keywords (Fig. 23.4) that can be used in place of several C++ operators. You can use operator keywords if you have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Operator	Operator keyword	Description
Logical operator keywords		
`&&`	`and`	logical AND
`||`	`or`	logical OR
`!`	`not`	logical NOT
Inequality operator keyword		
`!=`	`not_eq`	inequality

Fig. 23.4 | Operator keyword alternatives to operator symbols. (Part 1 of 2.)

Operator	Operator keyword	Description
Bitwise operator keywords		
&	bitand	bitwise AND
|	bitor	bitwise inclusive OR
^	xor	bitwise exclusive OR
~	compl	bitwise complement
Bitwise assignment operator keywords		
&=	and_eq	bitwise AND assignment
|=	or_eq	bitwise inclusive OR assignment
^=	xor_eq	bitwise exclusive OR assignment

Fig. 23.4 | Operator keyword alternatives to operator symbols. (Part 2 of 2.)

Figure 23.5 demonstrates the operator keywords. Microsoft Visual C++ 2010 requires the header <ciso646> (line 4) to use the operator keywords. In GNU C++ and LLVM, the operator keywords are always defined and this header is not required.

```
1  // Fig. 23.5: fig23_05.cpp
2  // Demonstrating operator keywords.
3  #include <iostream>
4  #include <ciso646> // enables operator keywords in Microsoft Visual C++
5  using namespace std;
6
7  int main()
8  {
9     bool a = true;
10    bool b = false;
11    int c = 2;
12    int d = 3;
13
14    // sticky setting that causes bool values to display as true or false
15    cout << boolalpha;
16
17    cout << "a = " << a << "; b = " << b
18       << "; c = " << c << "; d = " << d;
19
20    cout << "\n\nLogical operator keywords:";
21    cout << "\n   a and a: " << ( a and a );
22    cout << "\n   a and b: " << ( a and b );
23    cout << "\n    a or a: " << ( a or a );
24    cout << "\n    a or b: " << ( a or b );
25    cout << "\n     not a: " << ( not a );
26    cout << "\n     not b: " << ( not b );
27    cout << "\na not_eq b: " << ( a not_eq b );
28
```

Fig. 23.5 | Demonstrating operator keywords. (Part 1 of 2.)

```
29      cout << "\n\nBitwise operator keywords:";
30      cout << "\nc bitand d: " << ( c bitand d );
31      cout << "\n c bitor d: " << ( c bitor d );
32      cout << "\n   c xor d: " << ( c xor d );
33      cout << "\n   compl c: " << ( compl c );
34      cout << "\nc and_eq d: " << ( c and_eq d );
35      cout << "\n c or_eq d: " << ( c or_eq d );
36      cout << "\nc xor_eq d: " << ( c xor_eq d ) << endl;
37   } // end main
```

```
a = true; b = false; c = 2; d = 3

Logical operator keywords:
   a and a: true
   a and b: false
    a or a: true
    a or b: true
     not a: false
     not b: true
a not_eq b: true

Bitwise operator keywords:
c bitand d: 2
 c bitor d: 3
   c xor d: 1
   compl c: -3
c and_eq d: 2
 c or_eq d: 3
c xor_eq d: 0
```

Fig. 23.5 | Demonstrating operator keywords. (Part 2 of 2.)

The program declares and initializes two `bool` variables and two integer variables (lines 9–12). Logical operations (lines 21–27) are performed with `bool` variables `a` and `b` using the various logical operator keywords. Bitwise operations (lines 30–36) are performed with the `int` variables `c` and `d` using the various bitwise operator keywords. The result of each operation is output.

23.6 Pointers to Class Members (`.*` and `->*`)

C++ provides the `.*` and `->*` operators for accessing class members via pointers. This is a rarely used capability, primarily for advanced C++ programmers. We provide only a mechanical example of using pointers to class members here. Figure 23.6 demonstrates the pointer-to-class-member operators.

```
1  // Fig. 23.6: fig23_06.cpp
2  // Demonstrating operators .* and ->*.
3  #include <iostream>
4  using namespace std;
5
```

