

When you have completed a CRC card for each class in the application, you will have a good idea of each class's responsibilities and how the classes must interact.



Checkpoint

- 14.31 What are the benefits of having operator functions that perform object conversion?
- 14.32 Why are no return types listed in the prototypes or headers of operator functions that perform data type conversion?
- 14.33 Assume there is a class named `BlackBox`. Write the header for a member function that converts a `BlackBox` object to an `int`.
- 14.34 Assume there are two classes, `Big` and `Small`. The `Big` class has, as a member, an instance of the `Small` class. Write a sentence that describes the relationship between the two classes.

14.9

Focus on Object-Oriented Programming: Simulating the Game of Cho-Han

Cho-Han is a traditional Japanese gambling game in which a dealer uses a cup to roll two six-sided dice. The cup is placed upside down on a table so the value of the dice is concealed. Players then wager on whether the sum of the dice values is even (Cho) or odd (Han). The winner, or winners, take all of the wagers, or the house takes them if there are no winners.

We will develop a program that simulates a simplified variation of the game. The simulated game will have a dealer and two players. The players will not wager money, but will simply guess whether the sum of the dice values is even (Cho) or odd (Han). One point will be awarded to the player, or players, for correctly guessing the outcome. The game is played for five rounds, and the player with the most points is the grand winner.

In the program, we will use the `Die` class introduced in Chapter 13 to simulate the dice. We will create two instances of the `Die` class to represent two six-sided dice. In addition to the `Die` class, we will write the following classes:

- **Dealer class:** We will create an instance of this class to represent the dealer. It will have the ability to roll the dice, report the value of the dice, and report whether the total dice value is Cho or Han.
- **Player class:** We will create two instances of this class to represent the players. Instances of the `Player` class can store the player's name, make a guess between Cho or Han, and be awarded points.

First, let's look at the `Dealer` class.

Contents of Dealer.h

```

1 // Specification file for the Dealer class
2 #ifndef DEALER_H
3 #define DEALER_H
4 #include <string>
5 #include "Die.h"
6 using namespace std;
7

```

```

8 class Dealer
9 {
10 private:
11     Die die1;           // Object for die #1
12     Die die2;           // Object for die #2
13     int die1Value;      // Value of die #1
14     int die2Value;      // Value of die #2
15
16 public:
17     Dealer();          // Constructor
18     void rollDice();   // To roll the dice
19     string getChoOrHan(); // To get the result (Cho or Han)
20     int getDie1Value();  // To get the value of die #1
21     int getDie2Value();  // To get the value of die #2
22 };
23 #endif

```

Contents of Dealer.cpp

```

1 // Implementation file for the Dealer class
2 #include "Dealer.h"
3 #include "Die.h"
4 #include <string>
5 using namespace std;
6
7 //*****
8 // Constructor
9 //*****
10 Dealer::Dealer()
11 {
12     // Set the initial dice values to 0.
13     // (We will not use these values.)
14     die1Value = 0;
15     die2Value = 0;
16 }
17
18 //*****
19 // The rollDice member function rolls the
20 // dice and saves their values.
21 //*****
22 void Dealer::rollDice()
23 {
24     // Roll the dice.
25     die1.roll();
26     die2.roll();
27
28     // Save the dice values.
29     die1Value = die1.getValue();
30     die2Value = die2.getValue();
31 }
32

```

```

33 //*****
34 // The getChoOrHan member function returns *
35 // the result of the dice roll, Cho (even) *
36 // or Han (odd).
37 //*****
38 string Dealer::getChoOrHan()
39 {
40     string result; // To hold the result
41
42     // Get the sum of the dice.
43     int sum = die1Value + die2Value;
44
45     // Determine even or odd.
46     if (sum % 2 == 0)
47         result = "Cho (even)";
48     else
49         result = "Han (odd)";
50
51     // Return the result.
52     return result;
53 }
54
55 //*****
56 // The getDie1Value member function returns *
57 // the value of die #1.
58 //*****
59 int Dealer::getDie1Value()
60 {
61     return die1Value;
62 }
63
64 //*****
65 // The getDie2Value member function returns *
66 // the value of die #2.
67 //*****
68 int Dealer::getDie2Value()
69 {
70     return die2Value;
71 }

```

Here is a synopsis of the class members:

die1	Declared in line 11 of Dealer.h. This is an instance of the Die class, to represent die #1.
die2	Declared in line 12 of Dealer.h. This is an instance of the Die class, to represent die #2.
die1Value	Declared in line 13 of Dealer.h. This member variable will hold the value of die #1 after it has been rolled.
die2Value	Declared in line 14 of Dealer.h. This member variable will hold the value of die #2 after it has been rolled.
Constructor	Lines 10 through 16 of Dealer.cpp initializes the die1Value and die2Value fields to 0.

rollDice	Lines 22 through 31 of Dealer.cpp. This member function simulates the rolling of the dice. Lines 25 and 26 call the Die objects' roll method. Lines 29 and 30 save the value of the dice in the die1Value and die2Value member variables.
getChoOrHan	Lines 38 through 53 of Dealer.cpp. This member function returns a string indicating whether the sum of the dice is Cho (even) or Han (odd).
getDie1Value	Lines 59 through 62 of Dealer.cpp. This member function returns the value of first die (stored in the die1Value member variable).
getDie2Value	Lines 68 through 71 of Dealer.cpp. This member function returns the value of second die (stored in the die2Value member variable).

Now, let's look at the Player class.

Contents of Player.h

```
1 // Specification file for the Player class
2 #ifndef PLAYER_H
3 #define PLAYER_H
4 #include <string>
5 using namespace std;
6
7 class Player
8 {
9 private:
10     string name;           // The player's name
11     string guess;          // The player's guess
12     int points;            // The player's points
13
14 public:
15     Player(string);        // Constructor
16     void makeGuess();      // Causes player to make a guess
17     void addPoints(int);   // Adds points to the player
18     string getName();      // Returns the player's name
19     string getGuess();     // Returns the player's guess
20     int getPoints();       // Returns the player's points
21 };
22 #endif
```

Contents of Player.cpp

```
1 // Implementation file for the Player class
2 #include "Player.h"
3 #include <cstdlib>
4 #include <ctime>
5 #include <string>
6 using namespace std;
7
8 //*****
9 // Constructor
10 //*****
11 Player::Player(string playerName)
```

```
12  {
13      // Seed the random number generator.
14      srand(time(0));
15
16      name = playerName;
17      guess = "";
18      points = 0;
19  }
20
21 //*****
22 // The makeGuess member function causes the      *
23 // player to make a guess, either "Cho (even)"  *
24 // or "Han (odd)".                            *
25 //*****
26 void Player::makeGuess()
27 {
28     const int MIN_VALUE = 0;
29     const int MAX_VALUE = 1;
30
31     int guessNumber; // For the user's guess
32
33     // Get a random number, either 0 or 1.
34     guessNumber = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
35
36     // Convert the random number to Cho or Han.
37     if (guessNumber == 0)
38         guess = "Cho (even)";
39     else
40         guess = "Han (odd)";
41 }
42
43 //*****
44 // The addPoints member function adds a      *
45 // specified number of points to the player's  *
46 // current balance.                           *
47 //*****
48 void Player::addPoints(int newPoints)
49 {
50     points += newPoints;
51 }
52
53 //*****
54 // The getName member function returns a      *
55 // player's name.                            *
56 //*****
57 string Player::getName()
58 {
59     return name;
60 }
61
62 //*****
63 // The getGuess member function returns a      *
64 // player's guess.                            *
65 //*****
```

```
66 string Player::getGuess()
67 {
68     return guess;
69 }
70
71 //*****
72 // The getPoints member function returns a
73 // player's points.
74 //*****
75 int Player::getPoints()
76 {
77     return points;
78 }
```

Here is a synopsis of the class members:

name	Declared in line 10 of Player.h. This member variable will hold the player's name.
guess	Declared in line 11 of Player.h. This member variable will hold the player's guess.
points	Declared in line 12 of Player.h. This member variable will hold the player's points.
Constructor	Lines 11 through 19 of Player.cpp, accepts an argument for the player's name, which is assigned to the name field. The guess field is assigned an empty string, and the points field is set to 0.
makeGuess	Lines 26 through 41 of Player.cpp. This member function causes the player to make a guess. The function generates a random number that is either a 0 or a 1. The if statement that begins at line 37 assigns the string "Cho (even)" to the guess member variable if the random number is 0, or it assigns the string "Han (odd)" to the guess member variable if the random number is 1.
addPoints	Lines 48 through 51 of Player.cpp. This member function adds the number of points specified by the argument to the player's point member variable.
getName	Lines 57 through 60 of Player.cpp. This member function returns the player's name.
getGuess	Lines 66 through 69 of Player.cpp. This member function returns the player's guess.
getPoints	Lines 75 through 78 of Player.cpp. This member function returns the player's points.

Program 14-17 uses these classes to simulate the game. The `main` function simulates five rounds of the game, displaying the results of each round, then displays the overall game results.

Program 14-17

```
1 // This program simulates the game of Cho-Han.
2 #include <iostream>
3 #include <string>
4 #include "Dealer.h"
5 #include "Player.h"
6 using namespace std;
7
```

(program continues)

Program 14-17 (continued)

```

8 // Function prototypes
9 void roundResults(Dealer &, Player &, Player &);
10 void checkGuess(Player &, Dealer &);
11 void displayGrandWinner(Player, Player);
12
13 int main()
14 {
15     const int MAX_ROUNDS = 5;    // Number of rounds
16     string player1Name;        // First player's name
17     string player2Name;        // Second player's name
18
19     // Get the player's names.
20     cout << "Enter the first player's name: ";
21     cin >> player1Name;
22     cout << "Enter the second player's name: ";
23     cin >> player2Name;
24
25     // Create the dealer.
26     Dealer dealer;
27
28     // Create the two players.
29     Player player1(player1Name);
30     Player player2(player2Name);
31
32     // Play the rounds.
33     for (int round = 0; round < MAX_ROUNDS; round++)
34     {
35         cout << "-----\n";
36         cout << "Now playing round " << (round + 1)
37             << endl;
38
39         // Roll the dice.
40         dealer.rollDice();
41
42         // The players make their guesses.
43         player1.makeGuess();
44         player2.makeGuess();
45
46         // Determine the winner of this round.
47         roundResults(dealer, player1, player2);
48     }
49
50     // Display the grand winner.
51     displayGrandWinner(player1, player2);
52     return 0;
53 }
54
55 //*****
56 // The roundResults function determines the results *
57 // of the current round. *
58 //*****
59 void roundResults(Dealer &dealer, Player &player1, Player &player2)

```

```

149
148 cout << "Game over. Here are the results:\n";
147 cout << "-----\n";
146 }
145 void displayGrandwinner(Player player1, Player player2)
144 //*****
143 // game's grand winner.
142 // The displayGrandwinner function displays the
141 // ****
140 //*****
141 {
142 {
143 {
144 {
145 cout << "Awarding " << POINTS_TO_ADD
146 cout << "points(s) to " << player.getName();
147 cout << " " << endl;
148 cout << "Player " << player.getName();
149 cout << " guessed " << player.getGuess() << endl;
150 cout << "The player guessed correctly.";
151 cout << " " << endl;
152 if (guess == choHanResult)
153 {
154 cout << "Award points if the player guessed correctly.";
155 cout << " " << endl;
156 cout << "The player's guess ";
157 cout << " " << endl;
158 cout << " " << endl;
159 cout << " " << endl;
160 cout << " " << endl;
161 cout << " " << endl;
162 cout << " " << endl;
163 cout << " " << endl;
164 cout << " " << endl;
165 cout << " " << endl;
166 cout << " " << endl;
167 cout << " " << endl;
168 cout << " " << endl;
169 checkGuess(player1, dealer);
170 checkGuess(player2, dealer);
171 }
172 /**
173 // The checkGuess function checks a player's guess
174 // against the dealer's result.
175 // ****
176 // ****
177 void checkGuess(Player player, Dealer &dealer)
178 {
179 const int POINTS_TO_ADD = 1; // Points to award winner
180
181 cout << "Get the result (Cho or Han). ";
182 string guess = player.getGuess();
183 cout << "Get the player's guess ";
184 cout << "Get the result (Cho or Han). ";
185 string choHanResult = dealer.getChoHan();
186
187 cout << "Display the player's guess ";
188 cout << "The player " << player.getName();
189 cout << " guessed " << player.getGuess() << endl;
190 cout << " " << endl;
191 cout << " " << endl;
192 if (guess == choHanResult)
193 {
194 cout << "Award points if the player guessed correctly.";
195 cout << " " << endl;
196 cout << "The player guessed correctly.";
197 cout << " " << endl;
198 cout << " " << endl;
199 cout << " " << endl;
200 cout << " " << endl;
201 cout << " " << endl;
202 cout << " " << endl;
203 cout << " " << endl;
204 cout << " " << endl;
205 void displayGrandwinner(Player player1, Player player2)
206 {
207 cout << "-----\n";
208 cout << "Game over. Here are the results:\n";
209 }

```

Program 14-17 *(continued)*

```

110     // Display player #1's results.
111     cout << player1.getName() << ":" "
112         << player1.getPoints() << " points\n";
113
114     // Display player #2's results.
115     cout << player2.getName() << ":" "
116         << player2.getPoints() << " points\n";
117
118     // Determine the grand winner.
119     if (player1.getPoints() > player2.getPoints())
120     {
121         cout << player1.getName()
122             << " is the grand winner!\n";
123     }
124     else if (player2.getPoints() > player1.getPoints())
125     {
126         cout << player2.getName()
127             << " is the grand winner!\n";
128     }
129     else
130     {
131         cout << "Both players are tied!\n";
132     }
133 }
```

Program Output with Example Input Shown in Bold

Enter the first player's name: **Bill**
 Enter the second player's name: **Jill**

Now playing round 1
 The dealer rolled 4 and 6
 Result: Cho (even)
 The player Bill guessed Cho (even)
 Awarding 1 point(s) to Bill
 The player Jill guessed Cho (even)
 Awarding 1 point(s) to Jill

Now playing round 2
 The dealer rolled 5 and 2
 Result: Han (odd)
 The player Bill guessed Han (odd)
 Awarding 1 point(s) to Bill
 The player Jill guessed Han (odd)
 Awarding 1 point(s) to Jill

Now playing round 3
 The dealer rolled 4 and 2
 Result: Cho (even)
 The player Bill guessed Cho (even)
 Awarding 1 point(s) to Bill

```
The player Jill guessed Cho (even)
Awarding 1 point(s) to Jill
-----
Now playing round 4
The dealer rolled 3 and 2
Result: Han (odd)
The player Bill guessed Han (odd)
Awarding 1 point(s) to Bill
The player Jill guessed Cho (even)
-----
Now playing round 5
The dealer rolled 4 and 6
Result: Cho (even)
The player Bill guessed Han (odd)
The player Jill guessed Han (odd)
-----
Game over. Here are the results:
Bill: 4 points
Jill: 3 points
Bill is the grand winner!
```

Let's look at the code. Here is a summary of the `main` function:

- Lines 15 through 17 make the following declarations: `MAX_ROUNDS`—the number of rounds to play, `player1Name`—to hold the name of player #1, and `player2Name`—to hold the name of player #2.
- Lines 20 through 23 prompt the user to enter the player's names.
- Line 26 creates an instance of the `Dealer` class to represent the dealer.
- Line 29 creates an instance of the `Player` class to represent player #1. Notice `player1Name` is passed as an argument to the constructor.
- Line 30 creates another instance of the `Player` class to represent player #2. Notice `player2Name` is passed as an argument to the constructor.
- The for loop that begins in line 33 iterates five times, causing the simulation of five rounds of the game. The loop performs the following actions:
 - Line 40 causes the dealer to roll the dice.
 - Line 43 causes player #1 to make a guess (Cho or Han).
 - Line 44 causes player #2 to make a guess (Cho or Han).
 - Line 47 passes the `dealer`, `player1`, and `player2` objects to the `roundResults` function. The function displays the results of this round.
 - Line 51 passes the `player1` and `player2` objects to the `displayGrandWinner` function, which displays the grand winner of the game.

The `roundResults` function, which displays the results of a round, appears in lines 59 through 71. Here is a summary of the function:

- The function accepts references to the `dealer`, `player1`, and `player2` objects as arguments.
- The statement in lines 62 and 63 displays the value of the two dice.
- Line 66 calls the `dealer` object's `getChoOrHan` function to display the results, Cho or Han.
- Line 69 calls the `checkGuess` function, passing the `player1` and `dealer` objects as arguments. The `checkGuess` function compares a player's guess to the dealer's result (Cho or Han), and awards points to the player if the guess is correct.
- Line 70 calls the `checkGuess` function, passing the `player2` and `dealer` objects as arguments.

The `checkGuess` function, which compares a player's guess to the dealer's result, awarding points to the player for a correct guess, appears in lines 77 through 99. Here is a summary of the function:

- The function accepts references to a `Player` object and the `Dealer` object as arguments.
- Line 79 declares the constant `POINTS_TO_ADD`, set to the value 1, which is the number of points to add to the player's balance if the player's guess is correct.
- Line 82 assigns the player's guess to the `string` object `guess`.
- Line 85 assigns the dealer's results (Cho or Han) to the `string` object `choHanResult`.
- The statement in lines 88 and 89 displays the player's name and guess.
- The `if` statement in line 92 compares the player's guess to the dealer's result. If they match, then the player guessed correctly, and line 94 awards points to the player.

The `displayGrandWinner` function, which displays the grand winner of the game, appears in lines 105 through 133. Here is a summary of the function:

- The function accepts the `player1` and `player2` objects as arguments.
- The statements in lines 111 through 116 display both players' names and points.
- The `if-else-if` statement that begins in line 119 determines which of the two players has the highest score and displays that player's name as the grand winner. If both players have the same score, a tie is declared.

14.10

Rvalue References and Move Semantics

CONCEPT: A move operation transfers resources from a source object to a target object. A move operation is appropriate when the source object is temporary and is about to be destroyed.

C++ has always been regarded as a high-performance language. Typically, it is the language of choice for developers who need to write fast, efficient code. The C++ 11 standard introduced new features that make C++ even faster and more efficient. One of these new features is the ability to use *move semantics* in your classes. In certain situations, you can use move semantics to greatly improve the performance of assignment operations. You can also use move semantics to write a new type of constructor known as a move constructor. Before we can discuss move semantics, however, we must introduce another new topic: rvalue references.

Lvalues and Rvalues

During the execution of a program, numerous entities, such as variables, class objects, and so on, are created in memory. For example, look at the following `main` function:

```

1 int main()
2 {
3     int x = 0;
4
5     x = 12;
6     cout << x << endl;
7     return 0;
8 }
```

In this code, an `int` variable named `x` is created in line 3. Although the variable is created by the statement in line 3, it is meant to be accessible to other statements. In fact, any statement in the variable's scope can use the name `x` to access the variable. For example, line 5 assigns the value 12 to the `x` variable, and line 6 displays the contents of the `x` variable.

Some entities, such as the variable `x` in the previous example, are created by definition statements and meant to exist in memory long enough for other statements to access them. Other entities, however, are meant to exist only temporarily. For example, look at the following code fragment:

```
int x;
x = 2 * 6;
```

When the second statement executes, the system evaluates the expression on the right side of the `=` operator. In this case, it gets the value of `2 * 6` and stores the result, 12, as a temporary value. Then, it copies the temporary value to the `x` variable. After the statement executes, the temporary value is no longer needed, so the system discards it.

Values that are returned from functions are also stored as temporary values. When a function returns a value, you have to immediately do something with the value, or you will lose it. For example, look at the following code fragment:

```
1 int square(int a)
2 {
3     return a * a;
4 }
5
6 int main()
7 {
8     int x = 0;
9
10    x = square(5);
11    cout << x << endl;
12    return 0;
13 }
```

Consider the statement in line 10. We can summarize the statement by saying it calls the `square` function, and assigns the value that is returned from the `square` function to the variable `x`. But more specifically, all of the following things happen when this statement executes:

- The `square` function is called, and the value 5 is passed as an argument.
- The `square` function calculates $5 * 5$ and stores the result, 25, as a temporary value.
- The temporary value is copied (assigned) to the variable `x`.
- The temporary value is no longer needed, so the system discards it.

To further illustrate this, look at the following code fragment:

```
1 int square(int a)
2 {
3     return a * a;
4 }
5
6 int main()
```

```

7  {
8      int x = 0;
9
10     square(5);
11     cout << x << endl;
12     return 0;
13 }

```

In this code, the statement in line 10 calls the `square` function, but it doesn't do anything with the value that is returned. As in the previous example, the value 25 is stored as a temporary value. But, in this example, the value is lost forever after the statement has finished executing.

So, we have two types of values stored in memory during the execution of a program:

- Values that persist beyond the statement that created them, and have names that make them accessible to other statements in the program. In C++, these values are called *lvalues*.
- Values that are temporary, and cannot be accessed beyond the statement that created them. In C++, these values are called *rvalues*.



TIP: Perhaps the simplest way to think of an lvalue is this: it is an expression that can be written on the left side of an assignment operator.

Rvalue References

In Chapter 6, you learned that a reference variable is a variable that refers to the memory location of another variable. For example, look at the following code fragment:

```

int x = 34;
int &ref = x;

```

11

In this code, the variable `ref` is a reference variable. In a declaration, a reference is indicated by the presence of an ampersand (&) between the type and the variable's identifier. In C++ 11 and later, this type of reference is called an *lvalue reference* because it can refer only to lvalues.

C++ 11 also introduces the concept of an *rvalue reference*, which is a reference variable that can refer only to temporary objects that would otherwise have no name. An rvalue reference is declared similarly to an lvalue reference, except with a double ampersand (&&). The code below uses an rvalue reference to print the square of 5:

```

int &&rvalRef = square(5);
cout << rvalRef << endl;

```

The first statement calls the `square` function, passing 5 as an argument. The function returns the value 25, which is stored as a temporary value. The address of the temporary value is then assigned to `rvalRef`. So, now we have given a name to the temporary value, so it can be accessed by other statements.

When you assign an rvalue reference to a temporary value, you are giving a name to the temporary value and making it possible to access the value from other parts of the program. Therefore, you are transforming a temporary value into an lvalue. Because an rvalue reference cannot refer to an lvalue, the following code will not compile:

```
int x = 0;
int &&rvalRef = x; // Error!
```

Look at the following initialization statement:

```
int &&rvalRef1 = square(5);
```

After this statement executes, the memory location containing the value returned by `square(5)` has a name, `rvalRef1`, so `rvalRef1` itself becomes an lvalue. This means that a subsequent initialization statement such as:

```
int &&rvalRef2 = rvalRef1; // Error!
```

will not compile, because `rvalRef1` is no longer an rvalue. Fortunately, there is a positive consequence of all this: a temporary object can have at most one lvalue reference pointing to it. If a function has an lvalue reference to a temporary object, you can be sure no other part of the program has access to the same object. This fact is important in understanding how move operations, to be discussed next, work.

Move Semantics

When objects are passed as arguments into functions, and returned from functions, the program does a lot of work behind the scenes, allocating memory and copying values. To illustrate, let's look at the `Person` class.

```
#ifndef PERSON_H
#define PERSON_H
#include <cstring>
using namespace std;

class Person
{
private:
    char *name = nullptr;
public:
    // Default constructor
    Person()
    {   name = new char[1];
        *name = 0; }

    // Constructor with a parameter
    Person(char *n)
    {   name = new char[strlen(n) + 1];
        strcpy(name, n); }

    // Copy constructor
    Person(const Person &obj)
    {   name = new char[strlen(obj.name) + 1];
        strcpy(name, obj.name); }

    // Destructor
    ~Person()
    {   delete name; }
```

```

    // Copy assignment operator
    Person & operator=(const Person &right)
    { if (this != &right)
        {
            name = new char[strlen(right.name) + 1];
            strcpy(name, right.name);
        }
        return *this; }

    // getName member function
    char *getName() const
    { return name; }
};

#endif

```

The Person class has three constructors: a default constructor, a parameterized constructor, and a copy constructor. It also has an overloaded = operator, and a member function to get the name member variable. We are going to use this class to demonstrate some of the work that happens when an object is passed around in a program. But first, we are going to insert some cout statements so we can see when the constructors and the destructor are called, as well as the overloaded = operator. The modified class is shown here, followed by Program 14-18 which demonstrates the class.

Contents of Person.h

```

1 #ifndef PERSON_H
2 #define PERSON_H
3 #include <iostream>
4 #include <cstring>
5 using namespace std;
6
7 class Person
8 {
9 private:
10     char *name;
11 public:
12     // Default constructor
13     Person()
14     { cout << "**** default constructor ****\n";
15         name = nullptr; }
16
17     // Constructor with a parameter
18     Person(char *n)
19     { cout << "**** parameterized constructor ****\n";
20         name = new char[strlen(n) + 1];
21         strcpy(name, n); }
22
23     // Copy constructor
24     Person(const Person &obj)
25     { cout << "**** copy constructor ****\n";
26         name = new char[strlen(obj.name) + 1];
27         strcpy(name, obj.name); }
28
29

```

```
30     // Destructor
31     ~Person()
32     { cout << "**** destructor ***\n";
33         delete [] name; }
34
35     // Overloaded = operator
36     Person & operator=(const Person &right)
37     { cout << "**** assignment operator ***\n";
38         if (this != &right)
39         {
40             if (name != nullptr)
41                 delete[] name;
42             name = new char[strlen(right.name) + 1];
43             strcpy(name, right.name);
44         }
45         return *this; }
46
47     // getName member function
48     char *getName() const
49     { return name; }
50 }
51 #endif
```

Program 14-18

```
1 #include <iostream>
2 #include <string>
3 #include "Person.h"
4 using namespace std;
5
6 // Function prototype
7 Person makePerson();
8 void displayPerson(Person);
9
10 int main()
11 {
12     Person person;
13     person = makePerson();
14     displayPerson(person);
15     return 0;
16 }
17
18 Person makePerson()
19 {
20     Person p("Will MacKenzie");
21     return p;
22 }
23
24 void displayPerson(Person p2)
25 {
26     cout << p2.getName() << endl;
27 }
```

(program output continues)

Program 14-18 (continued)**Program Output**

```

*** default constructor ***
*** parameterized constructor ***
*** copy constructor ***
*** destructor ***
*** assignment operator ***
*** destructor ***
*** copy constructor ***
Will MacKenzie
*** destructor ***
*** destructor ***

```

(1) Object named person created in main
 (2) Object named p created in makePerson
 (3) Temporary object created as a copy of p
 (4) Object p destroyed
 (5) Temporary object assigned to person in main
 (6) Temporary object destroyed
 (7) p2 parameter object in displayPerson
 (8) Displayed by the cout statement in line 26
 (9) p2 parameter object destroyed
 (10) person object in main destroyed

In the program's output, the comments that are shown to the right, in a different color, are not part of the output. We have added those so we can refer to specific lines in our discussion.

Let's step through the execution of the program, and take a closer look at its output:

- In the `main` function, the statement in line 12 creates a local `Person` object named `person`. This statement causes the `Person` class's default constructor to execute, displaying the output in line (1).
- Line 13 calls the `makePerson` function. Inside the `makePerson` function, the statement in line 20 creates a `Person` object named `p`, passing "Will MacKenzie" to the constructor. This causes the output in line (2) to be displayed.
- In line 21, the `makePerson` function returns the `Person` object named `p`. Before the function terminates, however, a temporary object is created, as a copy of the `p` object. This causes the temporary object's copy constructor to be called, displaying the output in line (3).
- As the `makePerson` function terminates, the `p` object is destroyed, causing the output in line (4) to be displayed.
- In line 13, the temporary object is assigned to the `person` object. This causes the `Person` class's overloaded `=` operator to be called, displaying the output in line (5).
- The temporary object is destroyed, causing the `Person` class's destructor to be called. This displays the output in line (6).
- Line 14 calls the `displayPerson` function, passing the `person` object as an argument.
- In line 24, the `p2` parameter object is created as a copy of the `person` object. This calls the copy constructor, and displays the output in line (7).
- Line 26 displays the output shown in line (8).
- The `displayPerson` function terminates, causing the `p2` parameter to be destroyed. This calls the `Person` class's destructor, displaying the output in line (9).
- In `main`, line 15 ends the program, causing the `person` object to be destroyed. The output in line (10) is displayed by the `Person` class's destructor.

We are especially interested in line 13:

```
person = makePerson();
```

that calls the `makePerson()` function:

```
Person makePerson()
{
    Person p("Will MacKenzie");
    return p;
}
```

The `makePerson()` function creates a local `Person` object named `p`. When the function returns `p`, a temporary object is created as a copy of `p`. This causes the `Person` class's copy constructor to execute. The copy constructor allocates a block of memory to hold the name array, then copies the name array from `p` to the temporary object.

Then in line 13, the overloaded assignment operator executes to copy the temporary object to the `person` object. The overloaded assignment operator also allocates a block of memory, and copies the name array from the temporary object to the `person` object. As you can see, the copy assignment goes to great lengths to avoid pointer sharing between the temporary object and the `person` object, but then right after that, the temporary object is destroyed and its name array is deleted.

11

With C++ 11, we can add another assignment operator function, known as a *move assignment* operator, to the `Person` class. The idea behind the move assignment operator is to avoid all this work by simply swapping the members of the object being assigned with the temporary object. That way, when the temporary object is destroyed, its destructor deletes the memory that was previously owned by `person`, and `person` avoids having to copy the elements that were in the temporary object's name array.

The move assignment operator function for the `Person` class follows. To simplify the code, we have used the STL library function `swap` to swap the contents of two memory locations. The `swap` function is declared in the `<algorithm>` header file (you will learn more about the functions in the `<algorithm>` header file in Chapter 17).

```
// Move assignment operator
Person& operator=(Person&& right)
{
    if (this != &right)
    {
        swap(name, right.name);
    }
    return *this;
}
```

Compare the class's move assignment operator to the class's copy assignment operator:

```
// Copy assignment operator
Person & operator=(const Person &right)
{
    if (this != &right)
    {
        name = new char[strlen(right.name) + 1];
        strcpy(name, right.name);
    }
    return *this;
}
```

Notice the function header for the move assignment operator is different from the function header for the copy assignment operator in two ways:

- The move assignment operator takes an rvalue reference as parameter. This is because a move assignment should only be performed when the source of the assignment is a temporary object.
- The parameter in the move assignment operator is not `const`. This is because “moving” a resource from an object modifies it.

The move assignment, introduced in C++ 11, is much more efficient than copy assignment, and it should be used whenever the object being assigned from is temporary. There is also a *move constructor*, which should be used when creating a new object by initialization from a temporary object.

If you add a move assignment operator to a class, you should also add a *move constructor* to the class. A move constructor avoids unnecessary copying of data by “stealing” it from the temporary object. Here is the move constructor for the `Person` class. Again, note the parameter to the constructor is an rvalue reference, denoting that the parameter is a temporary object.

```
// Move constructor
Person(Person&& temp)
{ // Steal the name pointer from temp.
    name = temp.name;
    // Nullify the temp object's name pointer.
    temp.name = nullptr;
}
```

Note the parameter in the move constructor cannot be `const`. In addition, the temporary object that is the source for the move constructor must be left in a state that will allow its destructor to run without causing errors. That is why, in the `Person` class’s move constructor, we set the temporary object’s name pointer to `nullptr` before the function ends.

When to Implement Move Semantics in a Class

Anytime you write a class that contains a pointer or a reference to an outside piece of data, you should include a copy constructor, a move constructor, a copy assignment operator, and a move assignment operator. By adding move operations to such a class, you will ensure the best performance for your code.

At runtime, the move constructor and move assignment operator are automatically called when appropriate. Specifically, the compiler uses a move operation whenever an object is being assigned the value of a temporary object, or initialized with a copy of a temporary object.

Default Operations Provided by the Compiler

The motivation for writing copy constructors, move constructors, copy assignment operators, move assignment operators, and destructors is to make sure a class properly manages and disposes of resources, such as dynamically allocated memory. Writing all of these operations for each class can be tedious, so the compiler tries to help out

by generating default implementations for them. Let's assume we compile the following class:

```
class MyClass
{
private:
    int mydata = 0;
public:
    int getMyData()
    { return mydata; }
    void setMyData(int d)
    { mydata = d; }
};
```

The compiler will automatically generate the following:

- A default constructor `MyClass()`
- A copy constructor `MyClass(const MyClass &)`
- A copy assignment operator `MyClass & operator=(const MyClass &)`
- A move constructor `MyClass(MyClass &&)`
- A destructor `~MyClass()`

Keep in mind if you write your own version of *any* of these member functions, the compiler will not provide any of the default versions. So, if you write one of these member functions, you should write all of the others as well. For example, if you write a copy constructor for a class, then you should also write a move constructor, a copy assignment operator, a move assignment operator, and a destructor.

Review Questions and Exercises

Short Answer

1. Describe the difference between an instance member variable and a static member variable.
2. Assume a class named `Numbers` has the following static member function declaration:

```
static void showTotal();
```

Write a statement that calls the `showTotal` function.

3. A static member variable is declared in a class. Where is the static member variable defined?
4. What is a friend function?
5. Why is it not always a good idea to make an entire class a friend of another class?
6. What is memberwise assignment?
7. When is a copy constructor called?
8. How can the compiler determine if a constructor is a copy constructor?
9. Describe a situation where memberwise assignment is not desirable.
10. Why must the parameter of a copy constructor be a reference?
11. What is a default copy constructor?
12. Why would a programmer want to overload operators rather than use regular member functions to perform similar operations?
13. What is passed to the parameter of a class's `operator=` function?

14. Why shouldn't a class's overloaded = operator be implemented with a void operator function?
15. How does the compiler know whether an overloaded ++ operator should be used in prefix or postfix mode?
16. What is the this pointer?
17. What type of value should be returned from an overloaded relational operator function?
18. The class Stuff has both a copy constructor and an overloaded = operator. Assume blob and clump are both instances of the Stuff class. For each statement below, indicate whether the copy constructor or the overloaded = operator will be called:

```
Stuff blob = clump;
clump = blob;
blob.operator=(clump);
showValues(blob); // blob is passed by value.
```

19. Explain the programming steps necessary to make a class's member variable static.
20. Explain the programming steps necessary to make a class's member function static.
21. Consider the following class declaration:

```
class Thing
{
private:
    int x;
    int y;
    static int z;
public:
    Thing()
    { x = y = z; }
    static void putThing(int a)
    { z = a; }
};
```

Assume a program containing the class declaration defines three Thing objects with the following statement:

```
Thing one, two, three;
```

How many separate instances of the x member exist?

How many separate instances of the y member exist?

How many separate instances of the z member exist?

What value will be stored in the x and y members of each object?

Write a statement that will call the PutThing member function *before* the objects above are defined.

22. Describe the difference between making a class a member of another class (object aggregation), and making a class a friend of another class.
23. What is the purpose of a forward declaration of a class?
24. Explain why memberwise assignment can cause problems with a class that contains a pointer member.
25. Why is a class's copy constructor called when an object of that class is passed by value into a function?

Fill-in-the-Blank

26. If a member variable is declared _____, all objects of that class have access to the same variable.
27. Static member variables are defined _____ the class.
28. A(n) _____ member function cannot access any nonstatic member variables in its own class.
29. A static member function may be called _____ any instances of its class are defined.
30. A(n) _____ function is not a member of a class, but has access to the private members of the class.
31. A(n) _____ tells the compiler that a specific class will be declared later in the program.
32. _____ is the default behavior when an object is assigned the value of another object of the same class.
33. A(n) _____ is a special constructor, called whenever a new object is initialized with another object's data.
34. _____ is a special built-in pointer that is automatically passed as a hidden argument to all nonstatic member functions.
35. An operator may be _____ to work with a specific class.
36. When overloading the _____ operator, its function must have a dummy parameter.
37. Making an instance of one class a member of another class is called _____.
38. Object aggregation is useful for creating a(n) _____ relationship between two classes.

Algorithm Workbench

39. Assume a class named `Bird` exists. Write the header for a member function that overloads the `=` operator for that class.
40. Assume a class named `Dollars` exists. Write the headers for member functions that overload the prefix and postfix `++` operators for that class.
41. Assume a class named `Yen` exists. Write the header for a member function that overloads the `<` operator for that class.
42. Assume a class named `Length` exists. Write the header for a member function that overloads `cout`'s `<<` operator for that class.
43. Assume a class named `Collection` exists. Write the header for a member function that overloads the `[]` operator for that class.

True or False

44. T F Static member variables cannot be accessed by nonstatic member functions.
45. T F Static member variables are defined outside their class declaration.
46. T F A static member function may refer to nonstatic member variables of the same class, but only after an instance of the class has been defined.
47. T F When a function is declared a `friend` by a class, it becomes a member of that class.
48. T F A `friend` function has access to the private members of the class declaring it a `friend`.
49. T F An entire class may be declared a `friend` of another class.

50. T F In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access.
51. T F If a class has a pointer as a member, it's a good idea to also have a copy constructor.
52. T F You cannot use the = operator to assign one object's values to another object, unless you overload the operator.
53. T F If a class doesn't have a copy constructor, the compiler generates a default copy constructor for it.
54. T F If a class has a copy constructor, and an object of that class is passed by value into a function, the function's parameter will *not* call its copy constructor.
55. T F The this pointer is passed to static member functions.
56. T F All functions that overload unary operators must have a dummy parameter.
57. T F For an object to perform automatic type conversion, an operator function must be written.
58. T F It is possible to have an instance of one class as a member of another class.

Find the Errors

Each of the following class declarations has errors. Locate as many as you can.

```

59. class Box
{
    private:
        double width;
        double length;
        double height;
    public:
        Box(double w, l, h)
            { width = w; length = l; height = h; }
        Box(Box b) // Copy constructor
            { width = b.width;
              length = b.length;
              height = b.height; }

        ... Other member functions follow ...
};

60. class Circle
{
    private:
        double diameter;
        int centerX;
        int centerY;
    public:
        Circle(double d, int x, int y)
            { diameter = d; centerX = x; centerY = y; }
        // Overloaded = operator
        void Circle=(Circle &right)
            { diameter = right.diameter;
              centerX = right.centerX;
              centerY = right.centerY; }

        ... Other member functions follow ...
};

```

```
61. class Point
{
    private:
        int xCoord;
        int yCoord;
    public:
        Point (int x, int y)
        { xCoord = x; yCoord = y; }
        // Overloaded + operator
        void operator+(const &Point right)
        { xCoord += right.xCoord;
          yCoord += right.yCoord;
        }
        ... Other member functions follow ...
};

62. class Box
{
    private:
        double width;
        double length;
        double height;
    public:
        Box(double w, l, h)
        { width = w; length = l; height = h; }
        // Overloaded prefix ++ operator
        void operator++()
        { ++width; ++length; }
        // Overloaded postfix ++ operator
        void operator++()
        { width++; length++; }
        ... Other member functions follow ...
};

63. class Yard
{
    private:
        float length;
    public:
        yard(float l)
        { length = l; }
        // float conversion function
        void operator float()
        { return length; }
        ... Other member functions follow ...
};
```

Programming Challenges

1. Numbers Class

Design a class **Numbers** that can be used to translate whole dollar amounts in the range 0 through 9999 into an English description of the number. For example, the number 713 would be translated into the string *seven hundred thirteen*, and 8203 would be translated into *eight thousand two hundred three*. The class should have a single integer member variable:

```
int number;
```

and a static array of **string** objects that specify how to translate key dollar amounts into the desired format. For example, you might use static strings such as

```
string lessThan20[20] = {"zero", "one", ..., "eighteen", "nineteen"};
string hundred = "hundred";
string thousand = "thousand";
```

The class should have a constructor that accepts a nonnegative integer and uses it to initialize the **Numbers** object. It should have a member function **print()** that prints the English description of the **Numbers** object. Demonstrate the class by writing a main program that asks the user to enter a number in the proper range then prints out its English description.

2. Day of the Year

Assuming a year has 365 days, write a class named **DayOfYear** that takes an integer representing a day of the year and translates it to a string consisting of the month followed by day of the month. For example,

Day 2 would be *January 2*.

Day 32 would be *February 1*.

Day 365 would be *December 31*.

The constructor for the class should take as parameter an integer representing the day of the year, and the class should have a member function **print()** that prints the day in the month-day format. The class should have an integer member variable to represent the day, and should have static member variables holding **string** objects that can be used to assist in the translation from the integer format to the month-day format.

Test your class by inputting various integers representing days and printing out their representation in the month-day format.

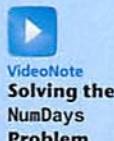
3. Day of the Year Modification

Modify the **DayOfYear** class, written in Programming Challenge 2, to add a constructor that takes two parameters: a **string** object representing a month and an integer in the range 0 through 31 representing the day of the month. The constructor should then initialize the integer member of the class to represent the day specified by the month and day of month parameters. The constructor should terminate the program with an appropriate error message if the number entered for a day is outside the range of days for the month given.

Add the following overloaded operators:

- ++ **prefix and postfix increment operators.** These operators should modify the `DayOfYear` object so it represents the next day. If the day is already the end of the year, the new value of the object will represent the first day of the year.
- **prefix and postfix decrement operators.** These operators should modify the `DayOfYear` object so it represents the previous day. If the day is already the first day of the year, the new value of the object will represent the last day of the year.

4. NumDays Class



VideoNote
Solving the
NumDays
Problem

Design a class called `NumDays`. The class's purpose is to store a value that represents a number of work hours and convert it to a number of days. For example, 8 hours would be converted to 1 day, 12 hours would be converted to 1.5 days, and 18 hours would be converted to 2.25 days. The class should have a constructor that accepts a number of hours, as well as member functions for storing and retrieving the hours and days. The class should also have the following overloaded operators:

- + **Addition operator.** When two `NumDays` objects are added together, the overloaded + operator should return the sum of the two objects' hours members.
- **Subtraction operator.** When one `NumDays` object is subtracted from another, the overloaded - operator should return the difference of the two objects' hours members.
- ++ **Prefix and postfix increment operators.** These operators should increment the number of hours stored in the object. When incremented, the number of days should be automatically recalculated.
- **Prefix and postfix decrement operators.** These operators should decrement the number of hours stored in the object. When decremented, the number of days should be automatically recalculated.

5. Time Off



NOTE: This assignment assumes you have already completed Programming Challenge 4.

Design a class named `TimeOff`. The purpose of the class is to track an employee's sick leave, vacation, and unpaid time off. It should have, as members, the following instances of the `NumDays` class described in Programming Challenge 4:

<code>maxSickDays</code>	A <code>NumDays</code> object that records the maximum number of days of sick leave the employee may take.
<code>sickTaken</code>	A <code>NumDays</code> object that records the number of days of sick leave the employee has already taken.
<code>maxVacation</code>	A <code>NumDays</code> object that records the maximum number of days of paid vacation the employee may take.
<code>vacTaken</code>	A <code>NumDays</code> object that records the number of days of paid vacation the employee has already taken.

maxUnpaid A `NumDays` object that records the maximum number of days of unpaid vacation the employee may take.

unpaidTaken A `NumDays` object that records the number of days of unpaid leave the employee has taken.

Additionally, the class should have members for holding the employee's name and identification number. It should have an appropriate constructor and member functions for storing and retrieving data in any of the member objects.

Input Validation: Company policy states that an employee may not accumulate more than 240 hours of paid vacation. The class should not allow the `maxVacation` object to store a value greater than this amount.

6. Personnel Report



NOTE: This assignment assumes you have already completed Programming Challenges 4 and 5.

Write a program that uses an instance of the `TimeOff` class you designed in Programming Challenge 5. The program should ask the user to enter the number of months an employee has worked for the company. It should then use the `TimeOff` object to calculate and display the employee's maximum number of sick leave and vacation days. Employees earn 12 hours of vacation leave and 8 hours of sick leave per month.

7. Month Class

Design a class named `Month`. The class should have the following private members:

name A string object that holds the name of a month, such as "January," "February," and so on.

monthNumber An integer variable that holds the number of the month. For example, January would be 1, February would be 2, and so on. Valid values for this variable are 1 through 12.

In addition, provide the following member functions:

- A default constructor that sets `monthNumber` to 1 and `name` to "January."
- A constructor that accepts the name of the month as an argument. It should set `name` to the value passed as the argument and set `monthNumber` to the correct value.
- A constructor that accepts the number of the month as an argument. It should set `monthNumber` to the value passed as the argument and set `name` to the correct month name.
- Appropriate set and get functions for the `name` and `monthNumber` member variables.
- Prefix and postfix overloaded `++` operator functions that increment `monthNumber` and set `name` to the name of next month. If `monthNumber` is set to 12 when these functions execute, they should set `monthNumber` to 1 and `name` to "January."
- Prefix and postfix overloaded `--` operator functions that decrement `monthNumber` and set `name` to the name of previous month. If `monthNumber` is set to 1 when these functions execute, they should set `monthNumber` to 12 and `name` to "December."

Also, you should overload `cout`'s `<<` operator and `cin`'s `>>` operator to work with the `Month` class. Demonstrate the class in a program.

8. Date Class Modification

Modify the Date class in Programming Challenge 1 of Chapter 13. The new version should have the following overloaded operators:

- ++ Prefix and postfix increment operators.** These operators should increment the object's day member.
- Prefix and postfix decrement operators.** These operators should decrement the object's day member.
- Subtraction operator.** If one Date object is subtracted from another, the operator should give the number of days between the two dates. For example, if April 10, 2014 is subtracted from April 18, 2014, the result will be 8.
- << cout's stream insertion operator.** This operator should cause the date to be displayed in the form
April 18, 2018
- >> cin's stream extraction operator.** This operator should prompt the user for a date to be stored in a Date object.

The class should detect the following conditions and handle them accordingly:

- When a date is set to the last day of the month and incremented, it should become the first day of the following month.
- When a date is set to December 31 and incremented, it should become January 1 of the following year.
- When a day is set to the first day of the month and decremented, it should become the last day of the previous month.
- When a date is set to January 1 and decremented, it should become December 31 of the previous year.

Demonstrate the class's capabilities in a simple program.

Input Validation: The overloaded >> operator should not accept invalid dates. For example, the date 13/45/2018 should not be accepted.

9. FeetInches Modification

Modify the FeetInches class discussed in this chapter so it overloads the following operators:

<=
>=
!=

Demonstrate the class's capabilities in a simple program.

10. Corporate Sales

A corporation has six divisions, each responsible for sales to different geographic locations. Design a DivSales class that keeps sales data for a division, with the following members:

- An array with four elements for holding four quarters of sales figures for the division.
- A private static variable for holding the total corporate sales for all divisions for the entire year.

- A member function that takes four arguments, each assumed to be the sales for a quarter. The value of the arguments should be copied into the array that holds the sales data. The total of the four arguments should be added to the static variable that holds the total yearly corporate sales.
- A function that takes an integer argument within the range of 0–3. The argument is to be used as a subscript into the division quarterly sales array. The function should return the value of the array element with that subscript.

Write a program that creates an array of six `DivSales` objects. The program should ask the user to enter the sales for four quarters for each division. After the data are entered, the program should display a table showing the division sales for each quarter. The program should then display the total corporate sales for the year.

Input Validation: Only accept positive values for quarterly sales figures.

11. `FeetInches` Class Copy Constructor and `multiply` Function

Add a copy constructor to the `FeetInches` class. This constructor should accept a `FeetInches` object as an argument. The constructor should assign to the `feet` attribute the value in the argument's `feet` attribute, and assign to the `inches` attribute the value in the argument's `inches` attribute. As a result, the new object will be a copy of the argument object.

Next, add a `multiply` member function to the `FeetInches` class. The `multiply` function should accept a `FeetInches` object as an argument. The argument object's `feet` and `inches` attributes will be multiplied by the calling object's `feet` and `inches` attributes, and a `FeetInches` object containing the result will be returned.

12. `LandTract` Class

Make a `LandTract` class that is composed of two `FeetInches` objects: one for the tract's length, and one for the width. The class should have a member function that returns the tract's area. Demonstrate the class in a program that asks the user to enter the dimensions for two tracts of land. The program should display the area of each tract of land, and indicate whether the tracts are of equal size.

13. Carpet Calculator

The Westfield Carpet Company has asked you to write an application that calculates the price of carpeting for rectangular rooms. To calculate the price, you multiply the area of the floor (width times length) by the price per square foot of carpet. For example, the area of floor that is 12 feet long and 10 feet wide is 120 square feet. To cover that floor with carpet that costs \$8 per square foot would cost \$960. ($12 \times 10 \times 8 = 960$.)

First, you should create a class named `RoomDimension` that has two `FeetInches` objects as attributes: one for the length of the room, and one for the width. (You should use the version of the `FeetInches` class you created in Programming Challenge 11 with the addition of a `multiply` member function. You can use this function to calculate the area of the room.) The `RoomDimension` class should have a member function that returns the area of the room as a `FeetInches` object.

Next, you should create a `RoomCarpet` class that has a `RoomDimension` object as an attribute. It should also have an attribute for the cost of the carpet per square foot. The `RoomCarpet` class should have a member function that returns the total cost of the carpet.

Once you have written these classes, use them in an application that asks the user to enter the dimensions of a room and the price per square foot of the desired carpeting. The application should display the total cost of the carpet.

14. Parking Ticket Simulator

For this assignment, you will design a set of classes that work together to simulate a police officer issuing a parking ticket. The classes you should design are:

- **The ParkedCar Class:** This class should simulate a parked car. The class's responsibilities are:
 - To know the car's make, model, color, license number, and the number of minutes that the car has been parked
- **The ParkingMeter Class:** This class should simulate a parking meter. The class's only responsibility is:
 - To know the number of minutes of parking time that has been purchased
- **The ParkingTicket Class:** This class should simulate a parking ticket. The class's responsibilities are:
 - To report the make, model, color, and license number of the illegally parked car
 - To report the amount of the fine, which is \$25 for the first hour or part of an hour that the car is illegally parked, plus \$10 for every additional hour or part of an hour that the car is illegally parked
 - To report the name and badge number of the police officer issuing the ticket
- **The PoliceOfficer Class:** This class should simulate a police officer inspecting parked cars. The class's responsibilities are:
 - To know the police officer's name and badge number
 - To examine a **ParkedCar** object and a **ParkingMeter** object, and determine whether the car's time has expired
 - To issue a parking ticket (generate a **ParkingTicket** object) if the car's time has expired

Write a program that demonstrates how these classes collaborate.

15. Car Instrument Simulator

For this assignment you will design a set of classes that work together to simulate a car's fuel gauge and odometer. The classes you will design are:

- **The FuelGauge Class:** This class will simulate a fuel gauge. Its responsibilities are:
 - To know the car's current amount of fuel, in gallons.
 - To report the car's current amount of fuel, in gallons.
 - To be able to increment the amount of fuel by 1 gallon. This simulates putting fuel in the car. (The car can hold a maximum of 15 gallons.)
 - To be able to decrement the amount of fuel by 1 gallon, if the amount of fuel is greater than 0 gallons. This simulates burning fuel as the car runs.
- **The Odometer Class:** This class will simulate the car's odometer. Its responsibilities are:
 - To know the car's current mileage.
 - To report the car's current mileage.

- To be able to increment the current mileage by 1 mile. The maximum mileage the odometer can store is 999,999 miles. When this amount is exceeded, the odometer resets the current mileage to 0.
- To be able to work with a FuelGauge object. It should decrease the FuelGauge object's current amount of fuel by 1 gallon for every 24 miles traveled. (The car's fuel economy is 24 miles per gallon.)

Demonstrate the classes by creating instances of each. Simulate filling the car up with fuel, then run a loop that increments the odometer until the car runs out of fuel. During each loop iteration, print the car's current mileage and amount of fuel.

TOPICS

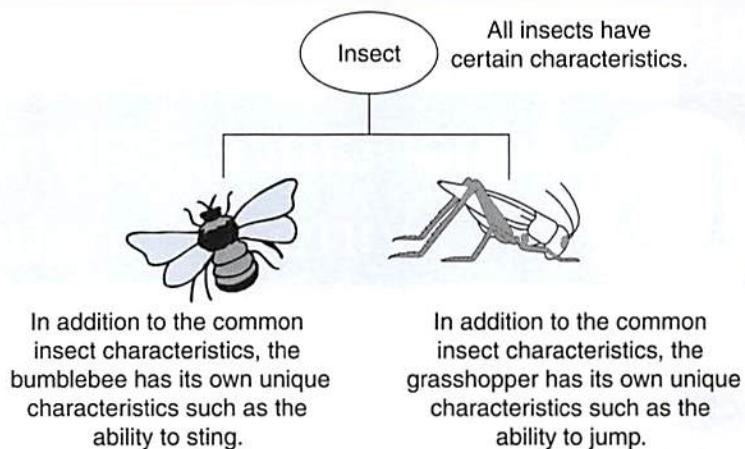
- | | | | |
|------|--|------|--|
| 15.1 | What Is Inheritance? | 15.5 | Class Hierarchies |
| 15.2 | Protected Members and Class Access | 15.6 | Polymorphism and Virtual Member Functions |
| 15.3 | Constructors and Destructors in Base and Derived Classes | 15.7 | Abstract Base Classes and Pure Virtual Functions |
| 15.4 | Redefining Base Class Functions | 15.8 | Multiple Inheritance |

15.1**What Is Inheritance?**

CONCEPT: Inheritance allows a new class to be based on an existing class. The new class automatically inherits all the member variables and functions, except the constructors and destructor, of the class it is based on. (In C++ 11, a class can optionally inherit some of the constructors of the class it is based on.)

Generalization and Specialization

In the real-world, you can find many objects that are specialized versions of other more general objects. For example, the term “insect” describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 15-1.

Figure 15-1 Specialized versions of an insect

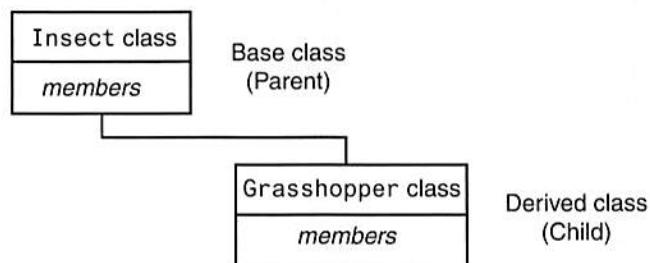
Inheritance and the “Is a” Relationship

When one object is a specialized version of another object, there is an “*is a*” relationship between them. For example, a grasshopper *is an* insect. Here are a few other examples of the “*is a*” relationship:

- A poodle *is a* dog.
- A car *is a* vehicle.
- A tree *is a* plant.
- A rectangle *is a* shape.
- A football player *is an* athlete.

When an “*is a*” relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an “*is a*” relationship between classes.

Inheritance involves a base class and a derived class. The *base class* is the general class and the *derived class* is the specialized class. The derived class is based on, or derived from, the base class. You can think of the base class as the parent, and the derived class as the child. This is illustrated in Figure 15-2.

Figure 15-2 Base class and derived class

The derived class inherits the member variables and member functions of the base class without any of them being rewritten. Furthermore, new member variables and functions may be added to the derived class to make it more specialized than the base class.

Let's look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can receive a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D, or F. The following `GradedActivity` class is designed to hold the numeric score and letter grade of a graded activity. When a numeric score is stored by the class, it automatically determines the letter grade. (These files can be found in the Student Source Code Folder Chapter 15\ `GradedActivity Version 1`.)

Contents of `GradedActivity.h` (Version 1)

```
1 #ifndef GRADEDACTIVITY_H
2 #define GRADEDACTIVITY_H
3
4 // GradedActivity class declaration
5
6 class GradedActivity
7 {
8 private:
9     double score; // To hold the numeric score
10 public:
11     // Default constructor
12     GradedActivity()
13     { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17     { score = s; }
18
19     // Mutator function
20     void setScore(double s)
21     { score = s; }
22
23     // Accessor functions
24     double getScore() const
25     { return score; }
26
27     char getLetterGrade() const;
28 };
29 #endif
```

Contents of `GradedActivity.cpp` (Version 1)

```
1 #include "GradedActivity.h"
2
3 //*****
4 // Member function GradedActivity::getLetterGrade      *
5 //*****
```

```

7  char GradedActivity::getLetterGrade() const
8  {
9      char letterGrade; // To hold the letter grade
10
11     if (score > 89)
12         letterGrade = 'A';
13     else if (score > 79)
14         letterGrade = 'B';
15     else if (score > 69)
16         letterGrade = 'C';
17     else if (score > 59)
18         letterGrade = 'D';
19     else
20         letterGrade = 'F';
21
22     return letterGrade;
23 }

```

The `GradedActivity` class has a default constructor that initializes the `score` member variable to 0.0. A second constructor accepts an argument for `score`. The `setScore` member function also accepts an argument for the `score` variable, and the `getLetterGrade` member function returns the letter grade that corresponds to the value in `score`. Program 15-1 demonstrates the `GradedActivity` class. (This file can also be found in the Student Source Code Folder Chapter 15\GradedActivity Version 1.)

Program 15-1

```

1 // This program demonstrates the GradedActivity class.
2 #include <iostream>
3 #include "GradedActivity.h"
4 using namespace std;
5
6 int main()
7 {
8     double testScore; // To hold a test score
9
10    // Create a GradedActivity object for the test.
11    GradedActivity test;
12
13    // Get a numeric test score from the user.
14    cout << "Enter your numeric test score: ";
15    cin >> testScore;
16
17    // Store the numeric score in the test object.
18    test.setScore(testScore);
19
20    // Display the letter grade for the test.
21    cout << "The grade for that test is "
22        << test.getLetterGrade() << endl;
23
24    return 0;
25 }

```

Program Output with Example Input Shown in BoldEnter your numeric test score: **89**

The grade for that test is B

Program Output with Different Example Input Shown in BoldEnter your numeric test score: **75**

The grade for that test is C

The `GradedActivity` class represents the general characteristics of a student's graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create derived classes to handle each one. For example, the following code shows the `FinalExam` class, which is derived from the `GradedActivity` class. It has member variables for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`). These files can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 1.

Contents of FinalExam.h

```
1 #ifndef FINALEXAM_H
2 #define FINALEXAM_H
3 #include "GradedActivity.h"
4
5 class FinalExam : public GradedActivity
6 {
7 private:
8     int numQuestions; // Number of questions
9     double pointsEach; // Points for each question
10    int numMissed; // Number of questions missed
11 public:
12     // Default constructor
13     FinalExam()
14     { numQuestions = 0;
15         pointsEach = 0.0;
16         numMissed = 0; }
17
18     // Constructor
19     FinalExam(int questions, int missed)
20     { set(questions, missed); }
21
22     // Mutator function
23     void set(int, int); // Defined in FinalExam.cpp
24
25     // Accessor functions
26     double getNumQuestions() const
27     { return numQuestions; }
28
29     double getPointsEach() const
30     { return pointsEach; }
31
```

```

32     int getNumMissed() const
33         { return numMissed; }
34     };
35 #endif

```

Contents of FinalExam.cpp

```

1  #include "FinalExam.h"
2
3 //*****
4 // set function
5 // The parameters are the number of questions and the
6 // number of questions missed.
7 //*****
8
9 void FinalExam::set(int questions, int missed)
10 {
11     double numericScore; // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26 }

```

The only new notation in this code is in line 5 of the FinalExam.h file, which reads

```
class FinalExam : public GradedActivity
```

This line indicates the name of the class being declared and the name of the base class it is derived from. `FinalExam` is the name of the class being declared, and `GradedActivity` is the name of the base class it inherits from.

If we want to express the relationship between the two classes, we can say a `FinalExam` is a `GradedActivity`.

The word `public`, which precedes the name of the base class in line 5 of the FinalExam.h file, is the *base class access specification*. It affects how the members of the base class are inherited by the derived class. When you create an object of a derived class, you can think of it as being built on top of an object of the base class. The members of the base class object become members of the derived class object. How the base class members appear in the derived class is determined by the base class access specification.

Although we will discuss this topic in more detail in the next section, let's see how it works in this example. The `GradedActivity` class has both private members and public members. The `FinalExam` class is derived from the `GradedActivity` class, using public access specification. This means that the public members of the `GradedActivity` class will become public members of the `FinalExam` class. The private members of the `GradedActivity` class cannot be accessed directly by code in the `FinalExam` class. Although the private members of the `GradedActivity` class are inherited, it's as though they are invisible to the code in the `FinalExam` class. They can only be accessed by the member functions of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class:

Private Members:

<code>int numQuestions</code>	Declared in the <code>FinalExam</code> class
<code>double pointsEach</code>	Declared in the <code>FinalExam</code> class
<code>int numMissed</code>	Declared in the <code>FinalExam</code> class

Public Members:

<code>FinalExam()</code>	Defined in the <code>FinalExam</code> class
<code>FinalExam(int, int)</code>	Defined in the <code>FinalExam</code> class
<code>set(int, int)</code>	Defined in the <code>FinalExam</code> class
<code>getNumQuestions()</code>	Defined in the <code>FinalExam</code> class
<code>getPointsEach()</code>	Defined in the <code>FinalExam</code> class
<code>getNumMissed()</code>	Defined in the <code>FinalExam</code> class
<code>setScore(double)</code>	Inherited from <code>GradedActivity</code>
<code>getScore()</code>	Inherited from <code>GradedActivity</code>
<code>getLetterGrade()</code>	Inherited from <code>GradedActivity</code>

The `GradedActivity` class has one private member, the variable `score`. Notice it is not listed as a member of the `FinalExam` class. It is still inherited by the derived class, but because it is a private member of the base class, only member functions of the base class may access it. It is truly private to the base class. Because the functions `setScore`, `getScore`, and `getLetterGrade` are public members of the base class, they also become public members of the derived class.

You will also notice the `GradedActivity` class constructors are not listed among the members of the `FinalExam` class. Although the base class constructors still exist, it makes sense that they are not members of the derived class because their purpose is to construct objects of the base class. In the next section, we discuss in more detail how base class constructors operate.

Let's take a closer look at the `FinalExam` class constructors. The default constructor appears in lines 13 through 16 of the `FinalExam.h` file. It simply assigns 0 to each of the class's member variables. Another constructor appears in lines 19 and 20. This constructor accepts two arguments: one for the number of questions on the exam, and one for the number of questions missed. This constructor merely passes those values as arguments to the `set` function.

The `set` function is defined in `FinalExam.cpp`. It accepts two arguments: the number of questions on the exam, and the number of questions missed by the student. In lines 14 and 15, these values are assigned to the `numQuestions` and `numMissed` member variables. In line 18, the number of points for each question is calculated. In line 21, the numeric test score is calculated. In line 25, the last statement in the function reads:

```
    setScore(numericScore);
```

This is a call to the `setScore` function. Although no `setScore` function appears in the `FinalExam` class, it is inherited from the `GradedActivity` class. Program 15-2 demonstrates the `FinalExam` class.