Fig. 23.6 | Demonstrating operatprs `.*` and `->*`. (Part 1 of 2.)

```
// class Test definition
class Test
{
public:
   void func()
   {
      cout << "In func\n";
   } // end function func

   int value; // public data member
}; // end class Test

void arrowStar( Test * ); // prototype
void dotStar( Test * ); // prototype

int main()
{
   Test test;
   test.value = 8; // assign value 8
   arrowStar( &test ); // pass address to arrowStar
   dotStar( &test ); // pass address to dotStar
} // end main

// access member function of Test object using ->*
void arrowStar( Test *testPtr )
{
   void ( Test::*memberPtr )() = &Test::func; // declare function pointer
   ( testPtr->*memberPtr )(); // invoke function indirectly
} // end arrowStar

// access members of Test object data member using .*
void dotStar( Test *testPtr2 )
{
   int Test::*vPtr = &Test::value; // declare pointer
   cout << ( *testPtr2 ).*vPtr << endl; // access value
} // end dotStar
```

```
In test function
8
```

Fig. 23.6 | Demonstrating operatprs .* and ->*. (Part 2 of 2.)

The program declares class `Test` (lines 7–16), which provides `public` member function `test` and `public` data member `value`. Lines 18–19 provide prototypes for the functions `arrowStar` (defined in lines 30–34) and `dotStar` (defined in lines 37–41), which demonstrate the `->*` and `.*` operators, respectively. Line 23 creates object `test`, and line 24 assigns `8` to its data member `value`. Lines 25–26 call functions `arrowStar` and `dotStar` with the address of the object `test`.

Line 32 in function `arrowStar` declares and initializes variable `memPtr` as a *pointer to a member function*. In this declaration, `Test::*` indicates that the variable `memPtr` is a *pointer to a member* of class `Test`. To declare a *pointer to a function*, enclose the pointer name preceded by `*` in parentheses, as in `(Test::*memPtr)`. A *pointer to a function* must

specify, as part of its type, both the *return type* of the *function it points to* and the *parameter list* of that function. The function's *return type* appears to the left of the left parenthesis and the *parameter list* appears in a separate set of parentheses to the right of the pointer declaration. In this case, the function has a `void` return type and no parameters. The pointer `memPtr` is initialized with the address of class `Test`'s member function named `test`. The header of the function must match the *function pointer's declaration*—i.e., function `test` must have a `void` return type and no parameters. Notice that the right side of the assignment uses the *address operator (&)* to get the address of the member function `test`. Also, notice that *neither the left side nor the right side of the assignment in line 32 refers to a specific object of class* `Test`. Only the class name is used with the scope resolution operator (`::`). Line 33 invokes the member function stored in `memPtr` (i.e., `test`), using the `->*` operator. Because `memPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

Line 39 declares and initializes `vPtr` as a pointer to an `int` data member of class `Test`. The right side of the assignment specifies the address of the data member `value`. Line 40 dereferences the pointer `testPtr2`, then uses the `.*` operator to access the member to which `vPtr` points. *The client code can create pointers to class members for only those class members that are accessible to the client code.* In this example, both member function `test` and data member `value` are publicly accessible.

Common Programming Error 23.2

Declaring a member-function pointer without enclosing the pointer name in parentheses is a syntax error.

Common Programming Error 23.3

Declaring a member-function pointer without preceding the pointer name with a class name followed by the scope resolution operator (`::`) is a syntax error.

Common Programming Error 23.4

Attempting to use the `->` or `*` operator with a pointer to a class member generates syntax errors.

23.7 Multiple Inheritance

In Chapters 11 and 12, we discussed *single inheritance, in which each class is derived from exactly one base class*. In C++, a class may be derived from *more than one* base class—a technique known as **multiple inheritance** in which a derived class inherits the members of two or more base classes. This powerful capability encourages interesting forms of software reuse but can cause a variety of ambiguity problems. *Multiple inheritance is a difficult concept that should be used only by experienced programmers.* In fact, some of the problems associated with multiple inheritance are so subtle that newer programming languages, such as Java and C#, do not enable a class to derive from more than one base class.

Software Engineering Observation 23.3

Great care is required in the design of a system to use multiple inheritance properly; it should not be used when single inheritance and/or composition will do the job.

A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the *same name*. This can lead to ambiguity problems when you attempt to compile. Consider the multiple-inheritance example (Figs. 23.7–23.11). Class `Base1` (Fig. 23.7) contains one `protected int` data member—`value` (line 20), a constructor (lines 10–13) that sets `value` and `public` member function `getData` (lines 15–18) that returns `value`.

```
1  // Fig. 23.7: Base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1
8  {
9  public:
10    Base1( int parameterValue )
11       : value( parameterValue )
12    {
13    } // end Base1 constructor
14
15    int getData() const
16    {
17       return value;
18    } // end function getData
19 protected: // accessible to derived classes
20    int value; // inherited by derived class
21 }; // end class Base1
22
23 #endif // BASE1_H
```

Fig. 23.7 | Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 23.8) is similar to class `Base1`, except that its `protected` data is a `char` named `letter` (line 20). Like class `Base1`, `Base2` has a `public` member function `getData`, but this function returns the value of `char` data member `letter`.

```
1  // Fig. 23.8: Base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2
8  {
9  public:
10    Base2( char characterData )
11       : letter( characterData )
12    {
13    } // end Base2 constructor
```

Fig. 23.8 | Demonstrating multiple inheritance—`Base2.h`. (Part 1 of 2.)

```
14
15    char getData() const
16    {
17       return letter;
18    } // end function getData
19 protected: // accessible to derived classes
20    char letter; // inherited by derived class
21 }; // end class Base2
22
23 #endif // BASE2_H
```

Fig. 23.8 | Demonstrating multiple inheritance—`Base2.h`. (Part 2 of 2.)

Class `Derived` (Figs. 23.9–23.10) inherits from both class `Base1` and class `Base2` through *multiple inheritance*. Class `Derived` has a `private` data member of type `double` named `real` (Fig. 23.9, line 20), a constructor to initialize all the data of class `Derived` and a `public` member function `getReal` that returns the value of `double` variable `real`.

```
1  // Fig. 23.9: Derived.h
2  // Definition of class Derived which inherits
3  // multiple base classes (Base1 and Base2).
4  #ifndef DERIVED_H
5  #define DERIVED_H
6
7  #include <iostream>
8  #include "Base1.h"
9  #include "Base2.h"
10 using namespace std;
11
12 // class Derived definition
13 class Derived : public Base1, public Base2
14 {
15    friend ostream &operator<<( ostream &, const Derived & );
16 public:
17    Derived( int, char, double );
18    double getReal() const;
19 private:
20    double real; // derived class's private data
21 }; // end class Derived
22
23 #endif // DERIVED_H
```

Fig. 23.9 | Demonstrating multiple inheritance—`Derived.h`.

```
1  // Fig. 23.10: Derived.cpp
2  // Member-function definitions for class Derived
3  #include "Derived.h"
4
5  // constructor for Derived calls constructors for
6  // class Base1 and class Base2.
```

Fig. 23.10 | Demonstrating multiple inheritance—`Derived.cpp`. (Part 1 of 2.)

```
// use member initializers to call base-class constructors
Derived::Derived( int integer, char character, double double1 )
   : Base1( integer ), Base2( character ), real( double1 ) { }

// return real
double Derived::getReal() const
{
   return real;
} // end function getReal

// display all data members of Derived
ostream &operator<<( ostream &output, const Derived &derived )
{
   output << "    Integer: " << derived.value << "\n  Character: "
        << derived.letter << "\nReal number: " << derived.real;
   return output; // enables cascaded calls
} // end operator<<
```

Fig. 23.10 | Demonstrating multiple inheritance—`Derived.cpp`. (Part 2 of 2.)

To indicate *multiple inheritance* (in Fig. 23.9) we follow the colon (`:`) after `class Derived` with a comma-separated list of base classes (line 13). In Fig. 23.10, notice that constructor `Derived` explicitly calls base-class constructors for each of its base classes—`Base1` and `Base2`—using the member-initializer syntax (line 9). The *base-class constructors are called in the order that the inheritance is specified, not in the order in which their constructors are mentioned*. Also, *if the base-class constructors are not explicitly called in the member-initializer list, their default constructors will be called implicitly*.

The overloaded stream insertion operator (Fig. 23.10, lines 18–23) uses its second parameter—a reference to a `Derived` object—to display a `Derived` object's data. This operator function is a `friend` of `Derived`, so `operator<<` can directly access *all* of class `Derived`'s `protected` and `private` members, including the `protected` data member `value` (inherited from class `Base1`), `protected` data member `letter` (inherited from class `Base2`) and `private` data member `real` (declared in class `Derived`).