Program 15-2

```

1 // This program demonstrates a base class and a derived class.
2 #include <iostream>
3 #include <iomanip>
4 #include "FinalExam.h"
5 using namespace std;
6
7 int main()
8 {
9     int questions; // Number of questions on the exam
10    int missed;    // Number of questions missed by the student
11
12    // Get the number of questions on the final exam.
13    cout << "How many questions are on the final exam? ";
14    cin >> questions;
15
16    // Get the number of questions the student missed.
17    cout << "How many questions did the student miss? ";
18    cin >> missed;
19
20    // Define a FinalExam object and initialize it with
21    // the values entered.
22    FinalExam test(questions, missed);
23
24    // Display the test results.
25    cout << setprecision(2);
26    cout << "\nEach question counts " << test.getPointsEach()
27        << " points.\n";
28    cout << "The exam score is " << test.getScore() << endl;
29    cout << "The exam grade is " << test.getLetterGrade() << endl;
30
31    return 0;
32 }
```

Program Output with Example Input Shown in Bold

How many questions are on the final exam? **20**

How many questions did the student miss? **3**

Each question counts 5 points.

The exam score is 85

The exam grade is B

Notice in lines 28 and 29, the public member functions of the `GradedActivity` class may be directly called by the `test` object:

```
cout << "The exam score is " << test.getScore() << endl;
cout << "The exam grade is " << test.getLetterGrade() << endl;
```

The `getScore` and `getLetterGrade` member functions are inherited as public members of the `FinalExam` class, so they may be accessed like any other public member.

Inheritance does not work in reverse. It is not possible for a base class to call a member function of a derived class. For example, the following classes will not compile in a program because the `BadBase` constructor attempts to call a function in its derived class:

```
class BadBase
{
    private:
        int x;
    public:
        BadBase() { x = getVal(); } // Error!
};

class Derived : public BadBase
{
    private:
        int y;
    public:
        Derived(int z) { y = z; }
        int getVal() { return y; }
};
```



Checkpoint

- 15.1 Here is the first line of a class declaration. What is the name of the base class?

```
class Truck : public Vehicle
```

- 15.2 What is the name of the derived class in the following declaration line?

```
class Truck : public Vehicle
```

- 15.3 Suppose a program has the following class declarations:

```
class Shape
{
private:
    double area;
public:
    void setArea(double a)
    { area = a; }

    double getArea()
    { return area; }
};

class Circle : public Shape
{
private:
    double radius;
```

```

public:
    void setRadius(double r)
    { radius = r;
      setArea(3.14 * r * r); }
    double getRadius()
    { return radius; }
};

```

Answer the following questions concerning these classes:

- When an object of the `Circle` class is created, what are its private members?
- When an object of the `Circle` class is created, what are its public members?
- What members of the `Shape` class are not accessible to member functions of the `Circle` class?

15.2

Protected Members and Class Access

CONCEPT: Protected members of a base class are like private members, but they may be accessed by derived classes. The base class access specification determines how private, public, and protected base class members are accessed when they are inherited by the derived classes.

Until now, you have used two access specifications within a class: `private` and `public`. C++ provides a third access specification, `protected`. Protected members of a base class are like private members, except they may be accessed by functions in a derived class. To the rest of the program, however, protected members are inaccessible.

The following code shows a modified version of the `GradedActivity` class declaration. The private member of the class has been made protected. This file can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 2. The implementation file, `GradedActivity.cpp` has not changed, so it is not shown again in this example.

Contents of `GradedActivity.h` (Version 2)

```

1 #ifndef GRADEDACTIVITY_H
2 #define GRADEDACTIVITY_H
3
4 // GradedActivity class declaration
5
6 class GradedActivity
7 {
8 protected:
9     double score; // To hold the numeric score
10 public:
11     // Default constructor
12     GradedActivity()
13     { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17     { score = s; }
18

```

```
19 // Mutator function
20 void setScore(double s)
21     { score = s; }
22
23 // Accessor functions
24 double getScore() const
25     { return score; }
26
27 char getLetterGrade() const;
28 };
29 #endif
```

Now we will look at a modified version of the `FinalExam` class, which is derived from this version of the `GradedActivity` class. This version of the `FinalExam` class has a new member function named `adjustScore`. This function directly accesses the `GradedActivity` class's `score` member variable. If the content of the `score` variable has a fractional part of 0.5 or greater, the function rounds `score` up to the next whole number. The `set` function calls the `adjustScore` function after it calculates the numeric score. (These files are available in the Student Source Code Folder Chapter 15\GradedActivity Version 2.)

Contents of `FinalExam.h` (Version 2)

```
1 #ifndef FINALEXAM_H
2 #define FINALEXAM_H
3 #include "GradedActivity.h"
4
5 class FinalExam : public GradedActivity
6 {
7 private:
8     int numQuestions; // Number of questions
9     double pointsEach; // Points for each question
10    int numMissed; // Number of questions missed
11 public:
12     // Default constructor
13     FinalExam()
14     { numQuestions = 0;
15      pointsEach = 0.0;
16      numMissed = 0; }
17
18     // Constructor
19     FinalExam(int questions, int missed)
20     { set(questions, missed); }
21
22     // Mutator functions
23     void set(int, int); // Defined in FinalExam.cpp
24     void adjustScore(); // Defined in FinalExam.cpp
25
26     // Accessor functions
27     double getNumQuestions() const
28     { return numQuestions; }
29
30     double getPointsEach() const
31     { return pointsEach; }
32
```

```
33     int getNumMissed() const
34         { return numMissed; }
35     };
36 #endif
```

Contents of FinalExam.cpp (Version 2)

```
1  #include "FinalExam.h"
2
3 //*****
4 // set function
5 // The parameters are the number of questions and the *
6 // number of questions missed.
7 //*****
8
9 void FinalExam::set(int questions, int missed)
10 {
11     double numericScore; // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26
27     // Call the adjustScore function to adjust
28     // the score.
29     adjustScore();
30 }
31
32 //*****
33 // Definition of Test::adjustScore. If score is within      *
34 // 0.5 points of the next whole point, it rounds the score up *
35 // and recalculates the letter grade.
36 //*****
37
38 void FinalExam::adjustScore()
39 {
40     double fraction = score - static_cast<int>(score);
41
42     if (fraction >= 0.5)
43     {
44         // Adjust the score variable in the GradedActivity class.
45         score += (1.0 - fraction);
46     }
47 }
```

Program 15-3 demonstrates these versions of the `GradedActivity` and `FinalExam` classes. (This file can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 2.)

Program 15-3

```
1 // This program demonstrates a base class with a
2 // protected member.
3 #include <iostream>
4 #include <iomanip>
5 #include "FinalExam.h"
6 using namespace std;
7
8 int main()
9 {
10     int questions; // Number of questions on the exam
11     int missed;    // Number of questions missed by the student
12
13     // Get the number of questions on the final exam.
14     cout << "How many questions are on the final exam? ";
15     cin >> questions;
16
17     // Get the number of questions the student missed.
18     cout << "How many questions did the student miss? ";
19     cin >> missed;
20
21     // Define a FinalExam object and initialize it with
22     // the values entered.
23     FinalExam test(questions, missed);
24
25     // Display the adjusted test results.
26     cout << setprecision(2) << fixed;
27     cout << "\nEach question counts "
28         << test.getPointsEach() << " points.\n";
29     cout << "The adjusted exam score is "
30         << test.getScore() << endl;
31     cout << "The exam grade is "
32         << test.getLetterGrade() << endl;
33
34     return 0;
35 }
```

Program Output with Example Input Shown in Bold

How many questions are on the final exam? **16**

How many questions did the student miss? **5**

Each question counts 6.25 points.

The adjusted exam score is 69.00

The exam grade is D

The program works as planned. In the example run, the student missed five questions, which are worth 6.25 points each. The unadjusted score would be 68.75. The score was adjusted to 69.

More about Base Class Access Specification

The first line of the `FinalExam` class declaration reads:

```
class FinalExam : public GradedActivity
```

This declaration gives public access specification to the base class. Recall from our earlier discussion that base class access specification affects how inherited base class members are accessed. Be careful not to confuse base class access specification with member access specification. Member access specification determines how members that are *defined* within the class are accessed. Base class access specification determines how *inherited* members are accessed.

When you create an object of a derived class, it inherits the members of the base class. The derived class can have its own private, protected, and public members, but what is the access specification of the inherited members? This is determined by the base class access specification. Table 15-1 summarizes how base class access specification affects the way that base class members are inherited.

Table 15-1 Base Class Access Specification

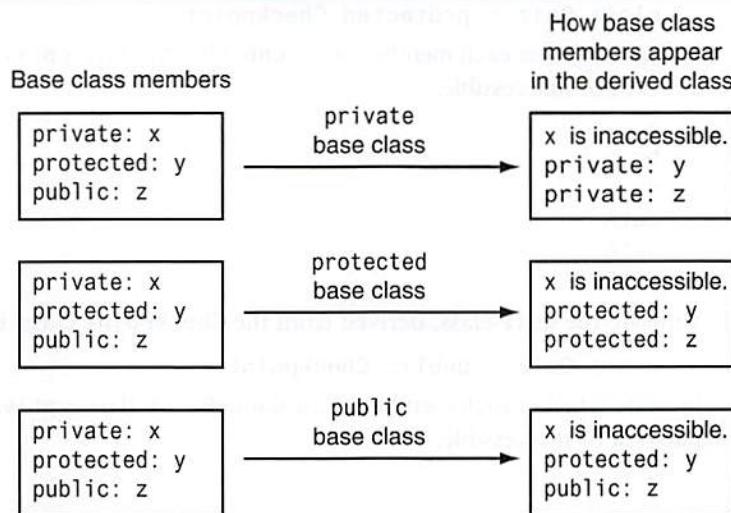
Base Class Access Specification	How Members of the Base Class Appear in the Derived Class
<code>private</code>	Private members of the base class are inaccessible to the derived class. Protected members of the base class become private members of the derived class. Public members of the base class become private members of the derived class.
<code>protected</code>	Private members of the base class are inaccessible to the derived class. Protected members of the base class become protected members of the derived class. Public members of the base class become protected members of the derived class.
<code>public</code>	Private members of the base class are inaccessible to the derived class. Protected members of the base class become protected members of the derived class. Public members of the base class become public members of the derived class.

As you can see from Table 15-1, class access specification gives you a great deal of flexibility in determining how base class members will appear in the derived class. Think of a base class's access specification as a filter that base class members must pass through when becoming inherited members of a derived class. This is illustrated in Figure 15-3.



NOTE: If the base class access specification is left out of a declaration, the default access specification is `private`. For example, in the following declaration, `Grade` is declared as a `private` base class:

```
class Test : Grade
```

Figure 15-3 Base class access specification

Checkpoint

- 15.4 What is the difference between private members and protected members?
- 15.5 What is the difference between member access specification and class access specification?
- 15.6 Suppose a program has the following class declaration:

```
// Declaration of CheckPoint class.
class CheckPoint
{
    private:
        int a;
    protected:
        int b;
        int c;
        void setA(int x) { a = x; }
    public:
        void setB(int y) { b = y; }
        void setC(int z) { c = z; }
};
```

Answer the following questions regarding the class:

- A) Suppose another class, Quiz, is derived from the `CheckPoint` class. Here is the first line of its declaration:

```
class Quiz : private CheckPoint
```

Indicate whether each member of the `CheckPoint` class is private, protected, public, or inaccessible:

a
b
c
setA
setB
setC

- B) Suppose the Quiz class, derived from the CheckPoint class, is declared as

```
class Quiz : protected Checkpoint
```

Indicate whether each member of the CheckPoint class is private, protected, public, or inaccessible:

a
b
c
setA
setB
setC

- C) Suppose the Quiz class, derived from the CheckPoint class, is declared as

```
class Quiz : public Checkpoint
```

Indicate whether each member of the CheckPoint class is private, protected, public, or inaccessible:

a
b
c
setA
setB
setC

- D) Suppose the Quiz class, derived from the CheckPoint class, is declared as

```
class Quiz : Checkpoint
```

Is the CheckPoint class a private, public, or protected base class?

15.3

Constructors and Destructors in Base and Derived Classes

CONCEPT: The base class's constructor is called before the derived class's constructor. The destructors are called in reverse order, with the derived class's destructor being called first.

In inheritance, the base class constructor is called before the derived class constructor. Destructors are called in reverse order. Program 15-4 shows a simple set of demonstration classes, each with a default constructor and a destructor. The `DerivedClass` class is derived from the `BaseClass` class. Messages are displayed by the constructors and destructors to demonstrate when each is called.

Program 15-4

```
1 // This program demonstrates the order in which base and
2 // derived class constructors and destructors are called.
3 #include <iostream>
4 using namespace std;
5
6 //***** BaseClass declaration *****
7 // BaseClass declaration      *
8 //***** BaseClass declaration *****
9
```

```
10 class BaseClass
11 {
12 public:
13     BaseClass()    // Constructor
14     { cout << "This is the BaseClass constructor.\n"; }
15
16     ~BaseClass()   // Destructor
17     { cout << "This is the BaseClass destructor.\n"; }
18 };
19
20 //*****
21 // DerivedClass declaration      *
22 //*****
23
24 class DerivedClass : public BaseClass
25 {
26 public:
27     DerivedClass()    // Constructor
28     { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass()   // Destructor
31     { cout << "This is the DerivedClass destructor.\n"; }
32 };
33
34 //*****
35 // main function                  *
36 //*****
37
38 int main()
39 {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46 }
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

Passing Arguments to Base Class Constructors

In Program 15-4, both the base class and derived class have default constructors, which are called automatically. But what if the base class's constructor takes arguments? What if there is more than one constructor in the base class? The answer to these questions is to let the derived class constructor pass arguments to the base class constructor. For example, consider the following class:

Contents of Rectangle.h

```

1  #ifndef RECTANGLE_H
2  #define RECTANGLE_H
3
4  class Rectangle
5  {
6  private:
7      double width;
8      double length;
9  public:
10     // Default constructor
11     Rectangle()
12     { width = 0.0;
13      length = 0.0; }
14
15     // Constructor #2
16     Rectangle(double w, double len)
17     { width = w;
18      length = len; }
19
20     double getWidth() const
21     { return width; }
22
23     double getLength() const
24     { return length; }
25
26     double getArea() const
27     { return width * length; }
28 };
29 #endif

```

This class is designed to hold data about a rectangle. It specifies two constructors. The default constructor, in lines 11 through 13, simply initializes the `width` and `length` member variables to 0.0. The second constructor, in lines 16 through 18, takes two arguments, which are assigned to the `width` and `length` member variables. Now let's look at a class that is derived from the `Rectangle` class:

Contents of Box.h

```

1  #ifndef BOX_H
2  #define BOX_H
3  #include "Rectangle.h"
4
5  class Box : public Rectangle
6  {
7  protected:
8      double height;
9      double volume;
10 public:
11     // Default constructor
12     Box() : Rectangle()
13     { height = 0.0; volume = 0.0; }
14

```

```

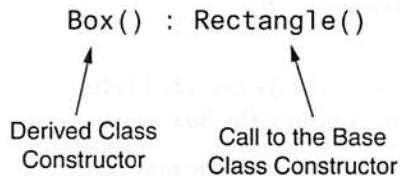
15 // Constructor #2
16 Box(double w, double len, double h) : Rectangle(w, len)
17 { height = h;
18     volume = getArea() * h; }
19
20     double getHeight() const
21     { return height; }
22
23     double getVolume() const
24     { return volume; }
25 };
26 #endif

```

The `Box` class is designed to hold data about boxes, which not only have a length and width, but a height and volume as well. Look at line 12, which is the first line of the `Box` class's default constructor:

```
Box() : Rectangle()
```

Notice the added notation in the header of the constructor. A colon is placed after the derived class constructor's parentheses, followed by a function call to a base class constructor. In this case, the base class's default constructor is being called. When this `Box` class constructor executes, it will first call the `Rectangle` class's default constructor. This is illustrated here:



The general format of this type of constructor declaration is

```
ClassName::ClassName(ParameterList) : BaseClassName(ArgumentList)
```

You can also pass arguments to the base class constructor, as shown in the `Box` class's second constructor. Look at line 16:

```
Box(double w, double len, double h) : Rectangle(w, len)
```

This `Box` class constructor has three parameters: `w`, `len`, and `h`. Notice the `Rectangle` class's constructor is called, and the `w` and `len` parameters are passed as arguments. This causes the `Rectangle` class's second constructor to be called.

You only write this notation in the definition of a constructor, not in a prototype. In this example, the derived class constructor is written inline (inside the class declaration), so the notation that contains the call to the base class constructor appears there. If the constructor were defined outside the class, the notation would appear in the function header. For example, the `Box` class could appear as follows:

```

class Box : public Rectangle
{
protected:
    double height;
    double volume;
public:
    // Default constructor
    Box() : Rectangle()
        { height = 0.0; volume = 0.0; }

    // Constructor #2
    Box(double, double, double);

    double getHeight() const
        { return height; }

    double getVolume() const
        { return volume; }
};

// Box class constructor #2
Box::Box(double w, double len, double h) : Rectangle(w, len)
{
    height = h;
    volume = getArea() * h;
}

```

The base class constructor is always executed before the derived class constructor. When the `Rectangle` constructor finishes, the `Box` constructor is then executed.

Any literal value or variable that is in scope may be used as an argument to the derived class constructor. Usually, one or more of the arguments passed to the derived class constructor are, in turn, passed to the base class constructor. The values that may be used as base class constructor arguments are as follows:

- Derived class constructor parameters
- Literal values
- Global variables that are accessible to the file containing the derived class constructor definition
- Expressions involving any of these items

Program 15-5 shows the `Rectangle` and `Box` classes in use.

Program 15-5

```

1 // This program demonstrates passing arguments to a base
2 // class constructor.
3 #include <iostream>
4 #include "Box.h"
5 using namespace std;
6

```

```
7 int main()
8 {
9     double boxWidth;    // To hold the box's width
10    double boxLength;   // To hold the box's length
11    double boxHeight;   // To hold the box's height
12
13    // Get the width, length, and height from the user.
14    cout << "Enter the dimensions of a box:\n";
15    cout << "Width: ";
16    cin >> boxWidth;
17    cout << "Length: ";
18    cin >> boxLength;
19    cout << "Height: ";
20    cin >> boxHeight;
21
22    // Define a Box object.
23    Box myBox(boxWidth, boxLength, boxHeight);
24
25    // Display the Box object's properties.
26    cout << "Here are the box's properties:\n";
27    cout << "Width: " << myBox.getWidth() << endl;
28    cout << "Length: " << myBox.getLength() << endl;
29    cout << "Height: " << myBox.getHeight() << endl;
30    cout << "Base area: " << myBox.getArea() << endl;
31    cout << "Volume: " << myBox.getVolume() << endl;
32
33 }
```

Program Output with Example Input Shown in Bold

Enter the dimensions of a box:

Width: **10** Enter

Length: **15** Enter

Height: **12** Enter

Here are the box's properties:

Width: 10

Length: 15

Height: 12

Base area: 150

Volume: 1800



NOTE: If the base class has no default constructor, then the derived class must have a constructor that calls one of the base class constructors.

In the Spotlight:

The Automobile, Car, Truck, and SUV Classes

Suppose we are developing a program that a car dealership can use to manage its inventory of used cars. The dealership's inventory includes three types of automobiles: cars, pickup trucks, and sport-utility vehicles (SUVs). Regardless of the type, the dealership keeps the following data about each automobile:

- Make
- Year model
- Mileage
- Price

Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics. For cars, the dealership keeps the following additional data:

- Number of doors (2 or 4)

For pickup trucks, the dealership keeps the following additional data:

- Drive type (two-wheel drive or four-wheel drive)

And, for SUVs, the dealership keeps the following additional data:

- Passenger capacity

In designing this program, one approach would be to write the following three classes:

- A **Car** class with attributes for the make, year model, mileage, price, and number of doors.
- A **Truck** class with attributes for the make, year model, mileage, price, and drive type.
- An **SUV** class with attributes for the make, year model, mileage, price, and passenger capacity.

This would be an inefficient approach, however, because all three classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later we need to add more common attributes, we would have to modify all three classes.

A better approach would be to write an **Automobile** base class to hold all the general data about an automobile, then write derived classes for each specific type of automobile. The following code shows the **Automobile** class. (This file can be found in the Student Source Code Folder Chapter 15\Automobile.)

Contents of Automobile.h

```

1  #ifndef AUTOMOBILE_H
2  #define AUTOMOBILE_H
3  #include <string>
4  using namespace std;
5
6  // The Automobile class holds general data
7  // about an automobile in inventory.
8  class Automobile
9  {

```

```
10 private:
11     string make; // The auto's make
12     int model; // The auto's year model
13     int mileage; // The auto's mileage
14     double price; // The auto's price
15
16 public:
17     // Default constructor
18     Automobile()
19     { make = "";
20         model = 0;
21         mileage = 0;
22         price = 0.0; }
23
24     // Constructor
25     Automobile(string autoMake, int autoModel,
26                 int autoMileage, double autoPrice)
27     { make = autoMake;
28         model = autoModel;
29         mileage = autoMileage;
30         price = autoPrice; }
31
32     // Accessors
33     string getMake() const
34     { return make; }
35
36     int getModel() const
37     { return model; }
38
39     int getMileage() const
40     { return mileage; }
41
42     double getPrice() const
43     { return price; }
44 };
45 #endif
```

Notice the class has a default constructor in lines 18 and 22, and a constructor that accepts arguments for all of the class's attributes in lines 25 through 30. The `Automobile` class is a complete class from which we can create objects. If we wish, we can write a program that creates instances of the `Automobile` class. However, the `Automobile` class holds only general data about an automobile. It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs. To hold data about those specific types of automobiles, we will write derived classes that inherit from the `Automobile` class. The following shows the code for the `Car` class. (This file is available in the Student Source Code Folder Chapter 15\Automobile.)

Contents of Car.h

```
1 #ifndef CAR_H
2 #define CAR_H
3 #include "Automobile.h"
4 #include <string>
5 using namespace std;
```

```

6
7 // The Car class represents a car.
8 class Car : public Automobile
9 {
10 private:
11     int doors;
12
13 public:
14     // Default constructor
15     Car() : Automobile()
16     { doors = 0; }
17
18     // Constructor #2
19     Car(string carMake, int carModel, int carMileage,
20          double carPrice, int carDoors) :
21         Automobile(carMake, carModel, carMileage, carPrice)
22     { doors = carDoors; }
23
24     // Accessor for doors attribute
25     int getDoors()
26     { return doors; }
27 };
28 #endif

```

The `Car` class defines a `doors` attribute in line 11 to hold the car's number of doors. The class has a default constructor in lines 15 and 16 that sets the `doors` attribute to 0. Notice in line 15 the default constructor calls the `Automobile` class's default constructor, which initializes all of the inherited attributes to their default values.

The `Car` class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the car's make, model, mileage, price, and number of doors. Line 21 calls the `Automobile` class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the `doors` attribute.

Now let's look at the `Truck` class, which also inherits from the `Automobile` class. (This file is available in the Student Source Code Folder Chapter 15\Automobile.)

Contents of `Truck.h`

```

1 #ifndef TRUCK_H
2 #define TRUCK_H
3 #include "Automobile.h"
4 #include <string>
5 using namespace std;
6
7 // The Truck class represents a truck.
8 class Truck : public Automobile
9 {
10 private:
11     string driveType;
12
13 public:
14     // Default constructor
15     Truck() : Automobile()
16     { driveType = ""; }
17

```

```
18 // Constructor #2
19 Truck(string truckMake, int truckModel, int truckMileage,
20       double truckPrice, string truckDriveType) :
21     Automobile(truckMake, truckModel, truckMileage, truckPrice)
22 { driveType = truckDriveType; }
23
24 // Accessor for driveType attribute
25 string getDriveType()
26 { return driveType; }
27 };
28 #endif
```

The `Truck` class defines a `driveType` attribute in line 11 to hold a string describing the truck's drive type. The class has a default constructor in lines 15 and 16 that sets the `driveType` attribute to an empty string. Notice in line 15 the default constructor calls the `Automobile` class's default constructor, which initializes all of the inherited attributes to their default values.

The `Truck` class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the truck's make, model, mileage, price, and drive type. Line 21 calls the `Automobile` class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the `driveType` attribute.

Now let's look at the `SUV` class, which also inherits from the `Automobile` class. (This file can be found in the Student Source Code Folder Chapter 15\Automobile.)

Contents of SUV.h

```
1 #ifndef SUV_H
2 #define SUV_H
3 #include "Automobile.h"
4 #include <string>
5 using namespace std;
6
7 // The SUV class represents a SUV.
8 class SUV : public Automobile
9 {
10 private:
11     int passengers;
12
13 public:
14     // Default constructor
15     SUV() : Automobile()
16     { passengers = 0; }
17
18     // Constructor #2
19     SUV(string SUVMake, int SUVModel, int SUVMileage,
20          double SUVPrice, int SUVPassengers) :
21       Automobile(SUVMake, SUVModel, SUVMileage, SUVPrice)
22     { passengers = SUVPassengers; }
23
24     // Accessor for passengers attribute
25     int getPassengers()
26     { return passengers; }
27 };
28 #endif
```

The SUV class defines a passengers attribute in line 11 to hold the number of passengers that the vehicle can accommodate. The class has a default constructor in lines 15 and 16 that sets the passengers attribute to 0. Notice in line 15 the default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The SUV class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the SUV's make, model, mileage, price, and number of passengers. Line 21 calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the passengers attribute.

Program 15-6 demonstrates each of the derived classes. It creates a Car object, a Truck object, and an SUV object. (This file can be found in the Student Source Code Folder Chapter 15\Automobile.)

Program 15-6

```
1 // This program demonstrates the Car, Truck, and SUV
2 // classes that are derived from the Automobile class.
3 #include <iostream>
4 #include <iomanip>
5 #include "Car.h"
6 #include "Truck.h"
7 #include "SUV.h"
8 using namespace std;
9
10 int main()
11 {
12     // Create a Car object for a used 2007 BMW with
13     // 50,000 miles, priced at $15,000, with 4 doors.
14     Car car("BMW", 2007, 50000, 15000.0, 4);
15
16     // Create a Truck object for a used 2006 Toyota
17     // pickup with 40,000 miles, priced at $12,000,
18     // with 4-wheel drive.
19     Truck truck("Toyota", 2006, 40000, 12000.0, "4WD");
20
21     // Create an SUV object for a used 2005 Volvo
22     // with 30,000 miles, priced at $18,000, with
23     // 5 passenger capacity.
24     SUV SUV("Volvo", 2005, 30000, 18000.0, 5);
25
26     // Display the automobiles we have in inventory.
27     cout << fixed << showpoint << setprecision(2);
28     cout << "We have the following car in inventory:\n"
29         << car.getModel() << " " << car.getMake()
30         << " with " << car.getDoors() << " doors and "
31         << car.getMileage() << " miles.\nPrice: $"
32         << car.getPrice() << endl << endl;
33
```

```

34     cout << "We have the following truck in inventory:\n"
35         << truck.getModel() << " " << truck.getMake()
36         << " with " << truck.getDriveType()
37         << " drive type and " << truck.getMileage()
38         << " miles.\nPrice: $" << truck.getPrice()
39         << endl << endl;
40
41     cout << "We have the following SUV in inventory:\n"
42         << suv.getModel() << " " << suv.getMake()
43         << " with " << suv.getMileage() << " miles and "
44         << suv.getPassengers() << " passenger capacity.\n"
45         << "Price: $" << suv.getPrice() << endl;
46
47     return 0;
48 }
```

Program Output

We have the following car in inventory:

2007 BMW with 4 doors and 50000 miles.

Price: \$15000.00

We have the following truck in inventory:

2006 Toyota with 4WD drive type and 40000 miles.

Price: \$12000.00

We have the following SUV in inventory:

2005 Volvo with 30000 miles and 5 passenger capacity.

Price: \$18000.00

Constructor Inheritance

11

C++ 11 provides a way for a derived class to inherit some of the base class's constructors. The constructors that cannot be inherited are the default constructor, the copy constructor, and the move constructor. Any other constructors in the base class can be inherited.

Constructor inheritance can be helpful in situations where the derived class's constructors simply invoke the base class's constructors. For example, look at the following classes:

```

class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase(int i)
    { ival = i; }

    MyBase(double d)
    { dval = d; }
};
```

```

class MyDerived : MyBase
{
public:
    MyDerived(int i) : MyBase(i)
    {}

    MyDerived(double d) : MyBase(d)
    {}

};

```

In this code, the `MyDerived(int i)` constructor simply calls the `MyBase(int i)` constructor, and the `MyDerived(double d)` constructor simply calls the `MyBase(double d)` constructor. In C++ 11, we rewrite the `MyDerived` class as follows, and achieve the same result:

```

class MyDerived : MyBase
{
    using MyBase::MyBase;
};

```

The `using` statement causes the `MyDerived` class to inherit the `MyBase` class's constructors. As a result, we can invoke the `MyBase` class's constructors when we instantiate the `MyDerived` class.

```

MyDerived d1(22); // Calls the MyBase(int i) constructor
MyDerived d2(3.14); // Calls the MyBase(double d) constructor

```

You can write the `using` statement anywhere in the derived class. The general format of the `using` statement in a derived class is

```
using BaseClassName::BaseClassName;
```

A derived class can have its own constructors, and inherit constructors from the base class. Here is an example:

```

class Derived : Base
{
private:
    string str;
public:
    using Base::Base;
    Derived(string s) : Base(0)
    { str = s; }
};

```

Keep in mind, however, if a derived class constructor has the same parameter list as a base class constructor, the base class constructor will not be inherited.



Checkpoint

15.7 What will the following program display?

```
#include <iostream>
using namespace std;
```

```

class Sky
{
public:
    Sky()
        { cout << "Entering the sky.\n"; }
    ~Sky()
        { cout << "Leaving the sky.\n"; }
};

class Ground : public Sky
{
public:
    Ground()
        { cout << "Entering the Ground.\n"; }
    ~Ground()
        { cout << "Leaving the Ground.\n"; }
};

int main()
{
    Ground object;
    return 0;
}

```

15.8 What will the following program display?

```

#include <iostream>
using namespace std;

class Sky
{
public:
    Sky()
        { cout << "Entering the sky.\n"; }
    Sky(string color)
        { cout << "The sky is " << color << endl; }
    ~Sky()
        { cout << "Leaving the sky.\n"; }
};

class Ground : public Sky
{
public:
    Ground()
        { cout << "Entering the Ground.\n"; }
    Ground(string c1, string c2) : Sky(c1)
        { cout << "The ground is " << c2 << endl; }
    ~Ground()
        { cout << "Leaving the Ground.\n"; }
};

int main()
{
    Ground object;
    return 0;
}

```

15.4

Redefining Base Class Functions



CONCEPT: A base class member function may be redefined in a derived class.

Inheritance is commonly used to extend a class, or to give it additional capabilities. Sometimes it may be helpful to overload a base class function with a function of the same name in the derived class. For example, recall the `GradedActivity` class that was presented earlier in this chapter:

```
class GradedActivity
{
protected:
    char letter;           // To hold the letter grade
    double score;          // To hold the numeric score
    void determineGrade(); // Determines the letter grade

public:
    // Default constructor
    GradedActivity()
        { letter = ' '; score = 0.0; }

    // Mutator function
    void setScore(double s)
        { score = s;
            determineGrade(); }

    // Accessor functions
    double getScore() const
        { return score; }

    char getLetterGrade() const
        { return letter; }
};
```

This class holds a numeric score and determines a letter grade based on that score. The `setScore` member function stores a value in `score`, then calls the `determineGrade` member function to determine the letter grade.

Suppose a teacher wants to “curve” a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class, she must multiply each student’s score by a certain percentage. This gives an adjusted score, which is used to determine the letter grade.

The following `CurvedActivity` class is derived from the `GradedActivity` class. It multiplies the numeric score by a percentage, and passes that value as an argument to the base class’s `setScore` function. (This file can be found in the Student Source Code Folder Chapter 15\CurvedActivity.)

Contents of CurvedActivity.h

```
1 #ifndef CURVEDACTIVITY_H
2 #define CURVEDACTIVITY_H
3 #include "GradedActivity.h"
4
5 class CurvedActivity : public GradedActivity
6 {
7 protected:
8     double rawScore;      // Unadjusted score
9     double percentage;   // Curve percentage
10 public:
11     // Default constructor
12     CurvedActivity() : GradedActivity()
13         { rawScore = 0.0; percentage = 0.0; }
14
15     // Mutator functions
16     void setScore(double s)
17         { rawScore = s;
18             GradedActivity::setScore(rawScore * percentage); }
19
20     void setPercentage(double c)
21         { percentage = c; }
22
23     // Accessor functions
24     double getPercentage() const
25         { return percentage; }
26
27     double getRawScore() const
28         { return rawScore; }
29 };
30 #endif
```

This CurvedActivity class has the following member variables:

- `rawScore` This variable holds the student's unadjusted score.
- `percentage` This variable holds the value that the unadjusted score must be multiplied by to get the curved score.

It also has the following member functions:

- A default constructor that calls the `GradedActivity` default constructor, then sets `rawScore` and `percentage` to 0.0.
- `setScore` This function accepts an argument that is the student's unadjusted score. The function stores the argument in the `rawScore` variable, then passes `rawScore * percentage` as an argument to the base class's `setScore` function.
- `setPercentage` This function stores a value in the `percentage` variable.
- `getPercentage` This function returns the value in the `percentage` variable.
- `getRawScore` This function returns the value in the `rawScore` variable.



NOTE: Although we are not using the `CurvedActivity` class as a base class, it still has a protected member section. This is because we might want to use the `CurvedActivity` class itself as a base class, as you will see in the next section.

Notice the `CurvedActivity` class has a `setScore` member function. This function has the same name as one of the base class member functions. When a derived class's member function has the same name as a base class member function, it is said the derived class function *redefines* the base class function. When an object of the derived class calls the function, it calls the derived class's version of the function.

There is a distinction between redefining a function and overloading a function. An overloaded function is one with the same name as one or more other functions, but with a different parameter list. The compiler uses the arguments passed to the function to tell which version to call. Overloading can take place with regular functions that are not members of a class. Overloading can also take place inside a class when two or more member functions of *the same class* have the same name. These member functions must have different parameter lists for the compiler to tell them apart in function calls.

Redefining happens when a derived class has a function with the same name as a base class function. The parameter lists of the two functions can be the same because the derived class function is always called by objects of the derived class type.

Let's continue our look at the `CurvedActivity` class. Here is the `setScore` member function:

```
void setScore(double s)
{
    rawScore = s;
    GradedActivity::setScore(rawScore * percentage); }
```

This function accepts an argument that should be the student's unadjusted numeric score, into the parameter `s`. This value is stored in the `rawScore` variable. Then, the following statement is executed:

```
GradedActivity::setScore(rawScore * percentage);
```

This statement calls the base class's version of the `setScore` function with the expression `rawScore * percentage` passed as an argument. Notice the name of the base class and the scope resolution operator precede the name of the function. This specifies that the base class's version of the `setScore` function is being called. A derived class function may call a base class function of the same name using this notation, which takes this form:

```
BaseClassName::functionName(ArgumentList);
```

Program 15-7 shows the `GradedActivity` and `CurvedActivity` classes used in a complete program. (This file is available in the Student Source Code Folder Chapter 15\CurvedActivity.)

Program 15-7

```
1 // This program demonstrates a class that redefines
2 // a base class function.
3 #include <iostream>
4 #include <iomanip>
5 #include "CurvedActivity.h"
6 using namespace std;
7
8 int main()
9 {
10    double numericScore; // To hold the numeric score
11    double percentage; // To hold curve percentage
12
13    // Define a CurvedActivity object.
14    CurvedActivity exam;
15
16    // Get the unadjusted score.
17    cout << "Enter the student's raw numeric score: ";
18    cin >> numericScore;
19
20    // Get the curve percentage.
21    cout << "Enter the curve percentage for this student: ";
22    cin >> percentage;
23
24    // Send the values to the exam object.
25    exam.setPercentage(percentage);
26    exam.setScore(numericScore);
27
28    // Display the grade data.
29    cout << fixed << setprecision(2);
30    cout << "The raw score is "
31        << exam.getRawScore() << endl;
32    cout << "The curved score is "
33        << exam.getScore() << endl;
34    cout << "The curved grade is "
35        << exam.getLetterGrade() << endl;
36
37    return 0;
38 }
```

Program Output with Example Input Shown in Bold

Enter the student's raw numeric score: **87**

Enter the curve percentage for this student: **1.06**

The raw score is 87.00

The curved score is 92.22

The curved grade is A

It is important to note even though a derived class may redefine a function in the base class, objects that are defined of the base class type still call the base class version of the function. This is demonstrated in Program 15-8.

Program 15-8

```
1 // This program demonstrates that when a derived class function
2 // overrides a class function, objects of the base class
3 // still call the base class version of the function.
4 #include <iostream>
5 using namespace std;
6
7 class BaseClass
8 {
9 public:
10     void showMessage()
11         { cout << "This is the Base class.\n"; }
12 };
13
14 class DerivedClass : public BaseClass
15 {
16 public:
17     void showMessage()
18         { cout << "This is the Derived class.\n"; }
19 };
20
21 int main()
22 {
23     BaseClass b;
24     DerivedClass d;
25
26     b.showMessage();
27     d.showMessage();
28
29     return 0;
30 }
```

Program Output

```
This is the Base class.
This is the Derived class.
```

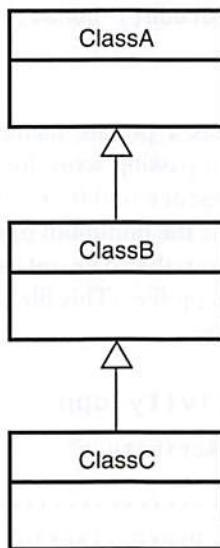
In Program 15-8, a class named `BaseClass` is declared with a member function named `showMessage`. A class named `DerivedClass` is then declared, also with a `showMessage` member function. As their names imply, `DerivedClass` is derived from `BaseClass`. Two objects, `b` and `d`, are defined in function `main`. The object `b` is a `BaseClass` object, and `d` is a `DerivedClass` object. When `b` is used to call the `showMessage` function, it is the `BaseClass` version that is executed. Likewise, when `d` is used to call `showMessage`, the `DerivedClass` version is used.

15.5 Class Hierarchies

CONCEPT: A base class can also be derived from another class.

Sometimes it is desirable to establish a hierarchy of classes in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by Figure 15-4. In some cases, the inheritance of classes goes on for many layers.

Figure 15-4 Class hierarchy



In Figure 15-4, ClassC inherits ClassB's members, including the ones that ClassB inherited from ClassA. Let's look at an example of such a chain of inheritance. Consider the following `PassFailActivity` class, which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of 'P' for passing, or 'F' for failing. (This file can be found in the Student Source Code Folder Chapter 15\PassFailActivity.)

Contents of PassFailActivity.h

```
1 #ifndef PASSFAILACTIVITY_H
2 #define PASSFAILACTIVITY_H
3 #include "GradedActivity.h"
4
5 class PassFailActivity : public GradedActivity
6 {
7 protected:
8     double minPassingScore; // Minimum passing score.
9 public:
10    // Default constructor
11    PassFailActivity() : GradedActivity()
12    { minPassingScore = 0.0; }
```

```

13
14    // Constructor
15    PassFailActivity(double mps) : GradedActivity()
16        { minPassingScore = mps; }
17
18    // Mutator
19    void setMinPassingScore(double mps)
20        { minPassingScore = mps; }
21
22    // Accessors
23    double getMinPassingScore() const
24        { return minPassingScore; }
25
26    char getLetterGrade() const;
27 };
28 #endif

```

The `PassFailActivity` class has a private member variable named `minPassingScore`. This variable holds the minimum passing score for an activity. The default constructor, in lines 11 and 12, sets `minPassingScore` to 0.0. An overloaded constructor in lines 15 and 16 accepts a `double` argument that is the minimum passing grade for the activity. This value is stored in the `minPassingScore` variable. The `getLetterGrade` member function is defined in the following `PassFailActivity.cpp` file. (This file can be found in the Student Source Code Folder Chapter 15\PassFailActivity.)

Contents of `PassFailActivity.cpp`

```

1 #include "PassFailActivity.h"
2
3 //*****
4 // Member function PassFailActivity::getLetterGrade      *
5 // This function returns 'P' if the score is passing,   *
6 // otherwise it returns 'F'.                            *
7 //*****
8
9 char PassFailActivity::getLetterGrade() const
10 {
11     char letterGrade;
12
13     if (score >= minPassingScore)
14         letterGrade = 'P';
15     else
16         letterGrade = 'F';
17
18     return letterGrade;
19 }

```

This `getLetterGrade` member function redefines the `getLetterGrade` member function of `GradedActivity` class. This version of the function returns a grade of 'P' if the numeric score is greater than or equal to `minPassingScore`. Otherwise, the function returns a grade of 'F'.

The `PassFailActivity` class represents the general characteristics of a student's pass-or-fail activity. There might be numerous types of pass-or-fail activities, however. Suppose we need a more specialized class, such as one that determines a student's grade for a pass-or-fail exam. The following `PassFailExam` class is an example. This class is derived from the `PassFailActivity` class. It inherits all of the members of `PassFailActivity`, including the ones that `PassFailActivity` inherits from `GradedActivity`. The `PassFailExam` class calculates the number of points that each question on the exam is worth, as well as the student's numeric score. (These files are available in the Student Source Code Folder Chapter 15\PassFailActivity.)

Contents of `PassFailExam.h`

```
1 #ifndef PASSFAILEXAM_H
2 #define PASSFAILEXAM_H
3 #include "PassFailActivity.h"
4
5 class PassFailExam : public PassFailActivity
6 {
7 private:
8     int numQuestions;      // Number of questions
9     double pointsEach;    // Points for each question
10    int numMissed;        // Number of questions missed
11 public:
12     // Default constructor
13     PassFailExam() : PassFailActivity()
14     { numQuestions = 0;
15         pointsEach = 0.0;
16         numMissed = 0; }
17
18     // Constructor
19     PassFailExam(int questions, int missed, double mps) :
20         PassFailActivity(mps)
21         { set(questions, missed); }
22
23     // Mutator function
24     void set(int, int); // Defined in PassFailExam.cpp
25
26     // Accessor functions
27     double getNumQuestions() const
28         { return numQuestions; }
29
30     double getPointsEach() const
31         { return pointsEach; }
32
33     int getNumMissed() const
34         { return numMissed; }
35 };
36 #endif
```

Contents of PassFailExam.cpp

```

1 #include "PassFailExam.h"
2
3 //*****
4 // set function
5 // The parameters are the number of questions and the
6 // number of questions missed.
7 //*****
8
9 void PassFailExam::set(int questions, int missed)
10 {
11     double numericScore; // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26 }
```

The PassFailExam class inherits all of the PassFailActivity class's members, including the ones that PassFailActivity inherited from GradedActivity. Because the public base class access specification is used, all of the protected members of PassFailActivity become protected members of PassFailExam, and all of the public members of PassFailActivity become public members of PassFailExam. Table 15-2 lists all of the member variables of the PassFailExam class, and Table 15-3 lists all the member functions. These include the members that were inherited from the base classes.

Table 15-2 Member Variables of the PassFailExam Class

Member Variable	Access	Inherited?
numQuestions	protected	No
pointsEach	protected	No
numMissed	protected	No
minPassingScore	protected	Yes, from PassFailActivity
score	protected	Yes, from PassFailActivity, which inherited it from GradedActivity

Table 15-3 Member Functions of the PassFailExam Class

Member Function	Access	Inherited?
set	public	No
getNumQuestions	public	No
getPointsEach	public	No
getNumMissed	public	No
setMinPassingScore	public	Yes, from PassFailActivity
getMinPassingScore	public	Yes, from PassFailActivity
getLetterGrade	public	Yes, from PassFailActivity
setScore	public	Yes, from PassFailActivity, which inherited it from GradedActivity
getScore	public	Yes, from PassFailActivity, which inherited it from GradedActivity

Program 15-9 demonstrates the PassFailExam class. This file can be found in the student source code folder Chapter 15\PassFailActivity.

Program 15-9

```

1 // This program demonstrates the PassFailExam class.
2 #include <iostream>
3 #include <iomanip>
4 #include "PassFailExam.h"
5 using namespace std;
6
7 int main()
8 {
9     int questions;           // Number of questions
10    int missed;             // Number of questions missed
11    double minPassing;      // The minimum passing score
12
13    // Get the number of questions on the exam.
14    cout << "How many questions are on the exam? ";
15    cin >> questions;
16
17    // Get the number of questions the student missed.
18    cout << "How many questions did the student miss? ";
19    cin >> missed;
20
21    // Get the minimum passing score.
22    cout << "Enter the minimum passing score for this test: ";
23    cin >> minPassing;
24
25    // Define a PassFailExam object.
26    PassFailExam exam(questions, missed, minPassing);
27

```

(program continues)

Program 15-9 (continued)

```

28     // Display the test results.
29     cout << fixed << setprecision(1);
30     cout << "\nEach question counts "
31         << exam.getPointsEach() << " points.\n";
32     cout << "The minimum passing score is "
33         << exam.getMinPassingScore() << endl;
34     cout << "The student's exam score is "
35         << exam.getScore() << endl;
36     cout << "The student's grade is "
37         << exam.getLetterGrade() << endl;
38     return 0;
39 }
```

Program Output with Example Input Shown in Bold

How many questions are on the exam? **100**

How many questions did the student miss? **25**

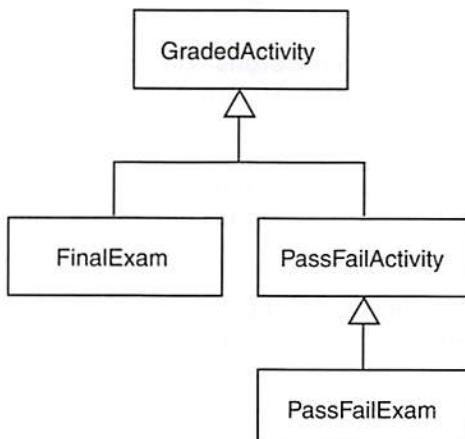
Enter the minimum passing score for this test: **60**

Each question counts 1.0 points.
 The minimum passing score is 60.0
 The student's exam score is 75.0
 The student's grade is P

This program uses the `PassFailExam` object to call the `getLetterGrade` member function in line 37. Recall that the `PassFailActivity` class redefines the `getLetterGrade` function to report only grades of 'P' or 'F'. Because the `PassFailExam` class is derived from the `PassFailActivity` class, it inherits the redefined `getLetterGrade` function.

Software designers often use class hierarchy diagrams. Like a family tree, a class hierarchy diagram shows the inheritance relationships between classes. Figure 15-5 shows a class hierarchy for the `GradedActivity`, `FinalExam`, `PassFailActivity`, and `PassFailExam` classes. The more general classes are toward the top of the tree, and the more specialized classes are toward the bottom.

Figure 15-5 Class hierarchy



15.6

Polymorphism and Virtual Member Functions

CONCEPT: Polymorphism allows an object reference variable or an object pointer to reference objects of different types and to call the correct member functions, depending upon the type of object being referenced.



Look at the following code for a function named `displayGrade`:

```
void displayGrade(const GradedActivity &activity)
{
    cout << setprecision(1) << fixed;
    cout << "The activity's numeric score is "
        << activity.getScore() << endl;
    cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

This function uses a `const GradedActivity` reference variable as its parameter. When a `GradedActivity` object is passed as an argument to this function, the function calls the object's `getScore` and `getLetterGrade` member functions to display the numeric score and letter grade. The following code shows how we might call the function:

```
GradedActivity test(88.0); // The score is 88
displayGrade(test); // Pass test to displayGrade
```

This code will produce the following output:

```
The activity's numeric score is 88.0
The activity's letter grade is B
```

Recall the `GradedActivity` class is also the base class for the `FinalExam` class. Because of the “is-a” relationship between a base class and a derived class, an object of the `FinalExam` class is not just a `FinalExam` object. It is also a `GradedActivity` object. (A final exam *is a* graded activity.) Because of this relationship, we can also pass a `FinalExam` object to the `displayGrade` function. For example, look at the following code:

```
// There are 100 questions. The student missed 25.
FinalExam test2(100, 25);
displayGrade(test2);
```

This code will produce the following output:

```
The activity's numeric score is 75.0
The activity's letter grade is C
```

Because the parameter in the `displayGrade` function is a `GradedActivity` reference variable, it can reference any object that is derived from `GradedActivity`. A problem can occur with this type of code, however, when redefined member functions are involved. For example, recall that the `PassFailActivity` class is derived from the `GradedActivity` class. The `PassFailActivity` class redefines the `getLetterGrade` function. Although we can pass a `PassFailActivity` object as an argument to the `displayGrade` function, we will not get the results we wish. This is demonstrated in Program 15-10. (This file is available in the Student Source Code Folder Chapter 15\PassFailActivity.)

Program 15-10

```

1 #include <iostream>
2 #include <iomanip>
3 #include "PassFailActivity.h"
4 using namespace std;
5
6 // Function prototype
7 void displayGrade(const GradedActivity &);
8
9 int main()
10 {
11     // Create a PassFailActivity object. Minimum passing
12     // score is 70.
13     PassFailActivity test(70);
14
15     // Set the score to 72.
16     test.setScore(72);
17
18     // Display the object's grade data. The letter grade
19     // should be 'P'. What will be displayed?
20     displayGrade(test);
21     return 0;
22 }
23
24 //*****
25 // The displayGrade function displays a GradedActivity object's *
26 // numeric score and letter grade. *
27 //*****
28
29 void displayGrade(const GradedActivity &activity)
30 {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33         << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35         << activity.getLetterGrade() << endl;
36 }
```

Program Output

The activity's numeric score is 72.0

The activity's letter grade is C

As you can see from the example output, the `getLetterGrade` member function returned 'C' instead of 'P'. This is because the `GradedActivity` class's `getLetterGrade` function was executed instead of the `PassFailActivity` class's version of the function.

This behavior happens because of the way C++ matches function calls with the correct function. This process is known as *binding*. In Program 15-10, C++ decides at compile time which version of the `getLetterGrade` function to execute when it encounters the call to the function in line 35. Even though we passed a `PassFailActivity` object to the `displayGrade` function, the `activity` parameter in the `displayGrade` function is a

GradedActivity reference variable. Because it is of the GradedActivity type, the compiler binds the function call in line 35 with the GradedActivity class's getLetterGrade function. When the program executes, it has already been determined by the compiler that the GradedActivity class's getLetterGrade function will be called. The process of matching a function call with a function at compile time is called *static binding*.

To remedy this, the getLetterGrade function can be made *virtual*. A *virtual function* is a member function that is dynamically bound to function calls. In *dynamic binding*, C++ determines which function to call at runtime, depending on the type of the object responsible for the call. If a GradedActivity object is responsible for the call, C++ will execute the GradedActivity::getLetterGrade function. If a PassFailActivity object is responsible for the call, C++ will execute the PassFailActivity::getLetterGrade function.

Virtual functions are declared by placing the key word `virtual` before the return type in the base class's function declaration, such as

```
virtual char getLetterGrade() const;
```

This declaration tells the compiler to expect `getLetterGrade` to be redefined in a derived class. The compiler does not bind calls to the function with the actual function. Instead, it allows the program to bind calls, at runtime, to the version of the function that belongs to the same class as the object responsible for the call.



NOTE: You place the `virtual` key word only in the function's declaration or prototype. If the function is defined outside the class, you do not place the `virtual` key word in the function header.

The following code shows an updated version of the GradedActivity class, with the `getLetterGrade` function declared `virtual`. This file can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 3. The `GradedActivity.cpp` file has not changed, so it is not shown again.

Contents of `GradedActivity.h` (Version 3)

```
1 #ifndef GRADEDACTIVITY_H
2 #define GRADEDACTIVITY_H
3
4 // GradedActivity class declaration
5
6 class GradedActivity
7 {
8 protected:
9     double score; // To hold the numeric score
10 public:
11     // Default constructor
12     GradedActivity()
13     { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17     { score = s; }
18
```

```

19      // Mutator function
20      void setScore(double s)
21          { score = s; }
22
23      // Accessor functions
24      double getScore() const
25          { return score; }
26
27      virtual char getLetterGrade() const;
28  };
29 #endif

```

The only change we have made to this class is to declare `getLetterGrade` as `virtual` in line 27. This tells the compiler not to bind calls to `getLetterGrade` with the function at compile time. Instead, calls to the function will be bound dynamically to the function at runtime.

When a member function is declared `virtual` in a base class, any redefined versions of the function that appear in derived classes automatically become `virtual`. So, it is not necessary to declare the `getLetterGrade` function in the `PassFailActivity` class as `virtual`. It is still a good idea to declare the function `virtual` in the `PassFailActivity` class for documentation purposes. A new version of the `PassFailActivity` class is shown here. This file can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 3. The `PassFailActivity.cpp` file has not changed, so it is not shown again.

Contents of PassFailActivity.h

```

1  #ifndef PASSFAILACTIVITY_H
2  #define PASSFAILACTIVITY_H
3  #include "GradedActivity.h"
4
5  class PassFailActivity : public GradedActivity
6  {
7  protected:
8      double minPassingScore; // Minimum passing score
9  public:
10     // Default constructor
11     PassFailActivity() : GradedActivity()
12         { minPassingScore = 0.0; }
13
14     // Constructor
15     PassFailActivity(double mps) : GradedActivity()
16         { minPassingScore = mps; }
17
18     // Mutator
19     void setMinPassingScore(double mps)
20         { minPassingScore = mps; }
21
22     // Accessors
23     double getMinPassingScore() const
24         { return minPassingScore; }
25
26     virtual char getLetterGrade() const;
27  };
28 #endif

```

The only change we have made to this class is to declare `getLetterGrade` as `virtual` in line 26. Program 15-11 is identical to Program 15-10, except it uses the corrected version of the `GradedActivity` and `PassFailActivity` classes. This file is also available in the student source code folder Chapter 15\GradedActivity Version 3.

Program 15-11

```
1 #include <iostream>
2 #include <iomanip>
3 #include "PassFailActivity.h"
4 using namespace std;
5
6 // Function prototype
7 void displayGrade(const GradedActivity &);
8
9 int main()
10 {
11     // Create a PassFailActivity object. Minimum passing
12     // score is 70.
13     PassFailActivity test(70);
14
15     // Set the score to 72.
16     test.setScore(72);
17
18     // Display the object's grade data. The letter grade
19     // should be 'P'. What will be displayed?
20     displayGrade(test);
21     return 0;
22 }
23
24 //*****
25 // The displayGrade function displays a GradedActivity object's *
26 // numeric score and letter grade. *
27 //*****
28
29 void displayGrade(const GradedActivity &activity)
30 {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33         << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35         << activity.getLetterGrade() << endl;
36 }
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

Now that the `getLetterGrade` function is declared `virtual`, the program works properly. This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms. Program 15-12 demonstrates polymorphism by passing objects of the `GradedActivity` and `PassFailExam` classes to the `displayGrade` function. This file can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 3.

Program 15-12

```

1 #include <iostream>
2 #include <iomanip>
3 #include "PassFailExam.h"
4 using namespace std;
5
6 // Function prototype
7 void displayGrade(const GradedActivity &);
8
9 int main()
10 {
11     // Create a GradedActivity object. The score is 88.
12     GradedActivity test1(88.0);
13
14     // Create a PassFailExam object. There are 100 questions,
15     // the student missed 25 of them, and the minimum passing
16     // score is 70.
17     PassFailExam test2(100, 25, 70.0);
18
19     // Display the grade data for both objects.
20     cout << "Test 1:\n";
21     displayGrade(test1);    // GradedActivity object
22     cout << "\nTest 2:\n";
23     displayGrade(test2);    // PassFailExam object
24     return 0;
25 }
26
27 //*****
28 // The displayGrade function displays a GradedActivity object's *
29 // numeric score and letter grade. *
30 //*****
31
32 void displayGrade(const GradedActivity &activity)
33 {
34     cout << setprecision(1) << fixed;
35     cout << "The activity's numeric score is "
36         << activity.getScore() << endl;
37     cout << "The activity's letter grade is "
38         << activity.getLetterGrade() << endl;
39 }
```

Program Output

Test 1:

The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:

The activity's numeric score is 75.0
The activity's letter grade is P

Polymorphism Requires References or Pointers

The `displayGrade` function in Programs 15-11 and 15-12 uses a `GradedActivity` reference variable as its parameter. When we call the function, we pass an object by reference. Polymorphic behavior is not possible when an object is passed by value, however. For example, suppose the `displayGrade` function had been written as shown here:

```
// Polymorphic behavior is not possible with this function.  
void displayGrade(const GradedActivity activity)  
{  
    cout << setprecision(1) << fixed;  
    cout << "The activity's numeric score is "  
        << activity.getScore() << endl;  
    cout << "The activity's letter grade is "  
        << activity.getLetterGrade() << endl;  
}
```

In this version of the function, the `activity` parameter is an object variable, not a reference variable. Suppose we call this version of the function with the following code:

```
// Create a GradedActivity object. The score is 88.  
GradedActivity test1(88.0);  
  
// Create a PassFailExam object. There are 100 questions,  
// the student missed 25 of them, and the minimum passing  
// score is 70.  
PassFailExam test2(100, 25, 70.0);  
  
// Display the grade data for both objects.  
cout << "Test 1:\n";  
displayGrade(test1); // Pass the GradedActivity object  
cout << "\nTest 2:\n";  
displayGrade(&test2); // Pass the PassFailExam object
```

This code will produce the following output:

Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is C

Even though the `getLetterGrade` function is declared `virtual`, static binding still takes place because `activity` is not a reference variable or a pointer.

Alternatively, we could have used a `GradedActivity` pointer in the `displayGrade` function, as shown in Program 15-13. This file is also available in the Student Source Code Folder Chapter 15\GradedActivity Version 3.

Program 15-13

```
1 #include <iostream>
2 #include <iomanip>
3 #include "PassFailExam.h"
4 using namespace std;
5
6 // Function prototype
7 void displayGrade(const GradedActivity * );
8
9 int main()
10 {
11     // Create a GradedActivity object. The score is 88.
12     GradedActivity test1(88.0);
13
14     // Create a PassFailExam object. There are 100 questions,
15     // the student missed 25 of them, and the minimum passing
16     // score is 70.
17     PassFailExam test2(100, 25, 70.0);
18
19     // Display the grade data for both objects.
20     cout << "Test 1:\n";
21     displayGrade(&test1); // Address of the GradedActivity object
22     cout << "\nTest 2:\n";
23     displayGrade(&test2); // Address of the PassFailExam object
24     return 0;
25 }
26
27 //*****
28 // The displayGrade function displays a GradedActivity object's *
29 // numeric score and letter grade. This version of the function *
30 // uses a GradedActivity pointer as its parameter. *
31 //*****
32
33 void displayGrade(const GradedActivity *activity)
34 {
35     cout << setprecision(1) << fixed;
36     cout << "The activity's numeric score is "
37         << activity->getScore() << endl;
38     cout << "The activity's letter grade is "
39         << activity->getLetterGrade() << endl;
40 }
```

Program Output

Test 1:

The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:

The activity's numeric score is 75.0
The activity's letter grade is P

Base Class Pointers

Pointers to a base class may be assigned the address of a derived class object. For example, look at the following code:

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
```

This statement dynamically allocates a `PassFailExam` object and assigns its address to `exam`, which is a `GradedActivity` pointer. We can then use the `exam` pointer to call member functions, as shown here:

```
cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Program 15-14 is an example that uses base class pointers to reference derived class objects. This file can be found in the Student Source Code Folder Chapter 15\GradedActivity Version 3.

Program 15-14

```
1 #include <iostream>
2 #include <iomanip>
3 #include "PassFailExam.h"
4 using namespace std;
5
6 // Function prototype
7 void displayGrade(const GradedActivity *);
8
9 int main()
10 {
11     // Constant for the size of an array.
12     const int NUM_TESTS = 4;
13
14     // tests is an array of GradedActivity pointers.
15     // Each element of tests is initialized with the
16     // address of a dynamically allocated object.
17     GradedActivity *tests[NUM_TESTS] =
18         { new GradedActivity(88.0),
19             new PassFailExam(100, 25, 70.0),
20             new GradedActivity(67.0),
21             new PassFailExam(50, 12, 60.0)
22         };
```

(program continues)

Program 15-14 (continued)

```

23
24 // Display the grade data for each element in the array.
25 for (int count = 0; count < NUM_TESTS; count++)
26 {
27   cout << "Test #" << (count + 1) << "\n";
28   displayGrade(tests[count]);
29   cout << endl;
30 }
31 return 0;
32 }
33 *****
34 // The displayGrade function displays a GradeActivity object's
35 // numeric score and letter grade. This version of the function
36 // uses a GradeActivity pointer as its parameter.
37 // numeric score and letter grade. This version of the function
38 // uses a GradeActivity pointer as its parameter.
39 *****
40 void displayGrade(const GradeActivity *activity)
41 {
42   cout << setprecision(1) << fixed;
43   cout << "The activity's numeric score is ";
44   cout << activity->getScore() << endl;
45   cout << "The activity's letter grade is ";
46   cout << activity->getLetterGrade() << endl;
47 }
```

Program Output

Test #1:	The activity's numeric score is 88.0 The activity's letter grade is B
Test #2:	The activity's numeric score is 75.0 The activity's letter grade is P
Test #3:	The activity's numeric score is 67.0 The activity's letter grade is D
Test #4:	The activity's numeric score is 76.0 The activity's letter grade is P

Let's take a closer look at this program. An array named `tests` is defined in lines 17 through 22. This is an array of `GradedActivity` pointers. The array elements are initialized with the addresses of dynamically allocated objects. The `tests[0]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new GradedActivity(88.0)
```

The `tests[1]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new PassFailExam(100, 25, 70.0)
```

The `tests[2]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new GradedActivity(67.0)
```

Finally, the `tests[3]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new PassFailExam(50, 12, 60.0)
```

Although each element in the array is a `GradedActivity` pointer, some of the elements point to `GradedActivity` objects, and some point to `PassFailExam` objects. The loop in lines 25 through 30 steps through the array, passing each pointer element to the `displayGrade` function.

Base Class Pointers and References Know Only about Base Class Members

Although a base class pointer can reference objects of any class that derives from the base class, there are limits to what the pointer can do with those objects. Recall that the `GradedActivity` class has, other than its constructors, only three member functions: `setScore`, `getScore`, and `getLetterGrade`. So, a `GradedActivity` pointer can be used to call only those functions, regardless of the type of object to which it points. For example, look at the following code:

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
cout << exam->getScore() << endl;           // This works.
cout << exam->getLetterGrade() << endl;        // This works.
cout << exam->getPointsEach() << endl;        // ERROR! Won't work!
```

In this code, `exam` is a `GradedActivity` pointer, and is assigned the address of a `PassFailExam` object. The `GradedActivity` class has only the `setScore`, `getScore`, and `getLetterGrade` member functions, so those are the only member functions the `exam` variable knows how to execute. The last statement in this code is a call to the `getPointsEach` member function, which is defined in the `PassFailExam` class. Because the `exam` variable only knows about member functions in the `GradedActivity` class, it cannot execute this function.