Now let's examine the `main` function (Fig. 23.11) that tests the classes in Figs. 23.7–23.10. Line 11 creates `Base1` object `base1` and initializes it to the `int` value `10`. Line 12 creates `Base2` object `base2` and initializes it to the `char` value `'Z'`. Line 13 creates `Derived` object `derived` and initializes it to contain the `int` value `7`, the `char` value `'A'` and the `double` value `3.5`.

```
// Fig. 23.11: fig23_11.cpp
// Driver for multiple-inheritance example.
#include <iostream>
#include "Base1.h"
#include "Base2.h"
#include "Derived.h"
using namespace std;
```

Fig. 23.11 | Demonstrating multiple inheritance. (Part 1 of 2.)

```
 8
 9  int main()
10  {
11     Base1 base1( 10 ); // create Base1 object
12     Base2 base2( 'Z' ); // create Base2 object
13     Derived derived( 7, 'A', 3.5 ); // create Derived object
14
15     // print data members of base-class objects
16     cout << "Object base1 contains integer " << base1.getData()
17        << "\nObject base2 contains character " << base2.getData()
18        << "\nObject derived contains:\n" << derived << "\n\n";
19
20     // print data members of derived-class object
21     // scope resolution operator resolves getData ambiguity
22     cout << "Data members of Derived can be accessed individually:"
23        << "\n    Integer: " << derived.Base1::getData()
24        << "\n  Character: " << derived.Base2::getData()
25        << "\nReal number: " << derived.getReal() << "\n\n";
26     cout << "Derived can be treated as an object of either base class:\n";
27
28     // treat Derived as a Base1 object
29     Base1 *base1Ptr = &derived;
30     cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
31
32     // treat Derived as a Base2 object
33     Base2 *base2Ptr = &derived;
34     cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
35  } // end main
```

```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 23.11 | Demonstrating multiple inheritance. (Part 2 of 2.)

Lines 16–18 display each object's data values. For objects `base1` and `base2`, we invoke each object's `getData` member function. Even though there are *two* `getData` functions in this example, the calls are *not ambiguous*. In line 16, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called. In line 17, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called. Line 18 displays the contents of object `derived` using the overloaded stream insertion operator.

Resolving Ambiguity Issues That Arise When a Derived Class Inherits Member Functions of the Same Name from Multiple Base Classes

Lines 22–25 output the contents of object `derived` again by using the *get* member functions of class `Derived`. However, there is an *ambiguity* problem, because this object contains two `getData` functions, one inherited from class `Base1` and one inherited from class `Base2`. This problem is easy to solve by using the scope resolution operator. The expression `derived.Base1::getData()` gets the value of the variable inherited from class `Base1` (i.e., the `int` variable named `value`) and `derived.Base2::getData()` gets the value of the variable inherited from class `Base2` (i.e., the `char` variable named `letter`). The `double` value in `real` is printed *without ambiguity* with the call `derived.getReal()`—there are no other member functions with that name in the hierarchy.

Demonstrating the* Is-A *Relationships in Multiple Inheritance

The *is-a* relationships of *single inheritance* also apply in *multiple-inheritance* relationships. To demonstrate this, line 29 assigns the address of object `derived` to the `Base1` pointer `base1Ptr`. This is allowed because an object of class `Derived` *is an* object of class `Base1`. Line 30 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`. Line 33 assigns the address of object `derived` to the `Base2` pointer `base2Ptr`. This is allowed because an object of class `Derived` *is an* object of class `Base2`. Line 34 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

23.8 Multiple Inheritance and `virtual` Base Classes

In Section 23.7, we discussed *multiple inheritance*, the process by which one class inherits from *two or more* classes. Multiple inheritance is used, for example, in the C++ standard library to form class `basic_iostream` (Fig. 23.12).

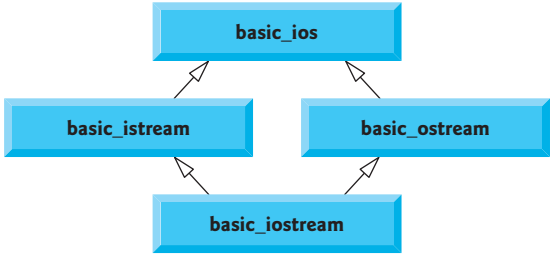

Fig. 23.12 | Multiple inheritance to form class `basic_iostream`.

Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`, each of which is formed with *single inheritance*. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. This enables class `basic_iostream` objects to provide the functionality of `basic_istream`s and `basic_ostream`s. In multiple-inheritance hierarchies, the inheritance described in Fig. 23.12 is referred to as **diamond inheritance**

Because classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, a potential problem exists for `basic_iostream`. Class `basic_iostream` could contain *two* copies of the members of class `basic_ios`—one inherited via class `basic_istream` and one

inherited via class `basic_ostream`). Such a situation would be *ambiguous* and would result in a compilation error, because the compiler would not know which version of the members from class `basic_ios` to use. In this section, you'll see how using `virtual` base classes solves the problem of inheriting duplicate copies of an indirect base class.

Compilation Errors Produced When Ambiguity Arises in Diamond Inheritance

Figure 23.13 demonstrates the *ambiguity* that can occur in *diamond inheritance*. Class `Base` (lines 8–12) contains pure `virtual` function `print` (line 11). Classes `DerivedOne` (lines 15–23) and `DerivedTwo` (lines 26–34) each publicly inherit from `Base` and override function `print`. Class `DerivedOne` and class `DerivedTwo` each contain a base-class subobject—i.e., the members of class `Base` in this example.