The “Is-a” Relationship Does Not Work in Reverse

It is important to note that the “is-a” relationship does not work in reverse. Although the statement “a final exam is a graded activity” is true, the statement “a graded activity is a

final exam” is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work:

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);

// Error! This will not work.
FinalExam *fePointer = gaPointer;
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` pointer. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the C++ compiler will let you make such an assignment if you use a type cast, as shown here:

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);

// This will work, but with limitations.
FinalExam *fePointer = static_cast<FinalExam *>(gaPointer);
```

After this code executes, the derived class pointer `fePointer` will be pointing to a base class object. We can use the pointer to access members of the object, but only the members that exist. The following code demonstrates:

```
// This will work. The object has a getScore function.
cout << fePointer->getScore() << endl;

// This will work. The object has a getLetterGrade function.
cout << fePointer->getLetterGrade() << endl;

// This will compile, but an error will occur at runtime.
// The object does not have a getPointsEach function.
cout << fePointer->getPointsEach() << endl;
```

In this code, `fePointer` is a `FinalExam` pointer, and it points to a `GradedActivity` object. The first two `cout` statements work because the `GradedActivity` object has `getScore` and `getLetterGrade` member functions. The last `cout` statement will cause an error, however, because it calls the `getPointsEach` member function. The `GradedActivity` object does not have a `getPointsEach` member function.

Redefining versus Overriding

Earlier in this chapter, you learned how a derived class can redefine a base class member function. When a class redefines a virtual function, it is said that the class *overrides* the function. In C++, the difference between overriding and redefining base class functions is that overridden functions are dynamically bound, and redefined functions are statically bound. Only virtual functions can be overridden.

Virtual Destructors

When you write a class with a destructor, and that class could potentially become a base class, you should always declare the destructor `virtual`. This is because the compiler will perform static binding on the destructor if it is not declared `virtual`. This can lead to problems when a base class pointer or reference variable references a derived class object. If the

derived class has its own destructor, it will not execute when the object is destroyed or goes out of scope. Only the base class destructor will execute. Program 15-15 demonstrates this.

Program 15-15

```
1 #include <iostream>
2 using namespace std;
3
4 // Animal is a base class.
5 class Animal
6 {
7 public:
8     // Constructor
9     Animal()
10    { cout << "Animal constructor executing.\n"; }
11
12    // Destructor
13    ~Animal()
14    { cout << "Animal destructor executing.\n"; }
15 };
16
17 // The Dog class is derived from Animal
18 class Dog : public Animal
19 {
20 public:
21     // Constructor
22     Dog() : Animal()
23     { cout << "Dog constructor executing.\n"; }
24
25     // Destructor
26     ~Dog()
27     { cout << "Dog destructor executing.\n"; }
28 };
29
30 //*****
31 // main function
32 //*****
33
34 int main()
35 {
36     // Create a Dog object, referenced by an
37     // Animal pointer.
38     Animal *myAnimal = new Dog;
39
40     // Delete the dog object.
41     delete myAnimal;
42
43 }
```

Program Output

```
Animal constructor executing.
Dog constructor executing.
Animal destructor executing.
```

This program declares two classes: `Animal` and `Dog`. `Animal` is the base class and `Dog` is the derived class. Each class has its own constructor and destructor. In line 38, a `Dog` object is created, and its address is stored in an `Animal` pointer. Both the `Animal` and the `Dog` constructors execute. In line 41, the object is deleted. When this statement executes, however, only the `Animal` destructor executes. The `Dog` destructor does not execute because the object is referenced by an `Animal` pointer. We can fix this problem by declaring the `Animal` class destructor `virtual`, as shown in Program 15-16.

Program 15-16

```

1 #include <iostream>
2 using namespace std;
3
4 // Animal is a base class.
5 class Animal
6 {
7 public:
8     // Constructor
9     Animal()
10    { cout << "Animal constructor executing.\n"; }
11
12    // Destructor
13    virtual ~Animal()
14    { cout << "Animal destructor executing.\n"; }
15};
16
17 // The Dog class is derived from Animal
18 class Dog : public Animal
19 {
20 public:
21     // Constructor
22     Dog() : Animal()
23     { cout << "Dog constructor executing.\n"; }
24
25     // Destructor
26     ~Dog()
27     { cout << "Dog destructor executing.\n"; }
28 };
29
30 //*****
31 // main function
32 //*****
33
34 int main()
35 {
36     // Create a Dog object, referenced by an
37     // Animal pointer.
38     Animal *myAnimal = new Dog;
39
40     // Delete the dog object.
41     delete myAnimal;
42
43 }
```

Program Output

```
Animal constructor executing.  
Dog constructor executing.  
Dog destructor executing.  
Animal destructor executing.
```

The only thing that has changed in this program is that the `Animal` class destructor is declared `virtual` in line 13. As a result, the destructor is dynamically bound at runtime. When the `Dog` object is destroyed, both the `Animal` and `Dog` destructors execute.

A good programming practice to follow is that any class that has a virtual member function should also have a virtual destructor. If the class doesn't require a destructor, it should have a virtual destructor that performs no statements. Remember, when a base class function is declared `virtual`, all overridden versions of the function in derived classes automatically become virtual. Including a virtual destructor in a base class, even one that does nothing, will ensure that any derived class destructors will also be virtual.

C++ 11's override and final Key Words**11**

C++ 11 introduces the `override` key word to help prevent subtle errors when overriding virtual functions. For example, can you find the mistake in Program 15-17?

Program 15-17

```
1 // This program has a subtle error in the virtual functions.  
2 #include <iostream>  
3 using namespace std;  
4  
5 class Base  
6 {  
7 public:  
8     virtual void functionA(int arg) const  
9         { cout << "This is Base::functionA" << endl; }  
10 };  
11  
12 class Derived : public Base  
13 {  
14 public:  
15     virtual void functionA(long arg) const  
16         { cout << "This is Derived::functionA" << endl; }  
17 };  
18  
19 int main()  
20 {  
21     // Allocate instances of the Derived class.  
22     Base *b = new Derived();  
23     Derived *d = new Derived();  
24  
25     // Call functionA with the two pointers.  
26     b->functionA(99);
```

(program continues)

Program 15-17 *(continued)*

```
27     d->functionA(99);  
28  
29     return 0;  
30 }
```

Program Output

```
This is Base::functionA  
This is Derived::functionA
```

Both the `Base` class and the `Derived` class have a virtual member function named `functionA`.

Notice in lines 22 and 23 in the `main` function, we allocate two instances of the `Derived` class. We reference one of the instances with a `Base` class pointer (`b`), and we reference the other instance with a `Derived` class pointer (`d`). When we call `functionA` in lines 26 and 27, we might expect that the `Derived` class's `functionA` would be called in both lines. This is not the case, however, as you can see from the program's output.

The `functionA` in the `Derived` class does not override the `functionA` in the `Base` class because the function signatures are different. The `functionA` in the `Base` class takes an `int` argument, but the one in the `Derived` class takes a `long` argument. So, `functionA` in the `Derived` class merely overloads `functionA` in the `Base` class.

To make sure a member function in a derived class overrides a virtual member function in a base class, you can use the `override` key word in the derived class's function prototype (or the function header, if the function is written inline). The `override` key word tells the compiler that the function is supposed to override a function in the base class. It will cause a compiler error if the function does not actually override any functions. Program 15-18 demonstrates how Program 15-17 can be fixed so the `Derived` class function does, in fact, override the `Base` class function. Notice in line 15 we have changed the parameter in the `Derived` class function to an `int`, and we have added the `override` key word to the function header.

Program 15-18

```
1 // This program demonstrates the override key word.  
2 #include <iostream>  
3 using namespace std;  
4  
5 class Base  
6 {  
7 public:  
8     virtual void functionA(int arg) const  
9     { cout << "This is Base::functionA" << endl; }  
10 };  
11  
12 class Derived : public Base  
13 {  
14 public:
```

```
15     virtual void functionA(int arg) const override
16     { cout << "This is Derived::functionA" << endl; }
17 };
18
19 int main()
20 {
21     // Allocate instances of the Derived class.
22     Base *b = new Derived();
23     Derived *d = new Derived();
24
25     // Call functionA with the two pointers.
26     b->functionA(99);
27     d->functionA(99);
28
29     return 0;
30 }
```

Program Output

This is Derived::functionA

This is Derived::functionA

Preventing a Member Function from Being Overridden

In some derived classes, you might want to make sure a virtual member function cannot be overridden any further down the class hierarchy. When a member function is declared with the `final` key word, it cannot be overridden in a derived class. The following member function prototype is an example that uses the `final` key word:

```
virtual void message() const final;
```

If a derived class attempts to override a `final` member function, the compiler generates an error.

15.7

Abstract Base Classes and Pure Virtual Functions

CONCEPT: An abstract base class cannot be instantiated, but other classes are derived from it. A pure virtual function is a virtual member function of a base class that must be overridden. When a class contains a pure virtual function as a member, that class becomes an abstract base class.

Sometimes it is helpful to begin a class hierarchy with an *abstract base class*. An abstract base class is not instantiated itself, but serves as a base class for other classes. The abstract base class represents the generic, or abstract, form of all the classes that are derived from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of planes: two different models of prop-driven planes, and one commuter jet model. The computer software that catalogs the planes might use an abstract base class called `Airplane`. That class has members representing the common characteristics of all airplanes. In addition, it has classes for each of the three specific airplane models the factory manufactures. These classes have members representing

the unique characteristics of each type of plane. The base class, `Airplane`, is never instantiated, but is used to derive the other classes.

A class becomes an abstract base class when one or more of its member functions is a *pure virtual function*. A pure virtual function is a virtual member function declared in a manner similar to the following:

```
virtual void showInfo() = 0;
```

The `= 0` notation indicates that `showInfo` is a pure virtual function. Pure virtual functions have no body, or definition, in the base class. They must be overridden in derived classes. Additionally, the presence of a pure virtual function in a class prevents a program from instantiating the class. The compiler will generate an error if you attempt to define an object of an abstract base class.

For example, look at the following abstract base class `Student`. It holds data common to all students, but does not hold all the data needed for students of specific majors.

Contents of Student.h

```

1 // Specification file for the Student class
2 #ifndef STUDENT_H
3 #define STUDENT_H
4 #include <string>
5 using namespace std;
6
7 class Student
8 {
9 protected:
10     string name;           // Student name
11     string idNumber;       // Student ID
12     int yearAdmitted;     // Year student was admitted
13 public:
14     // Default constructor
15     Student()
16     { name = "";
17         idNumber = "";
18         yearAdmitted = 0; }
19
20     // Constructor
21     Student(string n, string id, int year)
22     { set(n, id, year); }
23
24     // The set function sets the attribute data.
25     void set(string n, string id, int year)
26     { name = n;           // Assign the name
27         idNumber = id;     // Assign the ID number
28         yearAdmitted = year; } // Assign the year admitted
29
30     // Accessor functions
31     const string getName() const
32     { return name; }
33
34     const string getIdNum() const
35     { return idNumber; }
```

```
36
37     int getYearAdmitted() const
38         { return yearAdmitted; }
39
40     // Pure virtual function
41     virtual int getRemainingHours() const = 0;
42 };
43 #endif
```

The Student class contains members for storing a student's name, ID number, and year admitted. It also has constructors and a mutator function for setting values in the name, idNumber, and yearAdmitted members. Accessor functions are provided that return the values in the name, idNumber, and yearAdmitted members. A pure virtual function named getRemainingHours is also declared.

The pure virtual function must be overridden in classes derived from the Student class. It was made a pure virtual function because this class is intended to be the base for classes that represent students of specific majors. For example, a CsStudent class might hold the data for a computer science student, and a BiologyStudent class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the CsStudent class will calculate the number of hours taken in a different manner than the BiologyStudent class.

Let's look at an example of the CsStudent class.

Contents of CsStudent.h

```
1 // Specification file for the CsStudent class
2 #ifndef CSSTUDENT_H
3 #define CSSTUDENT_H
4 #include "Student.h"
5
6 // Constants for required hours
7 const int MATH_HOURS = 20;      // Math hours
8 const int CS_HOURS = 40;        // Computer science hours
9 const int GEN_ED_HOURS = 60;    // General Ed hours
10
11 class CsStudent : public Student
12 {
13 private:
14     int mathHours;      // Hours of math taken
15     int csHours;        // Hours of Computer Science taken
16     int genEdHours;    // Hours of general education taken
17
18 public:
19     // Default constructor
20     CsStudent() : Student()
21     { mathHours = 0;
22      csHours = 0;
23      genEdHours = 0; }
```

```

25      // Constructor
26      CsStudent(string n, string id, int year) :
27          Student(n, id, year)
28      { mathHours = 0;
29          csHours = 0;
30          genEdHours = 0; }
31
32      // Mutator functions
33      void setMathHours(int mh)
34      { mathHours = mh; }
35
36      void setCsHours(int csh)
37      { csHours = csh; }
38
39      void setGenEdHours(int geh)
40      { genEdHours = geh; }
41
42      // Overridden getRemainingHours function,
43      // defined in CsStudent.cpp
44      virtual int getRemainingHours() const;
45  };
46 #endif

```

This file declares the following `const int` member variables in lines 7 through 9: MATH_HOURS, CS_HOURS, and GEN_ED_HOURS. These variables hold the required number of math, computer science, and general education hours for a computer science student. The `CsStudent` class, which derives from the `Student` class, declares the following member variables in lines 14 through 16: `mathHours`, `csHours`, and `genEdHours`. These variables hold the number of math, computer science, and general education hours taken by the student. Mutator functions are provided to store values in these variables. In addition, the class overrides the pure virtual `getRemainingHours` function in the `CsStudent.cpp` file.

Contents of `CsStudent.cpp`

```

1 #include <iostream>
2 #include "CsStudent.h"
3 using namespace std;
4
5 //*****
6 // The CsStudent::getRemainingHours function returns *
7 // the number of hours remaining to be taken. *
8 //*****
9
10 int CsStudent::getRemainingHours() const
11 {
12     int reqHours,    // Total required hours
13     remainingHours; // Remaining hours
14
15     // Calculate the required hours.
16     reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;
17
18     // Calculate the remaining hours.
19     remainingHours = reqHours - (mathHours + csHours +
20                                     genEdHours);

```

```
21
22     // Return the remaining hours.
23     return remainingHours;
24 }
```

Program 15-19 provides a simple demonstration of the class.

Program 15-19

```
1 // This program demonstrates the CsStudent class, which is
2 // derived from the abstract base class, Student.
3 #include <iostream>
4 #include "CsStudent.h"
5 using namespace std;
6
7 int main()
8 {
9     // Create a CsStudent object for a student.
10    CsStudent student("Jennifer Haynes", "167W98337", 2006);
11
12    // Store values for Math, Computer Science, and General
13    // Ed hours.
14    student.setMathHours(12);      // Student has taken 12 Math hours
15    student.setCsHours(20);       // Student has taken 20 CS hours
16    student.setGenEdHours(40);    // Student has taken 40 Gen Ed hours
17
18    // Display the number of remaining hours.
19    cout << "The student " << student.getName()
20        << " needs to take " << student.getRemainingHours()
21        << " more hours to graduate.\n";
22
23    return 0;
24 }
```

Program Output

The student Jennifer Haynes needs to take 48 more hours to graduate.

Remember the following points about abstract base classes and pure virtual functions:

- When a class contains a pure virtual function, it is an abstract base class.
- Pure virtual functions are declared with the = 0 notation.
- Abstract base classes cannot be instantiated.
- Pure virtual functions have no body, or definition, in the base class.
- A pure virtual function *must* be overridden at some point in a derived class in order for it to become nonabstract.



Checkpoint

- 15.9 Explain the difference between overloading a function and redefining a function.
- 15.10 Explain the difference between static binding and dynamic binding.
- 15.11 Are virtual functions statically bound or dynamically bound?

15.12 What will the following program display?

```
#include <iostream>
using namespace std;

class First
{
protected:
    int a;
public:
    First(int x = 1)
        { a = x; }

    int getVal()
        { return a; }
};

class Second : public First
{
private:
    int b;

public:
    Second(int y = 5)
        { b = y; }
    int getVal()
        { return b; }
};

int main()
{
    First object1;
    Second object2;

    cout << object1.getVal() << endl;
    cout << object2.getVal() << endl;
    return 0;
}
```

15.13 What will the following program display?

```
#include <iostream>
using namespace std;

class First
{
protected:
    int a;
public:
    First(int x = 1)
        { a = x; }

    void twist()
        { a *= 2; }
    int getVal()
        { twist(); return a; }
};
```

```
class Second : public First
{
private:
    int b;
public:
    Second(int y = 5)
        { b = y; }

    void twist()
        { b *= 10; }
};

int main()
{
    First object1;
    Second object2;

    cout << object1.getVal() << endl;
    cout << object2.getVal() << endl;
    return 0;
}
```

- 15.14 What will the following program display?

```
#include <iostream>
using namespace std;

class First
{
protected:
    int a;
public:
    First(int x = 1)
        { a = x; }

    virtual void twist()
        { a *= 2; }

    int getVal()
        { twist(); return a; }
};

class Second : public First
{
private:
    int b;
public:
    Second(int y = 5)
        { b = y; }
    virtual void twist()
        { b *= 10; }
};

int main()
{
    First object1;
    Second object2;
```

```

        cout << object1.getVal() << endl;
        cout << object2.getVal() << endl;
        return 0;
    }
}

```

- 15.15 What will the following program display?

```

#include <iostream>
using namespace std;

class Base
{
protected:
    int baseVar;
public:
    Base(int val = 2)
        { baseVar = val; }

    int getVar()
        { return baseVar; }
};

class Derived : public Base
{
private:
    int derivedVar;

public:
    Derived(int val = 100)
        { derivedVar = val; }
    int getVar()
        { return derivedVar; }
};

int main()
{
    Base *optr = nullptr;
    Derived object;

    optr = &object;
    cout << optr->getVar() << endl;
    return 0;
}

```

15.8 Multiple Inheritance

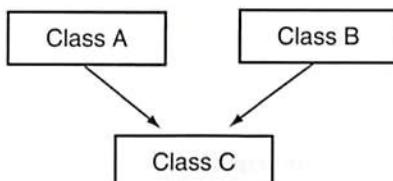
CONCEPT: Multiple inheritance is when a derived class has two or more base classes.

Previously, we discussed how a class may be derived from a second class that is itself derived from a third class. The series of classes establishes a chain of inheritance. In such a scheme, you might be tempted to think of the lowest class in the chain as having multiple base classes. A base class, however, should be thought of as the class that another class is

directly derived from. Even though there may be several classes in a chain, each class (below the topmost class) only has one base class.

Another way of combining classes is through multiple inheritance. *Multiple inheritance* is when a class has two or more base classes. This is illustrated in Figure 15-6.

Figure 15-6 Multiple inheritance



In Figure 15-6, class C is directly derived from classes A and B and inherits the members of both. Neither class A nor B, however, inherits members from the other. Their members are only passed down to class C. Let's look at an example of multiple inheritance. Consider the two classes declared here:

Contents of Date.h

```
1 // Specification file for the Date class
2 #ifndef DATE_H
3 #define DATE_H
4
5 class Date
6 {
7 protected:
8     int day;
9     int month;
10    int year;
11 public:
12     // Default constructor
13     Date(int d, int m, int y)
14         { day = 1; month = 1; year = 1900; }
15
16     // Constructor
17     Date(int d, int m, int y)
18         { day = d; month = m; year = y; }
19
20     // Accessors
21     int getDay() const
22         { return day; }
23
24     int getMonth() const
25         { return month; }
26
27     int getYear() const
28         { return year; }
29 };
30#endif
```

Contents of Time.h

```

1 // Specification file for the Time class
2 #ifndef TIME_H
3 #define TIME_H
4
5 class Time
6 {
7 protected:
8     int hour;
9     int min;
10    int sec;
11 public:
12     // Default constructor
13     Time()
14         { hour = 0; min = 0; sec = 0; }
15
16     // Constructor
17     Time(int h, int m, int s)
18         { hour = h; min = m; sec = s; }
19
20     // Accessor functions
21     int getHour() const
22         { return hour; }
23
24     int getMin() const
25         { return min; }
26
27     int getSec() const
28         { return sec; }
29 };
30 #endif

```

These classes are designed to hold integers that represent the date and time. They both can be used as base classes for a third class we will call **DateTime**:

Contents of DateTime.h

```

1 // Specification file for the DateTime class
2 #ifndef DATETIME_H
3 #define DATETIME_H
4 #include <string>
5 #include "Date.h"
6 #include "Time.h"
7 using namespace std;
8
9 class DateTime : public Date, public Time
10 {
11 public:
12     // Default constructor
13     DateTime();
14
15     // Constructor
16     DateTime(int, int, int, int, int, int);

```

```
17
18     // The showDateTime function displays the
19     // date and the time.
20     void showDateTime() const;
21 };
22 #endif
```

In line 9, the first line in the `DateTime` declaration reads

```
class DateTime : public Date, public Time
```

Notice there are two base classes listed, separated by a *comma*. Each base class has its own access specification. The general format of the first line of a class declaration with multiple base classes is

```
class DerivedClassName : AccessSpecification BaseClassName,
AccessSpecification BaseClassName [, ...]
```

The notation in the square brackets indicates that the list of base classes with their access specifications may be repeated. (It is possible to have several base classes.)

Contents of `DateTime.cpp`

```
1 // Implementation file for the DateTime class
2 #include <iostream>
3 #include <string>
4 #include "DateTime.h"
5 using namespace std;
6
7 //*****
8 // Default constructor
9 // Note that this constructor does nothing other
10 // than call default base class constructors.
11 //*****
12 DateTime::DateTime() : Date(), Time()
13 {}
14
15 //*****
16 // Constructor
17 // Note that this constructor does nothing other
18 // than call base class constructors.
19 //*****
20 DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :
21     Date(dy, mon, yr), Time(hr, mt, sc)
22 {}
23
24 //*****
25 // The showDateTime member function displays the
26 // date and the time.
27 //*****
28 void DateTime::showDateTime() const
29 {
30     // Display the date in the form MM/DD/YYYY.
31     cout << getMonth() << "/" << getDay() << "/" << getYear() << " ";
```

```

32
33     // Display the time in the form HH:MM:SS.
34     cout << getHour() << ":" << getMin() << ":" << getSec() << endl;
35 }

```

The class has two constructors: a default constructor and a constructor that accepts arguments for each component of a date and time. Let's look at the function header for the default constructor, in line 12:

```
DateTime::DateTime() : Date(), Time()
```

After the `DateTime` constructor's parentheses is a colon, followed by calls to the `Date` constructor and the `Time` constructor. The calls are separated by a comma. When using multiple inheritance, the general format of a derived class's constructor header is

```
DerivedClassName(ParameterList) : BaseClassName(ArgumentList),  
BaseClassName(ArgumentList)[, ...]
```

Look at the function header for the second constructor, which appears in lines 20 and 21:

```
DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :  
    Date(dy, mon, yr), Time(hr, mt, sc)
```

This `DateTime` constructor accepts arguments for the day (`dy`), month (`mon`), year (`yr`), hour (`hr`), minute (`mt`), and second (`sc`). The `dy`, `mon`, and `yr` parameters are passed as arguments to the `Date` constructor. The `hr`, `mt`, and `sc` parameters are passed as arguments to the `Time` constructor.

The order that the base class constructor calls appear in the list does not matter. They are always called in the order of inheritance. That is, they are always called in the order they are listed in the first line of the class declaration. Here is line 9 from the `DateTime.h` file:

```
class DateTime : public Date, public Time
```

Because `Date` is listed before `Time` in the `DateTime` class declaration, the `Date` constructor will always be called first. If the classes use destructors, they are always called in reverse order of inheritance. Program 15-20 shows these classes in use.

Program 15-20

```

1 // This program demonstrates a class with multiple inheritance.
2 #include "DateTime.h"
3 using namespace std;
4
5 int main()
6 {
7     // Define a DateTime object and use the default
8     // constructor to initialize it.
9     DateTime emptyDay;
10
11    // Display the object's date and time.
12    emptyDay.showDateTime();
13
14    // Define a DateTime object and initialize it
15    // with the date 2/4/1960 and the time 5:32:27.

```

```
16     DateTime pastDay(2, 4, 1960, 5, 32, 27);  
17  
18     // Display the object's date and time.  
19     pastDay.showDateTime();  
20     return 0;  
21 }
```

Program Output

```
1/1/1900 0:0:0  
4/2/1960 5:32:27
```

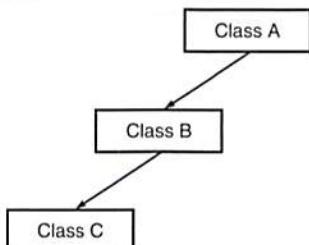


NOTE: It should be noted that multiple inheritance opens the opportunity for a derived class to have ambiguous members. That is, two base classes may have member variables or functions of the same name. In situations like these, the derived class should always redefine or override the member functions. Calls to the member functions of the appropriate base class can be performed within the derived class using the scope resolution operator (::). The derived class can also access the ambiguously named member variables of the correct base class using the scope resolution operator. If these steps aren't taken, the compiler will generate an error when it can't tell which member is being accessed.

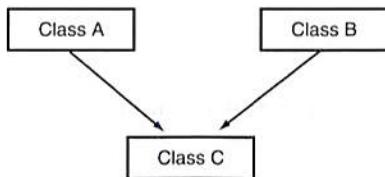


Checkpoint

15.16 Does the following diagram depict multiple inheritance or a chain of inheritance?



15.17 Does the following diagram depict multiple inheritance or a chain of inheritance?



15.18 Examine the following classes. The table lists the variables that are members of the Third class (some are inherited). Complete the table by filling in the access specification each member will have in the Third class. Write “inaccessible” if a member is inaccessible to the Third class.

```
class First  
{  
    private:
```

```

        int a;
protected:
    double b;
public:
    long c;
};

class Second : protected First
{
private:
    int d;
protected:
    double e;
public:
    long f;
};

class Third : public Second
{
private:
    int g;
protected:
    double h;
public:
    long i;
};

```

Member Variable	Access Specification in Third Class
a	
b	
c	
d	
e	
f	
g	
h	
i	

15.19 Examine the following class declarations:

```

class Van
{
protected:
    int passengers;
public:
    Van(int p)
    { passengers = p; }
};

```

```
class FourByFour
{
protected:
    double cargoWeight;
public:
    FourByFour(float w)
    { cargoWeight = w; }
};
```

Write the declaration of a class named `SportUtility`. The class should be derived from both the `Van` and `FourByFour` classes above. (This should be a case of multiple inheritance, where both `Van` and `FourByFour` are base classes.)

Review Questions and Exercises

Short Answer

1. What is an “is a” relationship?
2. A program uses two classes: `Dog` and `Poodle`. Which class is the base class, and which is the derived class?
3. How does base class access specification differ from class member access specification?
4. What is the difference between a protected class member and a private class member?
5. Can a derived class ever directly access the private members of its base class?
6. Which constructor is called first, that of the derived class or the base class?
7. What is the difference between redefining a base class function and overriding a base class function?
8. When does static binding take place? When does dynamic binding take place?
9. What is an abstract base class?
10. A program has a class `Potato`, which is derived from the class `Vegetable`, which is derived from the class `Food`. Is this an example of multiple inheritance? Why or why not?
11. What base class is named in the line below?
`class Pet : public Dog`
12. What derived class is named in the line below?
`class Pet : public Dog`
13. What is the class access specification of the base class named below?
`class Pet : public Dog`
14. What is the class access specification of the base class named below?
`class Pet : Fish`
15. Protected members of a base class are like _____ members, except they may be accessed by derived classes.
16. Complete the table on the next page by filling in private, protected, public, or inaccessible in the right-hand column:

In a private base class, this base class MEMBER access specification...	...becomes this access specification in the derived class.
--	---

private
protected
public

17. Complete the table below by filling in private, protected, public, or inaccessible in the right-hand column:

In a protected base class, this base class MEMBER access specification...	...becomes this access specification in the derived class.
--	---

private
protected
public

18. Complete the table below by filling in private, protected, public, or inaccessible in the right-hand column:

In a public base class, this base class MEMBER access specification...	...becomes this access specification in the derived class.
---	---

private
protected
public

Fill-in-the-Blank

19. A derived class inherits the _____ of its base class.
20. When both a base class and a derived class have constructors, the base class's constructor is called _____ (first/last).
21. When both a base class and a derived class have destructors, the base class's constructor is called _____ (first/last).
22. An overridden base class function may be called by a function in a derived class by using the _____ operator.
23. When a derived class redefines a function in a base class, which version of the function do objects that are defined of the base class call? _____
24. A(n) _____ member function in a base class expects to be overridden in a derived class.
25. _____ binding is when the compiler binds member function calls at compile time.
26. _____ binding is when a function call is bound at runtime.
27. _____ is when member functions in a class hierarchy behave differently, depending upon which object performs the call.
28. When a pointer to a base class is made to point to a derived class, the pointer ignores any _____ the derived class performs, unless the function is _____.

29. A(n) _____ class cannot be instantiated.
30. A(n) _____ function has no body, or definition, in the class in which it is declared.
31. A(n) _____ of inheritance is where one class is derived from a second class, which in turn is derived from a third class.
32. _____ is where a derived class has two or more base classes.
33. In multiple inheritance, the derived class should always _____ a function that has the same name in more than one base class.

Algorithm Workbench

34. Write the first line of the declaration for a `Poodle` class. The class should be derived from the `Dog` class with public base class access.
35. Write the first line of the declaration for a `SoundSystem` class. Use multiple inheritance to base the class on the `CDplayer` class, the `Tuner` class, and the `MP3Player` class. Use public base class access in all cases.
36. Suppose a class named `Tiger` is derived from both the `Felis` class and the `Carnivore` class. Here is the first line of the `Tiger` class declaration:

```
class Tiger : public Felis, public Carnivore
```

Here is the function header for the `Tiger` constructor:

```
Tiger(int x, int y) : Carnivore(x), Felis(y)
```

Which base class constructor is called first, `Carnivore` or `Felis`?

37. Write the declaration for class `B`. The class's members should be as follows:
 - `m`: an integer. This variable should not be accessible to code outside the class or to member functions in any class derived from class `B`.
 - `n`: an integer. This variable should not be accessible to code outside the class, but should be accessible to member functions in any class derived from class `B`.
 - `setM`, `getM`, `setN`, and `getN`: These are the set and get functions for the member variables `m` and `n`. These functions should be accessible to code outside the class.
 - `calc`: a public virtual member function that returns the value of `m` times `n`.

Next, write the declaration for class `D`, which is derived from class `B`. The class's members should be as follows:

- `q`: a `float`. This variable should not be accessible to code outside the class but should be accessible to member functions in any class derived from class `D`.
- `r`: a `float`. This variable should not be accessible to code outside the class, but should be accessible to member functions in any class derived from class `D`.
- `setQ`, `getQ`, `setR`, and `getR`: These are the set and get functions for the member variables `q` and `r`. These functions should be accessible to code outside the class.
- `calc`: a public member function that overrides the base class `calc` function. This function should return the value of `q` times `r`.

True or False

38. T F The base class's access specification affects the way base class member functions may access base class member variables.
39. T F The base class's access specification affects the way the derived class inherits members of the base class.

40. T F Private members of a private base class become inaccessible to the derived class.
41. T F Public members of a private base class become private members of the derived class.
42. T F Protected members of a private base class become public members of the derived class.
43. T F Public members of a protected base class become private members of the derived class.
44. T F Private members of a protected base class become inaccessible to the derived class.
45. T F Protected members of a public base class become public members of the derived class.
46. T F The base class constructor is called after the derived class constructor.
47. T F The base class destructor is called after the derived class destructor.
48. T F It isn't possible for a base class to have more than one constructor.
49. T F Arguments are passed to the base class constructor by the derived class constructor.
50. T F A member function of a derived class may not have the same name as a member function of the base class.
51. T F Pointers to a base class may be assigned the address of a derived class object.
52. T F A base class may not be derived from another class.

Find the Errors

Each of the class declarations and/or member function definitions below has errors. Find as many as you can.

```

53. class Car, public Vehicle
{
    public:
        Car();
        ~Car();
    protected:
        int passengers;
}

54. class Truck, public : Vehicle, protected
{
    private:
        double cargoWeight;
    public:
        Truck();
        ~Truck();
};

55. class SnowMobile : Vehicle
{
    protected:

```

```

        int horsePower;
        double weight;
    public:
        SnowMobile(int h, double w), Vehicle(h)
        { horsePower = h; }
        ~SnowMobile();
    };

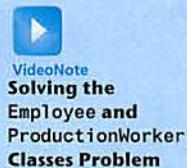
56. class Table : public Furniture
{
protected:
    int numSeats;
public:
    Table(int n) : Furniture(numSeats)
    { numSeats = n; }
    ~Table();
};

57. class Tank : public Cylinder
{
private:
    int fuelType;
    double gallons;
public:
    Tank();
    ~Tank();
    void setContents(double);
    void setContents(double);
};

58. class Three : public Two : public One
{
protected:
    int x;
public:
    Three(int a, int b, int c), Two(b), Three(c)
    { x = a; }
    ~Three();
};

```

Programming Challenges



1. Employee and ProductionWorker Classes

Design a class named `Employee`. The class should keep the following information:

- Employee name
- Employee number
- Hire date

Write one or more constructors, and the appropriate accessor and mutator functions, for the class.

Next, write a class named `ProductionWorker` that is derived from the `Employee` class. The `ProductionWorker` class should have member variables to hold the following information:

- Shift (an integer)
- Hourly pay rate (a double)

The workday is divided into two shifts: day and night. The shift variable will hold an integer value representing the shift that the employee works. The day shift is shift 1, and the night shift is shift 2. Write one or more constructors, and the appropriate accessor and mutator functions, for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

2. `ShiftSupervisor` Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that is derived from the `Employee` class you created in Programming Challenge 1 (Employee and Production Worker Classes). The `ShiftSupervisor` class should have a member variable that holds the annual salary, and a member variable that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3. `TeamLeader` Class

In a particular factory, a team leader is an hourly paid production worker who leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that extends the `ProductionWorker` class you designed in Programming Challenge 1 (Employee and Production Worker Classes). The `TeamLeader` class should have member variables for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

4. Time Format

In Program 15-20, the file `Time.h` contains a `Time` class. Design a class called `MilTime` that is derived from the `Time` class. The `MilTime` class should convert time in military (24-hour) format to the standard time format used by the `Time` class. The class should have the following member variables:

`milHours`: Contains the hour in 24-hour format. For example, 1:00 p.m. would be stored as 1300 hours, and 4:30 p.m. would be stored as 1630 hours.

`milSeconds`: Contains the seconds in standard format.

The class should have the following member functions:

Constructor: The constructor should accept arguments for the hour and seconds, in military format. The time should then be converted to standard time and stored in the `hours`, `min`, and `sec` variables of the `Time` class.

setTime: Accepts arguments to be stored in the `milHours` and `milSeconds` variables. The time should then be converted to standard time and stored in the `hours`, `min`, and `sec` variables of the `Time` class.

`getHour`: Returns the hour in military format.

`getStandHr`: Returns the hour in standard format.

Demonstrate the class in a program that asks the user to enter the time in military format. The program should then display the time in both military and standard format.

Input Validation: The `MilTime` class should not accept hours greater than 2359, or less than 0. It should not accept seconds greater than 59 or less than 0.

5. Time Clock

Design a class named `TimeClock`. The class should be derived from the `MilTime` class you designed in Programming Challenge 4 (Time Format). The class should allow the programmer to pass two times to it: starting time and ending time. The class should have a member function that returns the amount of time elapsed between the two times. For example, if the starting time is 900 hours (9:00 a.m.), and the ending time is 1300 hours (1:00 p.m.), the elapsed time is 4 hours.

Input Validation: The class should not accept hours greater than 2359 or less than 0.

6. Essay Class

Design an `Essay` class that is derived from the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives on an essay. The student's essay score can be up to 100, and is determined in the following manner:

- Grammar: 30 points
- Spelling: 20 points
- Correct length: 20 points
- Content: 30 points

Demonstrate the class in a simple program.

7. PersonData and CustomerData Classes

Design a class named `PersonData` with the following member variables:

- `lastName`
- `firstName`
- `address`
- `city`
- `state`
- `zip`
- `phone`

Write the appropriate accessor and mutator functions for these member variables.

Next, design a class named `CustomerData`, which is derived from the `PersonData` class. The `CustomerData` class should have the following member variables:

- `customerNumber`
- `mailingList`

The `customerNumber` variable will be used to hold a unique integer for each customer. The `mailingList` variable should be a `bool`. It will be set to `true` if the customer wishes to be on a mailing list, or `false` if the customer does not wish to be on a mailing list. Write appropriate accessor and mutator functions for these member variables. Demonstrate an object of the `CustomerData` class in a simple program.

8. PreferredCustomer Class

A retail store has a preferred customer plan where customers may earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store.

- When a preferred customer spends \$500, he or she gets a 5 percent discount on all future purchases.
- When a preferred customer spends \$1,000, he or she gets a 6 percent discount on all future purchases.
- When a preferred customer spends \$1,500, he or she gets a 7 percent discount on all future purchases.
- When a preferred customer spends \$2,000 or more, he or she gets a 10 percent discount on all future purchases.

Design a class named `PreferredCustomer`, which is derived from the `CustomerData` class you created in Programming Challenge 7. The `PreferredCustomer` class should have the following member variables:

- `purchasesAmount` (a `double`)
- `discountLevel` (a `double`)

The `purchasesAmount` variable holds the total of a customer's purchases to date. The `discountLevel` variable should be set to the correct discount percentage, according to the store's preferred customer plan. Write appropriate member functions for this class and demonstrate it in a simple program.

***Input Validation:** Do not accept negative values for any sales figures.*

9. File Filter

A file filter reads an input file, transforms it in some way, and writes the results to an output file. Write an abstract file filter class that defines a pure virtual function for transforming a character. Create one derived class of your file filter class that performs encryption, another that transforms a file to all uppercase, and another that creates an unchanged copy of the original file. The class should have the following member function:

```
void doFilter(ifstream &in, ofstream &out)
```

This function should be called to perform the actual filtering. The member function for transforming a single character should have the prototype:

```
char transform(char ch)
```

The encryption class should have a constructor that takes an integer as an argument and uses it as the encryption key.

10. File Double-Spacer

Create a derived class of the abstract filter class of Programming Challenge 9 (File Filter) that double-spaces a file, that is, it inserts a blank line between any two lines of the file.

11. Course Grades

In a course, a teacher gives the following tests and assignments:

- A lab activity that is observed by the teacher and assigned a numeric score.
- A pass/fail exam that has ten questions. The minimum passing score is 70.
- An essay that is assigned a numeric score.
- A final exam that has 50 questions.

Write a class named `CourseGrades`. The class should have a member named `grades` that is an array of `GradedActivity` pointers. The `grades` array should have four elements, one for each of the assignments previously described. The class should have the following member functions:

- `setLab:` This function should accept the address of a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` array should reference this object.
- `setPassFailExam:` This function should accept the address of a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` array should reference this object.
- `setEssay:` This function should accept the address of an `Essay` object as its argument. (See Programming Challenge 6 for the `Essay` class. If you have not completed Programming Challenge 6, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` array should reference this object.
- `setPassFailExam:` This function should accept the address of a `FinalExam` object as its argument. This object should already hold the student's score for the final exam. Element 3 of the `grades` array should reference this object.
- `print:` This function should display the numeric scores and grades for each element in the `grades` array.

Demonstrate the class in a program.

12. `Ship`, `CruiseShip`, and `CargoShip` Classes

Design a `Ship` class that has the following members:

- A member variable for the name of the ship (a string)
- A member variable for the year that the ship was built (a string)
- A constructor and appropriate accessors and mutators
- A virtual `print` function that displays the ship's name and the year it was built.

Design a `CruiseShip` class that is derived from the `Ship` class. The `CruiseShip` class should have the following members:

- A member variable for the maximum number of passengers (an `int`)
- A constructor and appropriate accessors and mutators
- A `print` function that overrides the `print` function in the base class. The `CruiseShip` class's `print` function should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that is derived from the `Ship` class. The `CargoShip` class should have the following members:

- A member variable for the cargo capacity in tonnage (an `int`)
- A constructor and appropriate accessors and mutators
- A `print` function that overrides the `print` function in the base class. The `CargoShip` class's `print` function should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has an array of `Ship` pointers. The array elements should be initialized with the addresses of dynamically allocated `Ship`, `CruiseShip`, and `CargoShip` objects. (See Program 15-14, lines 17 through 22, for an example of how to do this.) The program should then step through the array, calling each object's `print` function.

13. Pure Abstract Base Class Project

Define a pure abstract base class called `BasicShape`. The `BasicShape` class should have the following members:

Private Member Variable:

`area`: A `double` used to hold the shape's area.

Public Member Functions:

`getArea`: This function should return the value in the member variable `area`.

`calcArea`: This function should be a pure virtual function.

Next, define a class named `Circle`. It should be derived from the `BasicShape` class. It should have the following members:

Private Member Variables:

`centerX`: a long integer used to hold the x coordinate of the circle's center

`centerY`: a long integer used to hold the y coordinate of the circle's center

`radius`: a double used to hold the circle's radius

Public Member Functions:

constructor: accepts values for `centerX`, `centerY`, and `radius`. Should call the overridden `calcArea` function described below.

`getCenterX`: returns the value in `centerX`

`getCenterY`: returns the value in `centerY`

`calcArea`: calculates the area of the circle ($\text{area} = 3.14159 * \text{radius} * \text{radius}$) and stores the result in the inherited member `area`.

Next, define a class named `Rectangle`. It should be derived from the `BasicShape` class. It should have the following members:

Private Member Variables:

`width`: a long integer used to hold the width of the rectangle

`length`: a long integer used to hold the length of the rectangle

Public Member Functions:

constructor: accepts values for `width` and `length`. Should call the overridden `calcArea` function described below.

`getWidth`: returns the value in `width`.

`getLength`: returns the value in `length`.

`calcArea`: calculates the area of the rectangle ($\text{area} = \text{length} * \text{width}$) and stores the result in the inherited member `area`.

After you have created these classes, create a driver program that defines a `Circle` object and a `Rectangle` object. Demonstrate that each object properly calculates and reports its area.

Group Project

14. Bank Accounts

This program should be designed and written by a team of students. Here are some suggestions:

- One or more students may work on a single class.
- The requirements of the program should be analyzed so that each student is given about the same work load.
- The parameters and return types of each function and class member function should be decided in advance.
- The program will be best implemented as a multi-file program.

Design a generic class to hold the following information about a bank account:

- Balance
- Number of deposits this month
- Number of withdrawals
- Annual interest rate
- Monthly service charges

The class should have the following member functions:

`constructor:` Accepts arguments for the balance and annual interest rate.

`deposit:` A virtual function that accepts an argument for the amount of the deposit. The function should add the argument to the account balance. It should also increment the variable holding the number of deposits.

`withdraw:` A virtual function that accepts an argument for the amount of the withdrawal. The function should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.

`calcInt:` A virtual function that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:

$$\text{Monthly Interest Rate} = (\text{Annual Interest Rate} / 12)$$

$$\text{Monthly Interest} = \text{Balance} * \text{Monthly Interest Rate}$$

$$\text{Balance} = \text{Balance} + \text{Monthly Interest}$$

`monthlyProc:` A virtual function that subtracts the monthly service charges from the balance, calls the `calcInt` function, then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.

Next, design a savings account class, derived from the generic account class. The savings account class should have the following additional member:

`status` (to represent an active or inactive account)

If the balance of a savings account falls below \$25, it becomes inactive. (The `status` member could be a flag variable.) No more withdrawals may be made until the balance is raised above \$25, at which time the account becomes active again. The savings account class should have the following member functions:

- withdraw:** A function that checks to see if the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the base class version of the function.
- deposit:** A function that checks to see if the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above \$25, the account becomes active again. The deposit is then made by calling the base class version of the function.
- monthlyProc:** Before the base class function is called, this function checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of \$1 for each withdrawal above 4 is added to the base class variable that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below \$25, the account becomes inactive.)

Next, design a checking account class, also derived from the generic account class. It should have the following member functions:

- withdraw:** Before the base class function is called, this function will determine if a withdrawal (a check written) will cause the balance to go below \$0. If the balance goes below \$0, a service charge of \$15 will be taken from the account. (The withdrawal will not be made.) If there isn't enough in the account to pay the service charge, the balance will become negative and the customer will owe the negative amount to the bank.
- monthlyProc:** Before the base class function is called, this function adds the monthly fee of \$5 plus \$0.10 per withdrawal (check written) to the base class variable that holds the monthly service charges.

Write a complete program that demonstrates these classes by asking the user to enter the amounts of deposits and withdrawals for a savings account and checking account. The program should display statistics for the month, including beginning balance, total amount of deposits, total amount of withdrawals, service charges, and ending balance.



NOTE: You may need to add more member variables and functions to the classes than those listed above.

TOPICS

- | | |
|-------------------------|--|
| 16.1 Exceptions | 16.3 Focus on Software Engineering: Where to Start When Defining Templates |
| 16.2 Function Templates | 16.4 Class Templates |

16.1 Exceptions

CONCEPT: Exceptions are used to signal errors or unexpected events that occur while a program is running.

Error testing is usually a straightforward process involving if statements or other control mechanisms. For example, the following code segment will trap a division-by-zero error before it occurs:

```
if (denominator == 0)
    cout << "ERROR: Cannot divide by zero.\n";
else
    quotient = numerator / denominator;
```

But what if similar code is part of a function that returns the quotient, as in the following example?

```
// An unreliable division function
double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        cout << "ERROR: Cannot divide by zero.\n";
        return 0;
    }
    else
        return static_cast<double>(numerator) / denominator;
}
```



Functions commonly signal error conditions by returning a predetermined value. Apparently, the function in this example returns 0 when division by zero has been attempted. This is unreliable, however, because 0 is a valid result of a division operation. Even though the function displays an error message, the part of the program that calls the function will not know when an error has occurred. Problems like these require sophisticated error handling techniques.

Throwing an Exception

One way of handling complex error conditions is with *exceptions*. An exception is a value or an object that signals an error. When the error occurs, an exception is “thrown.” For example, the following code shows the *divide* function, modified to throw an exception when division by zero has been attempted:



```
double divide(int numerator, int denominator)
{
    if (denominator == 0)
        throw "ERROR: Cannot divide by zero.\n";
    else
        return static_cast<double>(numerator) / denominator;
}
```

The following statement causes the exception to be thrown:

```
throw "ERROR: Cannot divide by zero.\n";
```

The *throw* key word is followed by an argument, which can be any value. As you will see, the value of the argument is used to determine the nature of the error. The function above simply throws a string containing an error message.

The line containing a *throw* statement is known as the *throw point*. When a *throw* statement is executed, control is passed to another part of the program known as an *exception handler*. When an exception is thrown by a function, the function aborts.

Handling an Exception

To handle an exception, a program must have a *try/catch* construct. The general format of the *try/catch* construct is:

```
try
{
    // code here calls functions or object member
    // functions that might throw an exception.
}
catch(ExceptionParameter)
{
    // code here handles the exception
}
// Repeat as many catch blocks as needed.
```



The first part of the construct is the *try block*. This starts with the key word *try* and is followed by a block of code executing any statements that might directly or indirectly cause an exception to be thrown. The *try block* is immediately followed by one or more *catch blocks*,

which are the exception handlers. A catch block starts with the key word `catch`, followed by a set of parentheses containing the definition of an exception parameter. For example, here is a try/catch construct that can be used with the `divide` function:

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
    cout << exceptionString;
}
```

Because the `divide` function throws an exception whose value is a string, there must be an exception handler that catches a string. The catch block shown catches the error message in the `exceptionString` parameter then displays it with `cout`. Now let's look at an entire program to see how `throw`, `try`, and `catch` work together. In the first sample run of Program 16-1, valid data are given. This shows how the program should run with no errors. In the second sample running, a denominator of 0 is given. This shows the result of the exception being thrown.

Program 16-1

```
1 // This program demonstrates an exception being thrown and caught.
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 // Function prototype
7 double divide(int, int);
8
9 int main()
10 {
11     int num1, num2; // To hold two numbers
12     double quotient; // To hold the quotient of the numbers
13
14     // Get two numbers.
15     cout << "Enter two numbers: ";
16     cin >> num1 >> num2;
17
18     // Divide num1 by num2 and catch any
19     // potential exceptions.
20     try
21     {
22         quotient = divide(num1, num2);
23         cout << "The quotient is " << quotient << endl;
24     }
25     catch (string exceptionString)
26     {
27         cout << exceptionString;
28     }
29 }
```

(program continues)

Program 16-1

(continued)

```

30     cout << "End of the program.\n";
31     return 0;
32 }
33
34 //*****
35 // The divide function divides the numerator *
36 // by the denominator. If the denominator is *
37 // zero, the function throws an exception. *
38 //*****
39
40 double divide(int numerator, int denominator)
41 {
42     if (denominator == 0)
43     {
44         string exceptionString = "ERROR: Cannot divide by zero.\n";
45         throw exceptionString;
46     }
47
48     return static_cast<double>(numerator) / denominator;
49 }
```

Program Output with Example Input Shown in BoldEnter two numbers: **12 2**

The quotient is 6

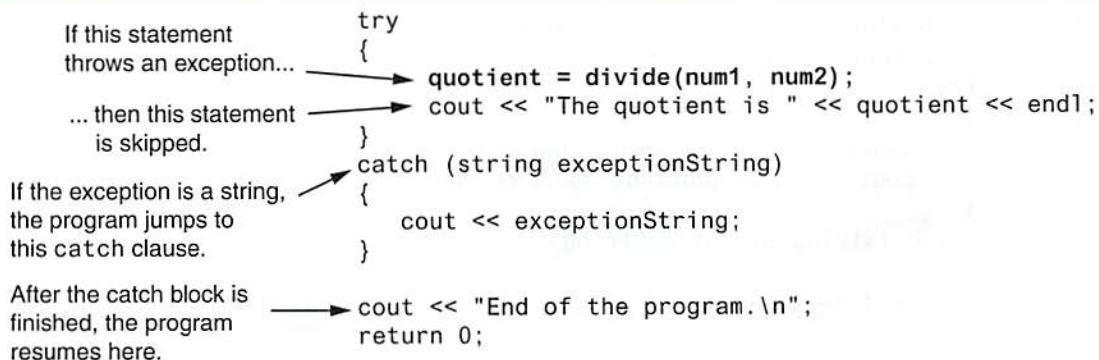
End of the program.

Program Output with Different Example Input Shown in BoldEnter two numbers: **12 0**

ERROR: Cannot divide by zero.

End of the program.

As you can see from the second output screen, the exception caused the program to jump out of the divide function and into the catch block. After the catch block has finished, the program resumes with the first statement after the try/catch construct. This is illustrated in Figure 16-1.

Figure 16-1 How the try/catch construct works

In the first output screen, the user entered nonnegative values. No exception was thrown in the try block, so the program skipped the catch block and jumped to the statement immediately following the try/catch construct, which is in line 30. This is illustrated in Figure 16-2.

Figure 16-2 No exception was thrown

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
    cout << exceptionString;
}
cout << "End of the program.\n";
return 0;
```

What if an Exception Is Not Caught?

There are two possible ways for a thrown exception to go uncaught. The first possibility is for the try/catch construct to contain no catch blocks with an exception parameter of the right data type. The second possibility is for the exception to be thrown from outside a try block. In either case, the exception will cause the entire program to abort execution.

Object-Oriented Exception Handling with Classes

Now that you have an idea of how the exception mechanism in C++ works, we will examine an object-oriented approach to exception handling. Recall the `Rectangle` class that was introduced in Chapter 13. That class had the mutator functions `setWidth` and `setLength` for setting the rectangle's width and length. If a negative value was passed to either of these functions, the class displayed an error message and aborted the program. The following code shows an improved version of the `Rectangle` class. This version throws an exception when a negative value is passed to `setWidth` or `setLength`. (These files can be found in the Student Source Code Folder Chapter 16\Rectangle Version 1.)

Contents of `Rectangle.h` (Version 1)

```
1 // Specification file for the Rectangle class
2 #ifndef RECTANGLE_H
3 #define RECTANGLE_H
4
5 class Rectangle
6 {
7     private:
8         double width;      // The rectangle's width
9         double length;     // The rectangle's length
10    public:
11        // Exception class
12        class NegativeSize
13            { };           // Empty class declaration
14
```

```

15         // Default constructor
16         Rectangle()
17         { width = 0.0; length = 0.0; }
18
19         // Mutator functions, defined in Rectangle.cpp
20         void setWidth(double);
21         void setLength(double);
22
23         // Accessor functions
24         double getWidth() const
25             { return width; }
26
27         double getLength() const
28             { return length; }
29
30         double getArea() const
31             { return width * length; }
32     };
33 #endif

```

Notice the empty class declaration that appears in the public section, in lines 12 and 13. The NegativeSize class has no members. The only important part of the class is its name, which will be used in the exception-handling code. Now look at the Rectangle.cpp file, where the setWidth and setLength member functions are defined.

Contents of Rectangle.cpp (Version 1)

```

1 // Implementation file for the Rectangle class.
2 #include "Rectangle.h"
3
4 //*****
5 // setWidth sets the value of the member variable width. *
6 //*****
7
8 void Rectangle::setWidth(double w)
9 {
10     if (w >= 0)
11         width = w;
12     else
13         throw NegativeSize();
14 }
15
16 //*****
17 // setLength sets the value of the member variable length. *
18 //*****
19
20 void Rectangle::setLength(double len)
21 {
22     if (len >= 0)
23         length = len;
24     else
25         throw NegativeSize();
26 }

```

In the `setWidth` function, the parameter `w` is tested by the `if` statement in line 10. If `w` is greater than or equal to 0, its value is assigned to the `width` member variable. If `w` holds a negative number, however, the statement in line 13 is executed.

```
throw NegativeSize();
```

The `throw` statement's argument, `NegativeSize()`, causes an instance of the `NegativeSize` class to be created and thrown as an exception.

The same series of events takes place in the `setLength` function. If the value in the `len` parameter is greater than or equal to 0, its value is assigned to the `length` member variable. If `len` holds a negative number, an instance of the `NegativeSize` class is thrown as an exception in line 25.

This way of reporting errors is much more graceful than simply aborting the program. Any code that uses the `Rectangle` class must simply have a catch block to handle the `NegativeSize` exceptions that the `Rectangle` class might throw. Program 16-2 shows an example. (This file can be found in the Student Source Code Folder Chapter 16\Rectangle Version 1.)

Program 16-2

```
1 // This program demonstrates Rectangle class exceptions.
2 #include <iostream>
3 #include "Rectangle.h"
4 using namespace std;
5
6 int main()
7 {
8     double width;
9     double length;
10
11    // Create a Rectangle object.
12    Rectangle myRectangle;
13
14    // Get the width and length.
15    cout << "Enter the rectangle's width: ";
16    cin >> width;
17    cout << "Enter the rectangle's length: ";
18    cin >> length;
19
20    // Store these values in the Rectangle object.
21    try
22    {
23        myRectangle.setWidth(width);
24        myRectangle.setLength(length);
25        cout << "The area of the rectangle is "
26            << myRectangle.getArea() << endl;
27    }
28    catch (Rectangle::NegativeSize)
29    {
30        cout << "Error: A negative value was entered.\n";
31    }
```

(program continues)

Program 16-2 (continued)

```

32     cout << "End of the program.\n";
33
34     return 0;
35 }
```

Program Output with Example Input Shown in Bold

Enter the rectangle's width: **10**
 Enter the rectangle's length: **20**
 The area of the rectangle is 200
 End of the program.

Program Output with Different Example Input Shown in Bold

Enter the rectangle's width: **5**
 Enter the rectangle's length: **-5**
 Error: A negative value was entered.
 End of the program.

The catch statement in line 28 catches the `NegativeSize` exception when it is thrown by any of the statements in the try block. Inside the catch statement's parentheses is the name of the `NegativeSize` class. Because the `NegativeSize` class is declared inside the `Rectangle` class, we have to fully qualify the class name with the scope resolution operator.

Notice we did not define a parameter of the `NegativeSize` class in the catch statement. In this case, the catch statement only needs to specify the type of exception it handles.

Multiple Exceptions

The programs we have studied so far test only for a single type of error and throw only a single type of exception. In many cases, a program will need to test for several different types of errors and signal which one has occurred. C++ allows you to throw and catch multiple exceptions. The only requirement is that each different exception be of a different type. You then code a separate catch block for each type of exception that may be thrown in the try block.

For example, suppose we wish to expand the `Rectangle` class so it throws one type of exception when a negative value is specified for the `width`, and another type of exception when a negative value is specified for the `length`. First, we declare two different exception classes, such as:

```

// Exception class for a negative width
class NegativeWidth
{
};

// Exception class for a negative length
class NegativeLength
{
};
```

An instance of `NegativeWidth` will be thrown when a negative value is specified for the `width`, and an instance of `NegativeLength` will be thrown when a negative value is specified for the `length`. The code for the modified `Rectangle` class is shown here. (These files can be found in the Student Source Code Folder Chapter 16\Rectangle Version 2.)

Contents of Rectangle.h (Version 2)

```

1 // Specification file for the Rectangle class
2 #ifndef RECTANGLE_H
3 #define RECTANGLE_H
4
5 class Rectangle
6 {
7     private:
8         double width;      // The rectangle's width
9         double length;    // The rectangle's length
10    public:
11        // Exception class for a negative width
12        class NegativeWidth
13        { };
14
15        // Exception class for a negative length
16        class NegativeLength
17        { };
18
19        // Default constructor
20        Rectangle()
21        { width = 0.0; length = 0.0; }
22
23        // Mutator functions, defined in Rectangle.cpp
24        void setWidth(double);
25        void setLength(double);
26
27        // Accessor functions
28        double getWidth() const
29        { return width; }
30
31        double getLength() const
32        { return length; }
33
34        double getArea() const
35        { return width * length; }
36    };
37 #endif

```

Contents of Rectangle.cpp (Version 2)

```

1 // Implementation file for the Rectangle class.
2 #include "Rectangle.h"
3
4 //*****
5 // setWidth sets the value of the member variable width. *
6 //*****
7
8 void Rectangle::setWidth(double w)
9 {
10     if (w >= 0)
11         width = w;

```

```

12     else
13         throw NegativeWidth();
14     }
15
16 //***** setLength *****
17 // setLength sets the value of the member variable length. *
18 //*****
19
20 void Rectangle::setLength(double len)
21 {
22     if (len >= 0)
23         length = len;
24     else
25         throw NegativeLength();
26 }
```

Notice in the definition of the `setWidth` function (in `Rectangle.cpp`) that an instance of the `NegativeWidth` class is thrown in line 13. In the definition of the `setLength` function, an instance of the `NegativeLength` class is thrown in line 25. Program 16-3 demonstrates this class. (This file can be found in the Student Source Code Folder Chapter 16\ Rectangle Version 2.)

Program 16-3

```

1 // This program demonstrates Rectangle class exceptions.
2 #include <iostream>
3 #include "Rectangle.h"
4 using namespace std;
5
6 int main()
7 {
8     double width;
9     double length;
10
11    // Create a Rectangle object.
12    Rectangle myRectangle;
13
14    // Get the width and length.
15    cout << "Enter the rectangle's width: ";
16    cin >> width;
17    cout << "Enter the rectangle's length: ";
18    cin >> length;
19
20    // Store these values in the Rectangle object.
21    try
22    {
23        myRectangle.setWidth(width);
24        myRectangle.setLength(length);
25        cout << "The area of the rectangle is "
26                      << myRectangle.getArea() << endl;
27    }
28    catch (Rectangle::NegativeWidth)
```

```

29     {
30         cout << "Error: A negative value was given "
31             << "for the rectangle's width.\n";
32     }
33     catch (Rectangle::NegativeLength)
34     {
35         cout << "Error: A negative value was given "
36             << "for the rectangle's length.\n";
37     }
38
39     cout << "End of the program.\n";
40
41 }

```

Program Output with Example Input Shown in Bold

Enter the rectangle's width: **10**

Enter the rectangle's length: **20**

The area of the rectangle is 200

End of the program.

Program Output with Different Example Input Shown in Bold

Enter the rectangle's width: **-5**

Enter the rectangle's length: **5**

Error: A negative value was given for the rectangle's width.

End of the program.

Program Output with Different Example Input Shown in Bold

Enter the rectangle's width: **5**

Enter the rectangle's length: **-5**

Error: A negative value was given for the rectangle's length.

End of the program.

The try block, in lines 21 through 27, contains code that can throw two different types of exceptions. The statement in line 23 can potentially throw a `NegativeWidth` exception, and the statement in line 24 can potentially throw a `NegativeLength` exception. To handle each of these types of exception, there are two `catch` statements. The statement in line 28 catches `NegativeWidth` exceptions, and the statement in line 33 catches `NegativeLength` exceptions.

When an exception is thrown by code in the try block, C++ searches the try/catch construct for a `catch` statement that can handle the exception. If the construct contains a `catch` statement that is compatible with the exception, control of the program is passed to the catch block.

Using Exception Handlers to Recover from Errors

Program 16-3 demonstrates how a try/catch construct can have several `catch` statements in order to handle different types of exceptions. However, the program does not use the exception handlers to recover from any of the errors. When the user enters a negative value for either the width or the length, this program still halts. Program 16-4 shows a better example of effective exception handling. It attempts to recover from the exceptions and get valid data from the user. (This file can be found in the Student Source Code Folder Chapter 16\Rectangle Version 2.)

Program 16-4

```
1 // This program handles the Rectangle class exceptions.
2 #include <iostream>
3 #include "Rectangle.h"
4 using namespace std;
5
6 int main()
7 {
8     double width;           // Rectangle's width
9     double length;          // Rectangle's length
10    bool tryAgain = true;   // Flag to reread input
11
12    // Create a Rectangle object.
13    Rectangle myRectangle;
14
15    // Get the rectangle's width.
16    cout << "Enter the rectangle's width: ";
17    cin >> width;
18
19    // Store the width in the myRectangle object.
20    while (tryAgain)
21    {
22        try
23        {
24            myRectangle.setWidth(width);
25            // If no exception was thrown, then the
26            // next statement will execute.
27            tryAgain = false;
28        }
29        catch (Rectangle::NegativeWidth)
30        {
31            cout << "Please enter a nonnegative value: ";
32            cin >> width;
33        }
34    }
35
36    // Get the rectangle's length.
37    cout << "Enter the rectangle's length: ";
38    cin >> length;
39
40    // Store the length in the myRectangle object.
41    tryAgain = true;
42    while (tryAgain)
43    {
44        try
45        {
46            myRectangle.setLength(length);
47            // If no exception was thrown, then the
48            // next statement will execute.
49            tryAgain = false;
50        }
51        catch (Rectangle::NegativeLength)
```

```

52         {
53             cout << "Please enter a nonnegative value: ";
54             cin >> length;
55         }
56     }
57
58     // Display the area of the rectangle.
59     cout << "The rectangle's area is "
60         << myRectangle.getArea() << endl;
61
62     return 0;
63 }
```

Program Output with Example Input Shown in Bold

```

Enter the rectangle's width: -1 
Please enter a nonnegative value: 10 
Enter the rectangle's length: -5 
Please enter a nonnegative value: 50 
The rectangle's area is 500
```

Let's look at how this program recovers from a `NegativeWidth` exception. In line 10, a `bool` flag variable, `tryAgain`, is defined and initialized with the value `true`. This variable will indicate whether we need to get a value from the user again. Lines 16 and 17 prompt the user to enter the rectangle's width. Then, the program enters the `while` loop in lines 20 through 34. The loop repeats as long as `tryAgain` is `true`. Inside the loop, the `Rectangle` class's `setWidth` member function is called in line 24. This statement is in a `try` block. If a `NegativeWidth` exception is thrown, the program will jump to the `catch` statement in line 29. In the `catch` block that follows, the user is asked to enter a nonnegative number. The program then jumps out of the `try/catch` construct. Because `tryAgain` is still `true`, the loop will repeat.

If a nonnegative number is passed to the `setWidth` member function in line 24, no exception will be thrown. In that case, the statement in line 27 will execute, which sets `tryAgain` to `false`. The program then jumps out of the `try/catch` construct. Because `tryAgain` is now `false`, the loop will not repeat.

The same strategy is used in lines 37 through 56 to get and validate the rectangle's length.

Extracting Data from the Exception Class

Sometimes we might want an exception object to pass data back to the exception handler. For example, suppose we would like the `Rectangle` class not only to signal when a negative value has been given, but also to pass the value back. This can be accomplished by giving the exception class members in which data can be stored.

In our next modification of the `Rectangle` class, the `NegativeWidth` and `NegativeLength` classes have been expanded, each with a member variable and a constructor. Here is the code for the `NegativeWidth` class:

```

class NegativeWidth
{
private:
    double value;
```

```

public:
    NegativeWidth(double val)
        { value = val; }

    double getValue() const
        { return value; }
};

```

When we throw this exception, we want to pass the invalid value as an argument to the class's constructor. This is done in the `setWidth` member function with the following statement:

```
throw NegativeWidth(w);
```

This `throw` statement creates an instance of the `NegativeWidth` class and passes a copy of the `w` variable to the constructor. The constructor then stores this number in `NegativeWidth`'s member variable, `value`. The class instance carries this member variable to the catch block that intercepts the exception.

In the catch block, the value is extracted with code such as

```

catch (Rectangle::NegativeWidth e)
{
    cout << "Error: " << e.getValue()
        << " is an invalid value for the"
        << " rectangle's width.\n";
}

```

Notice the catch block defines a parameter object named `e`. This is necessary because we want to call the class's `getValue` function to retrieve the value that caused the exception.

Here is the code for the `NegativeLength` class:

```

class NegativeLength
{
private:
    double value;
public:
    NegativeLength(double val)
        { value = val; }

    double getValue() const
        { return value; }
};

```

This class also has a member variable named `value`, and a constructor that initializes the variable. When we throw this exception, we follow the same general steps that were just described for the `NegativeWidth` exception. The complete code for the revised `Rectangle` class is shown here. Program 16-5 demonstrates these classes. (These files can be found in the Student Source Code Folder Chapter 16\Rectangle Version 3.)

Contents of Rectangle.h (Version 3)

```

1 // Specification file for the Rectangle class
2 #ifndef RECTANGLE_H
3 #define RECTANGLE_H
4

```

```
5 class Rectangle
6 {
7     private:
8         double width;      // The rectangle's width
9         double length;    // The rectangle's length
10    public:
11        // Exception class for a negative width
12        class NegativeWidth
13        {
14            private:
15                double value;
16            public:
17                NegativeWidth(double val)
18                    { value = val; }
19
20                double getValue() const
21                    { return value; }
22            };
23
24        // Exception class for a negative length
25        class NegativeLength
26        {
27            private:
28                double value;
29            public:
30                NegativeLength(double val)
31                    { value = val; }
32
33                double getValue() const
34                    { return value; }
35            };
36
37        // Default constructor
38        Rectangle()
39            { width = 0.0; length = 0.0; }
40
41        // Mutator functions, defined in Rectangle.cpp
42        void setWidth(double);
43        void setLength(double);
44
45        // Accessor functions
46        double getWidth() const
47            { return width; }
48
49        double getLength() const
50            { return length; }
51
52        double getArea() const
53            { return width * length; }
54    };
55 #endif
```

Contents of Rectangle.cpp (Version 3)

```

1 // Implementation file for the Rectangle class.
2 #include "Rectangle.h"
3
4 //*****
5 // setWidth sets the value of the member variable width. *
6 //*****
7
8 void Rectangle::setWidth(double w)
9 {
10    if (w >= 0)
11        width = w;
12    else
13        throw NegativeWidth(w);
14 }
15
16 //*****
17 // setLength sets the value of the member variable length. *
18 //*****
19
20 void Rectangle::setLength(double len)
21 {
22    if (len >= 0)
23        length = len;
24    else
25        throw NegativeLength(len);
26 }
```

Program 16-5

```

1 // This program demonstrates Rectangle class exceptions.
2 #include <iostream>
3 #include "Rectangle.h"
4 using namespace std;
5
6 int main()
7 {
8     double width;
9     double length;
10
11    // Create a Rectangle object.
12    Rectangle myRectangle;
13
14    // Get the width and length.
15    cout << "Enter the rectangle's width: ";
16    cin >> width;
17    cout << "Enter the rectangle's length: ";
18    cin >> length;
19
20    // Store these values in the Rectangle object.
21    try
```

If an exception is thrown by the member function of a class object, then the class destructor is called. If statements in the try block (or statements branching from the try block) created any other objects, their destructors will be called as well.

The function that executes a throw statement will immediately terminate. If that function was called by another function, and the exception is not caught, then the calling function will terminate as well. This process, known as *unwinding the stack*, continues for the entire chain of nested function calls, from the throw point, all the way back to the try block.

Once an exception has been thrown, the program cannot jump back to the throw point.

Unwinding the Stack

Program Output with Example Input Shown in Bold	
Error: -1 is an invalid value for the rectangle's width.	Enter the rectangle's width: 5 Enter
Error: -1 is an invalid value for the rectangle's length.	Enter the rectangle's length: -1 Enter
Error: -1 is an invalid value for the rectangle's width.	Enter the rectangle's width: 10 Enter
Error: -1 is an invalid value for the rectangle's length.	Enter the rectangle's length: -1 Enter

```
22 }  
23 myRectangle.setWidth(width);  
24 myRectangle.setHeight(height);  
25 cout << "The area of the rectangle is "  
26 << myRectangle.getArea() << endl;  
27 }  
28 catch (Rectangle::NegativeWidth e)  
29 {  
30 cout << "Error: " << e.getValue()  
31 << " is an invalid value for the"  
32 << " rectangle's width.\n";  
33 }  
34 catch (Rectangle::NegativeLength e)  
35 {  
36 cout << "Error: " << e.getValue()  
37 << " is an invalid value for the"  
38 << " rectangle's length.\n";  
39 }  
40 cout << "End of the program.\n";  
41  
42 return 0;  
43 }
```

Rethrowing an Exception

It is possible for try blocks to be nested. For example, look at this code segment:

```
try
{
    doSomething();
}
catch(exception1)
{
    // code to handle exception 1
}
catch(exception2)
{
    // code to handle exception 2
}
```

In this try block, the function `doSomething` is called. There are two catch blocks, one that handles `exception1`, and another that handles `exception2`. If the `doSomething` function also has a try block, then it is nested inside the one shown.

With nested try blocks, it is sometimes necessary for an inner exception handler to pass an exception to an outer exception handler. Sometimes both an inner and an outer catch block must perform operations when a particular exception is thrown. These situations require that the inner catch block *rethrow* the exception so the outer catch block has a chance to catch it.

A catch block can rethrow an exception with the `throw` statement. For example, suppose the `doSomething` function (called in the throw block above) calls the `doSomethingElse` function, which potentially can throw `exception1` or `exception3`. Suppose `doSomething` does not want to handle `exception1`. Instead, it wants to rethrow it to the outer block. The following code segment illustrates how this is done:

```
try
{
    doSomethingElse();
}
catch(exception1)
{
    throw; // Rethrow the exception
}
catch(exception3)
{
    // Code to handle exception 3
}
```

When the first catch block catches `exception1`, the `throw` statement simply throws the exception again. The catch block in the outer try/catch construct will then handle the exception.

Handling the `bad_alloc` Exception

Recall from Chapter 9 that when the `new` operator fails to allocate memory, an exception is thrown. Now that you've seen how to handle exceptions, you can write code that determines whether the `new` operator was successful.

When the `new` operator fails to allocate memory, C++ throws a `bad_alloc` exception. The `bad_alloc` exception type is defined in the `<new>` header file, so any program that attempts to catch this exception should have the following directive:

```
#include <new>
```

The `bad_alloc` exception is in the `std` namespace, so be sure to have the `using namespace std;` statement in your code as well.

Here is the general format of a try/catch construct that catches the `bad_alloc` exception:

```
try
{
    // Code that uses the new operator
}
catch (bad_alloc)
{
    // Code that responds to the error
}
```

Program 16-6 shows an example. The program uses the `new` operator to allocate a 10,000-element array of doubles. If the `new` operator fails, an error message is displayed.

Program 16-6

```
1 // This program demonstrates the bad_alloc exception.
2 #include <iostream>
3 #include <new>           // Needed for bad_alloc
4 using namespace std;
5
6 int main()
7 {
8     double *ptr = nullptr; // Pointer to double
9
10    try
11    {
12        ptr = new double [10000];
13    }
14    catch (bad_alloc)
15    {
16        cout << "Insufficient memory.\n";
17    }
18
19    return 0;
20 }
```



Checkpoint

- 16.1 What is the difference between a try block and a catch block?
- 16.2 What happens if an exception is thrown, but not caught?
- 16.3 If multiple exceptions can be thrown, how does the catch block know which exception to catch?
- 16.4 After the catch block has handled the exception, where does program execution resume?
- 16.5 How can an exception pass data back to the exception handler?

16.2 Function Templates

CONCEPT: A function template is a “generic” function that can work with any data type. The programmer writes the specifications of the function, but substitutes parameters for data types. When the compiler encounters a call to the function, it generates code to handle the specific data type(s) used in the call.

Introduction

Overloaded functions make programming convenient because only one function name must be remembered for a set of functions that perform similar operations. Each of the functions, however, must still be written individually, even if they perform the same operation. For example, suppose a program uses the following overloaded square functions:

```
int square(int number)
{
    return number * number;
}
double square(double number)
{
    return number * number;
}
```

The only differences between these two functions are the data types of their return values and their parameters. In situations like this, it is more convenient to write a *function template* than an overloaded function. Function templates allow you to write a single function definition that works with many different data types, instead of having to write a separate function for each data type used.

A function template is not an actual function, but a “mold” the compiler uses to generate one or more functions. When writing a function template, you do not have to specify actual types for the parameters, return value, or local variables. Instead, you use a *type parameter* to specify a generic data type. When the compiler encounters a call to the function, it examines the data types of its arguments and generates the function code that will work with those data types. (The generated code is known as a *template function*.)

Here is a function template for the square function:

```
template <class T>
T square(T number)
{
    return number * number;
}
```

The beginning of a function template is marked by a *template prefix*, which begins with the key word `template`. Next is a set of angled brackets that contains one or more generic data types used in the template. A generic data type starts with the key word `class` followed by a parameter name that stands for the data type. The example just given only uses one, which is named `T`. (If there were more, they would be separated by commas.) After this, the function definition is written as usual, except the type parameters are substituted for the actual data type names. In the example, the function header reads

`T square(T number)`



T is the type parameter, or generic data type. The header defines `square` as a function that returns a value of type T and uses a parameter, `number`, which is also of type T . As mentioned before, the compiler examines each call to `square` and fills in the appropriate data type for T . For example, the following call uses an `int` argument:

```
int y, x = 4;
y = square(x);
```

This code will cause the compiler to generate the function

```
int square(int number)
{
    return number * number;
}
```

while the following statements

```
double y, f = 6.2
y = square(f);
```

will generate the function

```
double square(double number)
{
    return number * number;
}
```

Program 16-7 demonstrates how this function template is used.

Program 16-7

```
1 // This program uses a function template.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 // Template definition for square function.
7 template <class T>
8 T square(T number)
9 {
10     return number * number;
11 }
12
13 int main()
14 {
15     int userInt;      // To hold integer input
16     double userDouble; // To hold double input
17
18     cout << setprecision(5);
19     cout << "Enter an integer and a floating-point value: ";
20     cin >> userInt >> userDouble;
21     cout << "Here are their squares: ";
22     cout << square(userInt) << " and "
23         << square(userDouble) << endl;
24
25 }
```

(program output continues)

Program 16-7

(continued)

Program Output with Example Input Shown in BoldEnter an integer and a floating-point value: **12 4.2** **Enter**

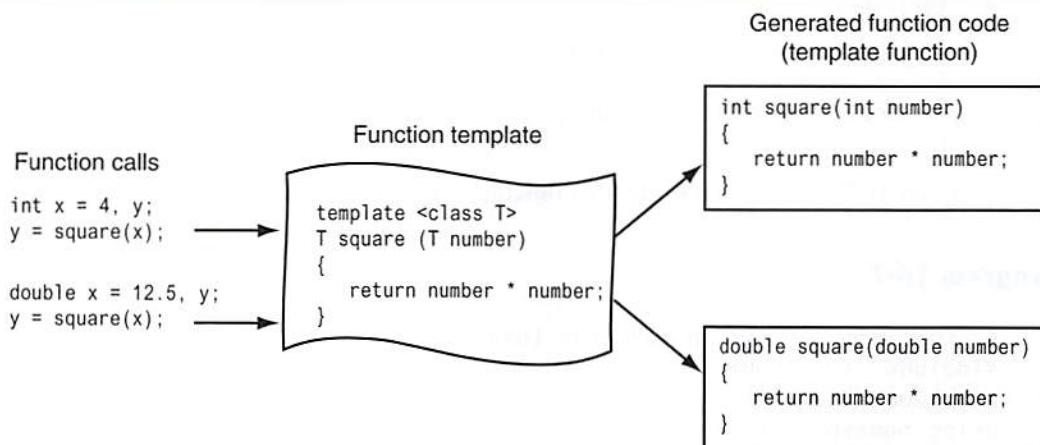
Here are their squares: 144 and 17.64



NOTE: All type parameters defined in a function template must appear at least once in the function parameter list.

Because the compiler encountered two calls to `square` in Program 16-7, each with a different parameter type, it generated the code for two instances of the function: one with an `int` parameter and `int` return type, the other with a `double` parameter and `double` return type. This is illustrated in Figure 16-3.

Figure 16-3 Generation of template functions



Notice in Program 16-7 that the template appears before all calls to `square`. As with regular functions, the compiler must already know the template's contents when it encounters a call to the template function. Templates, therefore, should be placed near the top of the program or in a header file.



NOTE: A function template is merely the specification of a function and by itself does not cause memory to be used. An actual instance of the function is created in memory when the compiler encounters a call to the template function.

Program 16-8 shows another example of a function template. The function, `swapVars`, uses two references to type `T` as parameters. The function swaps the contents of the variables referenced by the parameters.

Program 16-8

```

1 // This program demonstrates the swapVars function template.
2 #include <iostream>
3 using namespace std;
4
  
```

```

5  template <class T>
6  void swapVars(T &var1, T &var2)
7  {
8      T temp;
9
10     temp = var1;
11     var1 = var2;
12     var2 = temp;
13 }
14
15 int main()
16 {
17     char firstChar, secondChar;           // Two chars
18     int firstInt, secondInt;            // Two ints
19     double firstDouble, secondDouble; // Two doubles
20
21     // Get and swapVars two chars
22     cout << "Enter two characters: ";
23     cin >> firstChar >> secondChar;
24     swapVars(firstChar, secondChar);
25     cout << firstChar << " " << secondChar << endl;
26
27     // Get and swapVars two ints
28     cout << "Enter two integers: ";
29     cin >> firstInt >> secondInt;
30     swapVars(firstInt, secondInt);
31     cout << firstInt << " " << secondInt << endl;
32
33     // Get and swapVars two doubles
34     cout << "Enter two floating-point numbers: ";
35     cin >> firstDouble >> secondDouble;
36     swapVars(firstDouble, secondDouble);
37     cout << firstDouble << " " << secondDouble << endl;
38
39 }

```

Program Output with Example Input Shown in Bold

```

Enter two characters: A B Enter
B A
Enter two integers: 5 10 Enter
10 5
Enter two floating-point numbers: 1.2 9.6 Enter
9.6 1.2

```

Using Operators in Function Templates

The square template shown earlier uses the * operator with the number parameter. This works well as long as number is of a primitive data type such as int, float, and so on. If a user-defined class object is passed to the square function, however, the class must contain code for an overloaded * operator. If not, the compiler will generate a function with an error.

Always remember a class object passed to a function template must support all the operations the function will perform on the object. For instance, if the function performs a comparison on the object (with $>$, $<$, $==$, or another relational operator), those operators must be overloaded by the class object.

Function Templates with Multiple Types

More than one generic type may be used in a function template. Each type must have its own parameter, as shown in Program 16-9. This program uses a function template named `larger`. This template uses two type parameters: `T1` and `T2`. The sizes of the function parameters, `var1` and `var2`, are compared, and the function returns the number of bytes occupied by the larger of the two. Because the function parameters are specified with different types, the function generated from this template can accept two arguments of different types.

Program 16-9

```

1 // This program demonstrates a function template
2 // with two type parameters.
3 #include <iostream>
4 using namespace std;
5
6 template <class T1, class T2>
7 int largest(const T1 &var1, T2 &var2)
8 {
9     if (sizeof(var1) > sizeof(var2))
10        return sizeof(var1);
11    else
12        return sizeof(var2);
13 }
14
15 int main()
16 {
17     int i = 0;
18     char c = ' ';
19     float f = 0.0;
20     double d = 0.0;
21
22     cout << "Comparing an int and a double, the largest\n"
23         << "of the two is " << largest(i, d) << " bytes.\n";
24
25     cout << "Comparing a char and a float, the largest\n"
26         << "of the two is " << largest(c, f) << " bytes.\n";
27
28     return 0;
29 }
```

Program Output

```
Comparing an int and a double, the largest
of the two is 8 bytes.
Comparing a char and a float, the largest
of the two is 4 bytes.
```



NOTE: Each type parameter declared in the template prefix must be used somewhere in the template definition.

Overloading with Function Templates

Function templates may be overloaded. As with regular functions, function templates are overloaded by having different parameter lists. For example, there are two overloaded versions of the `sum` function in Program 16-10. The first version accepts two arguments, and the second version accepts three.

Program 16-10

```
1 // This program demonstrates an overloaded function template.
2 #include <iostream>
3 using namespace std;
4
5 template <class T>
6 T sum(T val1, T val2)
7 {
8     return val1 + val2;
9 }
10
11 template <class T>
12 T sum(T val1, T val2, T val3)
13 {
14     return val1 + val2 + val3;
15 }
16
17 int main()
18 {
19     double num1, num2, num3;
20
21     // Get two values and display their sum.
22     cout << "Enter two values: ";
23     cin >> num1 >> num2;
24     cout << "Their sum is " << sum(num1, num2) << endl;
25
26     // Get three values and display their sum.
27     cout << "Enter three values: ";
28     cin >> num1 >> num2 >> num3;
29     cout << "Their sum is " << sum(num1, num2, num3) << endl;
30
31 }
```

Program Output with Example Input Shown in Bold

Enter two values: **12.5 6.9** **Enter**

Their sum is 19.4

Enter three values: **45.76 98.32 10.51** **Enter**

Their sum is 154.59

There are other ways to perform overloading with function templates as well. For example, a program might contain a regular (nontemplate) version of a function as well as a template version. As long as each has a different parameter list, they can coexist as overloaded functions.

16.3

Focus on Software Engineering: Where to Start When Defining Templates

Quite often, it is easier to convert an existing function into a template than to write a template from scratch. With this in mind, you should start designing a function template by writing it first as a regular function. For example, the `swapVars` template in Program 16-8 would have been started as something like the following:

```
void swapVars(int &var1, int &var2)
{
    int temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

Once this function is properly tested and debugged, converting it to a template is a simple process. First, the template `<class T>` header is added, then all the references to `int` that must be changed are replaced with the data type parameter `T`.



Checkpoint

- 16.6 When does the compiler actually generate code for a function template?
- 16.7 The following function accepts an `int` argument and returns half of its value as a `double`:

```
double half(int number)
{
    return number / 2.0;
}
```

Write a template that will implement this function to accept an argument of any type.

- 16.8 What must you be sure of when passing a class object to a function template that uses an operator, such as `*` or `>`?
- 16.9 What is the best method for writing a function template?

16.4

Class Templates

CONCEPT: Templates may also be used to create generic classes and abstract data types. Class templates allow you to create one general version of a class without having to duplicate code to handle multiple data types.

Recall the `IntArray` class from Chapter 14. By overloading the `[]` operator, this class allows you to implement `int` arrays that perform bounds checking. But suppose you would like to have a version of this class for other data types. Of course, you could design specialized classes such as `LongArray`, `FloatArray`, `DoubleArray`, and so forth. A better solution, however, is to design a single class template that works with any primitive data type. In this section, we will convert the `IntArray` class into a generalized template named `SimpleVector`.

Declaring a class template is very similar to declaring a function template. First, a template prefix, such as `template <class T>`, is placed before the class declaration. As with function templates, `T` (or whatever identifier you choose to use) is a data type parameter. Then, throughout the class declaration, the data type parameter is used where you wish to support any data type. Below is the `SimpleVector` class template declaration.

Contents of SimpleVector.h

```
1 // SimpleVector class template
2 #ifndef SIMPLEVECTOR_H
3 #define SIMPLEVECTOR_H
4 #include <iostream>
5 #include <new>      // Needed for bad_alloc exception
6 #include <cstdlib>  // Needed for the exit function
7 using namespace std;
8
9 template <class T>
10 class SimpleVector
11 {
12 private:
13     T *aptr;          // To point to the allocated array
14     int arraySize;    // Number of elements in the array
15     void memError(); // Handles memory allocation errors
16     void subError(); // Handles subscripts out of range
17
18 public:
19     // Default constructor
20     SimpleVector()
21         { aptr = 0; arraySize = 0; }
22
23     // Constructor declaration
24     SimpleVector(int);
25
26     // Copy constructor declaration
27     SimpleVector(const SimpleVector &);
28
29     // Destructor declaration
30     ~SimpleVector();
31
32     // Accessor to return the array size
33     int size() const
34         { return arraySize; }
35
36     // Accessor to return a specific element
37     T getElementAt(int position);
38
39     // Overloaded [] operator declaration
40     T &operator[](const int &);
41 };
42
43 //*****
44 // Constructor for SimpleVector class. Sets the size of the *
45 // array and allocates memory for it. *
46 //*****
```

```
47
48 template <class T>
49 SimpleVector<T>::SimpleVector(int s)
50 {
51     arraySize = s;
52     // Allocate memory for the array.
53     try
54     {
55         aptr = new T [s];
56     }
57     catch (bad_alloc)
58     {
59         memError();
60     }
61
62     // Initialize the array.
63     for (int count = 0; count < arraySize; count++)
64         *(aptr + count) = 0;
65 }
66
67 //*****
68 // Copy Constructor for SimpleVector class. *
69 //*****
70
71 template <class T>
72 SimpleVector<T>::SimpleVector(const SimpleVector &obj)
73 {
74     // Copy the array size.
75     arraySize = obj.arraySize;
76
77     // Allocate memory for the array.
78     aptr = new T [arraySize];
79     if (aptr == 0)
80         memError();
81
82     // Copy the elements of obj's array.
83     for(int count = 0; count < arraySize; count++)
84         *(aptr + count) = *(obj.aptr + count);
85 }
86
87 //*****
88 // Destructor for SimpleVector class. *
89 //*****
90
91 template <class T>
92 SimpleVector<T>::~SimpleVector()
93 {
94     if (arraySize > 0)
95         delete [] aptr;
96 }
97
98 //*****
99 // memError function. Displays an error message and *
100 // terminates the program when memory allocation fails. *
101 //*****
```

```

102
103 template <class T>
104 void SimpleVector<T>::memError()
105 {
106     cout << "ERROR: Cannot allocate memory.\n";
107     exit(EXIT_FAILURE);
108 }
109
110 //***** *****
111 // subError function. Displays an error message and      *
112 // terminates the program when a subscript is out of range. *
113 //***** *****
114
115 template <class T>
116 void SimpleVector<T>::subError()
117 {
118     cout << "ERROR: Subscript out of range.\n";
119     exit(EXIT_FAILURE);
120 }
121
122 //***** *****
123 // getElementAt function. The argument is a subscript.   *
124 // This function returns the value stored at the          *
125 // subscript in the array.                                *
126 //***** *****
127
128 template <class T>
129 T SimpleVector<T>::getElementAt(int sub)
130 {
131     if (sub < 0 || sub >= arraySize)
132         subError();
133     return aptr[sub];
134 }
135
136 //***** *****
137 // Overloaded [] operator. The argument is a subscript.   *
138 // This function returns a reference to the element        *
139 // in the array indexed by the subscript.                  *
140 //***** *****
141
142 template <class T>
143 T &SimpleVector<T>::operator[](const int &sub)
144 {
145     if (sub < 0 || sub >= arraySize)
146         subError();
147     return aptr[sub];
148 }
149 #endif

```



NOTE: The `arraySize` member variable is declared as an `int`. This is because it holds the size of the array, which will be an integer value, regardless of the data type of the array. This is also why the `size` member function returns an `int`.

Defining Objects of the Class Template

Class template objects are defined like objects of ordinary classes, with one small difference: the data type you wish to pass to the type parameter must be specified. Placing the data type name inside angled brackets immediately following the class name does this. For example, the following statements create two `SimpleVector` objects: `intTable` and `doubleTable`.

```
SimpleVector<int> intTable(10);
SimpleVector<double> doubleTable(10);
```

In the definition of `intTable`, the data type `int` will be used in the template everywhere the type parameter `T` appears. This will cause `intTable` to store an array of `ints`. Likewise, the definition of `doubleTable` passes the data type `double` into the parameter `T`, causing it to store an array of `doubles`. This is demonstrated in Program 16-11.

Program 16-11

```
1 // This program demonstrates the SimpleVector template.
2 #include <iostream>
3 #include "SimpleVector.h"
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 10; // Number of elements
9     int count;           // Loop counter
10
11    // Create a SimpleVector of ints.
12    SimpleVector<int> intTable(SIZE);
13
14    // Create a SimpleVector of doubles.
15    SimpleVector<double> doubleTable(SIZE);
16
17    // Store values in the two SimpleVectors.
18    for (count = 0; count < SIZE; count++)
19    {
20        intTable[count] = (count * 2);
21        doubleTable[count] = (count * 2.14);
22    }
23
24    // Display the values in the SimpleVectors.
25    cout << "These values are in intTable:\n";
26    for (count = 0; count < SIZE; count++)
27        cout << intTable[count] << " ";
28    cout << endl;
29    cout << "These values are in doubleTable:\n";
30    for (count = 0; count < SIZE; count++)
31        cout << doubleTable[count] << " ";
32    cout << endl;
33
34    // Use the standard + operator on the elements.
35    cout << "\nAdding 5 to each element of intTable"
36                  << " and doubleTable.\n";
37    for (count = 0; count < SIZE; count++)
```

```

38     {
39         intTable[count] = intTable[count] + 5;
40         doubleTable[count] = doubleTable[count] + 5.0;
41     }
42
43     // Display the values in the SimpleVectors.
44     cout << "These values are in intTable:\n";
45     for (count = 0; count < SIZE; count++)
46         cout << intTable[count] << " ";
47     cout << endl;
48     cout << "These values are in doubleTable:\n";
49     for (count = 0; count < SIZE; count++)
50         cout << doubleTable[count] << " ";
51     cout << endl;
52
53     // Use the standard ++ operator on the elements.
54     cout << "\nIncrementing each element of intTable and"
55         << " doubleTable.\n";
56     for (count = 0; count < SIZE; count++)
57     {
58         intTable[count]++;
59         doubleTable[count]++;
60     }
61
62     // Display the values in the SimpleVectors.
63     cout << "These values are in intTable:\n";
64     for (count = 0; count < SIZE; count++)
65         cout << intTable[count] << " ";
66     cout << endl;
67     cout << "These values are in doubleTable:\n";
68     for (count = 0; count < SIZE; count++)
69         cout << doubleTable[count] << " ";
70     cout << endl;
71
72     return 0;
73 }
```

Program Output

These values are in intTable:
0 2 4 6 8 10 12 14 16 18
These values are in doubleTable:
0 2.14 4.28 6.42 8.56 10.7 12.84 14.98 17.12 19.26
Adding 5 to each element of intTable and doubleTable.
These values are in intTable:
5 7 9 11 13 15 17 19 21 23
These values are in doubleTable:
5 7.14 9.28 11.42 13.56 15.7 17.84 19.98 22.12 24.26
Incrementing each element of intTable and doubleTable.
These values are in intTable:
6 8 10 12 14 16 18 20 22 24
These values are in doubleTable:
6 8.14 10.28 12.42 14.56 16.7 18.84 20.98 23.12 25.26

Class Templates and Inheritance

Inheritance can easily be applied to class templates. For example, in the following template, SearchableVector is derived from the SimpleVector class.

Contents of SearchableVector.h

```

1  #ifndef SEARCHABLEVECTOR_H
2  #define SEARCHABLEVECTOR_H
3  #include "SimpleVector.h"
4
5  template <class T>
6  class SearchableVector : public SimpleVector<T>
7  {
8  public:
9      // Default constructor
10     SearchableVector() : SimpleVector<T>()
11         { }
12
13     // Constructor
14     SearchableVector(int size) : SimpleVector<T>(size)
15         { }
16
17     // Copy constructor
18     SearchableVector(const SearchableVector &);
19
20     // Accessor to find an item
21     int findItem(const T);
22 };
23
24 //*****
25 // Copy constructor
26 //*****
27
28 template <class T>
29 SearchableVector<T>::SearchableVector(const SearchableVector &obj) :
30             SimpleVector<T>(obj.size())
31 {
32     for(int count = 0; count < this->size(); count++)
33         this->operator[](count) = obj[count];
34 }
35
36 //*****
37 // findItem function
38 // This function searches for item. If item is found
39 // the subscript is returned. Otherwise -1 is returned.
40 //*****
41
42 template <class T>
43 int SearchableVector<T>::findItem(const T item)
44 {
45     for (int count = 0; count <= this->size(); count++)

```

```

46      {
47          if (getElementAt(count) == item)
48              return count;
49      }
50      return -1;
51  }
52 #endif

```

This class template defines a searchable version of the `SimpleVector` class. The member function `findItem` accepts an argument and performs a simple linear search to determine whether the argument's value is stored in the array. If the value is found in the array, its subscript is returned. Otherwise, `-1` is returned.

Notice each time the name `SimpleVector` is used in the class template, the type parameter `T` is used with it. For example, here is the first line of the class declaration, in line 6, which names `SimpleVector` as the base class:

```
class SearchableVector : public SimpleVector<T>
```

Also, here are the function headers for the class constructors:

```
SearchableVector() : SimpleVector<T>()
SearchableVector(int size) : SimpleVector<T>(size)
```

Because `SimpleVector` is a class template, the type parameter must be passed to it.

Program 16-12 demonstrates the class by storing values in two `SearchableVector` objects, then searching for a specific value in each.

Program 16-12

```

1 // This program demonstrates the SearchableVector template.
2 #include <iostream>
3 #include "SearchableVector.h"
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 10;    // Number of elements
9     int count;            // Loop counter
10    int result;           // To hold search results
11
12    // Create two SearchableVector objects.
13    SearchableVector<int> intTable(SIZE);
14    SearchableVector<double> doubleTable(SIZE);
15
16    // Store values in the objects.
17    for (count = 0; count < SIZE; count++)
18    {
19        intTable[count] = (count * 2);
20        doubleTable[count] = (count * 2.14);
21    }

```

(program continues)

Program 16-12 *(continued)*

```

22      // Display the values in the objects.
23      cout << "These values are in intTable:\n";
24      for (count = 0; count < SIZE; count++)
25          cout << intTable[count] << " ";
26      cout << endl << endl;
27      cout << "These values are in doubleTable:\n";
28      for (count = 0; count < SIZE; count++)
29          cout << doubleTable[count] << " ";
30      cout << endl;
31
32
33      // Search for the value 6 in intTable.
34      cout << "\nSearching for 6 in intTable.\n";
35      result = intTable.findItem(6);
36      if (result == -1)
37          cout << "6 was not found in intTable.\n";
38      else
39          cout << "6 was found at subscript " << result << endl;
40
41      // Search for the value 12.84 in doubleTable.
42      cout << "\nSearching for 12.84 in doubleTable.\n";
43      result = doubleTable.findItem(12.84);
44      if (result == -1)
45          cout << "12.84 was not found in doubleTable.\n";
46      else
47          cout << "12.84 was found at subscript " << result << endl;
48      return 0;
49  }

```

Program Output

These values are in intTable:
0 2 4 6 8 10 12 14 16 18

These values are in doubleTable:
0 2.14 4.28 6.42 8.56 10.7 12.84 14.98 17.12 19.26

Searching for 6 in intTable.
6 was found at subscript 3

Searching for 12.84 in doubleTable.
12.84 was found at subscript 6

The `SearchableVector` class demonstrates that a class template may be derived from another class template. In addition, class templates may be derived from ordinary classes, and ordinary classes may be derived from class templates.

Specialized Templates

Suppose you have a template that works for all data types but one. For example, the `SimpleVector` and `SearchableVector` classes work well with numeric data, and even character data. But they will not work with C-strings. Situations like this require the use of *specialized templates*. A specialized template is one that is designed to work with a specific data type. In the declaration, the actual data type is used instead of a type parameter. For example, the declaration of a specialized version of the `SimpleVector` class might start like this:

```
class SimpleVector<char *>
```

The compiler would know that this version of the `SimpleVector` class is intended for the `char *` data type. Anytime an object is defined of the type `SimpleVector <char *>`, the compiler will use this template to generate the code.



Checkpoint

- 16.10 Suppose your program uses a class template named `List`, which is defined as

```
template<class T>
class List
{
    // Members are declared here...
};
```

Give an example of how you would use `int` as the data type in the definition of a `List` object. (Assume the class has a default constructor.)

- 16.11 As the following `Rectangle` class is written, the `width` and `length` members are `doubles`. Rewrite the class as a template that will accept any data type for these members.

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setData(double w, double l)
            { width = w; length = l; }
        double getWidth()
            { return width; }
        double getLength()
            { return length; }
        double getArea()
            { return width * length; }
};
```

Review Questions and Exercises

Short Answer

1. What is a throw point?
2. What is an exception handler?
3. Explain the difference between a try block and a catch block.
4. What happens if an exception is thrown, but not caught?
5. What is “unwinding the stack”?
6. What happens if an exception is thrown by a class’s member function?
7. How do you prevent a program from halting when the new operator fails to allocate memory?
8. Why is it more convenient to write a function template than a series of overloaded functions?
9. Why must you be careful when writing a function template that uses operators such as [] with its parameters?

Fill-in-the-Blank

10. The line containing a throw statement is known as the _____.
11. The _____ block contains code that directly or indirectly might cause an exception to be thrown.
12. The _____ block handles an exception.
13. When writing function or class templates, you use a(n) _____ to specify a generic data type.
14. The beginning of a template is marked by a(n) _____.
15. When defining objects of class templates, the _____ you wish to pass into the type parameter must be specified.
16. A(n) _____ template works with a specific data type.

Algorithm Workbench

17. Write a function that searches a numeric array for a specified value. The function should return the subscript of the element containing the value if it is found in the array. If the value is not found, the function should throw an exception.
18. Write a function that dynamically allocates a block of memory and returns a char pointer to the block. The function should take an integer argument that is the amount of memory to be allocated. If the new operator cannot allocate the memory, the function should return a null pointer.
19. Make the function you wrote in Question 17 a template.
20. Write a template for a function that displays the contents of an array of any type.

True or False

21. T F There can be only one catch block in a program.
22. T F When an exception is thrown, but not caught, the program ignores the error.

23. T F Data may be passed with an exception by storing it in members of an exception class.
24. T F Once an exception has been thrown, it is not possible for the program to jump back to the throw point.
25. T F All type parameters defined in a function template must appear at least once in the function parameter list.
26. T F The compiler creates an instance of a function template in memory as soon as it encounters the template.
27. T F A class object passed to a function template must overload any operators used on the class object by the template.
28. T F Only one generic type may be used with a template.
29. T F In the function template definition, it is not necessary to use each type parameter declared in the template prefix.
30. T F It is possible to overload two function templates.
31. T F It is possible to overload a function template and an ordinary (nontemplate) function.
32. T F A class template may not be derived from another class template.
33. T F A class template may not be used as a base class.
34. T F Specialized templates work with a specific data type.

Find the Error

Each of the following declarations or code segments has errors. Locate as many as possible.

```
35. catch
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
try (string exceptionString)
{
    cout << exceptionString;
}

36. try
{
    quotient = divide(num1, num2);
}
cout << "The quotient is " << quotient << endl;
catch (string exceptionString)
{
    cout << exceptionString;
}

37. template <class T>
T square(T number)
{
    return T * T;
}
```

38. template <class T>
 int square(int number)
 {
 return number * number;
 }
39. template <class T1, class T2>
 T1 sum(T1 x, T1 y)
 {
 return x + y;
 }
40. Assume the following definition appears in a program that uses the `SimpleVector` class template presented in this chapter.
- ```
int <SimpleVector> array(25);
```
41. Assume the following statement appears in a program that has defined `valueSet` as an object of the `SimpleVector` class presented in this chapter. Assume `valueSet` is a vector of ints, and has 20 elements.
- ```
cout << valueSet<int>[2] << endl;
```

Programming Challenges

1. Date Exceptions

Modify the `Date` class you wrote for Programming Challenge 1 of Chapter 13 (Date). The class should implement the following exception classes:

`InvalidDay` Throw when an invalid day (< 1 or > 31) is passed to the class.
`InvalidMonth` Throw when an invalid month (< 1 or > 12) is passed to the class.

Demonstrate the class in a driver program.

2. Time Format Exceptions

Modify the `MillTime` class you created for Programming Challenge 4 of Chapter 15 (Time Format). The class should implement the following exceptions:

`BadHour` Throw when an invalid hour (< 0 or > 2359) is passed to the class.
`BadSeconds` Throw when an invalid number of seconds (< 0 or > 59) is passed to the class.

Demonstrate the class in a driver program.

3. Minimum/Maximum Templates

Write templates for the two functions `minimum` and `maximum`. The `minimum` function should accept two arguments and return the value of the argument that is the lesser of the two. The `maximum` function should accept two arguments and return the value of the argument that is the greater of the two. Design a simple driver program that demonstrates the templates with various data types.

4. Absolute Value Template

Write a function template that accepts an argument and returns its absolute value. The absolute value of a number is its value with no sign. For example, the absolute value of -5 is 5, and the absolute value of 2 is 2. Test the template in a simple driver program.

5. Total Template

Write a template for a function called `total`. The function should keep a running total of values entered by the user, then return the total. The argument sent into the function should be the number of values the function is to read. Test the template in a simple driver program that sends values of various types as arguments and displays the results.

6. IntArray Class Exception

Chapter 14 presented an `IntArray` class that dynamically creates an array of integers and performs bounds checking on the array. If an invalid subscript is used with the class, it displays an error message and aborts the program. Modify the class so it throws an exception instead.

7. TestScores Class

Write a class named `TestScores`. The class constructor should accept an array of test scores as its argument. The class should have a member function that returns the average of the test scores. If any test score in the array is negative or greater than 100, the class should throw an exception. Demonstrate the class in a program.

8. SimpleVector Modification

Modify the `SimpleVector` class template presented in this chapter to include the member functions `push_back` and `pop_back`. The `push_back` function should accept an argument and insert its value at the end of the array. The `pop_back` function should accept no argument and remove the last element from the array. Test the class with a driver program.

9. SearchableVector Modification

Modify the `SearchableVector` class template presented in this chapter so it performs a binary search instead of a linear search. Test the template in a driver program.

10. SortableVector Class Template

Write a class template named `SortableVector`. The class should be derived from the `SimpleVector` class presented in this chapter. It should have a member function that sorts the array elements in ascending order. (Use the sorting algorithm of your choice.) Test the template in a driver program.

11. Inheritance Modification

Assuming you have completed Programming Challenges 9 and 10, modify the inheritance hierarchy of the `SearchableVector` class template so it is derived from the `SortableVector` class instead of the `SimpleVector` class. Implement a member function named `sortAndSearch`, both a sort and a binary search.

12. Specialized Templates

In this chapter, the section *Specialized Templates* within Section 16.4 describes how to design templates that are specialized for one particular data type. The section introduces a method for specializing a version of the `SimpleVector` class template so it will work with strings. Complete the specialization for both the `SimpleVector` and `SearchableVector` templates. Demonstrate them with a simple driver program.



13. Exception Project

This assignment assumes you have completed Programming Challenge 1 of Chapter 15 (`Employee` and `ProductionWorker` Classes). Modify the `Employee` and `ProductionWorker` classes so they throw exceptions when the following errors occur:

- The `Employee` class should throw an exception named `InvalidEmployeeNumber` when it receives an employee number that is less than 0 or greater than 9999.
- The `ProductionWorker` class should throw an exception named `InvalidShift` when it receives an invalid shift.
- The `ProductionWorker` class should throw an exception named `InvalidPayRate` when it receives a negative number for the hourly pay rate.

Write a driver program that demonstrates how each of these exception conditions works.

TOPICS

- | | |
|--|--|
| 17.1 Introduction to the Standard Template Library | 17.5 The <code>set</code> , <code>multiset</code> , and <code>unordered_set</code> Classes |
| 17.2 STL Container and Iterator Fundamentals | 17.6 Algorithms |
| 17.3 The <code>vector</code> Class | 17.7 Introduction to Function Objects and Lambda Expressions |
| 17.4 The <code>map</code> , <code>multimap</code> , and <code>unordered_map</code> Classes | |

17.1

Introduction to the Standard Template Library

CONCEPT: The Standard Template Library is an extensive collection of templates for useful data structures and algorithms.

In addition to its runtime library, which you have used throughout this book, C++ also provides an extensive library of templates. The *Standard Template Library* (or *STL*) contains numerous generic templates for classes and functions. Most of the templates in the STL can be grouped into the following categories:

- **Containers:** Class templates for objects that store and organize data
- **Iterators:** Class templates for objects that behave like pointers, and are used to access the individual data elements in a container
- **Algorithms:** Function templates that perform various operations on elements of containers

This chapter introduces many of the STL's class and function templates.

17.2

STL Container and Iterator Fundamentals

CONCEPT: A container is an object that holds a collection of values or other objects. An iterator is an object that is used to iterate over the items in a collection, providing access to them.

Containers

There are two types of container classes in the STL: *sequence* and *associative*. A sequence container stores its data sequentially in memory, in a fashion similar to an array. An associative container stores its data in a nonsequential way that makes it faster to locate elements.

The sequence containers currently provided in the STL are listed in Table 17-1. (Note that the `deque` container will be discussed in Chapter 19, and the `list` and `forward_list` containers will be discussed in Chapter 18.)

Table 17-1 Sequence Containers

Container Class	Description
<code>array</code>	A fixed-size container that is similar to an array
<code>deque</code>	A double-ended queue. Like a <code>vector</code> , but designed so that values can be quickly added to or removed from the front and back. (This container will be discussed in Chapter 19.)
<code>forward_list</code>	A singly linked list of data elements. Values may be inserted to or removed from any position. (This container will be discussed in Chapter 18.)
<code>list</code>	A doubly linked list of data elements. Values may be inserted to or removed from any position. (This container will be discussed in Chapter 18.)
<code>vector</code>	A container that works like an expandable array. Values may be added to or removed from a <code>vector</code> . The <code>vector</code> automatically adjusts its size to accommodate the number of elements it contains.

The associative containers currently provided in the STL are listed in Table 17-2.

Table 17-2 Associative Containers

Container Class	Description
<code>set</code>	Stores a set of unique values that are sorted. No duplicates are allowed.
<code>multiset</code>	Stores a set of unique values that are sorted. Duplicates are allowed.
<code>map</code>	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed. The elements are sorted in order of their keys.
<code>multimap</code>	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed. The elements are sorted in order of their keys.
<code>unordered_set</code>	Like a <code>set</code> , except that the elements are not sorted
<code>unordered_multiset</code>	Like a <code>multiset</code> , except that the elements are not sorted
<code>unordered_map</code>	Like a <code>map</code> , except that the elements are not sorted
<code>unordered_multimap</code>	Like a <code>multimap</code> , except that the elements are not sorted

In addition to the classes listed in Tables 17-1 and 17-2, the STL also provides the three container adapter classes listed in Table 17-3. A container *adapter class* is not itself a container,

but a class that adapts one of the other containers to a specific use. Note we will discuss these classes in Chapter 19.

Table 17-3 Container Adapter Classes

Container Adapter Class	Description
stack	An adapter class that stores elements in a deque (by default). A stack is a last-in, first-out (LIFO) container. When you retrieve an element from a stack, the stack always gives you the last element that was inserted. (This class will be discussed in Chapter 19.)
queue	An adapter class that stores elements in a deque (by default). A queue is a first-in, first-out (FIFO) container. When you retrieve an element from a stack, the stack always gives you the first, or earliest, element that was inserted. (This class will be discussed in Chapter 19.)
priority_queue	An adapter class that stores elements in a vector (by default). A data structure in which the element that you retrieve is always the element with the greatest value. (This class will be discussed in Chapter 19.)

The container and container adapter classes are declared in various STL header files. Table 17-4 lists the necessary header files that you will need to include, depending on the containers you intend to use.

Table 17-4 Header Files

Header File	Classes
<array>	array
<deque>	deque
<forward_list>	forward_list
<list>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<unordered_map>	unordered_map, unordered_multimap
<unordered_set>	unordered_set, unordered_multiset
<vector>	vector

Introduction to the array Class

Perhaps the simplest container in the STL is the `array` class, which was introduced in C++ 11. An `array` object works very much like a regular array. It is a fixed-size container that holds elements of the same data type. In fact, an `array` object uses a traditional array, internally, to store its elements. An advantage `array` objects have over regular arrays is that `array` objects have a `size()` member function that returns the number of elements contained in the object.



When you define an **array** object, you must specify the data type of its elements, and the number of elements the object will store. Here is an example:

```
array<int, 5> numbers;
```

This statement defines an **array** object named **numbers**. Inside the angled brackets **<>**, the data type **int** is the type of each element, and the value **5** is the number of elements the object will have. You can provide an initialization list to initialize an **array**, as shown here:

```
array<int, 5> numbers = {1, 2, 3, 4, 5};
```

Here is an example of defining an **array** to hold four strings:

```
array<string, 4> names = {"Jamie", "Ashley", "Doug", "Claire"};
```

The **array** class overloads the **[]** operator, giving you the ability to access elements using a subscript, just as you would with a regular array. Program 17-1 gives a simple demonstration of creating and initializing an **array** object, and displaying its elements.

Program 17-1

```

1 #include <iostream>
2 #include <string>
3 #include <array>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 4;
9
10    // Store some names in an array object.
11    array<string, SIZE> names = {"Jamie", "Ashley", "Doug", "Claire"};
12
13    // Display the names.
14    cout << "Here are the names:\n";
15    for (int index = 0; index < names.size(); index++)
16        cout << names[index] << endl;
17
18    return 0;
19 }
```

Program Output

Here are the names:
 Jamie
 Ashley
 Doug
 Claire

Take a closer look at the following lines of code in Program 17-1:

- The statement in line 11 defines an **array** object named **names**, initialized with four strings.
- The **for** loop in lines 15 and 16 displays all of the strings. Notice the **array** object's **size()** member function is used in the loop's test expression.
- In line 16, the **[]** operator is used to access the object's elements.

The `for` loop in lines 15 and 16 of Program 17-1 is useful to demonstrate the use of the `array` class's `size()` member function, but we could easily rewrite it as a range-based `for` loop.

```
for (auto element : names)
    cout << element << endl;
```

WARNING! The `array` class's `[]` operator does not perform bounds checking. As with regular arrays, you must be careful not to use a subscript that is out of bounds with an `array` object.

As you can see, the `array` class is an object-oriented alternative to a traditional array in C++. The `array` class also provides several member functions that give additional capabilities. Table 17-5 lists the `array` class's member functions.

Table 17-5 The `array` Member Functions

Member Function	Description
<code>at(index)</code>	Returns a reference to the element located at the specified index. If the specified index is out of bounds, the <code>at</code> function throws an <code>out_of_bounds</code> exception.
<code>back()</code>	Returns a reference to the last element in the container.
<code>begin()</code>	Returns an iterator to the first element in the container.
<code>cbegin()</code>	Returns a <code>const_iterator</code> to the first element in the container.
<code>cend()</code>	Returns a <code>const_iterator</code> pointing to the end of the container.
<code>crbegin()</code>	Returns a <code>const_reverse_iterator</code> pointing to the last element in the container.
<code>crend()</code>	Returns a <code>const_reverse_iterator</code> pointing to the first element in the container.
<code>data()</code>	Returns a pointer to the first element in the container. The <code>array</code> class uses a traditional array to store its data, so the pointer returned from the <code>data()</code> function points to the first element in the underlying array.
<code>empty()</code>	Returns <code>true</code> if the container is empty, or <code>false</code> otherwise.
<code>end()</code>	Returns an iterator pointing to the end of the container.
<code>fill(value)</code>	Sets the value of each element to <code>value</code> .
<code>front()</code>	Returns a reference to the first element in the container.
<code>max_size()</code>	Returns the number of elements in the container (the same value returned by the <code>size()</code> member function).
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> pointing to the last element in the container.
<code>rend()</code>	Returns a <code>reverse_iterator</code> pointing to the first element in the container.
<code>size()</code>	Returns the number of elements in the container.
<code>swap(second)</code>	The <code>second</code> argument must be an <code>array</code> object of the same type and size as the calling object. The function swaps the contents of the calling object and the <code>second</code> object.

Notice many of the member functions listed in Table 17-5 return iterators. Iterators are a fundamental part of the STL, so it is important to have an understanding of them.



Iterators

Iterators are objects that work like pointers, and are used to access data stored in containers. As the name implies, you can use an iterator to iterate over the contents of a container. There are currently five categories of iterators, as described in Table 17-6.¹

Table 17-6 Categories of Iterators

Iterator Category	Description
Forward	Can only move forward in a container (uses the <code>++</code> operator).
Bidirectional	Can move forward or backward in a container (uses the <code>++</code> and <code>--</code> operators).
Random access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

Iterators are associated with containers. The type of container you have determines the category of iterator you use. For example:

- The `array`, `vector`, and `deque` containers use random-access iterators.
- The `list`, `set`, `multiset`, `map`, and `multimap` containers use bidirectional iterators.
- The `forward_list`, `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset` containers use forward iterators.

Iterators are like pointers in the following ways:

- An iterator can point to an element in a container.
- You can use the `*` operator to dereference an iterator, getting the element that it points to.
- When an iterator points to an object, you can use the `->` operator to access the object's members.
- You can use the `=` operator to assign an iterator to an element.
- You can use the `==` and `!=` operators to compare two iterators.
- You can use the `++` operator to increment an iterator, thus moving it to the next element in the container. You can also use the `+` operator to add an integer to an iterator, thus moving it forward in the container by a number of elements.
- If you are using a bidirectional or random-access iterator, you can use the `--` operator to decrement the iterator, thus moving it to the previous element in the container. You can also use the `-` operator to subtract an integer from an iterator, thus moving it backward in the container by a number of elements.

Defining an Iterator

Before you can define an iterator, you have to know the type of container with which you want to use the iterator. The general format of an iterator definition is:

```
containerType::iterator iteratorName;
```

¹ The C++17 standard will introduce a sixth category, the contiguous iterator. A contiguous iterator is a random-access iterator that points to elements known to be stored in contiguous memory locations.

In the general format, *containerType* is the STL container type, and *iteratorName* is the name of the iterator variable that you are defining. For example, suppose we have defined an **array** object, as follows:

```
array<string, 3> names = {"Sarah", "William", "Alfredo"};
```

We can define an iterator that will work with the **array** object with the following statement:

```
array<string, 3>::iterator it;
```

This statement defines an iterator named *it*, that can be used with an **array<string, 3>** object. The compiler automatically chooses the right type of iterator (in this case, a random-access iterator).

Getting an Iterator from a Container Object

All of the container classes in the STL provide a **begin()** member function and an **end()** member function. The **begin()** member function is straightforward: It returns an iterator pointing to the first element in a container. For example, the following code snippet does the following:

1. It defines an **array** object initialized with three strings.
2. It defines an iterator that can be used with the **array** object.
3. It makes the iterator point to the first element in the **array** object.
4. It uses the ***** operator to dereference the iterator, and displays the element to which the iterator points.