```
1  // Fig. 23.13: fig23_13.cpp
2  // Attempting to polymorphically call a function that is
3  // multiply inherited from two base classes.
4  #include <iostream>
5  using namespace std;
6
7  // class Base definition
8  class Base
9  {
10 public:
11    virtual void print() const = 0; // pure virtual
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : public Base
16 {
17 public:
18    // override print function
19    void print() const
20    {
21       cout << "DerivedOne\n";
22    } // end function print
23 }; // end class DerivedOne
24
25 // class DerivedTwo definition
26 class DerivedTwo : public Base
27 {
28 public:
29    // override print function
30    void print() const
31    {
32       cout << "DerivedTwo\n";
33    } // end function print
34 }; // end class DerivedTwo
35
36 // class Multiple definition
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
```

Fig. 23.13 | Attempting to call a multiply inherited function polymorphically. (Part 1 of 2.)

```
39  public:
40     // qualify which version of function print
41     void print() const
42     {
43        DerivedTwo::print();
44     } // end function print
45  }; // end class Multiple
46
47  int main()
48  {
49     Multiple both; // instantiate Multiple object
50     DerivedOne one; // instantiate DerivedOne object
51     DerivedTwo two; // instantiate DerivedTwo object
52     Base *array[ 3 ]; // create array of base-class pointers
53
54     array[ 0 ] = &both; // ERROR--ambiguous
55     array[ 1 ] = &one;
56     array[ 2 ] = &two;
57
58     // polymorphically invoke print
59     for ( int i = 0; i < 3; ++i )
60        array[ i ] -> print();
61  } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp9_examples\ch23\fig23_13\fig23_13.cpp(54) : error C2594: '=' :
   ambiguous conversions from 'Multiple *' to 'Base *'
```

Fig. 23.13 | Attempting to call a multiply inherited function polymorphically. (Part 2 of 2.)

Class `Multiple` (lines 37–45) inherits from *both* class `DerivedOne` and class `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 43). Notice that we must *qualify* the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.

Function `main` (lines 47–61) declares objects of classes `Multiple` (line 49), `DerivedOne` (line 50) and `DerivedTwo` (line 51). Line 52 declares an array of `Base *` pointers. Each array element is initialized with the address of an object (lines 54–56). An error occurs when the address of `both`—an object of class `Multiple`—is assigned to `array[ 0 ]`. The object `both` actually contains two subobjects of type `Base`, so the compiler does not know which subobject the pointer `array[ 0 ]` should point to, and it generates a compilation error indicating an *ambiguous conversion*.

Eliminating Duplicate Subobjects with `virtual` *Base-Class Inheritance*

The problem of *duplicate subobjects* is resolved with `virtual` inheritance. When a base class is inherited as `virtual`, only *one* subobject will appear in the derived class—a process called **`virtual` base-class inheritance**. Figure 23.14 revises the program of Fig. 23.13 to use a `virtual` base class.