```
// Define an array object. (1)
array<string, 3> names = {"Sarah", "William", "Alfredo"};

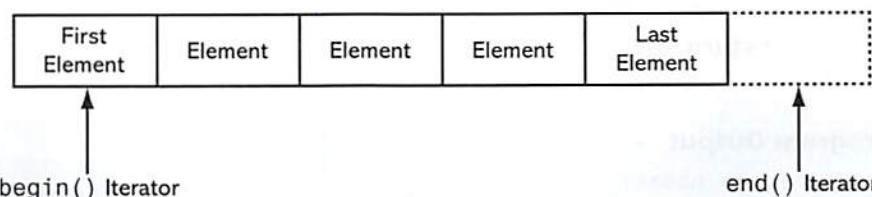
// Define an iterator for the array object. (2)
array<string, 3>::iterator it;

// Make the iterator point to the array object's first element. (3)
it = names.begin();

// Display the element that the iterator points to. (4)
cout << *it << endl;
```

The **end()** member function returns an iterator pointing to the position *after* the last element. This means it returns an iterator that does not point to an element, but points to the end of the container. This is illustrated in Figure 17-1.

Figure 17-1 Iterators returned from the **begin()** and **end()** member functions



You typically use the **end()** member function to know when you have reached the end of a container in algorithms that iterate over a range of elements. For example, the following

code snippet shows an iterator being used with a `while` loop to display the contents of an array object:

```
// Define an array object.
array<string, 3> names = {"Sarah", "William", "Alfredo"};

// Define an iterator for the array object.
array<string, 3>::iterator it;

// Make the iterator point to the array object's first element.
it = names.begin();

// Display the array object's contents.
while (it != names.end())
{
    cout << *it << endl;
    it++;
}
```

The code previously shown could be shortened somewhat by using a `for` loop instead of a `while` loop. Program 17-2 shows an example.

Program 17-2

```
1 #include <iostream>
2 #include <string>
3 #include <array>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 3;
9
10    // Store some names in an array object.
11    array<string, SIZE> names = {"Sarah", "William", "Alfredo"};
12
13    // Create an iterator for the array object.
14    array<string, SIZE>::iterator it;
15
16    // Display the names.
17    cout << "Here are the names:\n";
18    for (it = names.begin(); it != names.end(); it++)
19        cout << *it << endl;
20
21    return 0;
22 }
```

Program Output

Here are the names:

Sarah
William
Alfredo

Using auto to Define an Iterator

Quite often, you can use `auto` to simplify the definition of an iterator. For example, any time you are initializing an iterator with the value that is returned from a member function, such as `begin()`, you can use `auto` in the iterator's declaration. For example, look at the following code:

```
array<string, 3> names = {"Sarah", "William", "Alfredo"};
array<string, 3>::iterator it = names.begin();
```

The second statement defines an iterator named `it`, and calls the `names.begin()` member function to initialize it. This statement can be rewritten, as shown in the following code:

```
array<string, 3> names = {"Sarah", "William", "Alfredo"};
auto it = names.begin();
```

The `auto` declaration shown in the second statement works because the `names.begin()` function returns an object of the `array<string, 3>::iterator` type. So, the compiler can determine that `it` should be an object of the `array<string, 3>::iterator` type. This is especially useful in `for` loops that use iterators to step through the elements of a container. For example, look at Program 17-3. It is similar to Program 17-2, but in this version, the iterator is `auto` declared in the initialization expression of the `for` loop (in line 15).

Program 17-3

```
1 #include <iostream>
2 #include <string>
3 #include <array>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 4;
9
10    // Store some names in an array object.
11    array<string, SIZE> names = {"Jamie", "Ashley", "Doug", "Claire"};
12
13    // Display the names.
14    cout << "Here are the names:\n";
15    for (auto it = names.begin(); it != names.end(); it++)
16        cout << *it << endl;
17
18    return 0;
19 }
```

Program Output

Here are the names:

Sarah
William
Alfredo

Mutable Iterators and `const_iterator`

An iterator of the `iterator` type gives you read/write access to the element to which the iterator points. This is commonly known as a *mutable iterator*. The following code snippet shows an example:

```
// Define an array object.
array<int, 5> numbers = {1, 2, 3, 4, 5};

// Define an iterator for the array object.
array<int, 5>::iterator it;

// Make the iterator point to the array object's first element.
it = numbers.begin();

// Use the iterator to change the element.
*it = 99;
```

In this code snippet, the `numbers` object initially contains the values 1, 2, 3, 4, and 5. The last statement dereferences the `it` iterator (which points to the first element) and changes the element's value to 99. After the last statement executes, the `numbers` object contains the values 99, 2, 3, 4, and 5.

If you want to make sure that nothing changes the contents of a container, you can use the `const` modifier with the container definition to make the container's contents constant. Here is an example:

```
const array<string, 3> names = {"Sarah", "William", "Alfredo"};
```

This statement defines `names` as a `const array` object. As a result, the elements in the container are constant, and cannot be modified. (Attempting to modify the container's contents will result in an error at compile time.) If we need an iterator to work with the object, we must use a `const_iterator`, which provides read-only access to any element to which it points. The following statement shows an example of how to define a `const_iterator`:

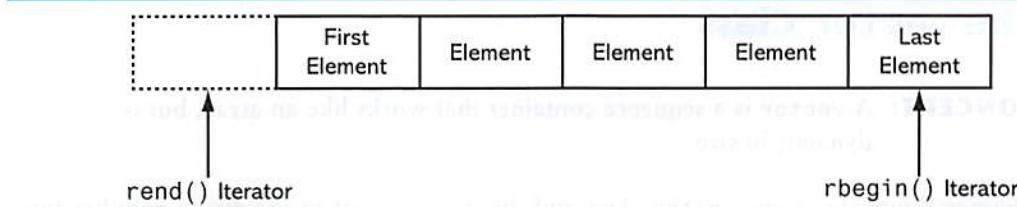
```
array<string, 3>::const_iterator it;
```

All of the container classes in the STL provide a `cbegin()` member function and a `cend()` member function. The `cbegin()` member function returns a `const_iterator` pointing to the first element in a container. The `cend()` member function returns a `const_iterator` pointing to the end of the container. When working with `const_iterators`, simply use the container class's `cbegin()` and `cend()` member functions instead of the `begin()` and `end()` member functions.

Reverse Iterators

A *reverse iterator* is a bidirectional or random-access iterator that works in reverse, allowing you to iterate backward over the elements in a container. When using a reverse iterator, the last element in a container is considered the first element, and the first element is considered the last element. The `++` operator moves a reverse iterator backward, and the `--` operator moves a reverse iterator forward.

The following STL containers support the use of reverse iterators: `array`, `deque`, `list`, `map`, `multimap`, `multiset`, `set`, and `vector`. All of these classes provide an `rbegin()` member function and an `rend()` member function. The `rbegin()` member function returns a reverse iterator pointing to the last element in a container, and the `rend()` member function returns an iterator pointing to the position *before* the first element. This is illustrated in Figure 17-2.

Figure 17-2 Iterators returned by the `rbegin()` and `rend()` member functions

To create a reverse iterator, you define it as `reverse_iterator`. The following code snippet shows an example of using a reverse iterator to display the contents of an `array` object in reverse order:

```
// Define an array object.  
array<int, 5> numbers = {1, 2, 3, 4, 5};  
  
// Define a reverse iterator for the array object.  
array<int, 5>::reverse_iterator it;  
  
// Display the elements in reverse order.  
for (it = numbers.rbegin(); it != numbers.rend(); it++)  
    cout << *it << endl;
```

Iterators that are declared as `reverse_iterator` are mutable, meaning they give you read/write access to the element being pointed to. If you need a reverse iterator to work with a `const` container, you must use a `const_reverse_iterator`, which provides read-only access to any element that it points to. The following code shows an example of how to create a `const_reverse_iterator`:

```
const array<string, 3> names = {"Sarah", "William", "Alfredo"};  
array<string, 3>::const_reverse_iterator it;
```

Each of the STL container classes that supports reverse iterators provides a `crbegin()` member function and a `crend()` member function. The `crbegin()` member function returns a `const_reverse_iterator` pointing to the last element in the container. The `crend()` member function returns a `const_reverse_iterator` pointing to the position before the first element in the container.



Checkpoint

- 17.1 What two types of containers does the STL provide?
- 17.2 What is a container adapter class?
- 17.3 What is an iterator?
- 17.4 Suppose you are writing a program that uses the `array`, `multimap`, and `vector` classes. What header files must you `#include` in the program, in order to use these classes?
- 17.5 What is the difference between a bidirectional iterator and a random-access iterator?
- 17.6 What does the `++` operator do when applied to an iterator?
- 17.7 What does a container's `begin()` and `end()` member functions return?
- 17.8 What is the difference between a mutable iterator and a `const_iterator`?
- 17.9 What is a reverse iterator?
- 17.10 What does a container's `rbegin()` and `rend()` member functions return?

17.3 The vector Class

CONCEPT: A **vector** is a sequence container that works like an array, but is dynamic in size.



Chapter 7 introduces the **vector** class and discusses several of the class's member functions. In this section, we will take a closer look at the class, and discuss how to use iterators to access a **vector** container's elements.

Recall from our previous discussions that a **vector** stores a sequence of elements, like an array. In fact, a **vector** uses an array internally to store its elements. A **vector**, however, offers many advantages over an array:

- You do not have to declare the size of a **vector** when you define it.
- A **vector** is dynamic in size. You can add elements to it, or delete elements from it at runtime. The **vector** automatically adjusts its size according to the number of elements it contains.
- A **vector** can report the number of elements it contains.

To use the **vector** class, you need to `#include <vector>` header file in your program. Then, you can define a **vector** object using one of the four **vector** constructors. Table 17-7 shows the general format of a **vector** definition statement using each constructor. Table 17-8 lists most of the **vector** class's member functions.

Table 17-7 **vector** Definition Statements

Default Constructor	<code>vector<dataType> name;</code>
	Creates an empty vector object. In the general format, <i>dataType</i> is the data type of each element, and <i>name</i> is the name of the vector .
Fill Constructor	<code>vector<dataType> name(size);</code>
	Creates a vector object of a specified size. In the general format, <i>dataType</i> is the data type of each element, and <i>name</i> is the name of the vector . The <i>size</i> argument is an unsigned integer that specifies the number of elements that the vector should initially have. If the elements are objects, they are initialized via their default constructors. Otherwise, the elements are initialized with the value 0.
Fill Constructor	<code>vector<dataType> name(size, value);</code>
	Creates a vector object of a specified size, where each element is initially given a specified value. In the general format, <i>dataType</i> is the data type of each element, and <i>name</i> is the name of the vector . The <i>size</i> argument is an unsigned integer that specifies the number of elements that the vector should initially have, and the <i>value</i> argument is the value with which to fill each element.

Table 17-7 (continued)

Range Constructor	<code>vector<dataType> name(iterator1, iterator2);</code>
	Creates a <code>vector</code> object that initially contains a range of values specified by two iterators. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the vector. The <code>iterator1</code> and <code>iterator2</code> arguments mark the beginning and end of a range of values that will be stored in the vector.
Copy Constructor	<code>vector<dataType> name(vector2);</code>
	Creates a <code>vector</code> object that is a copy of another <code>vector</code> or object. In the general format, <code>dataType</code> is the data type of each element, <code>name</code> is the name of the vector, and <code>vector2</code> is the vector to copy.

Table 17-8 The vector Member Functions

Member Function	Description
<code>assign(size, value)</code>	Assigns a new set of elements to the container, replacing its existing elements. The <code>size</code> argument is an unsigned integer that specifies the number of elements that the vector should have, and the <code>value</code> argument is the value with which to fill each element. After the function executes, the container will have <code>size</code> elements, each set to <code>value</code> .
<code>assign(iterator1, iterator2)</code>	Assigns a new set of elements to the container, replacing its existing contents with a range of values specified by iterators. The <code>iterator1</code> and <code>iterator2</code> arguments mark the beginning and end of a range of values that will be stored in the container.
<code>at(index)</code>	The <code>index</code> argument is an unsigned integer. The function returns a reference to the element located at the specified index. (The first element is at index 0, the second element is at index 1, and so on.) If the specified index is out of bounds, the <code>at</code> function throws an <code>out_of_bounds</code> exception.
<code>back()</code>	Returns a reference to the last element in the container.
<code>begin()</code>	Returns an iterator to the first element in the container.
<code>capacity()</code>	Returns the number of elements that the container's underlying array can hold without reallocating the array.
<code>cbegin()</code>	Returns a <code>const_iterator</code> to the first element in the container.
<code>cend()</code>	Returns a <code>const_iterator</code> pointing to the end of the container.
<code>clear()</code>	Erases all of the elements in the container.
<code>crbegin()</code>	Returns a <code>const_reverse_iterator</code> pointing to the last element in the container.
<code>crend()</code>	Returns a <code>const_reverse_iterator</code> pointing to the first element in the container.
<code>data()</code>	Returns a pointer to the first element in the container. The <code>vector</code> class uses a traditional array to store its data, so the pointer returned from the <code>data()</code> function points to the first element in the underlying array.

(table continues)

Table 17-8 (continued)

Member Function	Description
<code>emplace(it, value)</code>	Constructs a new element with <code>value</code> as its value. The <code>it</code> argument is an iterator pointing to an existing element in the container. The new element will be inserted before the one pointed to by <code>it</code> .
<code>emplace_back(value)</code>	Constructs a new element containing the specified <code>value</code> at the end of the container.
<code>empty()</code>	Returns <code>true</code> if the container is empty, or <code>false</code> otherwise.
<code>end()</code>	Returns an iterator pointing to the end of the container.
<code>erase(it)</code>	Erases the element pointed to by the iterator <code>it</code> . This function returns an iterator pointing to the element that follows the removed element (or the end of the container, if the removed element was the last one).
<code>erase(iterator1, iterator2)</code>	Erases a range of elements. The <code>iterator1</code> and <code>iterator2</code> arguments mark the beginning and end of a range of values that will be erased. This function returns an iterator pointing to the element that follows the removed elements (or the end of the container, if the last element was erased).
<code>front()</code>	Returns a reference to the first element in the container.
<code>insert(it, value)</code>	Inserts a new element with <code>value</code> as its value. The <code>it</code> argument is an iterator pointing to an existing element in the container. The new element will be inserted before the one pointed to by <code>it</code> . The function returns an iterator pointing to the newly inserted element.
<code>insert(it, n, value)</code>	Inserts <code>n</code> new elements with <code>value</code> as their value. The <code>it</code> argument is an iterator pointing to an existing element in the container, and <code>n</code> is an unsigned integer. The new elements will be inserted before the one pointed to by <code>it</code> . The function returns an iterator pointing to the first element of the newly inserted elements.
<code>insert(iterator1, iterator2, iterator3)</code>	Inserts a range of new elements. The <code>iterator1</code> argument points to an existing element in the container. The range of new elements will be inserted before the element pointed to by <code>iterator1</code> . The <code>iterator2</code> and <code>iterator3</code> arguments mark the beginning and end of a range of values that will be inserted. (The element pointed to by <code>iterator3</code> will not be included in the range.) The function returns an iterator pointing to the first element of the newly inserted range.
<code>max_size()</code>	Returns the theoretical maximum size of the container.
<code>pop_back()</code>	Removes the last element of the container.
<code>push_back(value)</code>	Adds a new element containing <code>value</code> to the end of the container.
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> pointing to the last element in the container.
<code>rend()</code>	Returns a <code>reverse_iterator</code> pointing to the first element in the container.
<code>resize(n)</code>	The <code>n</code> argument is an unsigned integer. This function resizes the container so it has <code>n</code> elements. If the current size of the container is larger than <code>n</code> , then the container is reduced in size so it keeps only the first <code>n</code> elements. If the current size of the container is smaller than <code>n</code> , then the container is increased in size so it has <code>n</code> elements.

Table 17-8 (continued)

Member Function	Description
<code>resize(<i>n</i>, <i>value</i>)</code>	Resizes the container so it has <i>n</i> elements (the <i>n</i> argument is an unsigned integer). If the current size of the container is larger than <i>n</i> , then the container is reduced in size so it keeps only the first <i>n</i> elements. If the current size of the container is smaller than <i>n</i> , then the container is increased in size so that it has <i>n</i> elements, and each of the new elements is initialized with <i>value</i> .
<code>shrink_to_fit()</code>	Requests that the <code>vector</code> be resized so its capacity is the same as its size.
<code>size()</code>	Returns the number of elements in the container.
<code>swap(<i>second</i>)</code>	The <i>second</i> argument must be a <code>vector</code> object of the same type as the calling object. The function swaps the contents of the calling object and the <i>second</i> object.

Review of Basic vector Operations

Here is an example of how you would use the default constructor to define a `vector` container to hold integers:

```
vector<int> numbers;
```

This statement defines an empty `vector` container named `numbers`. Inside the angled brackets `<>`, the data type `int` is the type of each element. If you are using a compiler that is compliant with C++11 or later, you can use an initialization list to initialize the `vector`, as shown here:

```
vector<int> numbers = {1, 2, 3, 4, 5};
```

Here is an example of defining a `vector` to hold strings:

```
vector<string> names = {"Joe", "Karen", "Lisa", "Jackie"};
```

The `vector` class overloads the `[]` operator, giving you the ability to access elements using a subscript, just as you would with an array. Program 17-4 gives a simple demonstration of creating and initializing a `vector` object, and displaying its elements.

Program 17-4

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7     const int SIZE = 10;
8
9     // Define a vector to hold 10 int values.
10    vector<int> numbers(SIZE);
11

```

(program continues)

Program 17-4 *(continued)*

```

12     // Store the values 0 through 9 in the vector.
13     for (int index = 0; index < numbers.size(); index++)
14         numbers[index] = index;
15
16     // Display the vector elements.
17     for (auto element : numbers)
18         cout << element << " ";
19     cout << endl;
20
21     return 0;
22 }
```

Program Output

0 1 2 3 4 5 6 7 8 9

Keep in mind the [] operator has its limitations. Specifically, you can use it only to access elements that already exist in a vector. In other words, you cannot add new elements to an existing vector using the [] operator. For example, the following code will cause an error at run time:

```

vector<int> numbers;           // Define an empty vector
numbers[0] = 99;                // Error!
```

This code will cause an error because `numbers` is an empty vector. Therefore, the element `numbers[0]` does not exist. If you want to add new elements to a vector, you must use one of the member functions that exist for that purpose in the `vector` class. For example, the `push_back()` member function adds a new element to the end of the container. You simply call the function, passing the new element's value as an argument. The following code shows an example:

```

vector <int> numbers;
numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);
```

The first statement creates an empty vector container named `numbers`. The statements that follow add the values 10, 20, and 30 to the vector.

You can use the `at()` member function to retrieve a vector element by its index with bounds checking. The following code shows an example:

```

vector<string> names = {"Joe", "Karen", "Lisa"};
cout << names.at(3) << endl; // Throws an exception
```

The first statement in this code snippet defines a vector with three elements: “Joe” is at index 0, “Karen” is at index 1, and “Lisa” is at index 2. The second statement tries to display the element at index 3, which does not exist. As a result, the `at()` member function throws an `out_of_bounds` exception.

Using an Iterator with a vector

The vector class supports the use of iterators of the following types:

- iterator
- const_iterator
- reverse_iterator
- const_reverse_iterator

Notice in Table 17-8 the vector class provides the following member functions for getting iterators: `begin()`, `end()`, `cbegin()`, `cend()`, `rbegin()`, `rend()`, `crbegin()`, and `crend()`.

The following code shows an example of using an iterator to display all of the elements of a vector:

```
// Create a vector containing names.
vector<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
vector<string>::iterator it;

// Use the iterator to display each element in the vector.
for (it = names.begin(); it != names.end(); it++)
{
    cout << *it << endl;
}
```

We can simplify this code by auto declaring the iterator in the initialization expression of the `for` loop, as shown here:

```
// Create a vector containing names.
vector<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Use an iterator to display each element in the vector.
for (auto it = names.begin(); it != names.end(); it++)
{
    cout << *it << endl;
}
```

Inserting New Elements into a vector

You can use the `insert` member function to insert one or more new elements at a specified position in a vector. There are three overloaded versions of the `insert` member function shown in Table 17-8. Program 17-5 shows an example of how to use the first version.

Program 17-5

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7     // Define a vector with 5 int values.
8     vector<int> numbers = {1, 2, 3, 4, 5};
9 }
```

(program continues)

Program 17-5 (continued)

```

10     // Define an iterator pointing to the second element.
11     auto it = numbers.begin() + 1;
12
13     // Insert a new element with the value 99.
14     numbers.insert(it, 99);
15
16     // Display the vector elements.
17     for (auto element : numbers)
18         cout << element << " ";
19     cout << endl;
20
21     return 0;
22 }
```

Program Output

1 99 2 3 4 5

Let's take a closer look at the program:

- Line 8 defines a vector of ints, initialized with the values 1, 2, 3, 4, and 5.
- Line 11 defines an iterator that is pointing at the second element in the vector.
- Line 14 inserts a new element just before the one that it points to. The new element contains the value 99.
- The loop in lines 17 and 18 display all of the vector's elements.

The second version of the `insert` member function allows you to insert multiple elements, each with the same value. The following code snippet shows an example:

```

// Define a vector with 5 int values.
vector<int> numbers = {1, 2, 3, 4, 5};

// Define an iterator pointing to the second element.
auto it = numbers.begin() + 1;

// Insert 3 new elements, each with the value 99.
numbers.insert(it, 3, 99);
```

The last statement inserts three elements, each with the value 99. After the statement executes, the vector will contain the following elements: 1 99 99 99 2 3 4 5

The third version of the `insert` member function allows you to insert a range of elements, perhaps from a different container, into the vector. The function takes three iterators as arguments. The first iterator specifies the position where the new elements will be inserted. The second and third iterators mark the beginning and end of a range of elements to be inserted. Program 17-6 demonstrates the function by inserting a range of elements from one vector into another.

Program 17-6

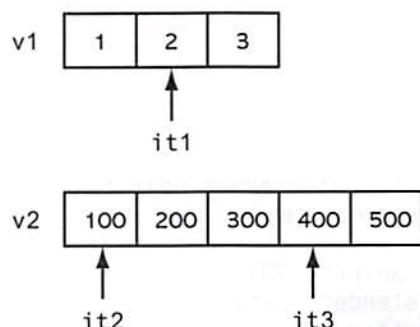
```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7     // Define two vectors.
8     vector<int> v1 = {1, 2, 3};
9     vector<int> v2 = {100, 200, 300, 400, 500};
10
11    // Define iterators
12    auto it1 = v1.begin() + 1;    // Points at 2 in v1
13    auto it2 = v2.begin();        // Points at 100 in v2
14    auto it3 = v2.begin() + 3;    // Points at 400 in v2
15
16    // Insert a range of elements into v1.
17    v1.insert(it1, it2, it3);
18
19    // Display the elements of v1.
20    for (auto element : v1)
21        cout << element << " ";
22    cout << endl;
23
24    return 0;
25 }
```

Program Output

1 100 200 300 2 3

Program 17-6 defines two vectors: `v1` and `v2`. Line 12 defines an iterator, `it1`, pointing to the second element of `v1`. Line 13 defines an iterator, `it2`, pointing to the first element of `v2`. Line 14 defines an iterator, `it3`, pointing to the fourth element of `v2`. These iterator positions are illustrated in Figure 17-3. Then, line 17 inserts the range of elements from `it2` to `it3` (not including `it3`) into `v1`, before the element at `it1`.

Figure 17-3 The iterators of Program 17-6



Storing Objects of Your Own Classes as Values in a vector

In the examples previously shown, we stored strings and primitive values in vectors. It is often useful to store objects of classes that you have written in a vector. For example, the `Product` class, shown here, keeps information about a product. Specifically, it keeps the product's name and the number of units on hand.

Contents of Product.h

```

1  #ifndef PRODUCT_H
2  #define PRODUCT_H
3  #include <string>
4  using namespace std;
5
6  class Product
7  {
8  private:
9      string name;
10     int units;
11 public:
12     Product(string n, int u)
13     { name = n;
14         units = u; }
15
16     void setName(string n)
17     { name = n; }
18
19     void setUnits(int u)
20     { units = u; }
21
22     string getName() const
23     { return name; }
24
25     int getUnits() const
26     { return units; }
27 };
28 #endif

```

Program 17-7 shows how we can define and initialize a vector to hold `Product` objects.

Program 17-7

```

1  #include <iostream>
2  #include <vector>
3  #include "Product.h"
4  using namespace std;
5
6  int main()
7  {
8      // Create a vector of Product objects.
9      vector<Product> products =
10     {
11         Product("T-Shirt", 20),
12         Product("Calendar", 25),
13         Product("Coffee Mug", 30)
14     };
15

```

```

16     // Display the vector elements.
17     for (auto element : products)
18     {
19         cout << "Product: " << element.getName() << endl
20             << "Units: " << element.getUnits() << endl;
21     }
22
23     return 0;
24 }
```

Program Output

```

Product: T-Shirt
Units: 20
Product: Calendar
Units: 25
Product: Coffee Mug
Units: 30
```

Program 17-8 shows an example of using the `push_back` member function to add `Product` objects to an existing `vector`, then using an iterator to display the contents of the `vector`. Notice the use of the `->` operator in lines 25 and 26. In those lines, it points to a `Product` object, so the `->` operator is used to call the object's `getName()` and `getUnits()` member functions.

Program 17-8

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include "Product.h"
5 using namespace std;
6
7 int main()
8 {
9     // Create Product objects.
10    Product prod1("T-Shirt", 20);
11    Product prod2("Calendar", 25);
12    Product prod3("Coffee Mug", 30);
13
14    // Create a vector to hold the Products
15    vector<Product> products;
16
17    // Add the products to the vector.
18    products.push_back(prod1);
19    products.push_back(prod2);
20    products.push_back(prod3);
```

(program continues)

Program 17-8

(continued)

```

22     // Use an iterator to display the vector contents.
23     for (auto it = products.begin(); it != products.end(); it++)
24     {
25         cout << "Product: " << it->getName() << endl
26         << "Units: " << it->getUnits() << endl;
27     }
28
29     return 0;
30 }
```

Program Output

```

Product: T-Shirt
Units: 20
Product: Calendar
Units: 25
Product: Coffee Mug
Units: 30
```

Inserting Elements with the `emplace()` and `emplace_back()` Member Functions

11

Member functions such as `insert()` and `push_back()` can cause temporary objects to be created in memory while the insertion is taking place. For programs that do not make a lot of insertions, the overhead of creating temporary objects is not usually a problem. However, for programs that insert a lot of objects into containers, member functions such as `insert()` and `push_back()` can be inefficient.

To improve runtime performance, C++11 introduced a new family of member functions that use a technique known as *emplacement* to insert new elements. The process of emplacement avoids the creation of temporary objects in memory while a new object is being inserted into a container. As a result, the emplacement functions are much more efficient than functions such as `insert()` and `push_back()`, especially when a lot of objects are being inserted into a container.

The `vector` class provides two member functions that use emplacement: `emplace()` and `emplace_back()`. When you use one of these member functions, it is not necessary to instantiate, ahead of time, the object you are going to insert. Instead, you pass to the emplacement function any arguments that you would normally pass to the constructor of the object you are inserting. The emplacement function handles the construction of the object, forwarding the arguments to its constructor.

Program 17-9 shows an example of how we can use the `emplace_back()` member function to add `Product` objects to a `vector`. Notice we do not instantiate the `Product` class in the program. Instead, in lines 12 through 14 we pass the arguments that we would normally pass to the `Product` class constructor, to the `emplace_back()` member function. The `emplace_back()` member function constructs the objects in the `vector`. (This type of object construction is known as *in-place construction*.)

Program 17-9

```

1 #include <iostream>
2 #include <vector>
3 #include "Product.h"
4 using namespace std;
5
6 int main()
7 {
8     // Create a vector to hold Products.
9     vector<Product> products;
10
11    // Add Products to the vector.
12    products.emplace_back("T-Shirt", 20);
13    products.emplace_back("Calendar", 25);
14    products.emplace_back("Coffee Mug", 30);
15
16    // Use an iterator to display the vector contents.
17    for (auto it = products.begin(); it != products.end(); it++)
18    {
19        cout << "Product: " << it->getName() << endl
20            << "Units: " << it->getUnits() << endl;
21    }
22
23    return 0;
24 }
```

Program Output

```

Product: T-Shirt
Units: 20
Product: Calendar
Units: 25
Product: Coffee Mug
Units: 30
```

The `emplace()` member function constructs an object at a specified location in the `vector`. The first argument you pass to the `emplace()` member function is an iterator. The new element will be constructed at the position just before the element that the iterator points to. After the iterator, you pass the list of arguments that is to be forwarded to the new object's constructor. Program 17-10 shows an example.

Program 17-10

```

1 #include <iostream>
2 #include <vector>
3 #include "Product.h"
4 using namespace std;
5
```

(program continues)

Program 17-10 (continued)

```

6 int main()
7 {
8     // Create a vector to hold Products.
9     vector<Product> products =
10    {
11        Product("T-Shirt", 20),
12        Product("Coffee Mug", 30)
13    };
14
15    // Get an iterator to the 2nd element.
16    auto it = products.begin() + 1;
17
18    // Insert another Product into the vector.
19    products.emplace(it, "Calendar", 25);
20
21    // Display the vector contents.
22    for (auto element : products)
23    {
24        cout << "Product: " << element.getName() << endl
25            << "Units: " << element.getUnits() << endl;
26    }
27
28    return 0;
29 }
```

Program Output

```

Product: T-Shirt
Units: 20
Product: Calendar
Units: 25
Product: Coffee Mug
Units: 30
```

Let's take a closer look at the program:

- Lines 9 through 13 create a vector named `products`, initialized with two `Product` objects.
- Line 16 defines an iterator, `it`, pointing to the second element in the vector.
- Line 19 inserts a new element into the vector, just before the element that `it` points to. The new element will be a `Product` object that is constructed, with the arguments “`Calendar`” and `25` passed as arguments to the constructor.

The `capacity()`, `max_size()`, `shrink_to_fit()`, and `reserve()` Member Functions

The `vector` container uses a dynamically allocated array to hold its elements. When the underlying array is full, and another element is added to the vector, the vector has to increase the size of its underlying array. Typically, this means that a new array must be

allocated, all of the elements of the old array must be copied to the new array, and the old array must be deallocated.

To prevent this process from happening each time a new element is added, it is common for the `vector` class to allocate more memory than it needs. That way, a certain number of elements can be added without causing the reallocation process to take place. So, a `vector` object has two sizes: (1) the number of elements that are stored in the `vector`, and (2) the number of elements that can be stored in the `vector` without causing its underlying array to be reallocated.

You already know that the `size()` member function returns the number of elements that are currently stored in the `vector`. The `vector` class also has a `capacity()` member function that returns the number of elements that the container's underlying array can currently store, without allocating more memory. In addition, the `max_size()` member function returns the theoretical maximum number of elements that the `vector` can ever store.

The value returned from a `vector`'s `capacity()` member function will always be greater than or equal to the value returned by the `size()` member function. If you need to increase a `vector`'s capacity, you can call the `reserve()` member function to make the request. You pass an integer argument to specify the desired number of elements. If you need to decrease a `vector`'s capacity, you can call the `shrink_to_fit()` member function to make the request. The function takes no arguments, and it simply shrinks the `vector`'s underlying array so its capacity is the same as the `vector`'s size.



Checkpoint

- 17.11 Write a statement that defines an empty `vector` object named `avect` that can hold strings.
- 17.12 Write a statement that defines a `vector` object named `avect` that can hold `ints`. The `vector` should have ten elements (initialized with the default value 0).
- 17.13 Write a statement that defines a `vector` object named `avect` that can hold `ints`. The `vector` should have 100 elements, each initialized with the value 1.
- 17.14 Write a statement that defines a `vector` object named `v1` that can hold `ints`. The `vector` should be a copy of another `vector` name `v2`.
- 17.15 What happens when you use an invalid index with the `vector` class's `at()` member function?
- 17.16 What is the difference between the `vector` class's `insert()` member function and `push_back()` member function?
- 17.17 If your program will be added a lot of objects to a `vector`, is it best to use the `inert()` member function, or the `emplace()` member function? Why?
- 17.18 Internally, how does a `vector` store its elements?
- 17.19 The `vector` class has a `size()` member function and a `capacity()` member function. What is the difference between these two member functions?

17.4 The map, multimap, and unordered_map Classes

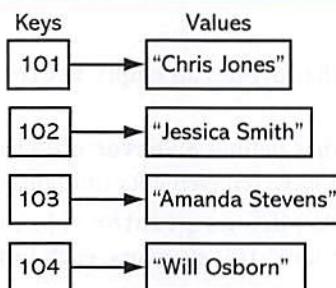
CONCEPT: Each element in a map has two parts: a key and a value. Each key is associated with a specific value, and can be used to locate that value.

A *map* is an associative container. Each element that is stored in a map has two parts: a *key* and a *value*. In fact, map elements are commonly referred to as *key-value pairs*. When you want to retrieve a specific value from a map, you use the key that is associated with that value. This is similar to the process of looking up a word in the dictionary, where the words are keys and the definitions are values.

For example, suppose each employee in a company has an ID number, and we want to write a program that lets us look up an employee's name by entering that employee's ID number. We could create a map in which each element contains an employee ID number as the key and that employee's name as the value. This is illustrated in Figure 17-4. If we know an employee's ID number, then we can retrieve that employee's name.

Another example would be a program that lets us enter a person's name and gives us that person's phone number. The program could use a map in which each element contains a person's name as the key and that person's phone number as the value. If we know a person's name, then we can retrieve that person's phone number.

Figure 17-4 Key-value pairs



NOTE: Key-value pairs are often referred to as *mappings* because each key is mapped to a value.

The map Class



The STL provides a `map` class template you can use to implement a map container in your programs. The `map` class provides many member functions for storing elements, retrieving elements, deleting elements, iterating over the map, and much more.

To use the `map` class, you need to `#include` the `<map>` header file in your program. Then, you can define a `map` object using one of the `map` class constructors. Table 17-9 shows the general format of a `map` definition statement using each constructor. Table 17-10 lists many (but not all) of the `map` class's member functions.

Table 17-9 map Definition Statements

Default Constructor	<code>map<keyDatatype, valueType> name;</code>
	Creates an empty map object. In the general format, <i>keyDatatype</i> is the data type of each element's key, <i>valueType</i> is the data type of each element's value, and <i>name</i> is the name of the map.
Range Constructor	<code>map<keyDatatype, valueType> name(iterator1, iterator2);</code>
	Creates a map object that initially contains a range of values specified by two iterators. In the general format, <i>keyDatatype</i> is the data type of each element's key, <i>valueType</i> is the data type of each element's value, and <i>name</i> is the name of the map. The <i>iterator1</i> and <i>iterator2</i> arguments mark the beginning and end of a range of elements that will be stored in the map.
Copy Constructor	<code>map<keyDatatype, valueType> name(map2);</code>
	Creates a map object that is a copy of another map or object. In the general format, <i>keyDatatype</i> is the data type of each element's key, <i>valueType</i> is the data type of each element's value, and <i>name</i> is the name of the map, and <i>map2</i> is the map to copy.

Table 17-10 Some of the map Member Functions

Member Function	Description
<code>at(key)</code>	Returns a reference to the element containing the specified key.
<code>begin()</code>	Returns an iterator pointing to the first element in the container.
<code>cbegin()</code>	Returns a <code>const_iterator</code> to the first element in the container.
<code>cend()</code>	Returns a <code>const_iterator</code> pointing to the end of the container.
<code>clear()</code>	Erases all of the elements in the container.
<code>count(key)</code>	Returns the number of elements containing the specified key.
<code>crbegin()</code>	Returns a <code>const_reverse_iterator</code> pointing to the last element in the container.
<code>crend()</code>	Returns a <code>const_reverse_iterator</code> pointing to the first element in the container.
<code>emplace(key, value)</code>	Inserts a new element containing the specified <i>key</i> and <i>value</i> into the container. If an element with the specified key already exists, the function call does nothing.
<code>empty()</code>	Returns <code>true</code> if the container is empty, or <code>false</code> otherwise.
<code>end()</code>	Returns an iterator pointing to the end of the container (the position <i>after</i> the last element).
<code>erase(key)</code>	Erases the element containing the specified key. The function returns 1 if the element was erased, or 0 if no matching element was found.
<code>find(key)</code>	Searches for an element with the specified key. If the element is found, the function returns an iterator to it. If the element is not found, the function returns an iterator to the end of the map.

(table continues)

Table 17-10 (continued)

Member Function	Description
<code>insert(pair)</code>	Inserts a <code>pair</code> object as an element to the map. If an element with the specified key already exists, the function call does nothing.
<code>lower_bound(key)</code>	Returns an iterator pointing to the first element with a key that is equal to or greater than <code>key</code> .
<code>max_size()</code>	Returns the theoretical maximum size of the container.
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> pointing to the last element in the container.
<code>rend()</code>	Returns a <code>reverse_iterator</code> pointing to the first element in the container.
<code>size()</code>	Returns the number of elements in the container.
<code>swap(second)</code>	The <code>second</code> argument must be a <code>map</code> object of the same type as the calling object. The function swaps the contents of the calling object and the <code>second</code> object.
<code>upper_bound(key)</code>	Returns an iterator pointing to the first element with a key that is greater than <code>key</code> .

Here is an example of how you might define a `map` container to hold employee ID numbers (as `ints`) and their corresponding employee names (as `strings`):

```
map<int, string> employees;
```

This statement defines a `map` container named `employees`. Inside the angled brackets `<>`, the first data type (`int`) is the type of each key, which in this case are the employee ID numbers. The second data type (`string`) is the type of the corresponding values, which in this case are the employee names. So, each element of this container will be a key-value pair where the key is an `int` and the value is a `string`.

The keys in a `map` container must be unique. No two elements in a `map` container can have the same key value.

Initializing a Map

You can initialize a `map` with an initialization list. The following code shows an example:

```
map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};
```

This code creates a `map` container named `employees`, and initializes it as follows:

- The first element is `{101, "Chris Jones"}`. In this element, 101 is the key and "Chris Jones" is the value.
- The second element is `{102, "Jessica Smith"}`. In this element, 102 is the key and "Jessica Smith" is the value.

- The third element is {103, "Amanda Stevens"}. In this element, 103 is the key and "Amanda Stevens" is the value.
- The fourth element is {104, "Will Osborn"}. In this element, 104 is the key and "Will Osborn" is the value.

Notice inside the initialization list:

- The initialization list is enclosed in a set of curly braces {}.
- Each element is enclosed in its own set of curly braces, with the key and the value separated by a comma.
- The elements are separated by commas.

As previously mentioned, the keys in a `map` must be unique. If two or more elements in an initialization list have the same key, only the first will be added to the `map`. The others will be ignored.

Adding Elements to an Existing Map

The `map` class overloads the [] operator, giving you the ability to add new elements to a `map` with an assignment statement in the following general format:

```
mapName[key] = value;
```

In the general format, `mapName` is the name of the map, and `key` is a key. If `key` already exists in the map, its associated value will be changed to `value`. If the `key` does not exist, it will be added to the map, along with `value` as its associated value. The following code shows an example:

```
map<int, string> employees;
employees[110] = "Beth Young";
employees[111] = "Jake Brown";
employees[112] = "Emily Davis";
```

The first statement creates an empty `map` container named `employees`. The statements that follow then add the following elements to the `map`:

- Key = 110, Value = "Beth Young"
- Key = 111, Value = "Jake Brown"
- Key = 112, Value = "Emily Davis"

Remember, the keys in a `map` must be unique. If you assign a new value to an existing key, the new value will replace the previously stored value. For example, look at the following code:

```
map<int, string> employees;
employees[110] = "Beth Young";
employees[110] = "Jake Brown";
```

The first statement creates an empty `map` container named `employees`. The second statement adds an element with the key 110 and the value "Beth Young". The third statement assigns the value "Jake Brown" to the element with the key 110.

Adding Elements with the `insert()` Member Function

When an element is stored in a `map`, it is stored as object of the `pair` type. The `pair` type is a struct that has two member variables: `first` and `second`. The element's key is stored in the `first` member variable, and the element's value is stored in the `second` member variable.

The `map` class provides an `insert()` member function that adds a `pair` object as an element to the `map`. You can use the STL function `make_pair` to construct a `pair` object, as shown in the following code:

```
map<int, string> employees;
employees.insert(make_pair(110, "Beth Young"));
employees.insert(make_pair(111, "Jake Brown"));
employees.insert(make_pair(112, "Emily Davis"));
```

The first statement creates an empty `map` container named `employees`. The statements that follow then add the following elements to the `map`:

- Key = 110, Value = “Beth Young”
- Key = 111, Value = “Jake Brown”
- Key = 112, Value = “Emily Davis”

If the element that you are inserting with the `insert()` member function has the same key as an existing element, the function will not insert the new element.



NOTE: The `pair` struct and the `make_pair` function template are declared in the `<utility>` header file. If you have included the `<map>` header file, the `<utility>` header file will be automatically included as well.

Adding Elements with the `emplace()` Member Function

The `map` class provides an `emplace()` member function that constructs a new element in the container. You simply call the member function, passing the new element's key and value as arguments. The following code shows an example:

```
map<int, string> employees;
employees.emplace(110, "Beth Young");
employees.emplace(111, "Jake Brown");
employees.emplace(112, "Emily Davis");
```

The first statement creates an empty `map` container named `employees`. The statements that follow then add the following elements to the `map`:

- Key = 110, Value = “Beth Young”
- Key = 111, Value = “Jake Brown”
- Key = 112, Value = “Emily Davis”

If the element that you are inserting with the `emplace()` member function has the same key as an existing element, the function will not insert the new element.

Retrieving Values from a Map

To retrieve a value from a `map`, you call the `at()` member function, passing the key that is associated with the desired value. If the specified key exists in the container, the function returns its associated value. If the key does not exist, the function throws an exception. The following code shows an example:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Retrieve a value from the map.
cout << employees.at(103) << endl;
```

This code retrieves and displays the value that is associated with the key 103 from the map. The code will display the string “Amanda Stevens”.

If the key you pass to the `at()` member function does not exist in the map, the `at()` member function will throw an exception. To prevent this, you should first call the `count()` member function to determine whether the key exists in the map. You call the `count()` member function, passing a key as the argument. If the specified key exists in the container, the function returns 1. If the key does not exist, the function returns 0. The following code shows an example:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Retrieve a value from the map.
if (employees.count(103))
    cout << employees.at(103) << endl;
else
    cout << "Employee not found.\n";
```

Deleting Elements

You can delete, or erase, an element from a `map` using the `erase()` member function. You call the `erase()` member function, passing the key that is associated with the element you want to erase. The following code shows an example:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Delete the employee with the ID 102.
employees.erase(102);
```

This code creates a `map` container with four elements. Then, it deletes the element with the key 102 (and the value “Jessica Smith”). The `map` then contains only three elements.

The `erase()` member function returns 1 if the specified element was successfully erased, or 0 if it was not found.

Iterating Over a Map with the Range-Based for Loop

The range-based for loop is a convenient way to iterate over all of the elements in a map. The following code shows an example of how to display all of the elements in a map:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Display each element.
for (auto element : employees)
{
    cout << "ID:" << element.first << "\tName:" << element.second << endl;
}
```

Notice we used the `auto` key word in the declaration of the range variable, `element`. The `auto` key word tells the compiler to determine the variable's data type from the context in which it is being used. If we had written the actual data type of the `element` variable, the loop would have looked like this:

```
for (pair<int, string> element : employees)
{
    cout << "ID: " << element.first << "\tName:" << element.second << endl;
}
```

As previously mentioned, elements are stored in a `map` container as objects of the `pair` type. The `pair` type is a `struct` that has two member variables: `first` and `second`. The element's key is stored in the `first` member variable, and the element's value is stored in the `second` member variable.

Using an Iterator with a Map

You can use a bidirectional iterator to access the elements of a `map`. The `map` class has a `begin()` member function that returns an iterator pointing to the first element in the `map`, and an `end()` member function that returns an iterator pointing to the end of the `map` (the position *after* the last element). Program 17-11 shows an example of using an iterator to display all of the elements of a `map`.

Program 17-11

```
1 // This program demonstrates an iterator with a map.
2 #include <iostream>
3 #include <string>
4 #include <map>
5 using namespace std;
6
7 int main()
8 {
9     // Create a map containing employee IDs and names.
10    map<int, string> employees =
11        { {101,"Chris Jones"}, {102,"Jessica Smith"},
12          {103,"Amanda Stevens"},{104,"Will Osborn"} };
```

```

14     // Create an iterator.
15     map<int, string>::iterator iter;
16
17     // Use the iterator to display each element in the map.
18     for (iter = employees.begin(); iter != employees.end(); iter++)
19     {
20         cout << "ID: " << iter->first
21             << "\tName: " << iter->second << endl;
22     }
23
24     return 0;
25 }
```

Program Output

```

ID: 101 Name: Chris Jones
ID: 102 Name: Jessica Smith
ID: 103 Name: Amanda Stevens
ID: 104 Name: Will Osborn
```

Let's take a closer look at Program 17-11:

- Lines 10 through 12 define a `map` container named `employees`, and initialize it with four elements. Notice the `map`'s type is `map<int, string>`.
- Line 15 defines an iterator that will work with an object of the type `map<int, string>`. The iterator's name is `iter`.
- The `for` loop that appears in lines 18 through 22 uses the iterator to step through the `map` container.
 - The loop's initialization expression `iter = employees.begin()` causes the iterator to point to the first element in the `map`.
 - The test expression `iter != employees.end()` causes the loop to execute as long as the iterator does not point to the end of the `map`.
 - The update expression increments the iterator, moving it to the next element in the `map`.
 - The body of the loop, in lines 20 and 21, displays the contents of the element to which the iterator is currently pointing. Remember, each element of the `map` is a `pair` object, with the key stored in the `first` member, and the value stored in the `second` member.



NOTE: When you access the elements of a `map` from first to last, you access them in order of their keys.



NOTE: The `end()` member function returns an iterator pointing to the end of the `map`, but it does not point to an actual element. It points to the position where an additional element would exist, if it appeared after the last element.

The `map` class also has a `find()` member function that searches for an element with a specified key. If the element is found, the `find()` function returns an iterator to it. If the element is not found, the `find()` function returns an iterator to the end of the `map`. The following code shows an example:

```
// Create a map containing employee IDs and names.
map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Create an iterator.
map<int, string>::iterator iter;

// Find employee 103.
iter = employees.find(103);

// Display the employee data.
if (iter != employees.end())
{
    cout << "ID: " << iter->first
        << "\tName: " << iter->second << endl;
}
else
{
    cout << "Employee not found.\n";
}
```

Storing vectors as Values in a map

In Program 17-12, we define a `map` in which the keys are student names (as strings), and the values are `vectors` containing test scores.

Program 17-12

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <map>
5 using namespace std;
6
7 int main()
8 {
9     // Create vectors to hold test scores.
10    vector<int> student1Scores = {88, 92, 100};
11    vector<int> student2Scores = {95, 74, 81};
12    vector<int> student3Scores = {72, 88, 91};
13    vector<int> student4Scores = {70, 75, 78};
14
15    // Create a map to hold all the test scores.
16    map<string, vector<int>> testScores;
17    testScores["Kayla"] = student1Scores;
18    testScores["Luis"] = student2Scores;
19    testScores["Sophie"] = student3Scores;
20    testScores["Ethan"] = student4Scores;
21 }
```

```

22     // Display each student's test scores.
23     for (auto element : testScores)
24     {
25         // Display the student name.
26         cout << "Student: " << element.first << endl;
27
28         // Display the student's test scores.
29         for (int i = 0; i < element.second.size(); i++)
30         {
31             cout << "\t" << element.second[i] << endl;
32         }
33     }
34     return 0;
35 }
```

Program Output

Student: Ethan

70

75

78

Student: Kayla

88

92

100

Student: Luis

95

74

81

Student: Sophie

72

88

91

Let's take a closer look at Program 17-12:

- Lines 10 through 13 create four vectors, each holding a different student's test scores.
- Line 16 defines a map container named `testScores`. Notice the map's type is `map<string, vector<int>>`. This indicates the keys will be of the `string` type, and the values will be of the `vector<int>` type.
- Lines 17 through 20 add the following elements to the map:

Key = "Kayla", Value = `student1Scores`, which is a `vector<int>` object

Key = "Luis", Value = `student2Scores`, which is a `vector<int>` object

Key = "Sophie", Value = `student3Scores`, which is a `vector<int>` object

Key = "Ethan", Value = `student4Scores`, which is a `vector<int>` object

- The range-based for loop that begins in line 23 iterates over each element in the map. Remember that the range variable, `element`, is an object of the `pair` data type. The `element` object's `first` member variable is the element's key, and its `second` member variable is the element's value. So, the expression `element.first` is a string, and the expression `element.second` is a `vector<int>` object. In line 31, the expression `element.second[i]` is an element in the vector.

Program 17-12 illustrates the process of storing vectors as values in a map, but the program can be simplified in two ways: (1) When we define the map, we can use an initialization list that also creates the vector objects. That will allow us to eliminate the vector definition statements in lines 10 through 13. (2) We can rewrite the inner for loop in lines 29 through 32 as a range-based for loop. Program 17-13 shows the program with these changes.

Program 17-13

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <map>
5 using namespace std;
6
7 int main()
8 {
9     // Create vectors to hold test scores.
10    map<string, vector<int>> testScores =
11        { {"Kayla", vector<int> {88, 92, 100 }},
12          {"Luis", vector<int> {95, 74, 81 }},
13          {"Sophie", vector<int> {72, 88, 91 }},
14          {"Ethan", vector<int> {70, 75, 78 }} };
15
16    // Display each student's test scores.
17    for (auto element : testScores)
18    {
19        // Display the student name.
20        cout << "Student: " << element.first << endl;
21
22        // Display the student's test scores.
23        for (auto score : element.second)
24        {
25            cout << "\t" << score << endl;
26        }
27    }
28    return 0;
29 }
```

Program Output

Student: Ethan

70

75

78

Student: Kayla

88

92

100

Student: Luis

95

74

81

Student: Sophie

72

88

91

Storing Objects of Your Own Classes as Values in a map

In the examples previously shown, we stored strings, primitive values, and vectors in maps. It is often useful to store objects of classes that you have written in a map. For example, suppose you are writing a program to keep a contact list, and you want to search for a person's contact information by entering that person's name. You could create a map that contains an element for each person in the contact list. In each element, the key is a person's name, and the value is an object containing that person's contact data. You can search the map, using a person's name as the key, and retrieve the object containing that person's contact data.

If you want to store an object as a value in a map, there is one requirement for that object's class: It must have a default constructor. For example, the Contact class shown here holds a person's name and e-mail address. Because it has a default constructor (in lines 12 through 14), objects of this class can be stored as values in a map.

Contents of Contact.h

```

1 #ifndef CONTACT_H
2 #define CONTACT_H
3 #include <string>
4 using namespace std;
5
6 class Contact
7 {
8 private:
9     string name;
10    string email;
11 public:
12     Contact()
13     {   name = "";
14      email = ""; }
15
16     Contact(string n, string em)
17     {   name = n;
18      email = em; }
19
20     void setName(string n)
21     {   name = n; }
22
23     void setEmail(string em)
24     {   email = em; }
25
26     string getName() const
27     {   return name; }
28
29     string getEmail() const
30     {   return email; }
31 };
32 #endif

```

Program 17-14 demonstrates how to store objects of the Contact class in a map. The program creates three Contact objects, then stores them in a map, using each Contact object's name field as the key. The user is prompted to enter a name to search for.

Program 17-14

```
1 #include <iostream>
2 #include <string>
3 #include <map>
4 #include "Contact.h"
5 using namespace std;
6
7 int main()
8 {
9     string searchName; // The name to search for
10
11    // Create some Contact objects
12    Contact contact1("Ashley Miller", "amiller@faber.edu");
13    Contact contact2("Jacob Brown", "jbrown@gotham.edu");
14    Contact contact3("Emily Ramirez", "eramirez@coolidge.edu");
15
16    // Create a map to hold the Contact objects.
17    map<string, Contact> contacts;
18
19    // Create an iterator for the map.
20    map<string, Contact>::iterator iter;
21
22    // Add the contact objects to the map.
23    contacts[contact1.getName()] = contact1;
24    contacts[contact2.getName()] = contact2;
25    contacts[contact3.getName()] = contact3;
26
27    // Get the name to search for.
28    cout << "Enter a name: ";
29    getline(cin, searchName);
30
31    // Search for the name.
32    iter = contacts.find(searchName);
33
34    // Display the results.
35    if (iter != contacts.end())
36    {
37        cout << "Name: " << iter->second.getName() << endl;
38        cout << "Email: " << iter->second.getEmail() << endl;
39    }
40    else
41    {
42        cout << "Contact not found.\n";
43    }
44
45    return 0;
46 }
```

Program Output (with Example Input Shown in Bold)Enter a name: **Emily Ramirez**

Name: Emily Ramirez

Email: eramirez@coolidge.edu

Program Output (with Example Input Shown in Bold)Enter a name: **Billy Clark**

Contact not found.

Let's take a closer look at Program 17-14:

- Lines 12 through 14 create three `Contact` objects named `contact1`, `contact2`, and `contact3`.
- Line 17 defines a `map` named `contacts`. Inside the angled brackets `<>`, the first data type (`string`) is the type of each key. The second data type (`Contact`) is the type of the corresponding values. So, each element of this container will be a key-value pair where the key is a `string` and the value is a `Contact` object.
- Line 20 defines an iterator that can be used with the `contacts` map.
- Line 23 adds an element to the `contacts` map. The key is the string “Ashley Miller”, and the value is the `contact1` object.
- Line 24 adds another element to the `contacts` map. The key is the string “Jacob Brown”, and the value is the `contact2` object.
- Line 25 adds another element to the `contacts` map. The key is the string “Emily Ramirez”, and the value is the `contact3` object.
- Lines 28 and 29 prompt the user to enter a name to search for. The user’s input is stored in `searchName`.
- Line 32 calls the `contacts.find()` member function to search for the name that was entered by the user. If the name is found as a key in the map, the member function returns an iterator to the element. If the name is not found, the member function returns an iterator to the end of the map.
- If the name was found, the statements in lines 37 and 38 display the name and e-mail address for the contact. Remember, the iterator is pointing to a `pair` object. The `pair` object’s `first` member variable is the element’s key, and the `pair` object’s `second` member variable is the element’s value. So, in lines 37 and 38, the expression `iter->second` references a `Contact` object. The expression `iter->second.getName()` returns that object’s `name` field, and the expression `iter->second.getEmail()` returns that object’s `email` field.

Program 17-15 shows another example in which we use a range-based `for` loop to iterate over a `map` containing `Contact` objects as values.

Program 17-15

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 #include "Contact.h"
5 using namespace std;
6

```

(program continues)

Program 17-15 *(continued)*

```

7 int main()
8 {
9     // Create some Contact objects
10    Contact contact1("Ashley Miller", "amiller@faber.edu");
11    Contact contact2("Jacob Brown", "jbrown@gotham.edu");
12    Contact contact3("Emily Ramirez", "eramirez@coolidge.edu");
13
14    // Create a map to hold the Contact objects.
15    map<string, Contact> contacts;
16
17    // Add the contact objects to the map.
18    contacts[contact1.getName()] = contact1;
19    contacts[contact2.getName()] = contact2;
20    contacts[contact3.getName()] = contact3;
21
22    // Display all objects in the map.
23    for (auto element : contacts)
24    {
25        cout << element.second.getName() << "\t"
26            << element.second.getEmail() << endl;
27    }
28
29    return 0;
30 }
```

Program Output

Ashley Miller	amiller@faber.edu
Emily Ramirez	eramirez@coolidge.edu
Jacob Brown	jbrown@gotham.edu

In lines 25 and 26, the range variable `element` is a `pair` object. The `pair` object's first member variable is the element's key, and the `pair` object's second member variable is the element's value. So, the expression `element.second` references a `Contact` object. The expression `element.second.getName()` returns that object's `name` field, and the expression `element.second.getEmail()` returns that object's `email` field.

Program 17-15 illustrates the process of using your own objects as values in a `map`, but the program can be simplified. When we define the `map`, we can use an initialization list that also creates the `Contact` objects. That will allow us to eliminate the `Contact` definition statements in lines 10 through 12. Program 17-16 shows the program with these changes.

Program 17-16

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 #include "Contact.h"
5 using namespace std;
6
```

```

7 int main()
8 {
9     // Create a map holding the Contact objects.
10    map<string, Contact> contacts =
11        {{"Ashley Miller", Contact("Ashley Miller", "amiller@faber.edu") },
12         {"Jacob Brown", Contact("Jacob Brown", "jbrown@gotham.edu") },
13         {"Emily Ramirez", Contact("Emily Ramirez", "eramirez@coolidge.edu") }
14     };
15
16     // Display all objects in the map.
17     for (auto element : contacts)
18     {
19         cout << element.second.getName() << "\t"
20             << element.second.getEmail() << endl;
21     }
22
23     return 0;
24 }
```

Program Output

```

Ashley Miller amiller@faber.edu
Emily Ramirez eramirez@coolidge.edu
Jacob Brown jbrown@gotham.edu
```

Using an Object of Your Own Class as a Key

You can use objects of a class that you have written as keys in a `map`, as long as the class has overloaded the `<` operator. For example, look at the `Customer` class shown here:

Contents of Customer.h

```

1 #ifndef CUSTOMER_H
2 #define CUSTOMER_H
3 #include<string>
4 using namespace std;
5
6 class Customer
7 {
8 private:
9     int custNumber;
10    string name;
11 public:
12    Customer(int cn, string n)
13    { custNumber = cn;
14        name = n; }
15
16    void setCustNumber(int cn)
17    { custNumber = cn; }
18
19    void setName(string n)
20    { name = n; }
21
22    int getCustNumber() const
23    { return custNumber; }
```

```

25     string getName() const
26     { return name; }
27
28     bool operator < (const Customer &right) const
29     { bool status = false;
30
31         if (custNumber < right.custNumber)
32             status = true;
33
34     return status; }
35 };
36 #endif

```

The `Customer` class holds two pieces of data about a customer: the customer number (`custNumber`) and the customer's name (`name`). The overloaded `<` operator (in lines 28 through 35) returns `true` if the `custNumber` member is less than the parameter's `custNumber` member. Otherwise, it returns `false`.

Suppose we are writing a program that assigns seats in a theater to customers, and we want to use `Customer` objects as the keys in a `map`. Program 17-17 shows an example. In line 15, the program defines a `map` that uses `Customer` objects as keys, and strings as values.

Program 17-17

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 #include "Customer.h"
5 using namespace std;
6
7 int main()
8 {
9     // Create some Customer objects.
10    Customer customer1(1001, "Sarah Scott");
11    Customer customer2(1002, "Austin Hill");
12    Customer customer3(1003, "Megan Cruz");
13
14    // Create a map to hold the seat assignments.
15    map<Customer, string> assignments;
16
17    // Use the map to store the seat assignments.
18    assignments[customer1] = "1A";
19    assignments[customer2] = "2B";
20    assignments[customer3] = "3C";
21
22    // Display all objects in the map.
23    for (auto element : assignments)
24    {
25        cout << element.first.getName() << "\t"
26            << element.second << endl;
27    }
28
29    return 0;
30 }

```

Program Output

Sarah Scott	1A
Austin Hill	2B
Megan Cruz	3C

Program 17-17 illustrates the process of using your own objects as keys in a `map`, but the program can be simplified. When we define the `map`, we can use an initialization list that also creates the `Customer` objects. That will allow us to eliminate the `Customer` definition statements in lines 10 through 12. Program 17-18 shows the program with these changes.

Program 17-18

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 #include "Customer.h"
5 using namespace std;
6
7 int main()
8 {
9     // Create a map to hold the seat assignments.
10    map<Customer, string> assignments =
11        { { Customer(1001, "Sarah Scott"), "1A" },
12          { Customer(1002, "Austin Hill"), "2B" },
13          { Customer(1003, "Megan Cruz"), "3C" } };
14
15    // Display all objects in the map.
16    for (auto element : assignments)
17    {
18        cout << element.first.getName() << "\t"
19                      << element.second << endl;
20    }
21
22    return 0;
23 }
```

Program Output

Sarah Scott	1A
Austin Hill	2B
Megan Cruz	3C

The unordered_map Class

11

Beginning in C++11, the STL provides a class template named `unordered_map` that is similar to the `map` class, except in two regards: (1) the keys in an `unordered_map` are not sorted in any particular way, and (2) the `unordered_map` class has better performance. If you will be making a lot of searches on a large number of elements, and you are not concerned with retrieving them in key order, you should probably use the `unordered_map` class instead of the `map` class.

To use the `unordered_map` class, you must write the `#include <unordered_map>` directive in your program. The `unordered_map` class has a similar set of member functions as the `map` class (many of which are listed in Table 17-10), and in most cases, working with an `unordered_map` is just like working with a `map`. The following code shows an `unordered_map` being initialized with elements, then an element being retrieved and displayed:

```
// Create an unordered_map containing employee IDs and names.
unordered_map<int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Retrieve a value from the map.
cout << employees.at(103) << endl;
```

The following code shows an example of using a range-based `for` loop to display all of the elements in a `map`:

```
// Create an unordered_map containing employee IDs and names.
unordered_map <int, string> employees =
    {{101, "Chris Jones"}, {102, "Jessica Smith"},
     {103, "Amanda Stevens"}, {104, "Will Osborn"}};

// Display each element.
for (auto element : employees)
{
    cout << "ID:" << element.first << "\tName:" << element.second << endl;
}
```



NOTE: The differences between the `unordered_map` class and the `map` class are a result of the way each internally stores its elements. The `map` class uses a data structure known as a *binary tree*, and the `unordered_map` class uses a *hash table*.

The `multimap` Class

The `multimap` class lets you create a map container in which multiple elements can have the same key. In other words, duplicate keys are allowed in a `multimap`. To use the `multimap` class, you will need to write the `#include<map>` directive in your program. Table 17-11 lists many (but not all) of the `multimap` class's member functions.

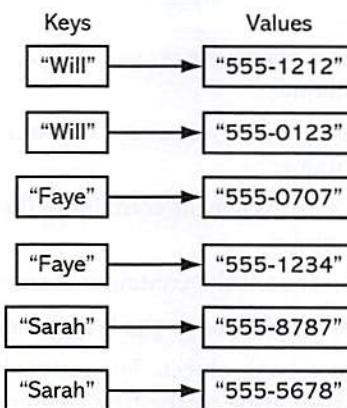
Table 17-11 Some of the `multimap` Member Functions

Member Function	Description
<code>begin()</code>	Returns an iterator to the first element in the container.
<code>cbegin()</code>	Returns a <code>const_iterator</code> to the first element in the container.
<code>cend()</code>	Returns a <code>const_iterator</code> pointing to the end of the container.
<code>clear()</code>	Erases all of the elements in the container.
<code>count(key)</code>	Returns the number of elements containing the specified key.

Table 17-11 (continued)

Member Function	Description
<code>crbegin()</code>	Returns a <code>const_reverse_iterator</code> pointing to the last element in the container.
<code>crend()</code>	Returns a <code>const_reverse_iterator</code> pointing to the first element in the container.
<code>emplace(key, value)</code>	Inserts a new element containing the specified <code>key</code> and <code>value</code> into the container.
<code>empty()</code>	Returns <code>true</code> if the container is empty, or <code>false</code> otherwise.
<code>end()</code>	Returns an iterator to the last element in the container.
<code>equal_range(key)</code>	Returns a <code>pair</code> object. The <code>pair</code> object's <code>first</code> member is an iterator pointing to the first element in the <code>multimap</code> that matches the specified <code>key</code> . The <code>pair</code> object's <code>second</code> member is an iterator pointing to the position <i>after</i> the last element that matches the specified <code>key</code> . If the specified <code>key</code> is not found, both iterators will point to the element that would naturally appear after the element that was searched for.
<code>erase(key)</code>	Erases all of the elements containing the specified <code>key</code> . The function returns the number of elements that were erased.
<code>find(key)</code>	Searches for an element with the specified <code>key</code> , and returns an iterator to the first matching element. If the element is not found, the function returns an iterator to the end of the container.
<code>insert(pair)</code>	Inserts a <code>pair</code> object as an element to the map.
<code>lower_bound(key)</code>	Returns an iterator pointing to the first element with a key that is equal to or greater than <code>key</code> .
<code>max_size()</code>	Returns the theoretical maximum size of the container.
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> pointing to the last element in the container.
<code>rend()</code>	Returns a <code>reverse_iterator</code> pointing to the first element in the container.
<code>size()</code>	Returns the number of elements in the container.
<code>swap(second)</code>	The <code>second</code> argument must be a <code>map</code> object of the same type as the calling object. The function swaps the contents of the calling object and the <code>second</code> object.
<code>upper_bound(key)</code>	Returns an iterator pointing to the first element with a key that is greater than <code>key</code> .

For example, in a phone book application, you could store names and phone numbers in a `multimap` container, using the names as keys and the phone numbers as values. If a person has multiple phone numbers, you could store multiple elements using that person's name as the key. Figure 17-5 illustrates this concept.

Figure 17-5 Duplicate keys stored in a multimap container

In Figure 17-5, there are two elements with the key “Will”, two elements with the key “Faye”, and two elements with the key “Sarah”. Program 17-19 shows how to define a multimap and initialize it with these elements.

Program 17-19

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 using namespace std;
5
6 int main()
7 {
8     // Define a phonebook multimap.
9     multimap<string, string> phonebook =
10        { {"Will", "555-1212"}, {"Will", "555-0123"}, 
11          {"Faye", "555-0707"}, {"Faye", "555-1234"}, 
12          {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };
13
14    // Display the elements in the multimap.
15    for (auto element : phonebook)
16    {
17        cout << element.first << "\t"
18           << element.second << endl;
19    }
20    return 0;
21 }
```

Program Output

```

Faye 555-0707
Faye 555-1234
Sarah 555-8787
Sarah 555-5678
Will 555-1212
Will 555-0123
```

As with the `map` class, the elements of a `multimap` are ordered. When you iterate over a `multimap`, you retrieve its elements in order of their keys. The `multimap` class also provides many of the same member functions as the `map` class. There are some differences between the `multimap` class and the `map` class, however, and we will focus on many of those differences in the rest of this section.

Adding Elements to a multimap

Unlike the `map` class, the `multimap` class does not overload the `[]` operator. So, you cannot write an assignment statement to add a new element to a `multimap`. Instead, you can use either the `emplace()` member function, or the `insert()` member function.

The `multimap` class's `emplace()` member function adds a new element to the container. You simply call the member function, passing the new element's key and value as arguments. The following code shows an example:

```
multimap<string, string> phonebook;
phonebook.emplace("Will", "555-1212");
phonebook.emplace("Will", "555-0123");
phonebook.emplace("Faye", "555-0707");
phonebook.emplace("Faye", "555-1234");
phonebook.emplace("Sarah", "555-8787");
phonebook.emplace("Sarah", "555-5678");
```

The first statement creates an empty `multimap` container named `phonebook`. The statements that follow use the `emplace()` member function to add the elements shown in Figure 17-5.

If the element that you are inserting with the `emplace()` member function has the same key as an existing element, the function adds a new element with the same key.

The `insert()` member function adds a `pair` object as an element to the `multimap`. You can use the STL function `make_pair` to construct a `pair` object, as shown in the following code:

```
multimap<string, string> phonebook;
phonebook.insert(make_pair("Will", "555-1212"));
phonebook.insert(make_pair("Will", "555-0123"));
phonebook.insert(make_pair("Faye", "555-0707"));
phonebook.insert(make_pair("Faye", "555-1234"));
phonebook.insert(make_pair("Sarah", "555-8787"));
phonebook.insert(make_pair("Sarah", "555-5678"));
```

The first statement creates an empty `multimap` container named `phonebook`. The statements that follow use the `insert()` member function to add the elements shown in Figure 17-5.

If the element that you are inserting with the `insert()` member function has the same key as an existing element, the function adds a new element with the same key.

Getting the Number of Elements with a Specified Key

The `multimap` class's `count()` member function accepts a key as its argument, and returns the number of elements that match the specified key. Program 17-20 demonstrates.

Program 17-20

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 using namespace std;
5
6 int main()
7 {
8     // Define a phonebook multimap.
9     multimap<string, string> phonebook =
10     { {"Will", "555-1212"}, {"Will", "555-0123"},
11     {"Faye", "555-0707"}, {"Faye", "555-1234"},
12     {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };
13
14     // Display the number of elements that match "Faye".
15     cout << "Faye has " << phonebook.count("Faye") << " elements.\n";
16
17 }

```

Program Output

Faye has 2 elements.

Retrieving the Elements with a Specified Key

Like the `map` class, the `multimap` class has a `find()` member function that searches for an element with a specified key. If the key is found, the `find()` function returns an iterator to the first element matching it. If the element is not found, the `find()` function returns an iterator to the end of the map.

If you want to retrieve all of the elements matching a specified key, use the `multimap` class's `equal_range` member function. The `equal_range` member function returns a `pair` object. The `pair` object's `first` member is an iterator pointing to the first element in the `multimap` that matches the specified key. The `pair` object's `second` member is an iterator pointing to the position *after* the last element that matches the specified key. Program 17-21 demonstrates how to use the function.

Program 17-21

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 using namespace std;
5
6 int main()
7 {
8     // Define a phonebook multimap.
9     multimap<string, string> phonebook =
10     { {"Will", "555-1212"}, {"Will", "555-0123"},
11     {"Faye", "555-0707"}, {"Faye", "555-1234"},
12     {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };
13

```

```
14 // Define a pair variable to receive the object that
15 // is returned from the equal_range member function.
16 pair<multimap<string, string>::iterator,
17     multimap<string, string>::iterator> range;
18
19 // Define an iterator for the multimap.
20 multimap<string, string>::iterator iter;
21
22 // Get the range of elements that match "Faye".
23 range = phonebook.equal_range("Faye");
24
25 // Display all of the elements that match "Faye".
26 for (iter = range.first; iter != range.second; iter++)
27 {
28     cout << iter->first << "\t" << iter->second << endl;
29 }
30
31 return 0;
32 }
```

Program Output

```
Faye    555-0707
Faye    555-1234
```

Let's take a closer look at the program:

- Lines 9 through 12 define a `multimap` container named `phonebook`, initialized with six elements.
- Lines 16 and 17 define a `pair` variable named `range`. The `range` variable will receive the object that is returned from the `equal_range` member function. The object's `first` and `second` members will both be iterators for the `multimap` container.
- Line 20 defines an iterator named `iter` for the `multimap` container.
- Line 23 calls the `phonebook` container's `equal_range` member function, specifying "Faye" as the key to search for. The `equal_range` function returns a `pair` object that is assigned to the `range` variable.
- The `for` loop in lines 26 through 29 iterates over all of the elements matching the key "Faye", displaying their values. Here are the details of the loop:
 - The `iter` iterator is initially set to `range.first`.
 - The loop repeats as long as `iter` is not equal to `range.second`.
 - At the end of each loop iteration, `iter` is incremented to the next element.
 - Each time the loop iterates, it displays the contents of the element to which `iter` is pointing (line 26).



NOTE: When searching for a range of elements with the `equal_range` member function, if the specified key is not found, both iterators will point to the element that would naturally appear after the element that was searched for.

Deleting Elements from a multimap

You can delete, or erase, elements from a `multimap` using the `erase()` member function. You call the `erase()` member function, passing the key that is associated with the elements that you want to erase. All of the elements matching the specified key will be erased from the `multimap`. The `erase()` member function returns the number of elements that were erased. The following code shows an example:

```
// Define a phonebook multimap.
multimap<string, string> phonebook =
    { {"Will", "555-1212"}, {"Will", "555-0123"}, {"Faye", "555-0707"}, {"Faye", "555-1234"}, {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };

// Delete Will's phone numbers from the multimap.
phonebook.erase("Will");
```

This code creates a `multimap` container with six elements. Then, it deletes all of the elements with the key “Will”. (Two elements will be deleted.) The `multimap` then contains only four elements.

The `unordered_multimap` Class

11

Beginning in C++11, the STL provides a class template named `unordered_multimap` that is similar to the `multimap` class, except in two regards: (1) the keys in an `unordered_multimap` are not sorted in any particular order, and (2) the `unordered_multimap` class has better performance. If you will be making a lot of searches on a large number of elements, and you are not concerned with retrieving them in key order, you should probably use the `unordered_multimap` class instead of the `multimap` class.

To use the `unordered_multimap` class, you must write the `#include <unordered_multimap>` directive in your program. The `unordered_multimap` class has a similar set of member functions as the `multimap` class, and in most cases, working with an `unordered_multimap` is just like working with a `multimap`.



NOTE: The differences between the `unordered_multimap` class and the `multimap` class are a result of the way each internally stores its elements. The `multimap` class uses a data structure known as a *binary tree*, and the `unordered_multimap` class uses a *hash table*.



Checkpoint

- 17.20 Each element that is stored in a map has two parts. What are they?
- 17.21 Write a statement that defines a `map` named `myMap`. The keys in `myMap` should be `ints`, and the values should be `strings`.
- 17.22 Suppose an empty `map` named `employee` has been created. What does the following statement do?
`employee[543] = "Joanne Manchester";`
- 17.23 Describe two ways in which you can retrieve an element with a particular key in a `map`.

- 17.24 If you want to store objects of a class you have written, as values in a `map`, what do you have to make sure that the class has?
- 17.25 If you want to store objects of a class you have written, as keys in a `map`, what do you have to make sure that the class has?
- 17.26 What is the difference between a `map` and an `unordered_map`?
- 17.27 What is the difference between a `map` and an `multimap`?

17.5

The set, multiset, and unordered_set Classes

CONCEPT: A set contains a collection of unique values.



A `set` is an associative container that stores a collection of unique values in a way that is similar to a mathematical set. You use the STL class template `set` to implement a set container in C++. Here are two important things to know about the STL `set` container:

- All the elements in a `set` must be unique. No two elements can have the same value.
- The elements in a `set` are automatically sorted in ascending order.

To use the `set` class, you will need to `#include` the `<set>` header file in your program. Then, you can define a `set` object using one of the `set` class constructors. Table 17-12 shows the general format of a `set` definition statement using each constructor. Table 17-13 lists many (but not all) of the `set` class's member functions.

Table 17-12 set Definition Statements

Default Constructor	<code>set<dataType> name;</code>
	Creates an empty <code>set</code> object. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the <code>set</code> .
Range Constructor	<code>set<dataType> name(iterator1, iterator2);</code>
	Creates a <code>set</code> object that initially contains a range of values specified by two iterators. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the <code>set</code> . The <code>iterator1</code> and <code>iterator2</code> arguments mark the beginning and end of a range of values that will be stored in the <code>set</code> . (<code>iterator2</code> marks the end of the range, but the element pointed to by <code>iterator2</code> will not be included in the range.) Any duplicate values in the range will be added only once to the <code>set</code> .
Copy Constructor	<code>set<dataType> name(set2);</code>
	Creates a <code>set</code> object that is a copy of another <code>set</code> object. In the general format, <code>dataType</code> is the data type of each element, <code>name</code> is the name of the <code>set</code> , and <code>set2</code> is the <code>set</code> to copy.

Table 17-13 Some of the set Member Functions

Member Function	Description
<code>begin()</code>	Returns an iterator to the first element in the container.
<code>cbegin()</code>	Returns a <code>const_iterator</code> to the first element in the container.
<code>cend()</code>	Returns a <code>const_iterator</code> pointing to the end of the container.
<code>clear()</code>	Erases all of the elements in the container.
<code>count(<i>value</i>)</code>	Returns the number of elements containing the specified value. (For the <code>set</code> class, the <code>count()</code> function returns either 0 or 1. For the <code>multiset</code> class, the function can return values greater than 1.)
<code>crbegin()</code>	Returns a <code>const_reverse_iterator</code> pointing to the last element in the container.
<code>crend()</code>	Returns a <code>const_reverse_iterator</code> pointing to the first element in the container.
<code>emplace(<i>args...</i>)</code>	Constructs a new element into the container, passing the list of arguments to the element's constructor. If an element with the specified value already exists, the function call does nothing.
<code>empty()</code>	Returns <code>true</code> if the container is empty, or <code>false</code> otherwise.
<code>end()</code>	Returns an iterator to the end of the container (the position after the last element).
<code>equal_range(<i>value</i>)</code>	Returns a <code>pair</code> object. The <code>pair</code> object's <code>first</code> member is an iterator pointing to the first element in the <code>set</code> that matches the specified <code>value</code> . The <code>pair</code> object's <code>second</code> member is an iterator pointing to the position <i>after</i> the last element that matches the specified <code>value</code> . If the specified <code>value</code> is not found, both iterators will point to the element that would naturally appear after the element that was searched for. (With the <code>set</code> class, the range will have at most one element. With the <code>multiset</code> class, the range can have multiple elements.)
<code>erase(<i>value</i>)</code>	Erases the element containing the specified value. The function returns 1 if the element was erased, or 0 if no matching element was found.
<code>find(<i>value</i>)</code>	Searches for an element with the specified value. If the element is found, the function returns an iterator to it. If the element is not found, the function returns an iterator to the end of the <code>set</code> .
<code>insert(<i>value</i>)</code>	Inserts a value as an element to the <code>set</code> . If an element with the specified value already exists, the function does not insert a new element. The function returns a <code>pair</code> object, with its <code>first</code> member being an iterator pointing to the newly inserted element (or the equivalent element, if it was already present), and with the <code>second</code> member being the <code>bool</code> value <code>true</code> if a new element was inserted, or <code>false</code> if the element was already present.
<code>lower_bound(<i>value</i>)</code>	Returns an iterator pointing to the first element with a key that is equal to or greater than <code>value</code> .
<code>max_size()</code>	Returns the theoretical maximum size of the container.
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> pointing to the last element in the container.
<code>rend()</code>	Returns a <code>reverse_iterator</code> pointing to the first element in the container.
<code>size()</code>	Returns the number of elements in the container.
<code>swap(<i>second</i>)</code>	The <code>second</code> argument must be a <code>map</code> object of the same type as the calling object. The function swaps the contents of the calling object and the <code>second</code> object.
<code>upper_bound(<i>value</i>)</code>	Returns an iterator pointing to the first element with a key that is greater than <code>value</code> .

Here is an example of how you might define a `set` container to hold integers:

```
set<int> numbers;
```

This statement defines a `set` container named `numbers`. Inside the angled brackets `<>`, the data type `int` is the type of each element. You can provide an initialization list to initialize the `set`, as shown here:

```
set<int> numbers = {1, 2, 3, 4, 5};
```

Here is an example of defining a set to hold strings:

```
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};
```

A `set` cannot contain duplicate items, so, if the same value appears more than once in an initialization list, it will be added to the `set` only one time. For example, the following `set` will contain the values 1, 2, 3, 4, and 5:

```
set<int> numbers = {1, 1, 2, 3, 3, 3, 4, 5, 5};
```

Adding Elements to an Existing set

The `set` class provides an `insert()` member function that adds a new element to the container. You simply call the member function, passing the new element's value as an argument. The following code shows an example:

```
set<int> numbers;
numbers.insert(10);
numbers.insert(20);
numbers.insert(30);
```

The first statement creates an empty `set` container named `numbers`. The statements that follow add the values 10, 20, and 30 to the `set`. If the value that you are inserting already exists in the `set`, the `insert()` member function will do nothing (no new element will be inserted).



NOTE: The `set` class also provides the `emplace()` member function, for inserting elements. You will see an example of it momentarily. For a review of the difference between the `emplace()` and `insert()` member functions, see the discussion on emplacement that appears in this chapter's section on vectors.

Iterating Over a set with the Range-Based for Loop

The range-based `for` loop is a convenient way to iterate over all of the elements in a `set`. The following code shows an example of how to display all of the elements in a `set`:

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Display each element.
for (string element : names)
{
    cout << element << endl;
}
```

Using an Iterator with a set

You can use a bidirectional iterator to access the elements of a `set`. The `set` class provides the `begin()` and `cbegin()` member functions that return an iterator pointing to the first element in the `set`, and the `end()` and `cend()` member functions that return an iterator pointing to the end of the `set` (the position *after* the last element). The following code shows an example of using an iterator to display all of the elements of a `set`:

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
set<string>::iterator iter;

// Use the iterator to display each element in the set.
for (iter = names.begin(); iter != names.end(); iter++)
{
    cout << *iter << endl;
}
```



NOTE: The `end()` and `cend()` member functions return an iterator pointing to the end of the `set`, but it does not point to an actual element. It points to the position where an additional element would exist, if it appeared after the last element.

Determining Whether a Value Exists in a set

You can use either the `count()` member function or the `find()` member function to determine whether a value exists in a `set`. The `count()` member function accepts the desired value as an argument, and returns 1 if the value is found in the `set`, or 0 if the value is not found. Here is an example:

```
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};
if (names.count("Lisa"))
    cout << "Lisa was found in the set.\n";
else
    cout << "Lisa was not found.\n";
```

The `find()` member function accepts the desired value as an argument, and returns an iterator that points to the element if it is found in the `set`. If the element is not found, the `find()` function returns an iterator to the end of the `set`. The following code shows an example:

```
// Create a set containing names.
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};

// Create an iterator.
set<string>::iterator iter;

// Find "Karen".
iter = names.find("Karen");

// Display the result.
if (iter != names.end())
```

```

    {
        cout << *iter << " was found.\n";
    }
    else
    {
        cout << "Karen was not found.\n";
    }
}

```

Storing Objects of Your Own Classes in a set

You can store objects of a class that you have written in a `set`, as long as the class has overloaded the `<` operator. The `set` class uses the `<` operator to determine the order in which to store elements, and to detect duplicates. (Suppose we are comparing two values: `value1` and `value2`. If `value1` is NOT less than `value2`, and `value2` is NOT less than `value1`, then the two values are equal. This is the logic that the `set` class uses to determine whether two values are duplicates.)

For example, recall the `Customer` class shown earlier in this chapter, in our discussion of the `map` class. (See the contents of the `Customer.h` file.) The `Customer` class holds two pieces of data about a customer: the customer number and the customer's name. The class also has an overloaded `<` operator that compares customer numbers. If we store objects of the `Customer` class in a `set` container, the objects will be sorted by customer number, and the container will not allow objects with the same customer number to be stored. Program 17-22 demonstrates this.

Program 17-22

```

1 #include <iostream>
2 #include <set>
3 #include "Customer.h"
4 using namespace std;
5
6 int main()
7 {
8     // Create a set of Customer objects.
9     set<Customer> customerset =
10    { Customer(1003, "Megan Cruz"),
11      Customer(1002, "Austin Hill"),
12      Customer(1001, "Sarah Scott")
13    };
14
15    // Try to insert a duplicate customer number.
16    customerset.emplace(1001, "Evan Smith");
17
18    // Display the set elements
19    cout << "List of customers:\n";
20    for (auto element : customerset)
21    {
22        cout << element.getCustNumber() << " "
23            << element.getName() << endl;
24    }
25

```

(program continues)

Program 17-22 *(continued)*

```

26 // Search for customer number 1002.
27 cout << "\nSearching for Customer Number 1002:\n";
28 auto it = customerset.find(Customer(1002, ""));
29
30 if (it != customerset.end())
31     cout << "Found: " << it->getName() << endl;
32 else
33     cout << "Not found.\n";
34
35 return 0;
36 }
```

Program Output

List of customers:

1001 Sarah Scott
 1002 Austin Hill
 1003 Megan Cruz

Searching for Customer Number 1002:

Found: Austin Hill

Let's take a closer look at the program:

- Lines 9 through 13 define a `set` container named `customerset`. The container is initialized with three `Customer` objects.
- Line 16 calls the `emplace()` member function to add a `Customer` object to the `set`. However, the customer number that is being passed to the `emplace()` function is already in the `set`. As a result, the object will not be added to the `set`.
- The loop in lines 20 through 24 displays all of the `set` container's elements. Notice in the program output, the elements are retrieved in order of their customer numbers.
- Next, the program searches the `set` container for an object with the customer number 1002. The statement in line 28 defines an iterator, assigning `it` the value returned from the `find()` member function.

Look carefully at the argument we are passing to the `find()` member function. The expression `Customer(1002, "")` constructs a temporary, nameless `Customer` object that has 1002 as the customer number, and "" as the customer name. (Because we are simply using this object to search for an element with customer number 1002, there is no need to provide a meaningful value for the customer name.) The `find()` member function will accept this `Customer` object as an argument, and it will search the container for an element that has a matching customer number. (Remember, the `set` container uses the `Customer` class's overloaded `<` operator to determine whether two objects are equal.) If a matching element is found, the function returns an iterator pointing to it. Otherwise, it returns an iterator pointing to the end of the container.

- The if/else statement in lines 30 through 33 tests the iterator to determine whether it is pointing to a matching element, or the end of the container, and displays a message indicating the results of the search.

The multiset Class

The `multiset` class lets you create a set container that can store duplicate elements. To use the `multiset` class, you will need to #include the `<set>` header file in your program. The class provides the same member functions as the `set` class, with these two differences:

- In the `set` class, the `count()` member function returns either 0 or 1. In the `multiset` class, the `count()` member function can return values greater than 1.
- In the `set` class, the `equal_range()` member function returns a range with, at most, one element. In the `multiset` class, the `equal_range()` member function can return a range with multiple elements.

The unordered_set and unordered_multiset Classes

11

Beginning in C++11, the STL provides class templates named `unordered_set` and `unordered_multiset`. These are similar to the `set` and `multiset` classes, except in two regards: (1) the values stored in an `unordered_set` or an `unordered_multiset` are not sorted in any particular way, and (2) the `unordered_set` and `unordered_multiset` classes has better performance than the `set` and `multiset` classes. If you will be making a lot of searches on a large number of elements, and you are not concerned with retrieving them in any order, you should probably use the `unordered_set` or `unordered_multiset` classes instead of `set` or `multiset`.

To use the `unordered_set` or `unordered_multiset` classes, you must write the `#include <unordered_set>` directive in your program. In most cases, working with an `unordered_set` or `unordered_multiset` is similar to working with a `set` or `multiset`.



Checkpoint

- 17.28 What are two differences between a `set` and a `vector`?
- 17.29 Write a statement that defines an empty `set` object named `aset`, that can hold strings.
- 17.30 Write a statement that defines a `set` object named `aset`, that can hold `ints`. The `set` should be initialized with these values: 10, 20, 30, 40
- 17.31 What happens when you use the `insert()` member function to insert a value into a `set`, and that value is already in the `set`?
- 17.32 What value does the `set` class's `count` member function return?
- 17.33 If you store objects of a class that you have written in a `set`, what must the class overload?
- 17.34 What is the difference between a `set` and a `multiset`?
- 17.35 In what two ways are the `unordered_set` and `unordered_multiset` different from the `set` and `multiset` classes?

17.6 Algorithms

CONCEPT: Many commonly used algorithms are written as function templates in the STL.

The STL provides a number of algorithms, implemented as function templates, in the `<algorithm>` header file. These functions perform various operations on ranges of elements. A *range of elements* is a sequence of elements denoted by two iterators. The first iterator points to the first element in the range, and the second iterator points to the end of the range (the element to which the second iterator points is not included in the range). The algorithms can be organized in the following categories:

- **Min/max algorithms:**
Determine the smallest (min) and largest (max) values in a range of elements
- **Sorting algorithms:**
Sort a range of elements, or determine whether a range is sorted
- **Search algorithms:**
Perform various searching operations on a sorted range of elements
- **Read-only sequence algorithms:**
Iterate over a range of elements, performing various operations that do not modify the elements
- **Copying and moving algorithms:**
Use various techniques to copy or move ranges of elements.
- **Swapping algorithms:**
Swap values, ranges of elements, or the values pointed to by iterators
- **Replacement algorithms:**
Replace elements of a specified value
- **Removal algorithms:**
Remove elements
- **Reversal algorithms:**
Reverse the order of elements in a range
- **Fill algorithms:**
Fill the elements in a range with values
- **Rotation algorithms:**
Rotate the elements in a range with values
- **Shuffling algorithms:**
Shuffle the elements in a range
- **Set algorithms:**
Perform common mathematical set operations on a range of elements
- **Transformation algorithm:**
Iterate over ranges of elements, performing an operation using each element
- **Partition algorithms:**
Partition a range of elements into two groups
- **Merge algorithms:**
Merge ranges of elements

- **Permutation algorithms:**
Rearrange a range of elements into all of its different possible permutations
- **Heap algorithms:**
Create and work with a heap data structure
- **Lexicographical comparison algorithm:**
Makes a lexicographical comparison between two ranges of elements

At the time this was written, there are 85 function templates in the `<algorithm>` header file. For a summary of all of the STL algorithms, see Appendix H on the Computer Science Portal at www.pearsonhighered.com/gaddis. We will not discuss all of the STL algorithms in this chapter, but we will look at examples of some of the more interesting ones.

Sorting and Searching Algorithms

The `<algorithm>` header file defines several function templates for sorting and searching. We will look at two of them: `sort()` and `binary_search()`. The `sort` function takes the following general format:

```
sort(iterator1, iterator2)
```

In the general format, `iterator1` and `iterator2` mark the beginning and end of a range of elements. The function sorts the range of elements in ascending order. The `binary_search()` function takes the following general format:

```
binary_search(iterator1, iterator2, value)
```

In the general format, the `iterator1` and `iterator2` arguments mark the beginning and end of a range of elements that are sorted in ascending order, and `value` is the value to search for. The function returns `true` if the `value` is found in the range of elements, or `false` otherwise. Program 17-23 demonstrates both the `sort()` and `binary_search()` functions.

Program 17-23

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main()
7 {
8     int searchValue; // Value to search for
9
10    // Create a vector of unsorted integers.
11    vector<int> numbers = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
12
13    // Sort the vector.
14    sort(numbers.begin(), numbers.end());
15

```

(program continues)

Program 17-24

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include "Customer.h"
5 using namespace std;
```

The sort() and binary-search() functions use the < operator to compare elements. If the particular customer number. Program 17-24 demonstrates this.

binary-search() functions to sort and search the container for an object containing store objects of the Customer class in a sequence container, we can use the STL sort() and name. The class also has an overloaded < operator that compares customer numbers. If we class holds two pieces of data about a customer: the customer number and the customer's name. The Customer class is part of the Customer file. The Customer class holds two pieces of data about a customer. (See the contents of the Customer file.) In our discussion of the map class, recall the Customer class shown earlier in this chapter. load the < operator. For example, recall the Customer class shown earlier in this chapter. elements that you are sorting and searching contain your own class objects, be sure to overload the < operator. For example, recall the Customer class shown earlier in this chapter. The sort() and binary-search() functions use the < operator to compare elements. If the

That value is not in the vector.
Enter a value to search for: 99
Here are the sorted values:
1 2 3 4 5 6 7 8 9 10

Program Output

That value is in the vector.
Enter a value to search for: 8

Here are the sorted values:
1 2 3 4 5 6 7 8 9 10

Program Output

```
16 // Display the vector.
17 cout << "Here are the sorted values:\n";
18 for (auto element : numbers)
19     cout << element << " ";
20 cout << endl;
21 // Get the value to search for.
22 cout << "Enter a value to search for: ";
23 cin >> searchValue;
24 cout << "Search for the value " << searchValue << endl;
25 if (binary-search(numbers.begin(), numbers.end(), searchValue))
26     cout << "That value is in the vector.\n";
27 else
28     cout << "That value is not in the vector.\n";
29 cout << endl;
30 cout << "Enter a value to search for: ";
31 cin >> searchValue;
32 return 0;
```

Program 17-23 (continued)

```

7 int main()
8 {
9     int searchValue; // Value to search for
10
11    // Create a vector of unsorted Customer objects.
12    vector<Customer> customers =
13        { Customer(1003, "Megan Cruz"),
14          Customer(1001, "Sarah Scott"),
15          Customer(1002, "Austin Hill")
16      };
17
18    // Sort the vector.
19    sort(customers.begin(), customers.end());
20
21    // Display the vector.
22    cout << "Here are the sorted customers:\n";
23    for (auto element : customers)
24    {
25        cout << element.getCustNumber() << " "
26            << element.getName() << endl;
27    }
28    cout << endl;
29
30    // Get the customer number to search for.
31    cout << "Enter a customer number to search for: ";
32    cin >> searchValue;
33
34    // Search for the customer number.
35    if (binary_search(customers.begin(), customers.end(),
36                      Customer(searchValue, "")))
37        cout << "That customer is in the vector.\n";
38    else
39        cout << "That customer is not in the vector.\n";
40
41    return 0;
42 }
```

Program Output

Here are the sorted customers:

1001 Sarah Scott
 1002 Austin Hill
 1003 Megan Cruz

Enter a customer number to search for: 1001
 That customer is in the vector.

Program Output

Here are the sorted customers:

1001 Sarah Scott
 1002 Austin Hill
 1003 Megan Cruz

Enter a customer number to search for: 1009
 That customer is not in the vector.

Let's take a closer look at the program:

- Lines 12 through 16 define a `vector` named `customers`, initialized with three `Customer` objects.
- Line 19 calls the `sort` function to sort the `vector`. Because the `Customer` class's overloaded `<` operator compares customer numbers, the objects in the `vector` will be sorted by customer number, in ascending order.
- The loop in lines 23 through 27 displays all of the `vector`'s elements.
- Line 31 prompts the user to enter a customer number, and line 32 stores the user's input in the `searchValue` variable.
- Next, the program searches the `vector` for an object with the customer number entered by the user. Look carefully at the third argument we are passing to the `binary_search()` function. The expression `Customer(searchValue, "")` constructs a temporary, nameless `Customer` object that has `searchValue`'s value as the customer number, and `" "` as the customer name. (Because we are simply using this object to search for an element with a specified customer number, there is no need to provide a meaningful value for the customer name.) The `binary_search()` function will accept this `Customer` object as an argument, and it will search the specified range of elements for an object that has a matching customer number. (The `binary_search` function will use the `Customer` class's overloaded `<` operator to determine whether two objects are equal.) If a matching element is found, the function returns `true`. Otherwise, it returns `false`.

Detecting Permutations

If a range has N elements, there are $N!$ possible arrangements of those elements. Each arrangement is called a *permutation*. For example, suppose we have a `vector` with the following integer elements:

1, 2, 3

There are three elements in the `vector`, so there are six possible ways to rearrange the elements. The six possible permutations are:

1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1

The STL's `is_permutation()` function determines whether one range of elements is a permutation of another range of elements. Here is the function's general format:

```
is_permutation(iterator1, iterator2, iterator3)
```

In the general format, `iterator1` and `iterator2` mark the beginning and end of the first range of elements. The `iterator3` argument marks the beginning of the second range of elements, assumed to have the same number of elements as the first range. The function returns `true` if the second range is a permutation of the first range, or `false` otherwise.

Program 17-25 demonstrates the `is_permutation()` function. It uses the function to determine whether a set of numbers matches the winning lottery numbers.

Program 17-25

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main()
7 {
8     const int MAX = 5;           // Numbers in 1 lottery ticket
9     vector<int> winning(MAX);   // The winning numbers
10    vector<int> player(MAX);    // Numbers on a ticket
11
12    // Get the winning numbers.
13    cout << "Enter the " << MAX << " winning numbers:\n";
14    for (auto &element : winning)
15    {
16        cout << "> ";
17        cin >> element;
18    }
19
20    // Get the numbers purchased on a lottery ticket.
21    cout << "\nEnter your " << MAX << " lottery numbers:\n";
22    for (auto &element : player)
23    {
24        cout << "> ";
25        cin >> element;
26    }
27
28    // Check for a winner.
29    if (is_permutation(winning.begin(), winning.end(),
30                        player.begin()))
31        cout << "You won the lottery!\n";
32    else
33        cout << "Sorry, you did not win.\n";
34
35    return 0;
36 }
```

Program Output

Enter the 5 winning numbers:

> 10
 > 25
 > 67
 > 88
 > 93

Enter your 5 lottery numbers:

> 25
 > 15
 > 62

(program output continues)

Program 17-25 (continued)

```
> 93
> 88
Sorry, you did not win.
```

Program Output

Enter the 5 winning numbers:

```
> 10
> 25
> 67
> 88
> 93
```

Enter your 5 lottery numbers:

```
> 67
> 10
> 93
> 88
> 25
```

You won the lottery!

Plugging Your Own Functions into an Algorithm

When a C++ program is running, the executable code for each of the functions in that program is stored in memory. You can use the name of a function to get that function's address in memory, in the same way that you can use the name of an array to get that array's address in memory. This capability allows you to get a *function pointer*, which is a pointer to a function's executable code.

Many of the function templates in the STL are designed to accept function pointers as arguments. This allows you to "plug" one of your own functions into the algorithm. For example, here is the general format of the `for_each` function:

```
for_each(iterator1, iterator2, function)
```

In the general format, the `iterator1` and `iterator2` arguments mark the beginning and end of a range of elements. The `function` argument is the address of a function that accepts an element as its argument (Any value returned from `function` is ignored). The `for_each()` function iterates over the range of elements, passing each element as an argument to `function`.

Program 17-26 demonstrates the `for_each()` function. The program defines a function named `doubleNumber()`. The `doubleNumber()` function accepts a reference to an `int` as its argument, and it doubles the value of the argument. The program uses the `for_each()` function to double the value of every element in a `vector` of integers.

Program 17-26

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
```

```

6 // Function prototype
7 void doubleNumber(int &);
8
9 int main()
10 {
11     vector<int> numbers = { 1, 2, 3, 4, 5 };
12
13     // Display the numbers before doubling.
14     for (auto element : numbers)
15         cout << element << " ";
16     cout << endl;
17
18     // Double the value of each vector element.
19     for_each(numbers.begin(), numbers.end(), doubleNumber);
20
21     // Display the numbers before doubling.
22     for (auto element : numbers)
23         cout << element << " ";
24     cout << endl;
25
26     return 0;
27 }
28
29 //*****
30 // The doubleNumber function doubles the value of n. *
31 //*****
32 void doubleNumber(int &n)
33 {
34     n = n * 2;
35 }
```

Program Output

```
1 2 3 4 5
2 4 6 8 10
```

Another example is the `count_if()` function. The general format of the `count_if()` function is:

$$\text{count_if}(\text{iterator1}, \text{iterator2}, \text{function})$$

In the general format, the `iterator1` and `iterator2` arguments mark the beginning and end of a range of elements. The `function` argument is the address of a function that accepts an element as its argument, and returns either `true` or `false`. (The function should not make changes to the element.) The `count_if` function iterates over the range of elements, passing each element as an argument to `function`. The `count_if` function returns the number of elements for which `function` returns `true`.

Program 17-27 demonstrates the `count_if()` function. The program defines a function named `outOfRange()`. The `outOfRange()` function accepts an `int` as its argument, and it returns `true` if the argument is less than 0 or greater than 100. The program uses the `count_if()` function to determine the number of elements in a `vector` that are not within the range of 1 through 100.

Program 17-27

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 // Function prototype
7 bool outOfRange(int);
8
9 int main()
10 {
11     // Create a vector of ints.
12     vector<int> numbers = { 0, 99, 120, -33, 10, 8, -1, 101 };
13
14     // Get the number of elements that are < 0 or > 100.
15     int invalid = count_if(numbers.begin(), numbers.end(), outOfRange);
16
17     // Display the results.
18     cout << "There are " << invalid << " elements out of range.\n";
19     return 0;
20 }
21
22 //***** The outOfRange function returns true if n is out of range. *****
23 //***** The outOfRange function returns true if n is out of range. *****
24
25 bool outOfRange(int n)
26 {
27     // Constants for min and max values
28     const int MIN = 0, MAX = 100;
29
30     // Flag to hold the status
31     bool status;
32
33     // Determine whether n out of range.
34     if (n < MIN || n > MAX)
35         status = true;
36     else
37         status = false;
38
39     return status;
40 }
```

Program Output

There are 4 elements out of range.

Using the STL to Perform Set Operations

The STL provides a number of function templates for performing the basic set operations you probably learned in math class. Table 17-14 summarizes these methods.

Table 17-14 STL Algorithms to Perform Set Operations

Function Template	Description
<code>set_union(iterator1, iterator2, iterator3, iterator4, iterator5)</code>	Finds the union of two sets. The union of two sets is a set that contains all the elements of both sets, excluding duplicates. The <i>iterator1</i> and <i>iterator2</i> arguments mark the beginning and end of the first set. The <i>iterator3</i> and <i>iterator4</i> arguments mark the beginning and end of the second set. The <i>iterator5</i> argument marks the beginning of the container that will hold the union of the two sets. The function returns an iterator pointing to the end of the range of elements in the union.
<code>set_intersection(iterator1, iterator2, iterator3, iterator4, iterator5)</code>	Finds the intersection of two sets. The intersection of two sets is a set that contains only the elements that are found in both sets. The <i>iterator1</i> and <i>iterator2</i> arguments mark the beginning and end of the first set. The <i>iterator3</i> and <i>iterator4</i> arguments mark the beginning and end of the second set. The <i>iterator5</i> argument marks the beginning of the container that will hold the intersection of the two sets. The function returns an iterator pointing to the end of range of elements in the intersection.
<code>set_difference(iterator1, iterator2, iterator3, iterator4, iterator5)</code>	Finds the difference of two sets. The difference of two sets is the set of elements that appear in one set, but not the other. The <i>iterator1</i> and <i>iterator2</i> arguments mark the beginning and end of the first set. The <i>iterator3</i> and <i>iterator4</i> arguments mark the beginning and end of the second set. The <i>iterator5</i> argument marks the beginning of the container that will hold the difference of the two sets. The function returns an iterator pointing to the end of the range of elements in the difference.
<code>set_symmetric_difference(iterator1, iterator2, iterator3, iterator4, iterator5)</code>	Finds the symmetric difference of two sets. The symmetric difference of two sets is the set of elements that are in one set, but not in both. The <i>iterator1</i> and <i>iterator2</i> arguments mark the beginning and end of the first set. The <i>iterator3</i> and <i>iterator4</i> arguments mark the beginning and end of the second set. The <i>iterator5</i> argument marks the beginning of the container that will hold the symmetric difference of the two sets. The function returns an iterator pointing to the end of the range of elements in the symmetric difference.
<code>includes(iterator1, iterator2, iterator3, iterator4)</code>	Determines whether one set includes another set. The <i>iterator1</i> and <i>iterator2</i> arguments mark the beginning and end of the first set. The <i>iterator3</i> and <i>iterator4</i> arguments mark the beginning and end of the second set. The function returns <code>true</code> if the first set contains all of the elements of the second set. Otherwise, the function returns <code>false</code> .

The ranges of elements with which these member functions work can be stored in `set` containers, `vectors`, arrays, or any other type of container that supports iterators. Keep in mind, however, the ranges of elements must be sorted in ascending order before you can use any of the functions in Table 17-14 with them. For that reason, the `set` container is convenient because it automatically sorts its elements.

Finding the Union of Sets with the `set_union` Function

The union of two sets is a set that contains all the elements of both sets, with no duplicates. For example, suppose `set1` contains the values 1, 2, 3, 4, and `set2` contains the values 3, 4, 5, 6. The union of `set1` and `set2` would be a set containing the values 1, 2, 3, 4, 5, 6.

You can call the STL algorithm function `set_union` to get the union of two sets. The function stores the union in a third container. Program 17-28 demonstrates how to use the function.

Program 17-28

```

1 #include <iostream>
2 #include <set>
3 #include <algorithm>
4 #include <vector>
5 using namespace std;
6
7 int main()
8 {
9     // Create two sets.
10    set<int> set1 = {1, 2, 3, 4};
11    set<int> set2 = {3, 4, 5, 6};
12
13    // Create a vector to hold the union. The
14    // vector must be large enough to hold both sets.
15    vector<int> result(set1.size() + set2.size());
16
17    // Get the union of the sets. The result vector
18    // will hold the union, and iter will point to
19    // the end of the result vector.
20    auto iter = set_union(set1.begin(), set1.end(),
21                          set2.begin(), set2.end(),
22                          result.begin());
23
24    // Resize the result vector to remove unused elements.
25    result.resize(iter - result.begin());
26
27    // Display the result vector's elements
28    cout << "The union of the sets is:\n";
29    for (auto element : result)
30    {
31        cout << element << " ";
32    }

```

```

33     cout << endl;
34
35     return 0;
36 }
```

Program Output

The union of the sets is:

1 2 3 4 5 6

Let's take a closer look at Program 17-28:

- Lines 10 and 11 define `set1` and `set2`, initialized with the values 1, 2, 3, 4, and 3, 4, 5, 6, respectively.
- Line 15 defines a vector named `result`. We will use the vector to hold the union of `set1` and `set2`. Notice the value we are passing to the vector constructor: `set1.size() + set2.size()`. This causes the vector to be large enough to hold both sets. (The vector elements will be initialized with 0.) Figure 17-6 shows the state of `set1`, `set2`, and `result` at this point in the program.
- Line 18 defines an iterator that can be used with the vector. The name of the iterator is `iter`.
- Lines 20 through 22 call the `set_union` function. The function gets the union of `set1` and `set2`, and stores the union in the `result` vector. The function returns an iterator that is assigned to `iter`. The `iter` iterator points to the *end* of the range of elements in the union. Figure 17-7 shows the state of `set1`, `set2`, `result`, and `iter` at this point in the program. (Notice the last two elements of the `result` vector are unused.)
- Line 25 resizes the `result` vector to remove the unused elements. Figure 17-8 shows the state of `set1`, `set2`, `result`, and `iter` at this point in the program.
- Lines 28 through 36 display the program's output, which includes the contents of the `result` vector.

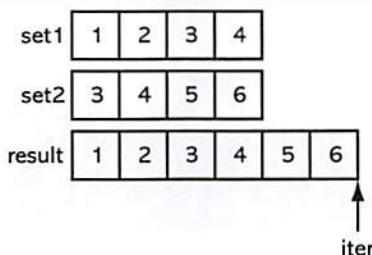
Figure 17-6 State of `set1`, `set2`, and `result` after line 15 has executed

set1	1	2	3	4				
set2	3	4	5	6				
result	0	0	0	0	0	0	0	0

Figure 17-7 State of `set1`, `set2`, `result`, and `iter` after lines 20 through 22 have executed

set1	1	2	3	4				
set2	3	4	5	6				
result	1	2	3	4	5	6	0	0

↑
iter

Figure 17-8 State of set1, set2, result, and iter after line 25 has executed

Finding the Intersection of Sets with the set_intersection Function

The intersection of two sets is a set that contains only the elements that are found in both sets. For example, suppose `set1` contains the values 1, 2, 3, 4, and `set2` contains the values 3, 4, 5, 6. The intersection of `set1` and `set2` would be a set containing the values 3, 4.

You can call the STL algorithm function `set_intersection` to get the intersection of two sets. The function stores the intersection in a third container. Program 17-29 demonstrates how to use the function.

Program 17-29

```

1 #include <iostream>
2 #include <set>
3 #include <algorithm>
4 #include <vector>
5 using namespace std;
6
7 int main()
8 {
9     // Create two sets.
10    set<int> set1 = {1, 2, 3, 4};
11    set<int> set2 = {3, 4, 5, 6};
12
13    // Create a vector to hold the intersection. The
14    // vector must be large enough to hold both sets.
15    vector<int> result(set1.size() + set2.size());
16
17    // Get the intersection of the sets. The result vector
18    // will hold the intersection, and iter will point to
19    // the end of the result vector.
20    auto iter = set_intersection(set1.begin(), set1.end(),
21                                set2.begin(), set2.end(),
22                                result.begin());
23
24    // Resize the result vector to remove unused elements.
25    result.resize(iter - result.begin());
26
27    // Display the result vector's elements
28    cout << "The intersection of the sets is:\n";
29    for (auto element : result)

```

```

30     {
31         cout << element << " ";
32     }
33     cout << endl;
34
35     return 0;
36 }
```

Program Output

The intersection of the sets is:

3 4

Let's take a closer look at Program 17-29:

- Lines 10 and 11 define `set1` and `set2`, initialized with the values 1, 2, 3, 4, and 3, 4, 5, 6, respectively.
- Line 15 defines a vector named `result`. We will use the `vector` to hold the intersection of `set1` and `set2`. Notice the value we are passing to the `vector` constructor: `set1.size() + set2.size()`. This causes the `vector` to be large enough to hold both sets. (The `vector` elements will be initialized with 0.)
- Line 18 defines an iterator that can be used with the `vector`. The name of the iterator is `iter`.
- Lines 20 through 22 call the `set_intersection` function. The function gets the intersection of `set1` and `set2`, and stores the intersection in the `result` vector. The function returns an iterator that is assigned to `iter`. The `iter` iterator points to the *end* of the range of elements in the intersection.
- Line 25 resizes the `result` vector to remove any unused elements.
- Lines 28 through 33 display the program's output, which includes the contents of the `result` vector.

Finding the Difference of Sets with the `set_difference` Function

The difference of two sets is the set of elements that appear in the first set, but not the second set. For example, suppose `set1` contains the values 1, 2, 3, 4, and `set2` contains the values 3, 4, 5, 6. The difference of `set1` and `set2` would be a set containing the values 1, 2.

You can call the STL algorithm function `set_difference` to get the difference of two sets. The function stores the difference in a third container. Program 17-30 demonstrates how to use the function.

Program 17-30

```

1 #include <iostream>
2 #include <set>
3 #include <algorithm>
4 #include <vector>
5 using namespace std;
6
7 int main()
```

(program continues)

Program 17-30 (continued)

```

8  {
9    // Create two sets.
10   set<int> set1 = {1, 2, 3, 4};
11   set<int> set2 = {3, 4, 5, 6};
12
13   // Create a vector to hold the difference. The
14   // vector must be large enough to hold both sets.
15   vector<int> result(set1.size() + set2.size());
16
17   // Get the difference of the sets. The result vector
18   // will hold the difference, and iter will point to
19   // the end of the result vector.
20   auto iter = set_difference(set1.begin(), set1.end(),
21                             set2.begin(), set2.end(),
22                             result.begin());
23
24   // Resize the result vector to remove unused elements.
25   result.resize(iter - result.begin());
26
27   // Display the result vector's elements
28   cout << "The difference of set1 and set2 is:\n";
29   for (auto element : result)
30   {
31     cout << element << " ";
32   }
33   cout << endl;
34
35   return 0;
36 }
```

Program Output

The difference of set1 and set2 is:

1 2

Let's take a closer look at Program 17-30:

- Lines 10 and 11 define `set1` and `set2`, initialized with the values 1, 2, 3, 4, and 3, 4, 5, 6, respectively.
- Line 15 defines a `vector` named `result`. We will use the `vector` to hold the difference of `set1` and `set2`. Notice the value we are passing to the `vector` constructor: `set1.size() + set2.size()`. This causes the `vector` to be large enough to hold both sets. (The `vector` elements will be initialized with 0.)
- Line 18 defines an iterator that can be used with the `vector`. The name of the iterator is `iter`.
- Lines 20 through 22 call the `set_difference` function. The function gets the difference of `set1` and `set2`, and stores the difference in the `result` vector. The function returns an iterator that is assigned to `iter`. The `iter` iterator points to the *end* of the range of elements in the difference.
- Line 25 resizes the `result` vector to remove any unused elements.
- Lines 28 through 33 display the program's output, which includes the contents of the `result` vector.

Finding the Symmetric Difference of Sets with the set_symmetric_difference Function

The symmetric difference of two sets is the set of elements that are in either set, but not in both. For example, suppose `set1` contains the values 1, 2, 3, 4, and `set2` contains the values 3, 4, 5, 6. The symmetric difference of `set1` and `set2` would be a set containing the values 1, 2, 5, 6.

You can call the STL algorithm function `set_symmetric_difference` to get the symmetric difference of two sets. The function stores the symmetric difference in a third container. Program 17-31 demonstrates how to use the function.

Program 17-31

```

1 #include <iostream>
2 #include <set>
3 #include <algorithm>
4 #include <vector>
5 using namespace std;
6
7 int main()
8 {
9     // Create two sets.
10    set<int> set1 = {1, 2, 3, 4};
11    set<int> set2 = {3, 4, 5, 6};
12
13    // Create a vector to hold the symmetric difference.
14    // The vector must be large enough to hold both sets.
15    vector<int> result(set1.size() + set2.size());
16
17    // Get the symmetric difference of the sets. The result
18    // vector will hold the symmetric difference, and iter
19    // will point to the end of the result vector.
20    auto iter = set_symmetric_difference(set1.begin(), set1.end(),
21                                         set2.begin(), set2.end(),
22                                         result.begin());
23
24    // Resize the result vector to remove unused elements.
25    result.resize(iter - result.begin());
26
27    // Display the result vector's elements
28    cout << "The symmetric difference of the sets is:\n";
29    for (auto element : result)
30    {
31        cout << element << " ";
32    }
33    cout << endl;
34
35    return 0;
36 }
```

Program Output

The symmetric difference of the sets is:
1 2 5 6

Let's take a closer look at Program 17-31:

- Lines 10 and 11 define `set1` and `set2`, initialized with the values 1, 2, 3, 4, and 3, 4, 5, 6, respectively.
- Line 15 defines a `vector` named `result`. We will use the `vector` to hold the difference of `set1` and `set2`. Notice the value we are passing to the `vector` constructor: `set1.size() + set2.size()`. This causes the `vector` to be large enough to hold both sets. (The `vector` elements will be initialized with 0.)
- Line 18 defines an iterator that can be used with the `vector`. The name of the iterator is `iter`.
- Lines 20 through 22 call the `set_symmetric_difference` function. The function gets the symmetric difference of `set1` and `set2`, and stores the symmetric difference in the `result` vector. The function returns an iterator that is assigned to `iter`. The `iter` iterator points to the *end* of the range of elements in the symmetric difference.
- Line 25 resizes the `result` vector to remove any unused elements.
- Lines 28 through 33 display the program's output, which includes the contents of the `result` vector.

Finding Subsets

Suppose you have two sets and one of those sets includes all of the elements of the other set. For example, suppose `set1` contains the values 1, 2, 3, 4, and `set2` contains the values 2, 3. In this example, `set1` contains all of the elements of `set2`, which means that `set2` is a *subset* of `set1`.

You can call the STL function `includes` to determine whether one set includes all of the elements of another set. You use iterators to pass two sorted ranges of elements to the `includes` function, and the function returns `true` if the first range includes all of the elements in the second range. Otherwise, the function returns `false`. Program 17-32 demonstrates how to use the function. The program determines whether the elements in `set1` include all of the elements in `set2`.

Program 17-32

```

1 #include <iostream>
2 #include <set>
3 #include <algorithm>
4 using namespace std;
5
6 int main()
7 {
8     // Create two sets.
9     set<int> set1 = {1, 2, 3, 4};
10    set<int> set2 = {2, 3};
11
12    // Determine whether set1 includes the
13    // elements of set2.
14    if (includes(set1.begin(), set1.end(),
15                  set2.begin(), set2.end()))
16    {
17        cout << "set2 is a subset of set1.\n";
18    }

```

```

19     else
20     {
21         cout << "set2 is NOT a subset of set1.\n";
22     }
23
24     return 0;
25 }
```

Program Output

set2 is a subset of set1.



In the Spotlight:

Set Operations

In this section, you will look at Program 17-33, which demonstrates various set operations. The program creates two sets: one that holds the names of students on the baseball team, and another that holds the names of students on the basketball team. The program then performs the following operations:

- It finds the intersection of the sets to display the names of students who play both sports.
- It finds the union of the sets to display the names of students who play either sport.
- It finds the difference of the `baseball` and `basketball` sets to display the names of students who play baseball but not basketball.
- It finds the difference of the `basketball` and `baseball` (`basketball-baseball`) sets to display the names of students who play basketball but not baseball. It also finds the difference of the `baseball` and `basketball` (`baseball-basketball`) sets to display the names of students who play baseball but not basketball.
- It finds the symmetric difference of the `basketball` and `baseball` sets to display the names of students who play one sport but not both.

Program 17-33

```

1 #include <iostream>
2 #include <string>
3 #include <algorithm>
4 #include <set>
5 #include <vector>
6 using namespace std;
7
8 // Function prototypes
9 void displaySet(set<string>);
10 void displayIntersection(set<string>, set<string>);
11 void displayUnion(set<string>, set<string>);
12 void displayDifference(set<string>, set<string>);
13 void displaySymmetricDifference(set<string>, set<string>);
14
```

(program continues)

Program 17-33 (continued)

```

15 int main()
16 {
17     // Create sets for the baseball & basketball teams.
18     set<string> baseball1 = {"Jodi", "Carmen", "Aida", "Alicia"};
19     set<string> basketball1 = {"Eva", "Carmen", "Alicia", "Sarah"};
20
21     // Display the elements of the basketball set.
22     cout << "The following students are on the basketball team:\n";
23     displaySet(basketball1);
24
25     // Display the elements of the basketball set.
26     cout << "\n\nThe following students are on the baseball team:\n";
27     displaySet(baseball1);
28
29     // Display the intersection of the two sets.
30     cout << "\n\nThe following students play both sports:\n";
31     displayIntersection(baseball1, basketball1);
32
33     // Display the union of the two sets.
34     cout << "\n\nThe following students play either sport:\n";
35     displayUnion(baseball1, basketball1);
36
37     // Display the difference of baseball1 and basketball.
38     cout << "\n\nThe following students play baseball,\n";
39     cout << "but not basketball:\n";
40     displayDifference(baseball1, basketball1);
41
42     // Display the difference of basketball and baseball.
43     cout << "\n\nThe following students play basketball,\n";
44     cout << "but not baseball:\n";
45     displayDifference(basketball1, baseball1);
46
47     // Display the symmetric difference of the two sets.
48     cout << "\n\nThe following students play one sport,\n";
49     cout << "but not both:\n";
50     displaySymmetricDifference(baseball1, basketball1);
51
52 }
53
54 // The displaySet function displays the contents of a set<string>.
55 void displaySet(set<string> s)
56 {
57     for (auto element : s)
58         cout << element << " ";
59 }
60
61 // The displayIntersection function displays the intersection
62 // of two set<string> objects.
63 void displayIntersection(set<string> set1, set<string> set2)
64 {
65 }
```

```
64  {
65      // Create a vector to hold the intersection.
66      vector<string> result(set1.size() + set2.size());
67
68      // Get the intersection of the sets.
69      auto iter = set_intersection(set1.begin(), set1.end(),
70                                  set2.begin(), set2.end(),
71                                  result.begin());
72
73      // Resize the result vector to remove unused elements.
74      result.resize(iter - result.begin());
75
76      // Display the result vector's elements
77      for (auto element : result)
78      {
79          cout << element << " ";
80      }
81  }
82
83 // The displayUnion function displays the union of two
84 // set<string> objects.
85 void displayUnion(set<string> set1, set<string> set2)
86 {
87     // Create a vector to hold the union.
88     vector<string> result(set1.size() + set2.size());
89
90     // Get the union of the sets.
91     auto iter = set_union(set1.begin(), set1.end(),
92                          set2.begin(), set2.end(),
93                          result.begin());
94
95     // Resize the result vector to remove unused elements.
96     result.resize(iter - result.begin());
97
98     // Display the result vector's elements
99     for (auto element : result)
100    {
101        cout << element << " ";
102    }
103}
104
105 // The displayDifference function displays the difference
106 // of two set<string> objects.
107 void displayDifference(set<string> set1, set<string> set2)
108 {
109     // Create a vector to hold the union.
110     vector<string> result(set1.size() + set2.size());
111
112     // Get the difference of the sets.
113     auto iter = set_difference(set1.begin(), set1.end(),
114                               set2.begin(), set2.end(),
115                               result.begin());
116
```

(program continues)

Program 17-33 *(continued)*

```

117     // Resize the result vector to remove unused elements.
118     result.resize(iter - result.begin());
119
120     // Display the result vector's elements
121     for (auto element : result)
122     {
123         cout << element << " ";
124     }
125 }
126
127 // The displaySymmetricDifference function displays the
128 // symmetric difference of two set<string> objects.
129 void displaySymmetricDifference(set<string> set1, set<string> set2)
130 {
131     // Create a vector to hold the union.
132     vector<string> result(set1.size() + set2.size());
133
134     // Get the symmetric difference of the sets.
135     auto iter = set_symmetric_difference(set1.begin(), set1.end(),
136                                         set2.begin(), set2.end(),
137                                         result.begin());
138
139     // Resize the result vector to remove unused elements.
140     result.resize(iter - result.begin());
141
142     // Display the result vector's elements
143     for (auto element : result)
144     {
145         cout << element << " ";
146     }
147 }
```

Program Output

The following students are on the baseball team:

Aida Alicia Carmen Jodi

The following students are on the basketball team:

Alicia Carmen Eva Sarah

The following students play both sports:

Alicia Carmen

The following students play either sport:

Aida Alicia Carmen Eva Jodi Sarah

The following students play baseball, but not basketball:

Aida Jodi

The following students play basketball, but not baseball:

Eva Sarah

The following students play one sport, but not both:

Aida Eva Jodi Sarah



Checkpoint

- 17.36 When a range of elements is denoted by two iterators, to what does the first iterator point? To what does the second iterator point?
- 17.37 What value will be stored in `v[0]` after the following code executes?
- ```
vector<int> v = {8, 4, 6, 1, 9};
sort(v.begin(), v.end());
```
- 17.38 What must you do to a range of elements before searching it with the `binary_search()` function?
- 17.39 If the elements that you are sorting with the `sort()` function contain your own class objects, you must be sure that the class overloads what operator?
- 17.40 If the elements that you are searching with the `binary_search()` function contain your own class objects, you must be sure that the class overloads what operator?
- 17.41 What is a function pointer?
- 17.42 Assume `vect` is a vector that contains 100 `int` elements, and the following statement appears in a program:
- ```
for_each(vect.begin(), vect.end(), myFunction);
```
- Without knowing anything else about the program, answer the following questions:
- What is `myFunction`?
 - How many arguments does `myFunction` accept? What are the data type(s) of the argument(s)?
 - What value does `myFunction` return?
 - How many times will the statement cause `myFunction` to be called?

17.7

Introduction to Function Objects and Lambda Expressions

CONCEPT: A function object is an object of a class that overloads the function call operator. Function objects behave just like functions, and can be passed as parameters to other functions. A lambda expression is a convenient way of creating a function object.



VideoNote
Function Objects and Lambda Expressions

A *function object*, also known as a *functor*, is an object that acts like a function. Function objects can be called, just like regular functions. They can accept arguments, and they can return values. To create a function object, you write a class that overloads the parentheses () operator, which is also known as the *function call operator*. Let's look at a simple example, the `Sum` class.

Contents of Sum.h

```

1  #ifndef SUM_H
2  #define SUM_H
3
4  class Sum
5  {
6  public:
7      int operator()(int a, int b)
8      { return a + b; }
9  };
10 #endif

```

The `Sum` class has one member function: the `operator()` function. When you write an `operator()` function, the function can have as many parameter variables as necessary. Notice in the `Sum` class, the `operator()` function has two `int` parameter variables: `a` and `b`. Also, notice in the `Sum` class the `operator()` function returns an `int`. The value it returns is the sum of `a` and `b`. Program 17-34 demonstrates the `Sum` class.

Program 17-34

```

1  #include <iostream>
2  #include "Sum.h"
3  using namespace std;
4
5  int main()
6  {
7      // Local variables
8      int x = 10;
9      int y = 2;
10     int z = 0;
11
12     // Create a Sum object.
13     Sum sum;
14
15     // Call the sum function object.
16     z = sum(x, y);
17
18     // Display the result.
19     cout << z << endl;
20
21     return 0;
22 }

```

Program Output

12

Let's take a closer look at Program 17-34:

- Lines 8 through 10 define three local `int` variables: `x`, `y`, and `z`.
- Line 13 creates an instance of the `Sum` class. The object's name is `sum`.
- Line 16 calls the `sum` object, as if it were a function. The statement passes `x` and `y` as arguments, and assigns the return value to the `z` variable.
- Line 19 displays the `z` variable.

In the previous section on STL algorithms, you saw examples of STL function templates that accept function pointers as arguments, allowing you to plug one of your own functions into the algorithm. If you prefer, you can pass a function object instead of a function pointer to any of these STL algorithms.

Let's look at an example, using the `count_if()` function. Recall the `count_if()` function iterates over a range of elements, passing each element as an argument to a function that you provide. The function you provide must accept one argument, and return either `true` or `false`. You can provide the function as either a function pointer, or a function object. The `count_if` function returns the number of elements for which *function* returns `true`. We will use a function object along with the `count_if()` function to count the number of even numbers that appear in a `vector`. Our function object will be an instance of the `IsEven` class, shown here:

Contents of IsEven.h

```

1 #ifndef IS_EVEN_H
2 #define IS_EVEN_H
3
4 class IsEven
5 {
6 public:
7     bool operator()(int x)
8     { return x % 2 == 0; }
9 };
10#endif

```

The `IsEven` class has one member function: the `operator()` function. The `operator()` function has one `int` parameter variable, `x`. The function returns `true` if `x` is an even number, or `false` otherwise. Program 17-35 creates an instance the `IsEven` class, and uses it as a function object with the `count_if()` function.

Program 17-35

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include "IsEven.h"
5 using namespace std;
6
7 int main()
8 {
9     // Create a vector of ints.
10    vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
11
12    // Create an instance of the IsEven class.
13    IsEven isNumberEven;
14
15    // Get the number of elements that even.
16    int evenNums = count_if(v.begin(), v.end(), isNumberEven);
17

```

(program continues)

Program 17-35 (continued)

```

18     // Display the results.
19     cout << "The vector contains " << evenNums << " even numbers.\n";
20     return 0;
21 }
```

Program Output

The vector contains 4 even numbers.

Constructing an Anonymous Function Object

In line 13 of Program 17-35, we created an instance of the `IsEven` class, and we named it `isNumberEven`. Then, in line 16 we passed the object to the `count_if()` function. Function objects can be called at the point of their creation, however, without ever being given a name. For example, the following code creates an `IsEven` function object and calls it with the argument 2, without ever giving the function object a name:

```

if (IsEven(2))
    cout << "The number is even.\n";
else
    cout << "The number is not even.\n";
```

Objects that are created and used without being given a name are said to be *anonymous*. Program 17-36 is similar to Program 17-35, except that an anonymous instance of the `IsEven` class is passed to the `count_if()` function.

Program 17-36

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include "IsEven.h"
5 using namespace std;
6
7 int main()
8 {
9     // Create a vector of ints.
10    vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
11
12    // Get the number of elements that even.
13    int evenNums = count_if(v.begin(), v.end(), IsEven());
14
15    // Display the results.
16    cout << "The vector contains " << evenNums << " even numbers.\n";
17    return 0;
18 }
```

Program Output

The vector contains 4 even numbers.

Predicate Terminology

A function or function object that returns a Boolean value is called a *predicate*. A predicate that takes only one argument is called a *unary predicate*. (An instance of the `IsEven` class, previously shown, would be a unary predicate.) A predicate that takes two arguments is called a *binary predicate*. It is important that you are familiar with this terminology because it is used in much of the available C++ documentation and literature.

Lambda Expressions

11

A *lambda expression* is a compact way of creating a function object without having to write a class declaration. It is an expression that contains only the logic of the object's `operator()` member function. When the compiler encounters a lambda expression, it automatically generates a function object in memory, using the code that you provide in the lambda expression for the `operator()` member function.

Typically, a lambda expression takes the following general format:

```
[ ](parameter list) { function body }
```

The `[]` at the beginning of the expression is known as the *lambda introducer*. It marks the beginning of a lambda expression. The *parameter list* is a list of parameter declarations for the function object's `operator()` member function, and the *function body* is the code that should be the body of the object's `operator()` member function.

For example, the lambda expression for a function object that computes the sum of two integers is:

```
[ ](int a, int b) { return x + y; }
```

And, the lambda expression for a function object that determines whether an integer is even is:

```
[ ](int x) { return x % 2 == 0; }
```

As another example, the lambda expression that takes an integer as input and prints the square of that integer is written like this:

```
[ ](int a) { cout << a * a << " "; }
```



NOTE: Optionally, a *capture list* can appear inside the brackets `[]` of a lambda expression. A capture list is a list of variables in the surrounding scope of the lambda expression that can be accessed from the lambda's function body.

When you call a lambda expression, you write a list of arguments, enclosed in parentheses, right after the expression. For example, the following code snippet displays 7, which is the sum of the variables `x` and `y`:

```
int x = 2;
int y = 5;
cout << [ ](int a, int b) {return a + b; }(x, y) << endl;
```

The following code segment counts the even numbers in a vector, in the manner of Program 17-36. However, it uses a lambda expression in place of the `IsEven()` function object as the third argument to the `count_if()` function.

```
// Create a vector of ints.
vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };

// Get the number of elements that are even.
int evenNums = count_if(v.begin(), v.end(), [](int x) {return x % 2 == 0;});

// Display the results.
cout << "The vector contains " << evenNums << " even numbers.\n";
```

Because lambda expressions generate function objects, you can assign a lambda expression to a variable of a suitable type then call it through the variable's name. For example, the lambda expression shown in the following code gives us a function object that adds two integers. The function object is assigned to a variable named `sum`:

```
auto sum = [](int a, int b) {return a + b;};
```

Once the statement has executed, we can use the `sum` variable to call the function object, as shown here:

```
int x = 2;
int y = 5;
int z = sum(x, y);
```

After this code has executed, the variable `z` will be assigned 7. Program 17-37 shows another example. It is similar to Program 17-36, except that a lambda expression (which determines whether a value is even) is assigned to a variable named `isEven`, and that variable is passed as the third argument to the `count_if()` function.