```
// Fig. 23.14: fig23_14.cpp
// Using virtual base classes.
#include <iostream>
using namespace std;

// class Base definition
class Base
{
public:
   virtual void print() const = 0; // pure virtual
}; // end class Base

// class DerivedOne definition
class DerivedOne : virtual public Base
{
public:
   // override print function
   void print() const
   {
      cout << "DerivedOne\n";
   } // end function print
}; // end DerivedOne class

// class DerivedTwo definition
class DerivedTwo : virtual public Base
{
public:
   // override print function
   void print() const
   {
      cout << "DerivedTwo\n";
   } // end function print
}; // end DerivedTwo class

// class Multiple definition
class Multiple : public DerivedOne, public DerivedTwo
{
public:
   // qualify which version of function print
   void print() const
   {
      DerivedTwo::print();
   } // end function print
}; // end Multiple class

int main()
{
   Multiple both; // instantiate Multiple object
   DerivedOne one; // instantiate DerivedOne object
   DerivedTwo two; // instantiate DerivedTwo object

   // declare array of base-class pointers and initialize
   // each element to a derived-class type
```

Fig. 23.14 | Using `virtual` base classes. (Part 1 of 2.)

```
54      Base *array[ 3 ];
55      array[ 0 ] = &both;
56      array[ 1 ] = &one;
57      array[ 2 ] = &two;
58
59      // polymorphically invoke function print
60      for ( int i = 0; i < 3; ++i )
61         array[ i ]->print();
62   } // end main
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 23.14 | Using `virtual` base classes. (Part 2 of 2.)

The key change is that classes `DerivedOne` (line 14) and `DerivedTwo` (line 25) each inherit from `Base` by specifying `virtual public Base`. Since both classes inherit from `Base`, they each contain a *`Base` subobject*. The benefit of *`virtual` inheritance* is not clear until class `Multiple` inherits from `DerivedOne` and `DerivedTwo` (line 36). Since each of the base classes used *`virtual` inheritance* to inherit class `Base`'s members, the compiler ensures that only *one* `Base` subobject is inherited into class `Multiple`. This eliminates the ambiguity error generated by the compiler in Fig. 23.13. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[ 0 ]` in line 55 in `main`. The `for` statement in lines 60–61 polymorphically calls `print` for each object.

Constructors in Multiple-Inheritance Hierarchies with `virtual` Base Classes

Implementing hierarchies with `virtual` base classes is simpler if *default constructors* are used for the base classes. Figures 23.13 and 23.14 use compiler-generated *default constructors*. If a `virtual` base class provides a constructor that requires arguments, the derived-class implementations become more complicated, because the most derived class must explicitly invoke the `virtual` base class's constructor.

Software Engineering Observation 23.4

Providing a default constructor for `virtual` base classes simplifies hierarchy design.

23.9 Wrap-Up

In this chapter, you learned how to use the `const_cast` operator to remove the `const` qualification of a variable. We showed how to use `namespaces` to ensure that every identifier in a program has a unique name and explained how `namespaces` can help resolve naming conflicts. You saw several operator keywords to use if your keyboards do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. We showed how the `mutable` storage-class specifier enables you to indicate that a data member should always be modifiable, even when it appears in an object that's currently being treated as a `const`. We also showed the mechanics of using pointers to class members and the `->*` and `.*` operators. Finally, we introduced multiple inheritance and discussed problems associated with allowing a

derived class to inherit the members of several base classes. As part of this discussion, we demonstrated how `virtual` inheritance can be used to solve those problems.

Summary

Section 23.2 `const_cast` Operator

- C++ provides the `const_cast` operator () for casting away `const` or `volatile` qualification.
- A program declares a variable with the `volatile` qualifier (p. 939) when that program expects the variable to be modified by other programs. Declaring a variable `volatile` indicates that the compiler should not optimize the use of that variable because doing so could affect the ability of those other programs to access and modify the `volatile` variable.
- In general, it is dangerous to use the `const_cast` operator, because it allows a program to modify a variable that was declared `const`, and thus was not supposed to be modifiable.
- There are cases in which it is desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions with non-`const` parameters and that do not modify their parameters. If you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report error messages.
- If you pass non-`const` data to a function that treats the data as if it were constant, then returns that data as a constant, you might need to cast away the `const`-ness of the returned data to access and modify that data.