Program 17-37

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main()
7 {
8     // Create a vector of ints.
9     vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };
10
11    // Use a lambda expression to create a function object.
12    auto isEven = [](int x) { return x % 2 == 0; };
13
14    // Get the number of elements that even.
15    int evenNums = count_if(v.begin(), v.end(), isEven);
16
17    // Display the results.
18    cout << "The vector contains " << evenNums << " even numbers.\n";
19
20 }
```

Program Output

The vector contains 4 even numbers.

Functional Classes in the STL

Function objects are so useful the C++ library defines a number of classes that you can instantiate to create function objects in your program. To use these classes, you must #include the <functional> header file. Table 17-15 lists those that can be used to compare objects, but you can find information on many more in various resources online.

Table 17-15 STL Function Object Classes

Functional Class	Description
<code>less<T></code>	<code>less<T>()(T a, T b)</code> is true if and only if <code>a < b</code>
<code>less_equal<T></code>	<code>less_equal()(T a, T b)</code> is true if and only if <code>a <= b</code>
<code>greater<T></code>	<code>greater<T>()(T a, T b)</code> is true if and only if <code>a > b</code>
<code>greater_equal<T></code>	<code>greater_equal<T>()(T a, T b)</code> is true if and only if <code>a >= b</code>

For example, the STL `sort` function normally sorts the elements in a range in ascending order. This is because the `sort` function, by default, uses the `<` operator to compare elements. But what if you want to use the function to sort a range of elements in descending order? All you have to do is make the function use the `>` operator instead.

As it turns out, the `sort` function takes a comparison function as an optional third argument. The argument can be a function pointer or a function object. The function must accept two arguments (of the same type as the range elements), and must return a Boolean value. When you pass a comparison function as the third argument to the `sort` function, the `sort` function will use the comparison function instead of the `<` operator to compare range elements. To make the `sort` function use the `>` operator, you can pass an instance of the `greater<T>` class as the comparison function. Program 17-38 demonstrates how this can be done to sort a `vector` in descending order.

Program 17-38

```

1 #include <iostream>
2 #include <vector>
3 #include <functional>
4 #include <algorithm>
5 using namespace std;
6
7 int main()
8 {
9     // Create an unsorted vector of ints.
10    vector<int> v = {8, 1, 7, 2, 6, 3, 5, 4};
11
12    // Display the vector contents.
13    cout << "Original order:\n";
14    for (auto element : v)
15        cout << element << " ";
16
17    // Sort the vector in ascending order.
18    sort(v.begin(), v.end());
19

```

(program continues)

Program 17-38 *(continued)*

```

20     // Display the vector contents again.
21     cout << "\nAscending order:\n";
22     for (auto element : v)
23         cout << element << " ";
24
25     // Sort the vector in descending order.
26     sort(v.begin(), v.end(), greater<int>());
27
28     // Display the vector contents again.
29     cout << "\nDescending order:\n";
30     for (auto element : v)
31         cout << element << " ";
32     cout << endl;
33
34     return 0;
35 }
```

Program Output

Original order:
8 1 7 2 6 3 5 4
Ascending order:
1 2 3 4 5 6 7 8
Descending order:
8 7 6 5 4 3 2 1

**Checkpoint**

- 17.43 What is a function object?
- 17.44 Which operator must be overloaded in a class before the class can be used to create function objects?
- 17.45 What is an anonymous function object?
- 17.46 What is a predicate?
- 17.47 What is a unary predicate?
- 17.48 What is a binary predicate?
- 17.49 What is a lambda expression?

Review Questions and Exercises**Short Answer**

1. What two emplacement member functions are provided by the `vector` class? How are these member functions different from the `insert()` and `push_back()` member functions?
2. What is the difference between the `vector` class's `size()` member function and the `capacity()` member function?

3. If you want to store objects of a class that you have written as values in a `map`, what requirement must the class meet?
4. If you want to store objects of a class that you have written as keys in a `map`, what requirement must the class meet?
5. What is the difference between a `map` and an `unordered_map`?
6. What is the difference between a `map` and a `multimap`?
7. What happens if you use the `insert()` member function to insert a value into a `set`, and that value already exists in the `set`?
8. If you want to store objects of a class that you have written as keys in a `set`, what requirement must the class meet?
9. What is the difference between a `set` and a `multiset`?
10. How does the behavior of the `count()` member function differ between the `set` class and the `multiset` class?
11. How does the behavior of the `equal_range()` member function differ between the `set` class and the `multiset` class?
12. What are two differences between the `set` and `unordered_set` classes?
13. When using one of the STL algorithm function templates, you typically work with a range of elements that are denoted by two iterators, to what does the first iterator point? To what does the second iterator point?
14. You have written a class, and you plan to store objects of that class in a `vector`. If you plan to use the `sort()` and/or `binary_search()` functions on the `vector`'s elements, what operator must the class overload?
15. What is a function object?
16. If you want to create function objects from a class, what must the class overload?
17. What is an anonymous function object?
18. What is a lambda expression?

Fill-in-the-Blank

19. There are two types of container classes in the STL: _____ and _____.
20. A(n) _____ container organizes data in a sequential fashion similar to an array.
21. A container _____ class is not itself a container, but a class that adapts a container to a specific use.
22. A(n) _____ container stores its data in a nonsequential way that makes it faster to locate elements.
23. _____ are pointer-like objects used to access data stored in a container.
24. Each element that is stored in a `map` has two parts: a _____ and a _____.
25. _____ is a map container that allows duplicate keys.
26. The _____ class is an associative container that stores a collection of unique, sorted values.
27. The _____ header file defines several function templates that implement useful algorithms.

28. A _____ is a pointer to a function's executable code.
29. A _____ object is an object that can be called, like a function.
30. A _____ is a function or function object that returns a Boolean value.
31. A _____ is a predicate that takes one argument.
32. A _____ is a predicate that takes two arguments.
33. A _____ is a compact way of creating a function object without having to write a class declaration.

True or False

34. T F The `array` class is a fixed-size container.
35. T F The `vector` class is a fixed-size container.
36. T F You use the `*` operator to dereference an iterator.
37. T F You can use the `++` operator to increment an iterator.
38. T F A container's `end()` member function returns an iterator pointing to the last element in the container.
39. T F A container's `rbegin()` member function returns a reverse iterator pointing to the first element in a container.
40. T F You do not have to declare the size of a `vector` when you define it.
41. T F A `vector` uses an array internally to store its elements.
42. T F A `map` is a sequence container.
43. T F You can store duplicate keys in a map container.
44. T F The `multimap` class's `erase()` member function erases only one element at a time. If you want to erase multiple elements that all have the same key, you will have to call the `erase()` member function multiple times.
45. T F All the elements in a `set` must be unique.
46. T F The elements in a `set` are sorted in ascending order.
47. T F If the same value appears more than once in the initialization list of a `set` definition, an exception will occur at runtime.
48. T F The `unordered_set` container has better performance than the `set` container.
49. T F If two iterators denote a range of elements that will be processed by an STL algorithm function, the element pointed to by the second iterator is not included in the range.
50. T F You must sort a range of elements before searching it with the `binary_search()` function.
51. T F Any class that will be used to create function objects must overload the `operator[]` member function.
52. T F Writing a lambda expression usually requires more code than writing a function object's class declaration and then instantiating the class.
53. T F You can assign a lambda expression to a variable, and then use the variable to call the lambda expression's function object.

Algorithm Workbench

54. Write a statement that defines an `array` object named `numbers` that can hold ten elements of the `double` data type.
55. Write a statement that defines an iterator that can be used with the `array` object that you defined in question 54. The iterator's name should be `it`.
56. Write a statement that defines a `vector` named `numbers` that can hold elements of the `int` data type. Initialize the `vector` with the values 10, 20, 30, 40, and 50.
57. The following statement defines a `vector` of `ints` named `v`. Write a statement that defines another `vector`, named `v2`, which is a copy of `v`.
- ```
vector<int> v = {1, 2, 3, 4, 5};
```
58. Write a range-based `for` loop that displays all of the elements of the `vector` that you defined in question 56.
59. Write a `for` loop that uses an iterator to display all of the elements of the `vector` that you defined in question 56.
60. The following code defines a `vector` and an iterator. Rewrite the statement that defines the iterator so it uses the `auto` key word.
- ```
vector<string> strv = {"one", "two", "three"};
vector<string>::iterator it = strv.begin();
```
61. The following statement defines a `vector` of `ints` named `v`. Write a statement that defines a `const_iterator` named `cit`, and initializes it to point to the first element in `v`.
- ```
vector<int> v = {1, 2, 3, 4, 5};
```
62. The following statement defines a `vector` of `ints` named `v`. The `vector`'s initial contents are: 1, 2, 3, 4, 5. Write code to insert the value 999 into the `vector`, so its contents are: 1, 2, 999, 3, 4, 5.
- ```
vector<int> v = {1, 2, 3, 4, 5};
```
63. Write a statement that defines a `map` named `food`. The keys should be `strings`, and the values should be `ints`.
64. Suppose a program defines a `map` as follows:
- ```
map<int, string> customers;
```
- Write statements that use the `map` class's `emplace` member function to insert the following elements:
- 9001, "Jen Williams"
  - 9002, "Frank Smith"
  - 9003, "Geri Rose"
65. Look at the following `vector` definition:
- ```
vector<int> v = {7, 2, 1, 6, 4, 3, 5};
```
- Write code that displays a message indicating whether the value 6 is found in the `vector`. Use the STL `binary_search()` algorithm to search the `vector` for the value 6.
66. Write a declaration for a class named `Display`. The class should be written in such a way that it can be used to create a function object. A function object created from the class should accept an `int` argument, and display that argument on the screen.

67. Write code that does the following:

- Uses a lambda expression that accepts two `int` arguments, `a` and `b`, and returns the value of `a * b`.
- Assigns the lambda expression to a variable named `multiply`.
- Uses the `multiply` variable to call the lambda expression, passing the values 2 and 10 as arguments. The result should be assigned to an `int` variable named `product`.

Find the Errors

Each of the following code snippets has errors. Find as many as you can.

68. // This code has an error.

```
array<int, 5> a;
a[5] = 99;
```

69. // This code has an error.

```
vector<string> strv = {"one", "two", "three"};
vector<string>::iterator it = strv.cbegin();
```

70. // This code has an error.

```
vector<int> numbers(10);
for (int index = 0; index < numbers.length(); index++)
    numbers[index] = index;
```

71. // This code has an error.

```
vector<int> numbers = {1, 2, 3};
numbers.insert(99);
```

72. // This code has an error.

```
map<string, string> contacts;
contacts.insert("Beth Young", "555-1212");
```

73. // This code has an error.

```
multimap<string, string> phonebook;
phonebook["Megan"] = "555-1212";
```

74. // This code has an error.

```
vector<int> v = {6, 5, 4, 2, 3, 1};
sort(v);
if (binary_search(v, 1))
    cout << "The value 1 is found in the vector.\n";
else
    cout << "The value 1 is NOT found in the vector.\n";
```

75. // This code has an error.

```
auto sum = ()[int a, int b] { return a + b; };
```

Programming Challenges

1. Unique Words

Write a program that opens a specified text file then displays a list of all the unique words found in the file.

Hint: Store each word as an element of a set.



2. Course Information

Write a program that creates a map containing course numbers and the room numbers of the rooms where the courses meet. The map should have the following key-value pairs:

Course Number (Key)	Room Number (Value)
CS101	3004
CS102	4501
CS103	6755
NT110	1244
CM241	1411

The program should also create a map containing course numbers and the names of the instructors that teach each course. The map should have the following key-value pairs:

Course Number (Key)	Instructor (Value)
CS101	Haynes
CS102	Alvarado
CS103	Rich
NT110	Burke
CM241	Lee

The program should also create a map containing course numbers and the meeting times of each course. The map should have the following key-value pairs:

Course Number (Key)	Meeting Time (Value)
CS101	8:00 a.m.
CS102	9:00 a.m.
CS103	10:00 a.m.
NT110	11:00 a.m.
CM241	1:00 p.m.

The program should let the user enter a course number, then it should display the course's room number, instructor, and meeting time.

3. Capital Quiz

Write a program that creates a map containing the U.S. states as keys, and their capitals as values. (Use the Internet to get a list of the states and their capitals.) The program should then randomly quiz the user by displaying the name of a state and ask the user to enter that state's capital. The program should keep a count of the number of correct and incorrect responses. (As an alternative to the U.S. states, the program can use the names of countries and their capitals.)

4. File Encryption and Decryption

Write a program that uses a map to assign "codes" to each letter of the alphabet. For example:

```
map<char, char> codes =
{ {'A', '%'}, {'a', '9'}, {'B', '@'}, {'b', '#'}, etc ...};
```

Using this example, the letter ‘A’ would be assigned the symbol %, the letter ‘a’ would be assigned the number 9, the letter ‘B’ would be assigned the symbol @, and so forth. The program should open a specified text file, read its contents, then use the `map` to write an encrypted version of the file’s contents to a second file. Each character in the second file should contain the code for the corresponding character in the first file. Write a second program that opens an encrypted file and displays its decrypted contents on the screen.

5. Text File Analysis

Write a program that reads the contents of two text files and compares them in the following ways:

- It should display a list of all the unique words contained in both files.
- It should display a list of the words that appears in both files.
- It should display a list of the words that appears in the first file, but not the second.
- It should display a list of the words that appears in the second file, but not the first.
- It should display a list of the words that appears in either the first or second file, but not in both.

Hint: Use set operations to perform these analyses. Also, see Chapter 10 for a discussion of string tokenizing.

6. Word Frequency

Write a program that reads the contents of a text file. The program should create a `map` in which the keys are the individual words found in the file and the values are the number of times each word appears. For example, if the word “the” appears 128 times, the `map` would contain an element with “the” as the key and 128 as the value. The program should either display the frequency of each word or create a second file containing a list of each word and its frequency.

Hint: See Chapter 10 for a discussion of string tokenizing.

7. Word Index

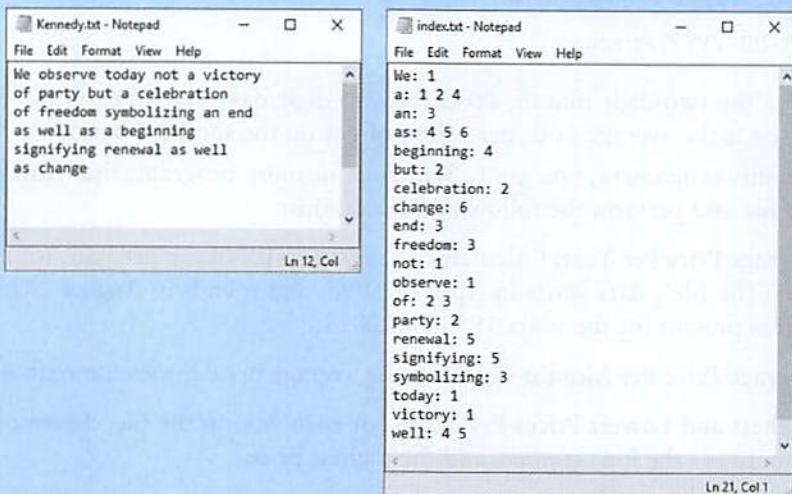
Write a program that reads the contents of a text file. The program should create a `map` in which the key–value pairs are described as follows:

- **Key**—The keys are the individual words found in the file.
- **Values**—Each value is a `set` that contains the line numbers in the file where the word (the key) is found.

For example, suppose the word “robot” is found in lines 7, 18, 94, and 138. The `map` would contain an element in which the key was the string “robot”, and the value was a `set` containing the numbers 7, 18, 94, and 138.

Once the `map` is built, the program should create another text file, known as a word index, listing the contents of the `map`. The word index file should contain an alphabetical listing of the words that are stored as keys in the `map`, along with the line numbers where the words appears in the original file. Figure 17-9 shows an example of an original text file (`Kennedy.txt`) and its index file (`index.txt`).

Hint: See Chapter 10 for a discussion of string tokenizing.

Figure 17-9 Example original file and index file

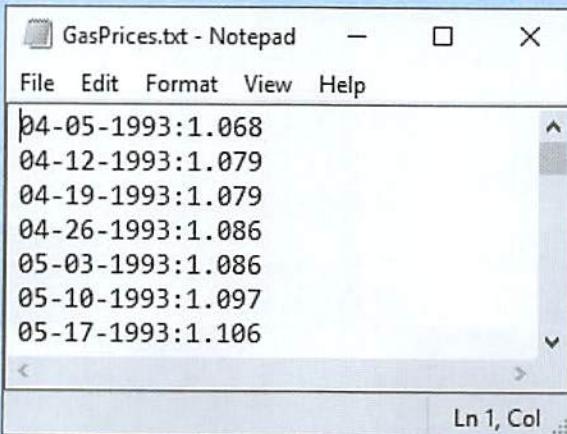
8. Prime Number Generation

A positive integer greater than 1 is said to be *prime* if it has no divisors other than 1 and itself. A positive integer greater than 1 is *composite* if it is not prime. Write a program that asks the user to enter an integer greater than 1, then displays all of the prime numbers that are less than or equal to the number entered. The program should work as follows:

- Once the user has entered a number, the program should populate a *vector* with all of the integers from 2, up through the value entered.
- The program should then use the STL's *for_each* function to step through the *vector*. The *for_each* function should pass each element to a function object that displays the element if it is a prime number.

9. Gas Prices

In the student sample program files for this chapter, you will find a text file named *GasPrices.txt*. The file contains the weekly average prices for a gallon of gas in the U.S., beginning on April 5, 1993, and ending on August 26, 2013. Figure 17-10 shows an example of the first few lines of the file's contents.

Figure 17-10 The *GasPrices.txt* file

Each line in the file contains the average price for a gallon of gas on a specific date. Each line is formatted in the following way:

MM-DD-YYYY:Price

MM is the two-digit month, *DD* is the two-digit day, and *YYYY* is the four-digit year. *Price* is the average price per gallon of gas on the specified date.

For this assignment, you are to write one or more programs that read the contents of the file and perform the following calculations:

Average Price Per Year: Calculate the average price of gas per year, for each year in the file. (The file's data starts in April of 1993, and it ends in August 2013. Use the data that is present for the years 1993 and 2013.)

Average Price Per Month: Calculate the average price for each month in the file.

Highest and Lowest Prices Per Year: For each year in the file, determine the date and amount for the lowest price, and the highest price.

List of Prices, Lowest to Highest: Generate a text file that lists the dates and prices, sorted from the lowest price to the highest.

List of Prices, Highest to lowest: Generate a text file that lists the dates and prices, sorted from the highest price to the lowest.

You can write one program to perform all of these calculations, or you can write different programs, one for each calculation. Regardless of the approach that you take, you should read the contents of the *GasPrices.txt* file, and extract its data into one or more STL containers appropriate for your algorithm.

Hint: See Chapter 10 for a discussion of string tokenizing.

18 Linked Lists

TOPICS

- | | |
|--|--|
| 18.1 Introduction to the Linked List ADT | 18.4 Variations of the Linked List |
| 18.2 Linked List Operations | 18.5 The STL <code>list</code> and <code>forward_list</code> |
| 18.3 A Linked List Template | Containers |

18.1

Introduction to the Linked List ADT

CONCEPT: Dynamically allocated data structures may be linked together in memory to form a chain.

A linked list is a series of connected *nodes*, where each node is a data structure. A linked list can grow or shrink in size as the program runs. This is possible because the nodes in a linked list are dynamically allocated. If new data need to be added to a linked list, the program simply allocates another node and inserts it into the series. If a particular piece of data needs to be removed from the linked list, the program deletes the node containing that data.

Advantages of Linked Lists over Arrays and vectors

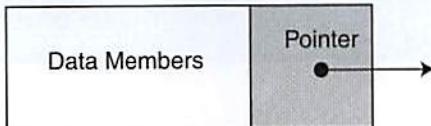
Although linked lists are more complex to code and manage than arrays, they have some distinct advantages. First, a linked list can easily grow or shrink in size. In fact, the programmer doesn't need to know how many nodes will be in the list. They are simply created in memory as they are needed.

One might argue that linked lists are not superior to `vectors` (found in the Standard Template Library), because `vectors`, too, can expand or shrink. The advantage that linked lists have over `vectors`, however, is the speed at which a node may be inserted into or deleted from the list. To insert a value into the middle of a `vector` requires all the elements below the insertion point to be moved one position toward the `vector`'s end, thus making room for the new value. Likewise, removing a value from a `vector` requires all the elements below the removal point to be moved one position toward the `vector`'s beginning. When a node is inserted into or deleted from a linked list, none of the other nodes have to be moved.

The Composition of a Linked List

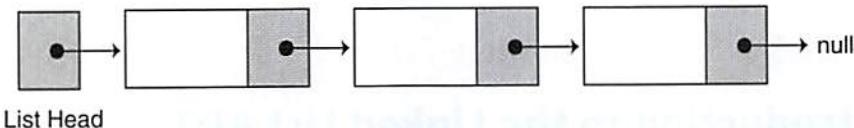
Each node in a linked list contains one or more members that represent data. (Perhaps the nodes hold inventory records, or customer names, addresses, and telephone numbers.) In addition to the data, each node contains a pointer, which can point to another node. The makeup of a node is illustrated in Figure 18-1.

Figure 18-1 A node



A linked list is called “linked” because each node in the series has a pointer that points to the next node in the list. This creates a chain where the first node points to the second node, the second node points to the third node, and so on. This is illustrated in Figure 18-2.

Figure 18-2 Nodes linked by pointers



The list depicted in Figure 18-2 has three nodes, plus a pointer known as the *list head*. The head simply points to the first node in the list. Each node, in turn, points to the next node in the list. The first node points to the second node, which points to the third node. Because the third node is the last one in the list, it points to the null address (address 0). This is usually how the end of a linked list is signified—by letting the last node point to null.



NOTE: Figure 18-2 depicts the nodes in the linked list as being very close to each other, neatly arranged in a row. In reality, the nodes may be scattered around various parts of memory.

Declarations

So how is a linked list created in C++? First you must declare a data structure that will be used for the nodes. For example, the following `struct` could be used to create a list where each node holds a `double`:

```
struct ListNode
{
    double value;
    ListNode *next;
};
```

The first member of the `ListNode` structure is a `double` named `value`. It will be used to hold the node’s data. The second member is a pointer named `next`. The pointer can hold the address of any object that is a `ListNode` structure. This allows each `ListNode` structure to point to the next `ListNode` structure in the list.

Because the `ListNode` structure contains a pointer to an object of the same type as that being declared, it is known as a *self-referential data structure*. This structure makes it possible to create nodes that point to other nodes of the same type.

The next step is to define a pointer to serve as the list head, as shown here:

```
ListNode *head;
```

Before you use the `head` pointer in any linked list operations, you must be sure it is initialized to `nullptr` because that marks the end of the list. Once you have declared a node data structure and have created a null `head` pointer, you have an empty linked list. The next step is to implement operations with the list.



Checkpoint

- 18.1 Describe the two parts of a node.
- 18.2 What is a list head?
- 18.3 What signifies the end of a linked list?
- 18.4 What is a self-referential data structure?

18.2

Linked List Operations

CONCEPT: The basic linked list operations are appending a node, traversing the list, inserting a node, deleting a node, and destroying the list.

In this section, we will develop an abstract data type that performs basic linked list operations using the `ListNode` structure and `head` pointer defined in the previous section. We will use the following class declaration, which is stored in `NumberList.h`.

Contents of NumberList.h

```
1 // Specification file for the NumberList class
2 #ifndef NUMBERLIST_H
3 #define NUMBERLIST_H
4
5 class NumberList
6 {
7 private:
8     // Declare a structure for the list
9     struct ListNode
10    {
11         double value;           // The value in this node
12         struct ListNode *next; // To point to the next node
13     };
14
15     ListNode *head;          // List head pointer
16 }
```

```

17 public:
18     // Constructor
19     NumberList()
20     { head = nullptr; }
21
22     // Destructor
23     ~NumberList();
24
25     // Linked list operations
26     void appendNode(double);
27     void insertNode(double);
28     void deleteNode(double);
29     void displayList() const;
30 };
31 #endif

```

Notice the constructor initializes the head pointer to `nullptr`. This establishes an empty linked list. The class has member functions for appending, inserting, and deleting nodes, as well as a `displayList` function that displays all the values stored in the list. The destructor destroys the list by deleting all its nodes. These functions are defined in `NumberList.cpp`. We will examine the member functions individually.

Appending a Node to the List



To append a node to a linked list means to add the node to the end of the list. The `appendNode` member function accepts a `double` argument, `num`. The function will allocate a new `ListNode` structure, store the value in `num` in the node's `value` member, and append the node to the end of the list. Here is a pseudocode representation of the general algorithm:

```

Create a new node.
Store data in the new node.
If there are no nodes in the list
    Make the new node the first node.
Else
    Traverse the list to find the last node.
    Add the new node to the end of the list.
End If.

```

Here is the actual C++ code for the function:

```

11 void NumberList::appendNode(double num)
12 {
13     ListNode *newNode; // To point to a new node
14     ListNode *nodePtr; // To move through the list
15
16     // Allocate a new node and store num there.
17     newNode = new ListNode;
18     newNode->value = num;
19     newNode->next = nullptr;
20
21     // If there are no nodes in the list
22     // make newNode the first node.

```

```

23     if (!head)
24         head = newNode;
25     else // Otherwise, insert newNode at end.
26     {
27         // Initialize nodePtr to head of list.
28         nodePtr = head;
29
30         // Find the last node in the list.
31         while (nodePtr->next)
32             nodePtr = nodePtr->next;
33
34         // Insert newNode as the last node.
35         nodePtr->next = newNode;
36     }
37 }
```

Let's examine the statements in detail. In lines 13 and 14, the function defines the following local variables:

```

ListNode *newNode; // To point to a new node
ListNode *nodePtr; // To move through the list
```

The `newNode` pointer will be used to allocate and point to the new node. The `nodePtr` pointer will be used to travel down the linked list, in search of the last node.

The following statements, in lines 17 through 19, create a new node and store `num` in its `value` member:

```

newNode = new ListNode;
newNode->value = num;
newNode->next = nullptr;
```

The statement in line 19 is important. Because this node will become the last node in the list, its `next` pointer must be a null pointer.

In line 23, we test the `head` pointer to determine whether there are any nodes already in the list. If `head` points to `nullptr`, we make the new node the first in the list. Making `head` point to the new node does this. Here is the code:

```

if (!head)
    head = newNode;
```

If `head` does not point to `nullptr`, however, there are nodes in the list. The `else` part of the `if` statement must contain code to find the end of the list and insert the new node. The code, in lines 25 through 36, is shown here:

```

else
{
    // Initialize nodePtr to head of list.
    nodePtr = head;

    // Find the last node in the list.
    while (nodePtr->next)
        nodePtr = nodePtr->next;

    // Insert newNode as the last node.
    nodePtr->next = newNode;
}
```

The code uses `nodePtr` to travel down the list. It does this by first assigning `nodePtr` to `head` in line 28.

```
nodePtr = head;
```

The `while` loop in lines 31 and 32 is then used to *traverse* (or travel through) the list searching for the last node. The last node will be the one whose `next` member points to `nullptr`.

```
while (nodePtr->next)
    nodePtr = nodePtr->next;
```

When `nodePtr` points to the last node in the list, we make its `next` member point to `newNode` in line 35 with the following statement:

```
nodePtr->next = newNode;
```

This inserts `newNode` at the end of the list. (Remember, `newNode->next` already points to `nullptr`.)

Program 18-1 demonstrates the function.

Program 18-1

```
1 // This program demonstrates a simple append
2 // operation on a linked list.
3 #include <iostream>
4 #include "NumberList.h"
5 using namespace std;
6
7 int main()
8 {
9     // Define a NumberList object.
10    NumberList list;
11
12    // Append some values to the list.
13    list.appendNode(2.5);
14    list.appendNode(7.9);
15    list.appendNode(12.6);
16    return 0;
17 }
```

(This program displays no output.)

Let's step through the program, observing how the `appendNode` function builds a linked list to store the three argument values used.

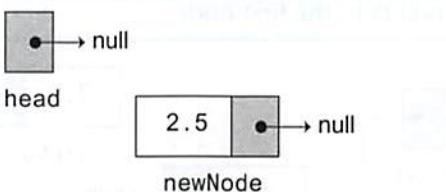
The `head` pointer is declared as a private member variable of the `NumberList` class. `head` is initialized to `nullptr` by the `NumberList` constructor, which indicates that the list is empty.

The first call to `appendNode` in line 13 passes 2.5 as the argument. In the following statements, a new node is allocated in memory, 2.5 is copied into its `value` member, and `nullptr` is assigned to the node's `next` pointer:

```
newNode = new ListNode;
newNode->value = num;
newNode->next = nullptr;
```

Figure 18-3 illustrates the state of the head pointer and the new node.

Figure 18-3 State of the head pointer and the new node

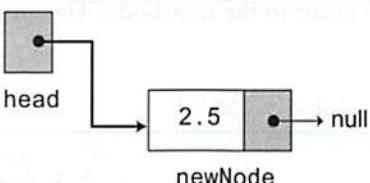


The next statement to execute is the following **if** statement:

```
if (!head)
    head = newNode;
```

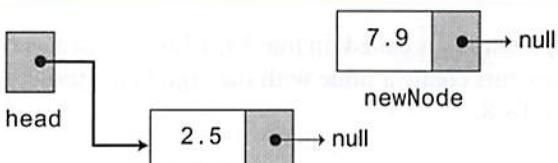
Because head is a null pointer, the condition `!head` is true. The statement `head = newNode;` is executed, making `newNode` the first node in the list. This is illustrated in Figure 18-4.

Figure 18-4 head points to newNode



There are no more statements to execute, so control returns to function `main`. In the second call to `appendNode`, in line 14, 7.9 is passed as the argument. Once again, the first three statements in the function create a new node, store the argument in the node's `value` member, and assign its `next` pointer to `nullptr`. Figure 18-5 illustrates the current state of the list and the new node.

Figure 18-5 A new node is created



Because head no longer points to `nullptr`, the `else` part of the `if` statement executes:

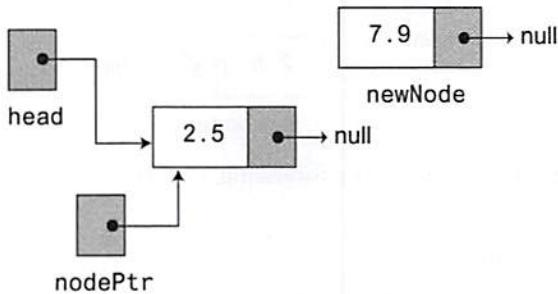
```
else // Otherwise, insert newNode at end.
{
    // Initialize nodePtr to head of list.
    nodePtr = head;

    // Find the last node in the list.
    while (nodePtr->next)
        nodePtr = nodePtr->next;

    // Insert newNode as the last node.
    nodePtr->next = newNode;
}
```

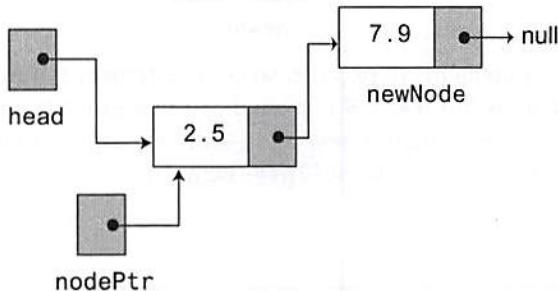
The first statement in the `else` block assigns the value in `head` to `nodePtr`. This causes `nodePtr` to point to the same node that `head` points to. This is illustrated in Figure 18-6.

Figure 18-6 `nodePtr` points to the first node



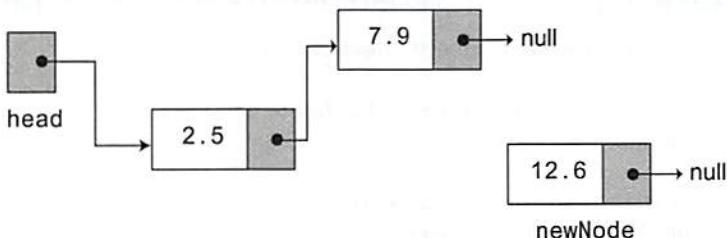
Look at the next member of the node that `nodePtr` points to. It is a null pointer, which means that `nodePtr->next` is also a null pointer. `nodePtr` is already at the end of the list, so the `while` loop immediately terminates. The last statement, `nodePtr->next = newNode;` causes `nodePtr->next` to point to the new node. This inserts `newNode` at the end of the list as shown in Figure 18-7.

Figure 18-7 The new node is added to the list

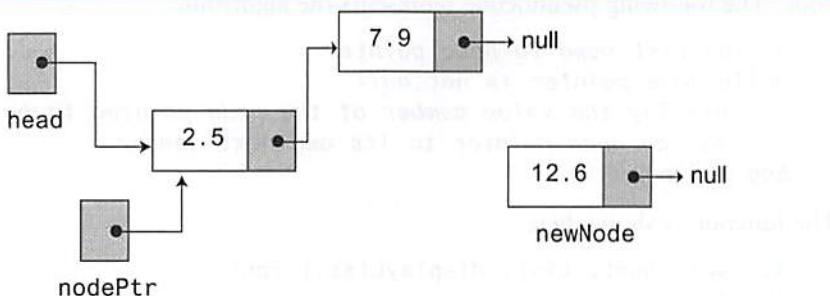


The third time `appendNode` is called, in line 15, 12.6 is passed as the argument. Once again, the first three statements create a node with the argument stored in the `value` member. This is shown in Figure 18-8.

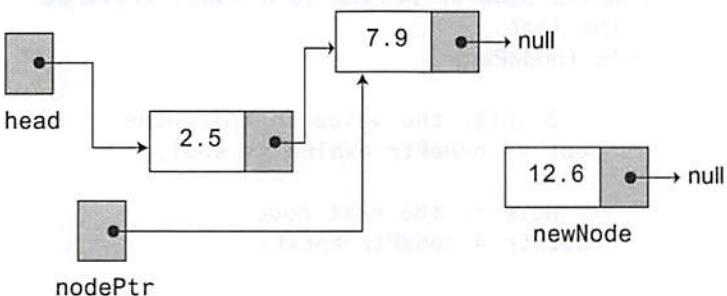
Figure 18-8 A new node is created



Next, the `else` part of the `if` statement executes. As before, `nodePtr` is made to point to the same node as `head`, as shown in Figure 18-9.

Figure 18-9 nodePtr points to the first node

Because `nodePtr->next` is not a null pointer, the `while` loop will execute. After its first iteration, `nodePtr` will point to the second node in the list. This is shown in Figure 18-10.

Figure 18-10 nodePtr points to the second node

The `while` loop's conditional test will fail after the first iteration because `nodePtr->next` is now a null pointer. The last statement, `nodePtr->next = newNode;` causes `nodePtr->next` to point to the new node. This inserts `newNode` at the end of the list as shown in Figure 18-11.

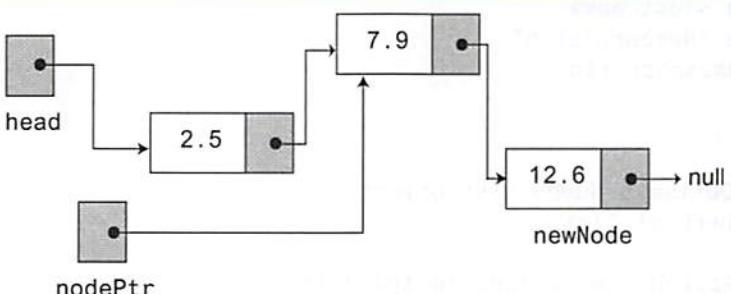
Figure 18-11 The new node is added to the list

Figure 18-11 depicts the final state of the linked list.

Traversing a Linked List

The `appendNode` function demonstrated in the previous section contains a `while` loop that traverses, or travels through the linked list. In this section, we will demonstrate the

`displayList` member function that traverses the list, displaying the `value` member of each node. The following pseudocode represents the algorithm:

```

Assign List head to node pointer.
While node pointer is not null
    Display the value member of the node pointed to by node pointer.
    Assign node pointer to its own next member.
End While.
```

The function is shown here:

```

45 void NumberList::displayList() const
46 {
47     ListNode *nodePtr; // To move through the list
48
49     // Position nodePtr at the head of the list.
50     nodePtr = head;
51
52     // While nodePtr points to a node, traverse
53     // the list.
54     while (nodePtr)
55     {
56         // Display the value in this node.
57         cout << nodePtr->value << endl;
58
59         // Move to the next node.
60         nodePtr = nodePtr->next;
61     }
62 }
```

Program 18-2, a modification of Program 18-1, demonstrates the function.

Program 18-2

```

1 // This program demonstrates the displayList member function.
2 #include <iostream>
3 #include "NumberList.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define a NumberList object.
9     NumberList list;
10
11    // Append some values to the list.
12    list.appendNode(2.5);
13    list.appendNode(7.9);
14    list.appendNode(12.6);
15
16    // Display the values in the list.
17    list.displayList();
18    return 0;
19 }
```

Program Output

```
2.5
7.9
12.6
```

Usually, when an operation is to be performed on some or all the nodes in a linked list, a traversal algorithm is used. You will see variations of this algorithm throughout this chapter.

Inserting a Node

Appending a node is a straightforward procedure. Inserting a node in the middle of a list, however, is more involved. For example, suppose the values in a list are sorted, and you wish all new values to be inserted in their proper position. This will preserve the order of the list. Using the `ListNode` structure again, the following pseudocode shows an algorithm for finding a new node's proper position in the list and inserting it there. The algorithm assumes the nodes in the list are already in order.



```
Create a new node.
Store data in the new node.
If there are no nodes in the list
    Make the new node the first node.
Else
    Find the first node whose value is greater than or equal to the new
        value, or the end of the list (whichever is first).
    Insert the new node before the found node, or at the end of the list
        if no such node was found.
End If.
```

Notice the new algorithm finds the first node whose value is greater than or equal to the new value. The new node is then inserted before the found node. This will require the use of two node pointers during the traversal: one to point to the node being inspected, and another to point to the previous node. The code for the traversal algorithm is as follows. (As before, `num` holds the value being inserted into the list.)

```
// Position nodePtr at the head of list.
nodePtr = head;

// Initialize previousNode to nullptr.
previousNode = nullptr;

// Skip all nodes whose value is less than num.
while (nodePtr != nullptr && nodePtr->value < num)
{
    previousNode = nodePtr;
    nodePtr = nodePtr->next;
}
```

This code segment uses the `ListNode` pointers `nodePtr` and `previousNode`. `previousNode` always points to the node before the one pointed to by `nodePtr`. The entire `insertNode` function is shown here:

```

69 void NumberList::insertNode(double num)
70 {
71     ListNode *newNode;           // A new node
72     ListNode *nodePtr;          // To traverse the list
73     ListNode *previousNode = nullptr; // The previous node
74
75     // Allocate a new node and store num there.
76     newNode = new ListNode;
77     newNode->value = num;
78
79     // If there are no nodes in the list
80     // make newNode the first node
81     if (!head)
82     {
83         head = newNode;
84         newNode->next = nullptr;
85     }
86     else // Otherwise, insert newNode
87     {
88         // Position nodePtr at the head of list.
89         nodePtr = head;
90
91         // Initialize previousNode to nullptr.
92         previousNode = nullptr;
93
94         // Skip all nodes whose value is less than num.
95         while (nodePtr != nullptr && nodePtr->value < num)
96         {
97             previousNode = nodePtr;
98             nodePtr = nodePtr->next;
99         }
100
101        // If the new node is to be the 1st in the list,
102        // insert it before all other nodes.
103        if (previousNode == nullptr)
104        {
105            head = newNode;
106            newNode->next = nodePtr;
107        }
108        else // Otherwise insert after the previous node.
109        {
110            previousNode->next = newNode;
111            newNode->next = nodePtr;
112        }
113    }
114 }
```

Program 18-3 is a modification of Program 18-2. It uses the `insertNode` member function to insert a value in its ordered position in the list.

Program 18-3

```

1 // This program demonstrates the insertNode member function.
2 #include <iostream>
3 #include "NumberList.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define a NumberList object.
9     NumberList list;
10
11    // Build the list with some values.
12    list.appendNode(2.5);
13    list.appendNode(7.9);
14    list.appendNode(12.6);
15
16    // Insert a node in the middle of the list.
17    list.insertNode(10.5);
18
19    // Display the list
20    list.displayList();
21    return 0;
22 }
```

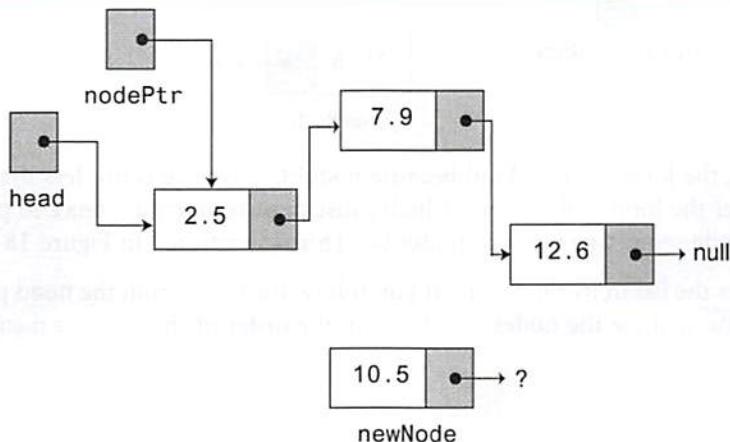
Program Output

2.5
7.9
10.5
12.6

Like Program 18-2, Program 18-3 calls the `appendNode` function three times to build the list with the values 2.5, 7.9, and 12.6. The `insertNode` function is then called, with the argument 10.5.

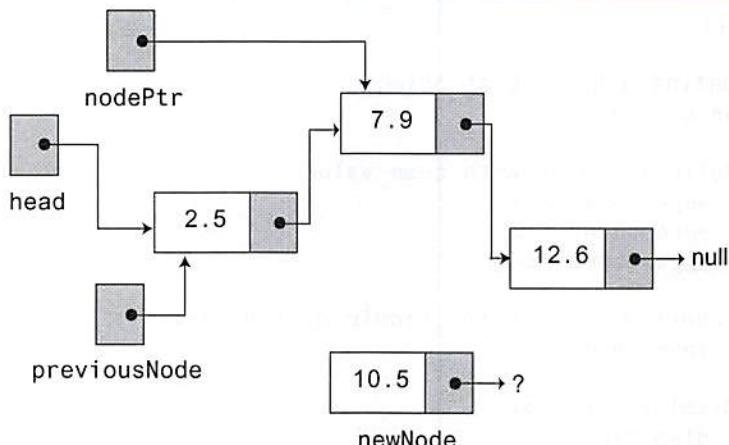
In `insertNode`, a new node is created, and the function argument is copied to its `value` member. Because the list already has nodes stored in it, the `else` part of the `if` statement will execute. It begins by assigning `nodePtr` to `head`. Figure 18-12 illustrates the state of the list at this point.

Figure 18-12 `nodePtr` points to the first node



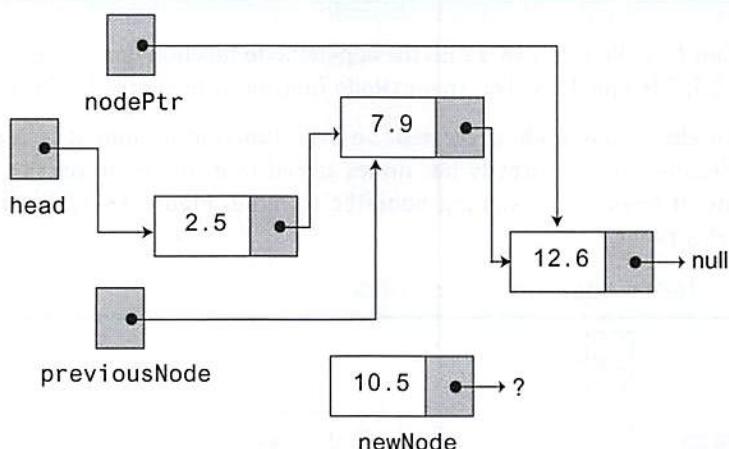
Because `nodePtr` is not a null pointer and `nodePtr->value` is less than `num`, the while loop will iterate. During the iteration, `previousNode` will be made to point to the node that `nodePtr` is pointing to. `nodePtr` will then be advanced to point to the next node. This is shown in Figure 18-13.

Figure 18-13 `previousNode` points to the first node and `nodePtr` points to the second node



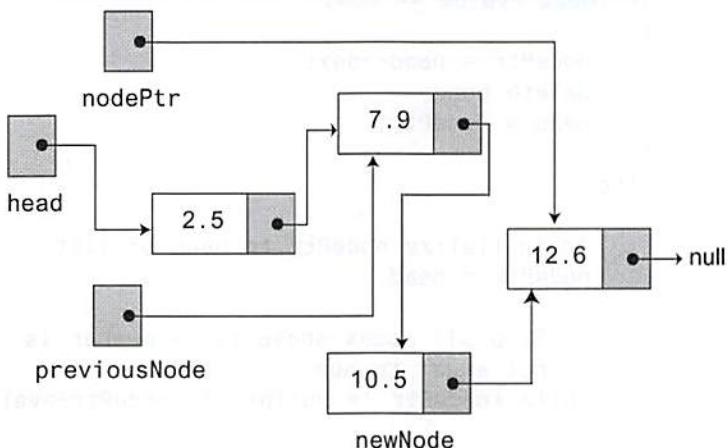
Once again, the loop performs its test. Because `nodePtr` is not a null pointer and `nodePtr->value` is less than `num`, the loop will iterate a second time. During the second iteration, both `previousNode` and `nodePtr` are advanced by one node in the list. This is shown in Figure 18-14.

Figure 18-14 `previousNode` points to the second node and `nodePtr` points to the third node



This time, the loop's test will fail because `nodePtr->value` is not less than `num`. The statements after the loop will execute, which cause `previousNode->next` to point to `newNode`, and `newNode->next` to point to `nodePtr`. This is illustrated in Figure 18-15.

This leaves the list in its final state. If you follow the links, from the `head` pointer to the `null` pointer, you will see the nodes are stored in the order of their `value` members.

Figure 18-15 The new node inserted

Checkpoint

- 18.5 What is the difference between appending a node to a list and inserting a node into a list?
- 18.6 Which is easier to code, appending or inserting?
- 18.7 Why does the `insertNode` function shown in this section use a `previousNode` pointer?

Deleting a Node

Deleting a node from a linked list requires two steps:

1. Remove the node from the list without breaking the links created by the `next` pointers.
2. Delete the node from memory.



The `deleteNode` member function searches for a node containing a particular value and deletes it from the list. It uses an algorithm similar to the `insertNode` function. Two node pointers, `nodePtr` and `previousNode`, are used to traverse the list. `previousNode` always points to the node whose position is just before the one pointed to by `nodePtr`. When `nodePtr` points to the node that is to be deleted, `previousNode->next` is made to point to `nodePtr->next`. This removes the node pointed to by `nodePtr` from the list. The final step performed by this function is to free the memory used by the node with the `delete` operator. The entire function is shown below:

```

122 void NumberList::deleteNode(double num)
123 {
124     ListNode *nodePtr;      // To traverse the list
125     ListNode *previousNode; // To point to the previous node
126
127     // If the list is empty, do nothing.
128     if (!head)
129         return;
130
  
```

```

131     // Determine if the first node is the one.
132     if (head->value == num)
133     {
134         nodePtr = head->next;
135         delete head;
136         head = nodePtr;
137     }
138     else
139     {
140         // Initialize nodePtr to head of list
141         nodePtr = head;
142
143         // Skip all nodes whose value member is
144         // not equal to num.
145         while (nodePtr != nullptr && nodePtr->value != num)
146         {
147             previousNode = nodePtr;
148             nodePtr = nodePtr->next;
149         }
150
151         // If nodePtr is not at the end of the list,
152         // link the previous node to the node after
153         // nodePtr, then delete nodePtr.
154         if (nodePtr)
155         {
156             previousNode->next = nodePtr->next;
157             delete nodePtr;
158         }
159     }
160 }
```

Program 18-4 demonstrates the function by first building a list of three nodes, then deleting them one by one.

Program 18-4

```

1 // This program demonstrates the deleteNode member function.
2 #include <iostream>
3 #include "NumberList.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define a NumberList object.
9     NumberList list;
10
11    // Build the list with some values.
12    list.appendNode(2.5);
13    list.appendNode(7.9);
14    list.appendNode(12.6);
15 }
```

```
16     // Display the list.  
17     cout << "Here are the initial values:\n";  
18     list.displayList();  
19     cout << endl;  
20  
21     // Delete the middle node.  
22     cout << "Now deleting the node in the middle.\n";  
23     list.deleteNode(7.9);  
24  
25     // Display the list.  
26     cout << "Here are the nodes left.\n";  
27     list.displayList();  
28     cout << endl;  
29  
30     // Delete the last node.  
31     cout << "Now deleting the last node.\n";  
32     list.deleteNode(12.6);  
33  
34     // Display the list.  
35     cout << "Here are the nodes left.\n";  
36     list.displayList();  
37     cout << endl;  
38  
39     // Delete the only node left in the list.  
40     cout << "Now deleting the only remaining node.\n";  
41     list.deleteNode(2.5);  
42  
43     // Display the list.  
44     cout << "Here are the nodes left.\n";  
45     list.displayList();  
46     return 0;  
47 }
```

Program Output

Here are the initial values:

2.5

7.9

12.6

Now deleting the node in the middle.

Here are the nodes left.

2.5

12.6

Now deleting the last node.

Here are the nodes left.

2.5

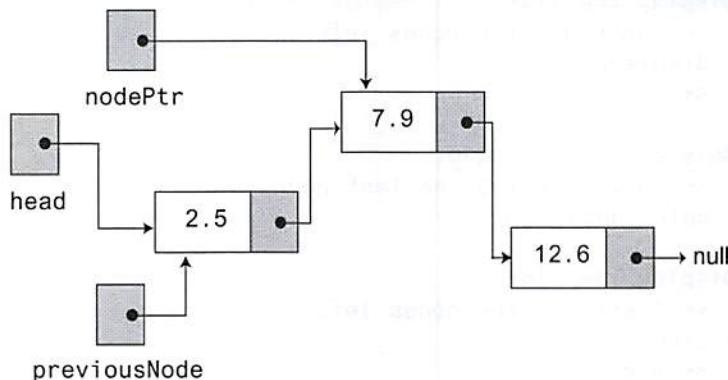
Now deleting the only remaining node.

Here are the nodes left.

To illustrate how `deleteNode` works, we will step through the first call, which deletes the node containing 7.9 as its value. This node is in the middle of the list.

In the `deleteNode` function, look at the `else` part of the second `if` statement. This is lines 138 through 159. This is where the function will perform its action since the list is not empty, and the first node does not contain the value 7.9. Just like `insertNode`, this function uses `nodePtr` and `previousNode` to traverse the list. The `while` loop in lines 145 through 149 terminates when the value 7.9 is located. At this point, the list and the other pointers will be in the state depicted in Figure 18-16.

Figure 18-16 `previousNode` points to the first node and `nodePtr` points to the second node

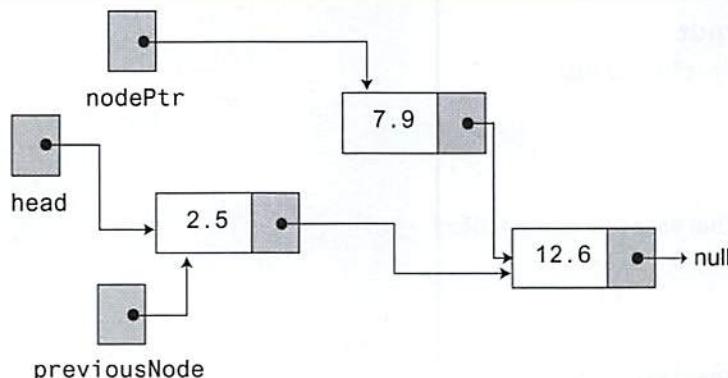


Next, the following statement in line 156 executes:

```
previousNode->next = nodePtr->next;
```

This statement causes the links in the list to bypass the node to which `nodePtr` points. Although the node still exists in memory, this removes it from the list, as illustrated in Figure 18-17.

Figure 18-17 Node removed from the list



The statement in line 157 uses the `delete` operator to complete the total deletion of the node.

Destroying the List

It's important for the class's destructor to release all the memory used by the list. It does so by stepping through the list, deleting one node at a time. The code is shown here:

```

167 NumberList::~NumberList()
168 {
169     ListNode *nodePtr; // To traverse the list
170     ListNode *nextNode; // To point to the next node
171
172     // Position nodePtr at the head of the list.
173     nodePtr = head;
174
175     // While nodePtr is not at the end of the list...
176     while (nodePtr != nullptr)
177     {
178         // Save a pointer to the next node.
179         nextNode = nodePtr->next;
180
181         // Delete the current node.
182         delete nodePtr;
183
184         // Position nodePtr at the next node.
185         nodePtr = nextNode;
186     }
187 }

```

Notice the use of `nextNode` instead of `previousNode`. The `nextNode` pointer is used to hold the position of the next node in the list so it will be available after the node pointed to by `nodePtr` is deleted.



Checkpoint

- 18.8 What are the two steps involved in deleting a node from a linked list?
- 18.9 When deleting a node, why can't you just use the `delete` operator to remove it from memory? Why must you take the steps you listed in response to Question 18.8?
- 18.10 In a program that uses several linked lists, what might eventually happen if the class destructor does not destroy its linked list?

18.3

A Linked List Template

CONCEPT: A template can be easily created to store linked lists of any type.

The limitation of the `NumberList` class is that it can only hold `double` values. The class can easily be converted to a template that will accept any data type, as shown in the following code. (This file can be found in the Student Source Code Folder Chapter 18\LinkedListTemplate Version 1.)

Contents of `LinkedList.h` (Version 1)

```

1 // A class template for holding a linked list.
2 #ifndef LINKEDLIST_H
3 #define LINKEDLIST_H
4 #include <iostream> // For cout
5 using namespace std;
6

```

```
7  template <class T>
8  class LinkedList
9  {
10 private:
11     // Declare a structure for the list.
12     struct ListNode
13     {
14         T value;           // The value in this node
15         struct ListNode *next; // To point to the next node
16     };
17
18     ListNode *head; // List head pointer
19
20 public:
21     // Constructor
22     NumberList()
23     { head = nullptr; }
24
25     // Destructor
26     ~NumberList();
27
28     // Linked list operations
29     void appendNode(T);
30     void insertNode(T);
31     void deleteNode(T);
32     void displayList() const;
33 };
34
35
36 //*****
37 // appendNode appends a node containing the value *
38 // passed into newValue, to the end of the list. *
39 //*****
40
41 template <class T>
42 void LinkedList<T>::appendNode(T newValue)
43 {
44     ListNode *newNode; // To point to a new node
45     ListNode *nodePtr; // To move through the list
46
47     // Allocate a new node and store num there.
48     newNode = new ListNode;
49     newNode->value = num;
50     newNode->next = nullptr;
51
52     // If there are no nodes in the list
53     // make newNode the first node.
54     if (!head)
55         head = newNode;
56     else // Otherwise, insert newNode at end.
57     {
58         // Initialize nodePtr to head of list.
59         nodePtr = head;
60     }
```

```
61         // Find the last node in the list.
62         while (nodePtr->next)
63             nodePtr = nodePtr->next;
64
65         // Insert newNode as the last node.
66         nodePtr->next = newNode;
67     }
68 }
69
70 //***** *****
71 // displayList shows the value
72 // stored in each node of the linked list
73 // pointed to by head.
74 //*****
75
76 template <class T>
77 void LinkedList<T>::displayList()
78 {
79     ListNode *nodePtr; // To move through the list
80
81     // Position nodePtr at the head of the list.
82     nodePtr = head;
83
84     // While nodePtr points to a node, traverse
85     // the list.
86     while (nodePtr)
87     {
88         // Display the value in this node.
89         cout << nodePtr->value << endl;
90
91         // Move to the next node.
92         nodePtr = nodePtr->next;
93     }
94 }
95
96 //*****
97 // The insertNode function inserts a node with
98 // newValue copied to its value member.
99 //*****
100
101 template <class T>
102 void LinkedList<T>::insertNode(T newValue)
103 {
104     ListNode *newNode;           // A new node
105     ListNode *nodePtr;          // To traverse the list
106     ListNode *previousNode = nullptr; // The previous node
107
108     // Allocate a new node and store num there.
109     newNode = new ListNode;
110     newNode->value = num;
111
112     // If there are no nodes in the list
113     // make newNode the first node.
```

```

114     if (!head)
115     {
116         head = newNode;
117         newNode->next = nullptr;
118         previousNode = newNode;
119     }
120     else // Otherwise, insert newNode.
121     {
122         nodePtr = head;
123         // Postition nodePtr at the head of list.
124         // Initialize previousNode to nullptr.
125         previousNode = nullptr;
126         while (nodePtr != nullptr && nodePtr->value < num)
127             // Skip all nodes whose value is less than num.
128         {
129             if (nodePtr->value == num)
130                 previousNode = nodePtr;
131             nodePtr = nodePtr->next;
132         }
133         // If the new node is to be the last in the list,
134         // insert it before all other nodes.
135         if (previousNode == nullptr)
136             // insert it before all other nodes.
137             {
138                 head = newNode;
139                 newNode->next = nodePtr;
140             }
141         else // Otherwise insert after the previous node.
142             {
143                 previousNode->next = newNode;
144                 newNode->next = nodePtr;
145             }
146         }
147     }
148 }
149 //*****
150 // The deleteNode function searches for a node
151 // with searchValue as its value. The node, if found,
152 // is deleted from the list and from memory.
153 // ****
154 template <class T>
155 void LinkedList<T>::deleteNode(T searchValue)
156 {
157     ListNode *nodePtr; // To traverse the list
158     ListNode *nodePtr; // To point to the previous node
159     ListNode *previousNode; // To point to the previous node
160     // If the list is empty, do nothing.
161     // If the list is empty, do nothing.
162     if (!head)
163     {
164         return;
165     }
166     if (head->value == num)
167     {
168         // Determine if the first node is the one.
169     }

```

```
167     {
168         nodePtr = head->next;
169         delete head;
170         head = nodePtr;
171     }
172 else
173 {
174     // Initialize nodePtr to head of list.
175     nodePtr = head;
176
177     // Skip all nodes whose value member is
178     // not equal to num.
179     while (nodePtr != nullptr && nodePtr->value != num)
180     {
181         previousNode = nodePtr;
182         nodePtr = nodePtr->next;
183     }
184
185     // If nodePtr is not at the end of the list,
186     // link the previous node to the node after
187     // nodePtr, then delete nodePtr.
188     if (nodePtr)
189     {
190         previousNode->next = nodePtr->next;
191         delete nodePtr;
192     }
193 }
194 }
195
196 //*****
197 // Destructor
198 // This function deletes every node in the list.
199 //*****
200
201 template <class T>
202 LinkedList<T>::~LinkedList()
203 {
204     ListNode *nodePtr;    // To traverse the list
205     ListNode *nextNode;  // To point to the next node
206
207     // Position nodePtr at the head of the list.
208     nodePtr = head;
209
210     // While nodePtr is not at the end of the list...
211     while (nodePtr != nullptr)
212     {
213         // Save a pointer to the next node.
214         nextNode = nodePtr->next;
215
216         // Delete the current node.
217         delete nodePtr;
218 }
```

```

219         // Position nodePtr at the next node.
220         nodePtr = nextNode;
221     }
222 }
223 #endif

```

Note the template uses the ==, !=, and < relational operators to compare node values, and it uses the << operator with cout to display node values. Any type passed to the template must support these operators.

Now let's see how the template can be used to create a list of objects. Recall the FeetInches class introduced in Chapter 14. That class overloaded numerous operators, including ==, <, and <<. In the Chapter 18\LinkedList Template Version 1 folder, we have included a modified version of the FeetInches class that also overloads the != operator. Program 18-5 is stored in that same folder. This program uses the LinkedList template to create a linked list of FeetInches objects.

Program 18-5

```

1 // This program demonstrates the linked list template.
2 #include <iostream>
3 #include "LinkedList.h"
4 #include "FeetInches.h"
5 using namespace std;
6
7 int main()
8 {
9     // Define a LinkedList object.
10    LinkedList<FeetInches> list;
11
12    // Define some FeetInches objects.
13    FeetInches distance1(5, 4); // 5 feet 4 inches
14    FeetInches distance2(6, 8); // 6 feet 8 inches
15    FeetInches distance3(8, 9); // 8 feet 9 inches
16
17    // Store the FeetInches objects in the list.
18    list.appendNode(distance1); // 5 feet 4 inches
19    list.appendNode(distance2); // 6 feet 8 inches
20    list.appendNode(distance3); // 8 feet 9 inches
21
22    // Display the values in the list.
23    cout << "Here are the initial values:\n";
24    list.displayList();
25    cout << endl;
26
27    // Insert another FeetInches object.
28    cout << "Now inserting the value 7 feet 2 inches.\n";
29    FeetInches distance4(7, 2);
30    list.insertNode(distance4);
31
32    // Display the values in the list.
33    cout << "Here are the nodes now.\n";
34    list.displayList();
35    cout << endl;

```

```

36     // Delete the last node.
37     cout << "Now deleting the last node.\n";
38     FeetInches distance5(8, 9);
39     list.deleteNode(distance5);
40
41     // Display the values in the list.
42     cout << "Here are the nodes left.\n";
43     list.displayList();
44
45     return 0;
46 }

```

Program Output

Here are the initial values:

5 feet, 4 inches
6 feet, 8 inches
8 feet, 9 inches

Now inserting the value 7 feet 2 inches.

Here are the nodes now.

5 feet, 4 inches
6 feet, 8 inches
7 feet, 2 inches
8 feet, 9 inches

Now deleting the last node.

Here are the nodes left.

5 feet, 4 inches
6 feet, 8 inches
7 feet, 2 inches

Using a Class Node Type

In the `LinkedList` class template, the following structure was used to create a data type for the linked list node:

```

struct ListNode
{
    T value;
    struct ListNode *next;
};

```

Another approach is to use a separate class template to create a data type for the node. Then, the class constructor can be used to store an item in the `value` member and set the `next` pointer to `nullptr`. Here is an example:

```

template <class T>
class ListNode
{
public:
    T value;           // Node value
    ListNode<T> *next; // Pointer to the next node
};

```

```

    // Constructor
    ListNode (T nodeValue)
    {
        value = nodeValue;
        next = nullptr;
    };
}

```

The `LinkedList` class template can then be written as the following:

```

template <class T>
class LinkedList
{
private:
    ListNode<T> *head; // List head pointer
public:
    // Constructor
    LinkedList()
        { head = nullptr; }

    // Destructor
    ~LinkedList();

    // Linked list operations
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList() const;
};

```

Because the `ListNode` class constructor assigns a value to the `value` member and sets the `next` pointer to `nullptr`, some of the code in the `LinkedList` class can be simplified. For example, the following code appears in the previous version of the `LinkedList` class template's `appendNode` function:

```

newNode = new ListNode;
newNode->value = newValue;
newNode->next = nullptr;

```

By using the `ListNode` class template with its constructor, these three lines of code can be reduced to one:

```

newNode = new ListNode<T>(newValue);

```

(This file can be found in the Student Source Code Folder Chapter 18\LinkedList Template Version 2.)

Contents of `LinkedList.h` (Version 2)

```

1 // A class template for holding a linked list.
2 // The node type is also a class template.
3 #ifndef LINKEDLIST_H
4 #define LINKEDLIST_H
5
6 //*****
7 // The ListNode class creates a type used to *
8 // store a node of the linked list.
9 //*****
10
11 template <class T>

```

```
12 class ListNode
13 {
14 public:
15     T value;           // Node value
16     ListNode<T> *next; // Pointer to the next node
17
18     // Constructor
19     ListNode (T nodeValue)
20     { value = nodeValue;
21         next = nullptr; }
22 };
23
24 //*****
25 // LinkedList class
26 //*****
27
28 template <class T>
29 class LinkedList
30 {
31 private:
32     ListNode<T> *head; // List head pointer
33
34 public:
35     // Constructor
36     LinkedList()
37     { head = nullptr; }
38
39     // Destructor
40     ~LinkedList();
41
42     // Linked list operations
43     void appendNode(T);
44     void insertNode(T);
45     void deleteNode(T);
46     void displayList() const;
47 };
48
49
50 //*****
51 // appendNode appends a node containing the value *
52 // passed into newValue, to the end of the list. *
53 //*****
54
55 template <class T>
56 void LinkedList<T>::appendNode(T newValue)
57 {
58     ListNode<T> *newNode; // To point to a new node
59     ListNode<T> *nodePtr; // To move through the list
60
61     // Allocate a new node and store newValue there.
62     newNode = new ListNode<T>(newValue);
63
64     // If there are no nodes in the list
65     // make newNode the first node.
```

```
66      if (!head)
67          head = newNode;
68      else // Otherwise, insert newNode at end.
69      {
70          // Initialize nodePtr to head of list.
71          nodePtr = head;
72
73          // Find the last node in the list.
74          while (nodePtr->next)
75              nodePtr = nodePtr->next;
76
77          // Insert newNode as the last node.
78          nodePtr->next = newNode;
79      }
80  }
81
82 //*****
83 // displayList shows the value stored in each node *
84 // of the linked list pointed to by head. *
85 //*****
86
87 template <class T>
88 void LinkedList<T>::displayList() const
89 {
90     ListNode<T> *nodePtr; // To move through the list
91
92     // Position nodePtr at the head of the list.
93     nodePtr = head;
94
95     // While nodePtr points to a node, traverse
96     // the list.
97     while (nodePtr)
98     {
99         // Display the value in this node.
100        cout << nodePtr->value << endl;
101
102        // Move to the next node.
103        nodePtr = nodePtr->next;
104    }
105 }
106
107 //*****
108 // The insertNode function inserts a node with      *
109 // newValue copied to its value member.           *
110 //*****
111
112 template <class T>
113 void LinkedList<T>::insertNode(T newValue)
114 {
115     ListNode<T> *newNode;                      // A new node
116     ListNode<T> *nodePtr;                      // To traverse the list
117     ListNode<T> *previousNode = nullptr; // The previous node
118 }
```

```
119     // Allocate a new node and store newValue there.
120     newNode = new ListNode<T>(newValue);
121
122     // If there are no nodes in the list
123     // make newNode the first node.
124     if (!head)
125     {
126         head = newNode;
127         newNode->next = nullptr;
128     }
129     else // Otherwise, insert newNode.
130     {
131         // Position nodePtr at the head of list.
132         nodePtr = head;
133
134         // Initialize previousNode to nullptr.
135         previousNode = nullptr;
136
137         // Skip all nodes whose value is less than newValue.
138         while (nodePtr != nullptr && nodePtr->value < newValue)
139         {
140             previousNode = nodePtr;
141             nodePtr = nodePtr->next;
142         }
143
144         // If the new node is to be the 1st in the list,
145         // insert it before all other nodes.
146         if (previousNode == nullptr)
147         {
148             head = newNode;
149             newNode->next = nodePtr;
150         }
151         else // Otherwise insert after the previous node.
152         {
153             previousNode->next = newNode;
154             newNode->next = nodePtr;
155         }
156     }
157 }
158
159 //*****
160 // The deleteNode function searches for a node
161 // with searchValue as its value. The node, if found,
162 // is deleted from the list and from memory.
163 //*****
164
165 template <class T>
166 void LinkedList<T>::deleteNode(T searchValue)
167 {
168     ListNode<T> *nodePtr; // To traverse the list
169     ListNode<T> *previousNode; // To point to the previous node
170 }
```