Section 23.3 `mutable` Class Members

- If a data member should always be modifiable, C++ provides the storage-class specifier `mutable` as an alternative to `const_cast`. A `mutable` data member (p. 941) is always modifiable, even in a `const` member function or `const` object. This reduces the need to cast away "`const`-ness."
- `mutable` and `const_cast` are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` must be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable.
- Operations involving `const_cast` are typically hidden in a member function's implementation. The user of a class might not be aware that a member is being modified.

Section 23.4 `namespace`s

- A program includes many identifiers defined in different scopes. Sometimes a variable of one scope will "overlap" with a variable of the same name in a different scope, possibly creating a naming conflict. The C++ standard solves this problem with `namespaces` (p. 943).
- Each `namespace` defines a scope in which identifiers are placed. To use a `namespace` member (p. 943), either the member's name must be qualified with the `namespace` name and the scope resolution operator (`::`) or a `using` directive or declaration must appear before the name is used in the program.
- Typically, `using` statements are placed at the beginning of the file in which members of the `namespace` are used.
- Not all `namespaces` are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their `namespace` names.
- A namespace can contain constants, data, classes, nested namespaces (p. 945), functions, etc. Definitions of namespaces must occupy the global scope or be nested within other namespaces.

- An unnamed namespace (p. 945) has an implicit `using` directive, so its members appear to occupy the global namespace, are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace.
- When accessing members of a nested namespace, the members must be qualified with the namespace name (unless the member is being used inside the nested namespace).
- Namespaces can be aliased (p. 946).

Section 23.5 Operator Keywords

- The C++ standard provides operator keywords (p. 946) that can be used in place of several C++ operators. Operator keywords are useful for programmers who have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Section 23.6 Pointers to Class Members (`.*` and `->*`)

- C++ provides the `.*` and `->*` operators (p. 948) for accessing class members via pointers. This is a rarely used capability that's used primarily by advanced C++ programmers.
- Declaring a pointer to a function requires that you enclose the pointer name preceded by an `*` in parentheses. A pointer to a function must specify, as part of its type, both the return type of the function it points to and the parameter list of that function.

Section 23.7 Multiple Inheritance

- In C++, a class may be derived from more than one base class—a technique known as multiple inheritance (p. 950), in which a derived class inherits the members of two or more base classes.
- A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the same name. This can lead to ambiguity problems when you attempt to compile.
- The *is-a* relationships of single inheritance also apply in multiple-inheritance relationships.
- Multiple inheritance is used in the C++ Standard Library to form class `basic_iostream`. Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. In multiple-inheritance hierarchies, the situation described here is referred to as diamond inheritance.

Section 23.8 Multiple Inheritance and `virtual` Base Classes

- The ambiguity in diamond inheritance (p. 955) occurs when a derived-class object inherits two or more base-class subobjects (p. 956). The problem of duplicate subobjects is resolved with `virtual` inheritance. When a base class is inherited as `virtual`, only one subobject will appear in the derived class—a process called `virtual` base-class inheritance (p. 957).
- Implementing hierarchies with `virtual` base classes is simpler if default constructors are used for the base classes. If a `virtual` base class provides a constructor that requires arguments, the implementation of the derived classes becomes more complicated, because the most derived class (p. 959) must explicitly invoke the virtual base class's constructor to initialize the members inherited from the `virtual` base class.

Self-Review Exercises

23.1 Fill in the blanks for each of the following:

a) The _____ operator qualifies a member with its namespace.

b) The _____ operator allows an object's "`const`-ness" to be cast away.

c) Because an unnamed `namespace` has an implicit `using` directive, its members appear to occupy the _____, are accessible directly and do not have to be qualified with a `namespace` name.

d) Operator _____ is the operator keyword for inequality.
e) _____ allows a class to be derived from more than one base class.
f) When a base class is inherited as _____, only one subobject of the base class will appear in the derived class.

23.2 State which of the following are *true* and which are *false*. If a statement is *false*, explain why.
a) When passing a non-`const` argument to a `const` function, the `const_cast` operator should be used to cast away the "`const`-ness" of the function.
b) A `mutable` data member cannot be modified in a `const` member function.
c) `namespaces` are guaranteed to be unique.
d) Like class bodies, `namespace` bodies also end in semicolons.
e) `namespaces` cannot have `namespaces` as members.

Answers to Self-Review Exercises

23.1 a) binary scope resolution (`::`). b) `const_cast`. c) global `namespace`. d) `not_eq`. e) multiple inheritance. f) `virtual`.

23.2 a) False. It's legal to pass a non-`const` argument to a `const` function. However, when passing a `const` reference or pointer to a non-`const` function, the `const_cast` operator should be used to cast away the "`const`-ness" of the reference or pointer
b) False. A `mutable` data member is always modifiable, even in a `const` member function.
c) False. Programmers might inadvertently choose the `namespace` already in use.
d) False. `namespace` bodies do not end in semicolons.
e) False. `namespaces` can be nested.

Exercises

23.3 ***(Fill in the Blanks)*** Fill in the blanks for each of the following:
a) Keyword _____ specifies that a `namespace` or `namespace` member is being used.
b) Operator _____ is the operator keyword for logical OR.
c) Storage specifier _____ allows a member of a `const` object to be modified.
d) The _____ qualifier specifies that an object can be modified by other programs.
e) Precede a member with its _____ name and the scope resolution operator if the possibility exists of a scoping conflict.
f) The body of a namespace is delimited by _____.
g) For a `const` object with no _____ data members, operator _____ must be used every time a member is to be modified.

23.4 ***(`Currency namespace`)*** Write a `namespace`, `Currency`, that defines constant members `ONE`, `TWO`, `FIVE`, `TEN`, `TWENTY`, `FIFTY` and `HUNDRED`. Write two short programs that use `Currency`. One program should make all constants available and the other should make only `FIVE` available.

23.5 ***(Namespaces)*** Given the `namespaces` in Fig. 23.15, determine whether each statement is *true* or *false*. Explain any *false* answers.
a) Variable `kilometers` is visible within `namespace Data`.
b) Object `string1` is visible within `namespace Data`.
c) Constant `POLAND` is not visible within `namespace Data`.
d) Constant `GERMANY` is visible within `namespace Data`.
e) Function `function` is visible to `namespace Data`.
f) Namespace `Data` is visible to `namespace CountryInformation`.
g) Object `map` is visible to `namespace CountryInformation`.
h) Object `string1` is visible within `namespace RegionalInformation`.

```
1  namespace CountryInformation
2  {
3     using namespace std;
4     enum Countries { POLAND, SWITZERLAND, GERMANY,
5                      AUSTRIA, CZECH_REPUBLIC };
6     int kilometers;
7     string string1;
8
9     namespace RegionalInformation
10    {
11       short getPopulation(); // assume definition exists
12       MapData map; // assume definition exists
13    } // end RegionalInformation
14 } // end CountryInformation
15
16 namespace Data
17 {
18    using namespace CountryInformation::RegionalInformation;
19    void *function( void *, int );
20 } // end Data
```

Fig. 23.15 | namespaces for Exercise 23.5.

23.6 *(`mutable` vs. `const_cast`)* Compare and contrast `mutable` and `const_cast`. Give at least one example of when one might be preferred over the other. [*Note:* This exercise does not require any code to be written.]

23.7 *(Modifying a `const` Variable)* Write a program that uses `const_cast` to modify a `const` variable. [*Hint:* Use a pointer in your solution to point to the `const` identifier.]

23.8 *(`virtual` Base Classes)* What problem do `virtual` base classes solve?

23.9 *(`virtual` Base Classes)* Write a program that uses `virtual` base classes. The class at the top of the hierarchy should provide a constructor that takes at least one argument (i.e., do not provide a default constructor). What challenges does this present for the inheritance hierarchy?

23.10 *(Find the Code Errors)* Find the error(s) in each of the following. When possible, explain how to correct each error.

a)
```
namespace Name {
   int x;
   int y;
   mutable int z;
};
```

b)
```
int integer = const_cast< int >( double );
```

c)
```
namespace PCM( 111, "hello" ); // construct namespace
```