```
171     // If the list is empty, do nothing.
172     if (!head)
173         return;
174
175     // Determine if the first node is the one.
176     if (head->value == searchValue)
177     {
178         nodePtr = head->next;
179         delete head;
180         head = nodePtr;
181     }
182     else
183     {
184         // Initialize nodePtr to head of list
185         nodePtr = head;
186
187         // Skip all nodes whose value member is
188         // not equal to num.
189         while (nodePtr != nullptr && nodePtr->value != searchValue)
190         {
191             previousNode = nodePtr;
192             nodePtr = nodePtr->next;
193         }
194
195         // If nodePtr is not at the end of the list,
196         // link the previous node to the node after
197         // nodePtr, then delete nodePtr.
198         if (nodePtr)
199         {
200             previousNode->next = nodePtr->next;
201             delete nodePtr;
202         }
203     }
204 }
205 ****
206 // Destructor
207 // This function deletes every node in the list.
208 // ****
209
210
211 template <class T>
212 LinkedList<T>::~LinkedList()
213 {
214     ListNode<T> *nodePtr; // To traverse the list
215     ListNode<T> *nextNode; // To point to the next node
216
217     // Position nodePtr at the head of the list.
218     nodePtr = head;
219
220     // While nodePtr is not at the end of the list...
221     while (nodePtr != nullptr)
222     {
223         // Save a pointer to the next node.
224         nextNode = nodePtr->next;
```

```

225
226     // Delete the current node.
227     delete nodePtr;
228
229     // Position nodePtr at the next node.
230     nodePtr = nextNode;
231 }
232 }
233#endif

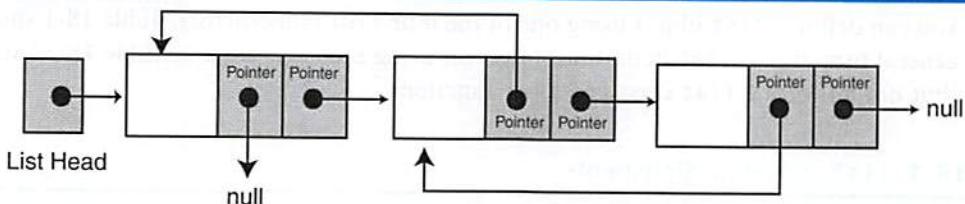
```

18.4 Variations of the Linked List

CONCEPT: There are many ways to link dynamically allocated data structures together. Two variations of the linked list are the doubly linked list and the circularly linked list.

The linked list examples we have discussed are considered *singly linked lists*. Each node is linked to a single other node. A variation of this is the *doubly linked list*. In this type of list, each node points not only to the next node, but also to the previous one. This is illustrated in Figure 18-18.

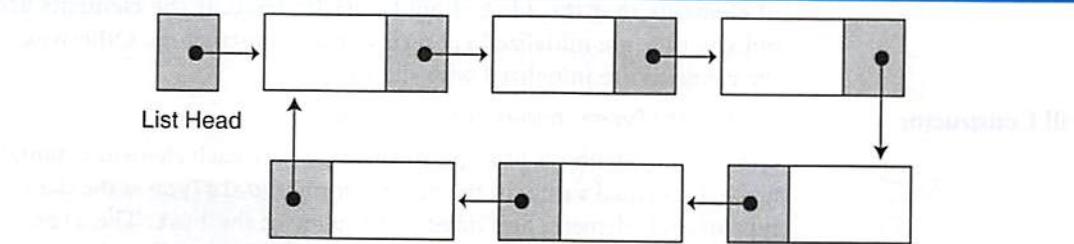
Figure 18-18 A doubly linked list



In Figure 18-18, the last node and the first node in the list have pointers to the null address. When the program traverses the list, it knows when it has reached either end.

Another variation is the *circularly linked list*. The last node in this type of list points to the first, as shown in Figure 18-19.

Figure 18-19 A circularly linked list



18.5 The STL `list` and `forward_list` Containers

CONCEPT: The Standard Template Library provides two containers that are implemented as linked lists. The `list` container is a doubly linked list, and the `forward_list` container is a singly linked list.

The `list` container, found in the Standard Template Library, is implemented as a doubly linked list, and the `forward_list` container is implemented as a singly linked list. These containers can perform insertion and emplacement operations quicker than vectors because linked lists do not have to shift their existing elements in memory when a new element is added. The `list` and `forward_list` containers are also efficient at adding elements at their back because they have a built-in pointer to the last element in the list (no traversal required).

A disadvantage of the `list` and `forward_list` containers, compared to other sequence containers, is that you cannot use an index to access a list element. If you want to access an element that is in the middle of a `list` or `forward_list`, you have to step through each element until you reach the one you are looking for.

The `list` Container

To use the `list` class, you need to `#include` the `<list>` header file in your program. Then, you can define a `list` object using one of the four `list` constructors. Table 18-1 shows the general format of a `list` definition statement using each constructor. Table 18-2 lists many (but not all) of the `list` class's member functions.

Table 18-1 `list` Definition Statements

Default Constructor	<code>list<dataType> name;</code>
	Creates an empty <code>list</code> object. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the <code>list</code> .
Fill Constructor	<code>list<dataType> name(size);</code>
	Creates a <code>list</code> object of a specified size. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the <code>list</code> . The <code>size</code> argument is an unsigned integer that specifies the number of elements that the <code>list</code> should initially have. If the elements are objects, they are initialized via their default constructors. Otherwise, the elements are initialized with the value 0.
Fill Constructor	<code>list<dataType> name(size, value);</code>
	Creates a <code>list</code> object of a specified size, where each element is initially given a specified value. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the <code>list</code> . The <code>size</code> argument is an unsigned integer that specifies the number of elements that the <code>list</code> should initially have, and the <code>value</code> argument is the value to fill each element with.

Table 18-1 (continued)

Range Constructor	<code>list<dataType> name(iterator1, iterator2);</code>
	Creates a <code>list</code> object that initially contains a range of values specified by two iterators. In the general format, <code>dataType</code> is the data type of each element, and <code>name</code> is the name of the <code>list</code> . The <code>iterator1</code> and <code>iterator2</code> arguments mark the beginning and end of a range of values that will be stored in the <code>list</code> .
Copy Constructor	<code>list<dataType> name(list2);</code>
	Creates a <code>list</code> object that is a copy of another <code>list</code> object. In the general format, <code>dataType</code> is the data type of each element, <code>name</code> is the name of the <code>list</code> , and <code>list2</code> is the <code>list</code> to copy.

Table 18-2 Many of the `list` Member Functions

Member Function	Description
<code>back()</code>	Returns a reference to the last element in the container.
<code>begin()</code>	Returns an iterator to the first element in the container.
<code>cbegin()</code>	Returns a <code>const_iterator</code> to the first element in the container.
<code>cend()</code>	Returns a <code>const_iterator</code> pointing to the end of the container.
<code>clear()</code>	Erases all of the elements in the container.
<code>crbegin()</code>	Returns a <code>const_reverse_iterator</code> pointing to the last element in the container.
<code>crend()</code>	Returns a <code>const_reverse_iterator</code> pointing to the first element in the container.
<code>emplace(it, args...)</code>	Constructs a new object as an element, passing <code>args...</code> as arguments to the element's constructor. The <code>it</code> argument is an iterator pointing to an existing element in the container. The new element will be inserted before the one pointed to by <code>it</code> .
<code>emplace_back(args...)</code>	Constructs a new object as an element at the end of the container. The <code>args...</code> arguments are passed to the element's constructor.
<code>emplace_front(args...)</code>	Constructs a new object as an element at the front of the container. The <code>args...</code> arguments are passed to the element's constructor.
<code>empty()</code>	Returns <code>true</code> if the container is empty, or <code>false</code> otherwise.
<code>end()</code>	Returns an iterator pointing to the end of the container.
<code>erase(it)</code>	Erases the element pointed to by the iterator <code>it</code> . This function returns an iterator pointing to the element that follows the removed element (or the end of the container, if the removed element was the last one).
<code>erase(iterator1, iterator2)</code>	Erases a range of elements. The <code>iterator1</code> and <code>iterator2</code> arguments mark the beginning and end of a range of values that will be erased. This function returns an iterator pointing to the element that follows the removed elements (or the end of the container, if the last element was erased).

(table continues)

Table 18-2 (continued)

Member Function	Description
<code>front()</code>	Returns a reference to the first element in the container.
<code>insert(<i>it</i>, <i>value</i>)</code>	Inserts a new element with <i>value</i> as its value. The <i>it</i> argument is an iterator pointing to an existing element in the container. The new element will be inserted before the one pointed to by <i>it</i> . The function returns an iterator pointing to the newly inserted element.
<code>insert(<i>it</i>, <i>n</i>, <i>value</i>)</code>	Inserts <i>n</i> new elements with <i>value</i> as their value. The <i>it</i> argument is an iterator pointing to an existing element in the container, and <i>n</i> is an unsigned integer. The new elements will be inserted before the one pointed to by <i>it</i> . The function returns an iterator pointing to the first element of the newly inserted elements.
<code>insert(<i>iterator1</i>, <i>iterator2</i>, <i>iterator3</i>)</code>	Inserts a range of new elements. The <i>iterator1</i> argument points to an existing element in the container. The range of new elements will be inserted before the element pointed to by <i>iterator1</i> . The <i>iterator2</i> and <i>iterator3</i> arguments mark the beginning and end of a range of values that will be inserted. (The element pointed to by <i>iterator3</i> will not be included in the range.) The function returns an iterator pointing to the first element of the newly inserted range.
<code>max_size()</code>	Returns the theoretical maximum size of the container.
<code>merge(<i>second</i>)</code>	The <i>second</i> argument must be a <code>list</code> object of the same type as the calling object. The elements of both the calling <code>list</code> object and the <i>second</i> <code>list</code> object must be sorted prior to calling the <code>merge</code> member function. The function merges the contents of the calling object and the <i>second</i> object. The elements of the <i>second</i> object will be inserted at their ordered positions. After the function executes, the <i>second</i> <code>list</code> object will be empty.
<code>pop_back()</code>	Removes the last element of the container.
<code>pop_front()</code>	Removes the first element of the container.
<code>push_back(<i>value</i>)</code>	Adds a new element containing <i>value</i> to the end of the container.
<code>push_front(<i>value</i>)</code>	Adds a new element containing <i>value</i> to the beginning of the container.
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> pointing to the last element in the container.
<code>remove(<i>value</i>)</code>	Removes all elements with a value equal to <i>value</i> .
<code>remove_if(<i>function</i>)</code>	<i>function</i> is a unary predicate function (a function or function object that accepts one argument, and returns a Boolean value). This member function passes each element as an argument to <i>function</i> , and removes all elements that cause the function to return <code>true</code> .
<code>rend()</code>	Returns a <code>reverse_iterator</code> pointing to the first element in the container.

Table 18-2 (continued)

Member Function	Description
<code>resize(n)</code>	The <i>n</i> argument is an unsigned integer. This function resizes the container so it has <i>n</i> elements. If the current size of the container is larger than <i>n</i> , then the container is reduced in size so it keeps only the first <i>n</i> elements. If the current size of the container is smaller than <i>n</i> , then the container is increased in size so it has <i>n</i> elements.
<code>resize(n, value)</code>	Resizes the container so that it has <i>n</i> elements (the <i>n</i> argument is an unsigned integer). If the current size of the container is larger than <i>n</i> , then the container is reduced in size so it keeps only the first <i>n</i> elements. If the current size of the container is smaller than <i>n</i> , then the container is increased in size so it has <i>n</i> elements, and each of the new elements is initialized with <i>value</i> .
<code>reverse()</code>	Reverses the order of the elements in the container.
<code>size()</code>	Returns the number of elements in the container.
<code>sort()</code>	Sorts the elements in ascending order. Uses the < operator to compare elements.
<code>swap(second)</code>	The <i>second</i> argument must be a <code>list</code> object of the same type as the calling object. The function swaps the contents of the calling object and the <i>second</i> object.
<code>unique()</code>	If the container has any consecutive elements that are duplicates, all are removed except the first one.

Program 18-6 demonstrates some simple operations with the `list` container.

Program 18-6

```

1 // This program demonstrates the STL list container.
2 #include <iostream>
3 #include <list>
4 using namespace std;
5
6 int main()
7 {
8     // Define an empty list.
9     list<int> myList;
10
11    // Add some values to the list.
12    for (int x = 0; x < 100; x += 10)
13        myList.push_back(x);
14
15    // Use an iterator to display the values.
16    for (auto it = myList.begin(); it != myList.end(); it++)
17        cout << *it << " ";
18    cout << endl;
19

```

(program continues)

Program 18-6 (continued)

```

20     // Now reverse the order of the elements.
21     myList.reverse();
22
23     // Display the values again, with a range-based for loop
24     for (auto element : myList)
25         cout << element << " ";
26     cout << endl;
27
28     return 0;
29 }
```

Program Output

```
0 10 20 30 40 50 60 70 80 90
90 80 70 60 50 40 30 20 10 0
```

The forward_list Container

Because a `forward_list` is implemented as a singly linked list, each node keeps only one pointer: a pointer to the next node. Compared to a `list` container, which keeps two pointers per node (one to the next node, and one to the previous node), a `forward_list` uses less memory than a `list`. However, you can step only forward through a `forward_list`. If you need to move both forward and backward, you will need to use a `list` container.

The `forward_list` container provides most of the same member functions as the `list` container, with a few exceptions. You can find documentation for the `forward_list` container with a good online reference site, such as www.cppreference.com or www.cplusplus.com.

Review Questions and Exercises**Short Answer**

1. What are some of the advantages that linked lists have over arrays?
2. What advantage does a linked list have over the STL `vector`?
3. What is a list head?
4. What is a self-referential data structure?
5. How is the end of a linked list usually signified?
6. Name five basic linked list operations.
7. What is the difference between appending a node and inserting a node?
8. What does “traversing the list” mean?
9. What are the two steps required to delete a node from a linked list?
10. What is the advantage of using a template to implement a linked list?
11. What is a singly linked list? What is a doubly linked list? What is a circularly linked list?
12. What type of linked list is the STL `list` container? What type of linked list is the STL `forward_list` container?

Fill-in-the-Blank

13. The _____ points to the first node in a linked list.
14. A data structure that points to an object of the same type as itself is known as a(n) _____ data structure.
15. After creating a linked list's head pointer, you should make sure it points to _____ before using it in any operations.
16. _____ a node means adding it to the end of a list.
17. _____ a node means adding it to a list, but not necessarily to the end.
18. _____ a list means traveling through the list.
19. In a(n) _____ list, the last node has a pointer to the first node.
20. In a(n) _____ list, each node has a pointer to the one before it and the one after it.

Algorithm Workbench

21. Consider the following code:

```
struct ListNode
{
    int value;
    struct ListNode *next;
};

ListNode *head; // List head pointer
```

Assume a linked list has been created and `head` points to the first node. Write code that traverses the list displaying the contents of each node's `value` member.

22. Write code that destroys the linked list described in Question 21.
23. Write code that defines an STL `list` container for holding `float` values.
24. Write code that stores the values 12.7, 9.65, 8.72, and 4.69 in the `list` container you defined for Question 23.
25. Write code that reverses the order of the items you stored in the `list` container in Question 24.

True or False

26. T F The programmer must know in advance how many nodes will be needed in a linked list.
27. T F It is not necessary for each node in a linked list to have a self-referential pointer.
28. T F In physical memory, the nodes in a linked list may be scattered around.
29. T F When the head pointer points to `nullptr`, it signifies an empty list.
30. T F Linked lists are not superior to STL vectors.
31. T F Deleting a node in a linked list is a simple matter of using the `delete` operator to free the node's memory.
32. T F A class that builds a linked list should destroy the list in the class destructor.

Find the Errors

Each of the following member functions has errors in the way it performs a linked list operation. Find as many mistakes as you can.

```

33. void NumberList::appendNode(double num)
{
    ListNode *newNode, *nodePtr;
    // Allocate a new node & store num
    newNode = new ListNode;
    newNode->value = num;

    // If there are no nodes in the list
    // make newNode the first node.
    if (!head)
        head = newNode;
    else      // Otherwise, insert newNode.
    {
        // Find the last node in the list.
        while (nodePtr->next)
            nodePtr = nodePtr->next;
        // Insert newNode as the last node.
        nodePtr->next = newNode;
    }
}

34. void NumberList::deleteNode(double num)
{
    ListNode *nodePtr, *previousNode;
    // If the list is empty, do nothing.
    if (!head)
        return;
    // Determine if the first node is the one.

    if (head->value == num)
        delete head;
    else
    {
        // Initialize nodePtr to head of list.
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr->value != num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }
    }
}

```

```
        // Link the previous node to the node after
        // nodePtr, then delete nodePtr.
        previousNode->next = nodePtr->next;
        delete nodePtr;
    }
}

35. NumberList::~NumberList()
{
    ListNode *nodePtr, *nextNode;
    nodePtr = head;
    while (nodePtr != nullptr)
    {
        nextNode = nodePtr->next;
        nodePtr->next = nullptr;
        nodePtr = nextNode;
    }
}
```

Programming Challenges

1. Your Own Linked List

Design your own linked list class to hold a series of integers. The class should have member functions for appending, inserting, and deleting nodes. Don't forget to add a destructor that destroys the list. Demonstrate the class with a driver program.

2. List Print

Modify the linked list class you created in Programming Challenge 1 to add a `print` member function. The function should display all the values in the linked list. Test the class by starting with an empty list, adding some elements, then printing the resulting list out.

3. List Copy Constructor

Modify your linked list class of Programming Challenges 1 and 2 to add a copy constructor. Test your class by making a list, making a copy of the list, then displaying the values in the copy.

4. List Reverse

Modify the linked list class you created in the previous programming challenges by adding a member function named `reverse` that rearranges the nodes in the list so their order is reversed. Demonstrate the function in a simple driver program.

5. List Search

Modify the linked list class you created in the previous programming challenges to include a member function named `search` that returns the position of a specific value, `x`, in the linked list. The first node in the list is at position 0, the second node is at position 1, and so on. If `x` is not found on the list, the search should return -1. Test the new member function using an appropriate driver program.



6. Member Insertion by Position

Modify the list class you created in the previous programming challenges by adding a member function for inserting a new item at a specified position. A position of 0 means that the value will become the first item on the list, a position of 1 means the value will become the second item on the list, and so on. A position equal to or greater than the length of the list means the value is placed at the end of the list.

7. Member Removal by Position

Modify the list class you created in the previous programming challenges by adding a member function for deleting a node at a specified position. A value of 0 for the position means the first node in the list (the current head) is deleted. The function does nothing if the specified position is greater than or equal to the length of the list.

8. List Template

Create a list class template based on the list class you created in the previous programming challenges.

9. Rainfall Statistics Modification

Modify Programming Challenge 2 in Chapter 7 (Rainfall Statistics) to let the user decide how many months of data will be entered. Use a linked list instead of an array to hold the monthly data.

10. Payroll Modification

Modify Programming Challenge 9 in Chapter 7 (Payroll) to use three linked lists instead of three arrays to hold the employee IDs, hours worked, and wages. When the program starts, it should ask the user to enter the employee IDs. There should be no limit on the number of IDs the user can enter.

11. List Search

Modify the `LinkedList` template shown in this chapter to include a member function named `search`. The function should search the list for a specified value. If the value is found, it should return a number indicating its position in the list. (The first node is node 1, the second node is node 2, and so forth.) If the value is not found, the function should return 0. Demonstrate the function in a driver program.

12. Double Merge

Modify the `NumberList` class shown in this chapter to include a member function named `mergeArray`. The `mergeArray` function should take an array of `doubles` as its first argument, and an integer as its second argument. (The second argument will specify the size of the array being passed into the first argument.)

The function should merge the values in the array into the linked list. The value in each element of the array should be inserted (not appended) into the linked list. When the values are inserted, they should be in numerical order. Demonstrate the function with a driver program. When you are satisfied with the function, incorporate it into the `LinkedList` template.

13. Rainfall Statistics Modification #2

Modify the program you wrote for Programming Challenge 9 so it saves the data in the linked list to a file. Write a second program that reads the data from the file into a linked list and displays it on the screen.

14. Overloaded [] Operator

Modify the linked list class you created in Programming Challenge 1 (or the `LinkedList` template presented in this chapter) by adding an overloaded `[]` operator function. This will give the linked list the ability to access nodes using a subscript, like an array. The subscript 0 will reference the first node in the list, the subscript 1 will reference the second node in the list, and so forth. The subscript of the last node will be the number of nodes minus 1. If an invalid subscript is used, the function should throw an exception.

15. pop and push Member Functions

The STL `list` container has member functions named `pop_back`, `pop_front`, `push_back`, and `push_front`, as described in Table 18-2. Modify the linked list class you created in Programming Challenge 1 (or the `LinkedList` template presented in this chapter) by adding your own versions of these member functions.

CHAPTER

19**Stacks and Queues****TOPICS**

- | | |
|---|--|
| 19.1 Introduction to the Stack ADT
19.2 Dynamic Stacks
19.3 The STL stack Container | 19.4 Introduction to the Queue ADT
19.5 Dynamic Queues
19.6 The STL deque and queue Containers |
|---|--|

19.1**Introduction to the Stack ADT**

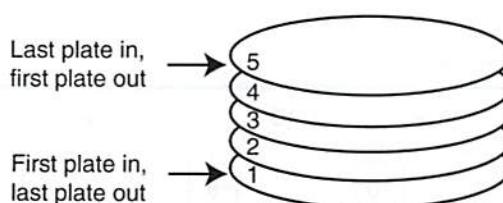
CONCEPT: A stack is a data structure that stores and retrieves items in a last-in, first-out manner.

Definition

Like an array or a linked list, a stack is a data structure that holds a sequence of elements. Unlike arrays and lists, however, stacks are *last-in, first-out (LIFO)* structures. This means when a program retrieves elements from a stack, the last element inserted into the stack is the first one retrieved (and likewise, the first element inserted is the last one retrieved).

When visualizing the way a stack works, think of a stack of plates at the beginning of a cafeteria line. When a cafeteria worker replenishes the supply of plates, the first one he or she puts on the stack is the last one taken off. This is illustrated in Figure 19-1.

Figure 19-1 A stack of plates



The LIFO characteristic of a stack of plates in a cafeteria is also the primary characteristic of a stack data structure. The last data element placed on the stack is the first data retrieved from the stack.

Applications of Stacks

Stacks are useful data structures for algorithms that work first with the last saved element of a series. For example, computer systems use stacks while executing programs. When a function is called, they save the program's return address on a stack. They also create local variables on a stack. When the function terminates, the local variables are removed from the stack and the return address is retrieved. Also, some calculators use a stack for performing mathematical operations.

Static and Dynamic Stacks

There are two types of stack data structures: static and dynamic. Static stacks have a fixed size and are implemented as arrays. Dynamic stacks grow in size as needed and are implemented as linked lists. In this section, you will see examples of both static and dynamic stacks.

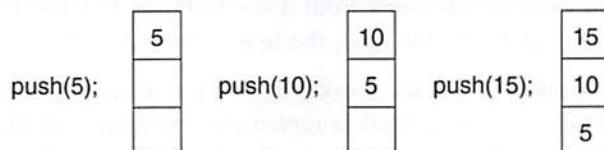
Stack Operations

A stack has two primary operations: *push* and *pop*. The push operation causes a value to be stored, or pushed onto the stack. For example, suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack, we execute the following push operations.

```
push(5);
push(10);
push(15);
```

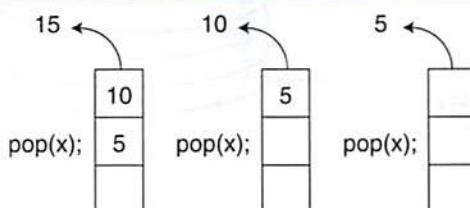
Figure 19-2 illustrates the state of the stack after each of these push operations.

Figure 19-2 State of the stack



The pop operation retrieves (and hence, removes) a value from the stack. Suppose, we execute three consecutive pop operations on the stack shown in Figure 19-2. Figure 19-3 depicts the results.

Figure 19-3 State of the stack



As you can see from Figure 19-3, the last pop operation leaves the stack empty.

For a static stack (one with a fixed size), we will need a Boolean `isFull` operation. The `isFull` operation returns `true` if the stack is full, and `false` otherwise. This operation is necessary to prevent a stack overflow in the event a push operation is attempted when all the stack's elements have values stored in them.

For both static and dynamic stacks, we will need a Boolean `isEmpty` operation. The `isEmpty` operation returns `true` when the stack is empty, and `false` otherwise. This prevents an error from occurring when a pop operation is attempted on an empty stack.

A Static Stack Class

Now we examine a class, `IntStack`, that stores a static stack of integers and performs the `isFull` and `isEmpty` operations. The class has the member variables described in Table 19-1.

Table 19-1 The `IntStack` Class's Member Variables

Member Variable	Description
<code>stackArray</code>	A pointer to <code>int</code> . When the constructor is executed, it uses <code>stackArray</code> to dynamically allocate an array for storage.
<code>stackSize</code>	An integer that holds the size of the stack.
<code>top</code>	An integer that is used to mark the top of the stack.

The class's member functions are listed in Table 19-2.

Table 19-2 The `IntStack` Class's Member Functions

Member Functions	Description
Constructor	The class constructor accepts an integer argument that specifies the size of the stack. An integer array of this size is dynamically allocated and assigned to <code>stackArray</code> . Also, the variable <code>top</code> is initialized to <code>-1</code> .
Destructor	The destructor frees the memory that was allocated by the constructor.
<code>isFull</code>	Returns <code>true</code> if the stack is full, and <code>false</code> otherwise. The stack is full when <code>top</code> is equal to <code>stackSize - 1</code> .
<code>isEmpty</code>	Returns <code>true</code> if the stack is empty, and <code>false</code> otherwise. The stack is empty when <code>top</code> is set to <code>-1</code> .
<code>pop</code>	The <code>pop</code> function uses an integer reference parameter. The value at the top of the stack is removed and copied into the reference parameter.
<code>push</code>	The <code>push</code> function accepts an integer argument, which is pushed onto the top of the stack.



NOTE: Even though the constructor dynamically allocates the stack array, it is still a static stack. The size of the stack does not change once it is allocated.

The code for the class is as follows:

Contents of IntStack.h

```

1 // Specification file for the IntStack class
2 #ifndef INTSTACK_H
3 #define INTSTACK_H
4
5 class IntStack
6 {
7 private:
8     int *stackArray; // Pointer to the stack array
9     int stackSize; // The stack size
10    int top; // Indicates the top of the stack
11
12 public:
13     // Constructor
14     IntStack(int);
15
16     // Copy constructor
17     IntStack(const IntStack &);
18
19     // Destructor
20     ~IntStack();
21
22     // Stack operations
23     void push(int);
24     void pop(int &);
25     bool isFull() const;
26     bool isEmpty() const;
27 };
28 #endif

```

Contents of IntStack.cpp

```

1 // Implementation file for the IntStack class
2 #include <iostream>
3 #include "IntStack.h"
4 using namespace std;
5
6 //*****
7 // Constructor
8 // This constructor creates an empty stack. The *
9 // size parameter is the size of the stack. *
10 //*****
11
12 IntStack::IntStack(int size)
13 {
14     stackArray = new int[size];
15     stackSize = size;
16     top = -1;
17 }
18

```

```
19 //*****
20 // Copy constructor
21 //*****
22
23 IntStack::IntStack(const IntStack &obj)
24 {
25     // Create the stack array.
26     if (obj.stackSize > 0)
27         stackArray = new int[obj.stackSize];
28     else
29         stackArray = nullptr;
30
31     // Copy the stackSize attribute.
32     stackSize = obj.stackSize;
33
34     // Copy the stack contents.
35     for (int count = 0; count < stackSize; count++)
36         stackArray[count] = obj.stackArray[count];
37
38     // Set the top of the stack.
39     top = obj.top;
40 }
41
42 //*****
43 // Destructor
44 //*****
45
46 IntStack::~IntStack()
47 {
48     delete [] stackArray;
49 }
50
51 //*****
52 // Member function push pushes the argument onto
53 // the stack.
54 //*****
55
56 void IntStack::push(int num)
57 {
58     if (isFull())
59     {
60         cout << "The stack is full.\n";
61     }
62     else
63     {
64         top++;
65         stackArray[top] = num;
66     }
67 }
68 }
```

```
69 //*****
70 // Member function pop pops the value at the top      *
71 // of the stack off, and copies it into the variable   *
72 // passed as an argument.                            *
73 //*****
74
75 void IntStack::pop(int &num)
76 {
77     if (isEmpty())
78     {
79         cout << "The stack is empty.\n";
80     }
81     else
82     {
83         num = stackArray[top];
84         top--;
85     }
86 }
87
88 //*****
89 // Member function isFull returns true if the stack   *
90 // is full, or false otherwise.                      *
91 //*****
92
93 bool IntStack::isFull() const
94 {
95     bool status;
96
97     if (top == stackSize - 1)
98         status = true;
99     else
100        status = false;
101
102    return status;
103 }
104
105 //*****
106 // Member function isEmpty returns true if the stack   *
107 // is empty, or false otherwise.                     *
108 //*****
109
110 bool IntStack::isEmpty() const
111 {
112     bool status;
113
114     if (top == -1)
115         status = true;
116     else
117         status = false;
118
119     return status;
120 }
```

The class has two constructors, one that accepts an argument for the stack size (lines 12 through 17 of IntStack.cpp), and a copy constructor (lines 23 through 40). The first constructor dynamically allocates the stack array in line 14, initializes the `stackSize` member variable in line 15, and initializes the `top` member variable in line 16. Remember items are stored to and retrieved from the top of the stack. In this class, the top of the stack is actually the end of the array. The variable `top` is used to mark the top of the stack by holding the subscript of the last element. When `top` holds `-1`, it indicates the stack is empty. (See the `isEmpty` function, which returns `true` if `top` is `-1`, or `false` otherwise.) The stack is full when `top` is at the maximum subscript, which is `stackSize - 1`. This is the value for which `isFull` tests. It returns `true` if the stack is full, or `false` otherwise.

Program 19-1 is a simple driver that demonstrates the `IntStack` class.

Program 19-1

```

1 // This program demonstrates the IntStack class.
2 #include <iostream>
3 #include "IntStack.h"
4 using namespace std;
5
6 int main()
7 {
8     int catchVar; // To hold values popped off the stack
9
10    // Define a stack object to hold 5 values.
11    IntStack stack(5);
12
13    // Push the values 5, 10, 15, 20, and 25 onto the stack.
14    cout << "Pushing 5\n";
15    stack.push(5);
16    cout << "Pushing 10\n";
17    stack.push(10);
18    cout << "Pushing 15\n";
19    stack.push(15);
20    cout << "Pushing 20\n";
21    stack.push(20);
22    cout << "Pushing 25\n";
23    stack.push(25);
24
25    // Pop the values off the stack.
26    cout << "Popping...\n";
27    stack.pop(catchVar);
28    cout << catchVar << endl;
29    stack.pop(catchVar);
30    cout << catchVar << endl;
31    stack.pop(catchVar);
32    cout << catchVar << endl;
33    stack.pop(catchVar);
34    cout << catchVar << endl;
35    stack.pop(catchVar);
36    cout << catchVar << endl;
37
38 }
```

(program output continues)

Program 19-1

(continued)

Program Output

```
Pushing 5
Pushing 10
Pushing 15
Pushing 20
Pushing 25
Popping...
25
20
15
10
5
```

In Program 19-1, the constructor is called with the argument 5. This sets up the member variables as shown in Figure 19-4. Because top is set to -1, the stack is empty.

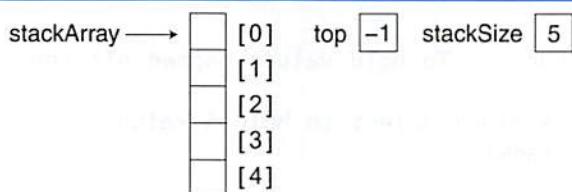
Figure 19-4 State of the stack

Figure 19-5 shows the state of the member variables after the `push` function is called the first time (with 5 as its argument). The top of the stack is now at element 0.

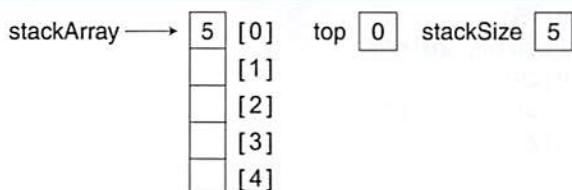
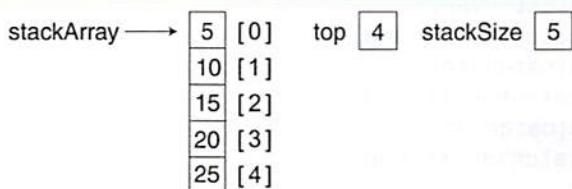
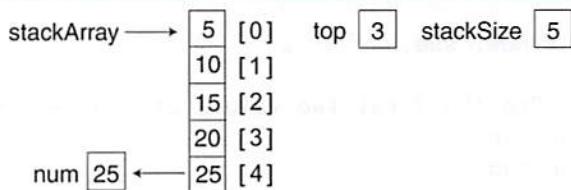
Figure 19-5 State of the stack

Figure 19-6 shows the state of the member variables after all five calls to the `push` function. Now, the top of the stack is at element 4, and the stack is full.

Figure 19-6 State of the stack

Notice the pop function uses a reference parameter, num. The value that is popped off the stack is copied into num so it can be used later in the program. Figure 19-7 depicts the state of the class members, and the num parameter, just after the first value is popped off the stack.

Figure 19-7 State of the stack



The program continues to call the pop function until all the values have been retrieved from the stack.

Implementing Other Stack Operations

More complex operations may be built on the basic stack class previously shown. In this section, we will discuss a class, MathStack, that is derived from IntStack. The MathStack class has two member functions: add() and sub(). The add() function pops the first two values off the stack, adds them together, then pushes the sum onto the stack. The sub() function pops the first two values off the stack, subtracts the second value from the first, then pushes the difference onto the stack. The class declaration is as follows:

Contents of MathStack.h

```

1 // Specification file for the MathStack class
2 #ifndef MATHSTACK_H
3 #define MATHSTACK_H
4 #include "IntStack.h"
5
6 class MathStack : public IntStack
7 {
8 public:
9     // Constructor
10    MathStack(int s) : IntStack(s) {}
11
12    // MathStack operations
13    void add();
14    void sub();
15 };
16 #endif

```

The definitions of the member functions are shown here:

Contents of MathStack.cpp

```

1 // Implementation file for the MathStack class
2 #include "MathStack.h"
3

```

```

4 //*****
5 // Member function add. add pops      *
6 // the first two values off the stack and      *
7 // adds them. The sum is pushed onto the stack. *
8 //*****
9
10 void MathStack::add()
11 {
12     int num, sum;
13
14     // Pop the first two values off the stack.
15     pop(sum);
16     pop(num);
17
18     // Add the two values, store in sum.
19     sum += num;
20
21     // Push sum back onto the stack.
22     push(sum);
23 }
24
25 //*****
26 // Member function sub. sub pops the      *
27 // first two values off the stack. The      *
28 // second value is subtracted from the      *
29 // first value. The difference is pushed      *
30 // onto the stack.      *
31 //*****
32
33 void MathStack::sub()
34 {
35     int num, diff;
36
37     // Pop the first two values off the stack.
38     pop(diff);
39     pop(num);
40
41     // Subtract num from diff.
42     diff -= num;
43
44     // Push diff back onto the stack.
45     push(diff);
46 }

```

The class is demonstrated in Program 19-2, a simple driver.

Program 19-2

```

1 // This program demonstrates the MathStack class.
2 #include <iostream>
3 #include "MathStack.h"
4 using namespace std;
5

```

```
6 int main()
7 {
8     int catchVar; // To hold values popped off the stack
9
10    // Create a MathStack object.
11    MathStack stack(5);
12
13    // Push 3 and 6 onto the stack.
14    cout << "Pushing 3\n";
15    stack.push(3);
16    cout << "Pushing 6\n";
17    stack.push(6);
18
19    // Add the two values.
20    stack.add();
21
22    // Pop the sum off the stack and display it.
23    cout << "The sum is ";
24    stack.pop(catchVar);
25    cout << catchVar << endl << endl;
26
27    // Push 7 and 10 onto the stack
28    cout << "Pushing 7\n";
29    stack.push(7);
30    cout << "Pushing 10\n";
31    stack.push(10);
32
33    // Subtract 7 from 10.
34    stack.sub();
35
36    // Pop the difference off the stack and display it.
37    cout << "The difference is ";
38    stack.pop(catchVar);
39    cout << catchVar << endl;
40
41 }
```

Program Output

```
Pushing 3
Pushing 6
The sum is 9

Pushing 7
Pushing 10
The difference is 3
```

It will be left as a Programming Challenge for you to implement the `mult()`, `div()`, and `mod()` functions that will complete the `MathStack` class.

A Static Stack Template

The stack classes shown previously in this chapter work only with integers. A stack template can be easily designed to work with any data type, as shown by the following example:

Contents of Stack.h

```
1 #ifndef STACK_H
2 #define STACK_H
3 #include <iostream>
4 using namespace std;
5
6 // Stack template
7 template <class T>
8 class Stack
9 {
10 private:
11     T *stackArray;
12     int stackSize;
13     int top;
14
15 public:
16     // Constructor
17     Stack(int);
18
19     // Copy constructor
20     Stack(const Stack&);
21
22     // Destructor
23     ~Stack();
24
25     // Stack operations
26     void push(T);
27     void pop(T &);
28     bool isFull();
29     bool isEmpty();
30 };
31
32 //*****
33 // Constructor
34 //*****
35
36 template <class T>
37 Stack<T>::Stack(int size)
38 {
39     stackArray = new T[size];
40     stackSize = size;
41     top = -1;
42 }
43
44 //*****
45 // Copy constructor
46 //*****
```

```

48 template <Class T>
49 Stack<T>;:Stack(const Stack &obj)
50 {
51 // Create the stack array.
52 if (obj.stackSize > 0)
53 stackArray = new T[obj.stackSize];
54 else
55 stackArray = nullptr;
56
57 // Copy the stack size attribute.
58 stackSize = obj.stackSize;
59
60 // Copy the stack contents.
61 for (int count = 0; count < stackSize; count++)
62 stackArray[count] = obj.stackArray[count];
63
64 // Set the top of the stack.
65 top = obj.top;
66 }
67
68 //*****
69 // Destructor
70 //*****
71
72 template <Class T>
73 Stack<T>;:~Stack()
74 {
75 if (stackSize > 0)
76 delete [] stackArray;
77 }
78
79 //*****
80 // Member function push pushes the argument onto
81 // the stack.
82
83 template <Class T>
84 void Stack<T>;:push(T item)
85 {
86 tempalte <Class T>
87 if (isFull())
88 {
89 cout << "The stack is full.\n";
90 }
91 }
92 else
93 {
94 stackArray[top] = item;
95
96 }
97
98 //*****
99 // Member function pop pops the value at the top
100 // off the stack and copies it into the variable
101 // passed as an argument.
102

```

```
103
104 template <class T>
105 void Stack<T>::pop(T &item)
106 {
107     if (isEmpty())
108     {
109         cout << "The stack is empty.\n";
110     }
111     else
112     {
113         item = stackArray[top];
114         top--;
115     }
116 }
117 //*****
118 // Member function isFull returns true if the stack      *
119 // is full, or false otherwise.                      *
120 //*****                                                 *
121
122
123 template <class T>
124 bool Stack<T>::isFull()
125 {
126     bool status;
127
128     if (top == stackSize - 1)
129         status = true;
130     else
131         status = false;
132
133     return status;
134 }
135 //*****
136 // Member function isEmpty returns true if the stack      *
137 // is empty, or false otherwise.                      *
138 //*****                                                 *
139
140
141 template <class T>
142 bool Stack<T>::isEmpty()
143 {
144     bool status;
145
146     if (top == -1)
147         status = true;
148     else
149         status = false;
150
151     return status;
152 }
153 #endif
```

Program 19-3 demonstrates the `Stack` template. It creates a stack of strings, then presents a menu that allows the user to push an item onto the stack, pop an item from the stack, or quit the program.

Program 19-3

```

1 // This program demonstrates the Stack template.
2 #include <iostream>
3 #include <string>
4 #include "Stack.h"
5 using namespace std;
6
7 // Constants for the menu choices
8 const int PUSH_CHOICE = 1,
9         POP_CHOICE = 2,
10        QUIT_CHOICE = 3;
11
12 // Function prototypes
13 void menu(int &);
14 void getStackSize(int &);
15 void pushItem(Stack<string>&);
16 void popItem(Stack<string>&);
17
18 int main()
19 {
20     int stackSize; // The stack size
21     int choice;    // To hold a menu choice
22
23     // Get the stack size.
24     getStackSize(stackSize);
25
26     // Create the stack.
27     Stack<string> stack(stackSize);
28
29     do
30     {
31         // Get the user's menu choice.
32         menu(choice);
33
34         // Perform the user's choice.
35         if (choice != QUIT_CHOICE)
36         {
37             switch (choice)
38             {
39                 case PUSH_CHOICE:
40                     pushItem(stack);
41                     break;
42                 case POP_CHOICE:
43                     popItem(stack);
44             }
45         }
46     } while (choice != QUIT_CHOICE);
47
48     return 0;
49 }
```

(program continues)

Program 19-3*(continued)*

```
50 //*****
51 // The getStackSize function gets the desired *
52 // stack size, which is assigned to the *
53 // reference parameter. *
54 //*****
55 //*****
56 void getStackSize(int &size)
57 {
58     // Get the desired stack size.
59     cout << "How big should I make the stack? ";
60     cin >> size;
61
62     // Validate the size.
63     while (size < 1)
64     {
65         cout << "Enter 1 or greater: ";
66         cin >> size;
67     }
68 }
69
70 //*****
71 // The menu function displays the menu and gets *
72 // the user's choice, which is assigned to the *
73 // reference parameter. *
74 //*****
75 void menu(int &choice)
76 {
77     // Display the menu and get the user's choice.
78     cout << "\nWhat do you want to do?\n"
79         << PUSH_CHOICE
80         << " - Push an item onto the stack\n"
81         << POP_CHOICE
82         << " - Pop an item off the stack\n"
83         << QUIT_CHOICE
84         << " - Quit the program\n"
85         << "Enter your choice: ";
86     cin >> choice;
87
88     // Validate the choice
89     while (choice < PUSH_CHOICE || choice > QUIT_CHOICE)
90     {
91         cout << "Enter a valid choice: ";
92         cin >> choice;
93     }
94 }
95
96 //*****
97 // The pushItem function gets an item from the *
98 // user and pushes it onto the stack. *
99 //*****
100 void pushItem(Stack<string> &stack)
```

(program output continues)

Program Output with Example Input shown in Bold	How big should I make the stack? 3 <input type="button" value="Enter"/>	What do you want to do?	1 - Push an item onto the stack 2 - Pop an item off the stack 3 - Quit the program	Enter an item: The Adventures of Huckleberry Finn <input type="button" value="Enter"/>
Program Output with Example Input shown in Bold	How big should I make the stack? 3 <input type="button" value="Enter"/>	What do you want to do?	1 - Push an item onto the stack 2 - Pop an item off the stack 3 - Quit the program	Enter an item: All Quiet on the Western Front <input type="button" value="Enter"/>
Program Output with Example Input shown in Bold	How big should I make the stack? 3 <input type="button" value="Enter"/>	What do you want to do?	1 - Push an item onto the stack 2 - Pop an item off the stack 3 - Quit the program	Enter an item: Brave New World <input type="button" value="Enter"/>

```

101 {
102     string item;
103
104     // Get an item to push onto the stack.
105     cin.ignore();
106     cout << "Enter an item: ";
107     getline(cin, item);
108     stack.push(item);
109 }
110
111 // *****
112 // The popItem function pops an item from the stack.
113 // *****
114 void popItem(stack<string> &stack)
115 {
116     string item = "";
117
118     // Pop the item.
119     stack.pop(item);
120
121     // Display the item.
122     if (item != "")
123         cout << "item << " was popped.\n";
124 }

```

Program 19-3

(continued)

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

Brave New World was popped.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

All Quiet on the Western Front was popped.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

The Adventures of Huckleberry Finn was popped.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

The stack is empty.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **3**

19.2 Dynamic Stacks

CONCEPT: A stack may be implemented as a linked list, and expand or shrink with each push or pop operation.

A dynamic stack is built on a linked list instead of an array. A linked list-based stack offers two advantages over an array-based stack. First, there is no need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, then expands by one node each time a value is pushed. Second, a dynamic stack will never be full, as long as the system has enough free memory.

In this section, we will look at a dynamic stack class, `DynIntStack`. This class is a dynamic version of the `IntStack` class previously discussed. The class declaration is shown here:

Contents of DynIntStack.h

```

1 // Specification file for the DynIntStack class
2 #ifndef DYNINTSTACK_H
3 #define DYNINTSTACK_H
4
5 class DynIntStack
6 {
7 private:
8     // Structure for stack nodes
9     struct StackNode
10    {
11         int value;           // Value in the node
12         StackNode *next;   // Pointer to the next node
13     };
14
15     StackNode *top;        // Pointer to the stack top
16
17 public:
18     // Constructor
19     DynIntStack()
20     { top = nullptr; }
21
22     // Destructor
23     ~DynIntStack();
24
25     // Stack operations
26     void push(int);
27     void pop(int &);
28     bool isEmpty();
29 };
30 #endif

```

The `StackNode` structure is the data type of each node in the linked list. It has a `value` member and a `next` pointer. Notice instead of a `head` pointer, a `top` pointer is defined. This member will always point to the first node in the list, which will represent the top of the stack. It is initialized to `nullptr` by the constructor, to signify the stack is empty.

The definitions of the other member functions are shown here:

Contents of DynIntStack.cpp

```

1 #include <iostream>
2 #include "DynIntStack.h"
3 using namespace std;
4
5 //*****
6 // Destructor
7 // This function deletes every node in the list. *
8 //*****
9
10 DynIntStack::~DynIntStack()
11 {
12     StackNode *nodePtr = nullptr, *nextNode = nullptr;
13

```

```
14     // Position nodePtr at the top of the stack.
15     nodePtr = top;
16
17     // Traverse the list deleting each node.
18     while (nodePtr != nullptr)
19     {
20         nextNode = nodePtr->next;
21         delete nodePtr;
22         nodePtr = nextNode;
23     }
24 }
25
26 //*****
27 // Member function push pushes the argument onto *
28 // the stack. *
29 //*****
30
31 void DynIntStack::push(int num)
32 {
33     StackNode *newNode = nullptr; // Pointer to a new node
34
35     // Allocate a new node and store num there.
36     newNode = new StackNode;
37     newNode->value = num;
38
39     // If there are no nodes in the list
40     // make newNode the first node.
41     if (isEmpty())
42     {
43         top = newNode;
44         newNode->next = nullptr;
45     }
46     else // Otherwise, insert NewNode before top.
47     {
48         newNode->next = top;
49         top = newNode;
50     }
51 }
52
53 //*****
54 // Member function pop pops the value at the top      *
55 // of the stack off, and copies it into the variable   *
56 // passed as an argument. *
57 //*****
58
59 void DynIntStack::pop(int &num)
60 {
61     StackNode *temp = nullptr; // Temporary pointer
62
63     // First make sure the stack isn't empty.
64     if (isEmpty())
65     {
66         cout << "The stack is empty.\n";
67     }
68     else // pop value off top of stack
```

```

69     } // Insert new node at top of stack
70     to_be_deleted->value = num;
71     temp = to_be_deleted->next;
72     delete to_be_deleted;
73     to_be_deleted = temp;
74 }
75 }
76
77 //***** Member function isEmpty *****
78 // Member function isEmpty returns true if the stack *
79 // is empty, or false otherwise.
80 //*****
81
82 bool DynIntStack::isEmpty()
83 {
84     bool status;
85
86     if (!top)
87         status = true;
88     else
89         status = false;
90
91     return status;
92 }
```

Let's look at the push operation in lines 31 through 51 of `DynIntStack.cpp`. First, in lines 36 and 37, a new node is allocated in memory, and the function argument is copied into its value member:

```

newNode = new StackNode;
newNode->value = num;
```

Next, in line 41, an if statement calls the `isEmpty` function to determine whether the stack is empty:

```

if (isEmpty())
{
    top = newNode;
    newNode->next = nullptr;
}
```

If `isEmpty` returns `true`, `top` is made to point at the new node, and the new node's `next` pointer is set to `nullptr`. After these statements execute, there will be one node in the list (and one value on the stack).

If `isEmpty` returns `false` in the if statement, the following statements in lines 46 through 50 are executed:

```

else // Otherwise, insert newNode before top
{
    newNode->next = top;
    top = newNode;
}
```

Notice `newNode` is being inserted in the list before the node to which `top` points. The `top` pointer is then updated to point to the new node. When this is done, `newNode` is at the top of the stack.

Now, let's look at the pop function in lines 59 through 75. Just as the push function must insert nodes at the head of the list, pop must delete nodes at the head of the list. First, the function calls isEmpty in line 64 to determine whether there are any nodes in the stack. If there are none, an error message is displayed:

```
if (isEmpty())
{
    cout << "The stack is empty.\n";
}
```

If isEmpty returns false, then the following statements in lines 68 through 74 are executed.

```
else // pop value off top of stack
{
    num = top->value;
    temp = top->next;
    delete top;
    top = temp;
}
```

First, the value member of the top node is copied into the num reference parameter. This saves the value for later use in the program. Next, a temporary StackNode pointer, temp, is made to point to top->next. If there are other nodes in the list, this causes temp to point to the second node. (If there are no more nodes, this will cause temp to point to nullptr.) Now, it is safe to delete the top node. After the top node is deleted, the top pointer is set equal to temp. This action moves the top pointer down the list by one node. The node that was previously second in the list becomes first.

The isEmpty function, in lines 82 through 92, is simple. If top is a null pointer, then the list (the stack) is empty.

Program 19-4 is a driver that demonstrates the DynIntStack class.

Program 19-4

```
1 // This program demonstrates the dynamic stack.
2 // class DynIntClass.
3 #include <iostream>
4 #include "DynIntStack.h"
5 using namespace std;
6
7 int main()
8 {
9     int catchVar; // To hold values popped off the stack
10
11    // Create a DynIntStack object.
12    DynIntStack stack;
13
14    // Push 5, 10, and 15 onto the stack.
15    cout << "Pushing 5\n";
16    stack.push(5);
17    cout << "Pushing 10\n";
18    stack.push(10);
```

```

19     cout << "Pushing 15\n";
20     stack.push(15);
21
22     // Pop the values off the stack and display them.
23     cout << "Popping...\n";
24     stack.pop(catchVar);
25     cout << catchVar << endl;
26     stack.pop(catchVar);
27     cout << catchVar << endl;
28     stack.pop(catchVar);
29     cout << catchVar << endl;
30
31     // Try to pop another value off the stack.
32     cout << "\nAttempting to pop again... ";
33     stack.pop(catchVar);
34
35 }

```

Program Output

```

Pushing 5
Pushing 10
Pushing 15
Popping...
15
10
5
Attempting to pop again... The stack is empty.

```

A Dynamic Stack Template

The dynamic stack class shown previously in this chapter works only with integers. A dynamic stack template can be easily designed to work with any data type, as shown by the following example:

Contents of DynamicStack.h

```

1 #ifndef DYNAMICSTACK_H
2 #define DYNAMICSTACK_H
3 #include <iostream>
4 using namespace std;
5
6 // Stack template
7 template <class T>
8 class DynamicStack
9 {
10 private:
11     // Structure for the stack nodes
12     struct StackNode
13     {
14         T value;           // Value in the node
15         StackNode *next; // Pointer to the next node
16     };

```

```

17
18     StackNode *top; // Pointer to the stack top
19
20 public:
21     //Constructor
22     DynamicStack()
23     { top = nullptr; }
24
25     // Destructor
26     ~DynamicStack();
27
28     // Stack operations
29     void push(T);
30     void pop(T &);
31     bool isEmpty();
32 };
33
34 //*****
35 // Destructor
36 //*****
37 template <class T>
38 DynamicStack<T>::~DynamicStack()
39 {
40     StackNode *nodePtr, *nextNode;
41
42     // Position nodePtr at the top of the stack.
43     nodePtr = top;
44
45     // Traverse the list deleting each node.
46     while (nodePtr != nullptr)
47     {
48         nextNode = nodePtr->next;
49         delete nodePtr;
50         nodePtr = nextNode;
51     }
52 }
53
54 //*****
55 // Member function push pushes the argument onto
56 // the stack.
57 //*****
58
59 template <class T>
60 void DynamicStack<T>::push(T item)
61 {
62     StackNode *newNode = nullptr; // Pointer to a new node
63
64     // Allocate a new node and store num there.
65     newNode = new StackNode;
66     newNode->value = item;
67
68     // If there are no nodes in the list
69     // make newNode the first node.
70     if (isEmpty())

```

```
71     {
72         top = newNode;
73         newNode->next = nullptr;
74     }
75     else // Otherwise, insert NewNode before top.
76     {
77         newNode->next = top;
78         top = newNode;
79     }
80 }
81
82 //*****
83 // Member function pop pops the value at the top
84 // of the stack off, and copies it into the variable
85 // passed as an argument.
86 //*****
87
88 template <class T>
89 void DynamicStack<T>::pop(T &item)
90 {
91     StackNode *temp = nullptr; // Temporary pointer
92
93     // First make sure the stack isn't empty.
94     if (isEmpty())
95     {
96         cout << "The stack is empty.\n";
97     }
98     else // pop value off top of stack
99     {
100         item = top->value;
101         temp = top->next;
102         delete top;
103         top = temp;
104     }
105 }
106
107 //*****
108 // Member function isEmpty returns true if the stack
109 // is empty, or false otherwise.
110 //*****
111
112 template <class T>
113 bool DynamicStack<T>::isEmpty()
114 {
115     bool status;
116
117     if (!top)
118         status = true;
119     else
120         status = false;
121
122     return status;
123 }
124 #endif
```

Program 19-5 demonstrates the DynamicStack template. This program is a modification of Program 19-3. It creates a stack of strings then presents a menu that allows the user to push an item onto the stack, pop an item from the stack, or quit the program.

Program 19-5

```
1 #include <iostream>
2 #include <string>
3 #include "DynamicStack.h"
4 using namespace std;
5
6 // Constants for the menu choices
7 const int PUSH_CHOICE = 1,
8         POP_CHOICE = 2,
9         QUIT_CHOICE = 3;
10
11 // Function prototypes
12 void menu(int &);
13 void getStackSize(int &);
14 void pushItem(DynamicStack<string> &);
15 void popItem(DynamicStack<string> &);
16
17 int main()
18 {
19     int choice; // To hold a menu choice
20
21     // Create the stack.
22     DynamicStack<string> stack;
23
24     do
25     {
26         // Get the user's menu choice.
27         menu(choice);
28
29         // Perform the user's choice.
30         if (choice != QUIT_CHOICE)
31         {
32             switch (choice)
33             {
34                 case PUSH_CHOICE:
35                     pushItem(stack);
36                     break;
37                 case POP_CHOICE:
38                     popItem(stack);
39             }
40         }
41     } while (choice != QUIT_CHOICE);
42
43     return 0;
44 }
```

```
46 //*****
47 // The menu function displays the menu and gets *
48 // the user's choice, which is assigned to the *
49 // reference parameter.
50 //*****
51 void menu(int &choice)
52 {
53     // Display the menu and get the user's choice.
54     cout << "What do you want to do?\n"
55         << PUSH_CHOICE
56         << " - Push an item onto the stack\n"
57         << POP_CHOICE
58         << " - Pop an item off the stack\n"
59         << QUIT_CHOICE
60         << " - Quit the program\n"
61         << "Enter your choice: ";
62     cin >> choice;
63
64     // Validate the choice
65     while (choice < PUSH_CHOICE || choice > QUIT_CHOICE)
66     {
67         cout << "Enter a valid choice: ";
68         cin >> choice;
69     }
70 }
71 //*****
72 // The pushItem function gets an item from the *
73 // user and pushes it onto the stack. *
74 //*****
75 void pushItem(DynamicStack<string> &stack)
76 {
77     string item;
78
79     // Get an item to push onto the stack.
80     cin.ignore();
81     cout << "\nEnter an item: ";
82     getline(cin, item);
83     stack.push(item);
84 }
85
86 //*****
87 // The popItem function pops an item from the stack *
88 //*****
89 void popItem(DynamicStack<string> &stack)
90 {
91     string item = "";
92
93     // Pop the item.
94     stack.pop(item);
```

(program continues)

Program 19-5

(continued)

```

97      // Display the item.
98      if (item != "")
99          cout << item << " was popped.\n";
100 }

```

Program Output with Example Input Shown in Bold

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **1**

Enter an item: **The Catcher in the Rye**

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **1**

Enter an item: **Crime and Punishment**

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

Crime and Punishment was popped.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

The Catcher in the Rye was popped.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **2**

The stack is empty.

What do you want to do?

- 1 – Push an item onto the stack
- 2 – Pop an item off the stack
- 3 – Quit the program

Enter your choice: **3**

19.3

The STL stack Container



CONCEPT: The Standard Template Library offers a stack template, which may be implemented as a `vector`, a `list`, or a `deque`.

So far, the STL containers you have learned about are `vectors` and `lists`. The STL `stack` container may be implemented as a `vector` or a `list`. (It may also be implemented as a `deque`, which you will learn about later in this chapter.) Because the `stack` container is used to adapt these other containers, it is often referred to as a *container adapter*.

Here are examples of how to define a stack of `ints`, implemented as a `vector`, a `list`, and a `deque`:

```
stack<int, vector<int>> iStack; // Vector stack
stack<int, list<int>> iStack; // List stack
stack<int> iStack; // Default - deque stack
```



11

NOTE: If you are using a compiler that is older than C++ 11, be sure to put spaces between the angled brackets that appear next to each other. Older compilers will mistake the `>>` characters for the stream extraction operator, `>>`. Here is an example of how to write the definitions for an older compiler:

```
stack< int, vector<int> > iStack; // Vector stack
stack< int, list<int> > iStack; // List stack
stack< int > iStack; // Default - deque stack
```

Table 19-3 lists and describes some of the `stack` template's member functions.

Table 19-3 Some of the `stack` Template Member Functions

Member Function	Examples and Description
<code>empty</code>	<code>if (myStack.empty())</code> The <code>empty</code> member function returns <code>true</code> if the stack is empty. If the stack has elements, it returns <code>false</code> .
<code>pop</code>	<code>myStack.pop();</code> The <code>pop</code> function removes the element at the top of the stack.
<code>push</code>	<code>myStack.push(x);</code> The <code>push</code> function pushes an element with the value <code>x</code> onto the stack.
<code>size</code>	<code>cout << myStack.size() << endl;</code> The <code>size</code> function returns the number of elements in the list.
<code>top</code>	<code>x = myStack.top();</code> The <code>top</code> function returns a reference to the element at the top of the stack.



NOTE: The `pop` function in the `stack` template does not retrieve the value from the top of the stack, it only removes it. To retrieve the value, you must call the `top` function first.

Program 19-6 is a driver that demonstrates an STL `stack` implemented as a `vector`.

Program 19-6

```

1 // This program demonstrates the STL stack
2 // container adapter.
3 #include <iostream>
4 #include <vector>
5 #include <stack>
6 using namespace std;
7
8 int main()
9 {
10    const int MAX = 8;      // Max value to store in the stack
11    int count;             // Loop counter
12
13    // Define an STL stack
14    stack< int, vector<int> > iStack;
15
16    // Push values onto the stack.
17    for (count = 2; count < MAX; count += 2)
18    {
19        cout << "Pushing " << count << endl;
20        iStack.push(count);
21    }
22
23    // Display the size of the stack.
24    cout << "The size of the stack is ";
25    cout << iStack.size() << endl;
26
27    // Pop the values off the stack.
28    for (count = 2; count < MAX; count += 2)
29    {
30        cout << "Popping " << iStack.top() << endl;
31        iStack.pop();
32    }
33    return 0;
34 }
```

Program Output

```

Pushing 2
Pushing 4
Pushing 6
The size of the stack is 3
Popping 6
Popping 4
Popping 2
```

**Checkpoint**

- 19.1 Describe what LIFO means.
- 19.2 What is the difference between static and dynamic stacks? What advantages do dynamic stacks have over static stacks?
- 19.3 What are the two primary stack operations? Describe them both.
- 19.4 What STL types does the STL stack container adapt?

19.4 Introduction to the Queue ADT

CONCEPT: A queue is a data structure that stores and retrieves items in a first-in, first-out manner.

Definition

Like a stack, a queue (pronounced “cue”) is a data structure that holds a sequence of elements. A queue, however, provides access to its elements in *first-in, first-out (FIFO)* order. The elements in a queue are processed like customers standing in a grocery checkout line: The first customer in line is the first one served.

Application of Queues

Queue data structures are commonly used in computer operating systems. They are especially important in multiuser/multitasking environments where several users or tasks may be simultaneously requesting the same resource. Printing, for example, is controlled by a queue because only one document may be printed at a time. A queue is used to hold print jobs submitted by users of the system, while the printer services those jobs one at a time.

Communications software also uses queues to hold data received over networks. Sometimes data is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

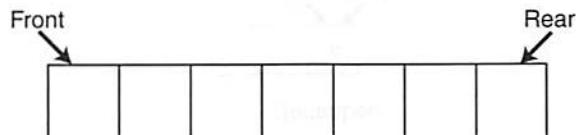
Static and Dynamic Queues

Just as stacks are implemented as arrays or linked lists, so are queues. Dynamic queues offer the same advantages over static queues that dynamic stacks offer over static stacks. In fact, the primary difference between queues and stacks is the way data elements are accessed in each structure.

Queue Operations

Just like checkout lines in a grocery store, think of queues as having a front and a rear. This is illustrated in Figure 19-8.

Figure 19-8 A queue has a front and a rear



When an element is added to a queue, it is added to the rear. When an element is removed from a queue, it is removed from the front. The two primary queue operations are *enqueueing* and *dequeueing*. To enqueue means to insert an element at the rear of a queue, and to

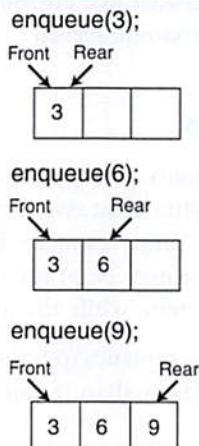
dequeue means to remove an element from the front of a queue. There are several different algorithms for implementing these operations. We will begin by looking at the most simple.

Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue, we execute the following enqueue operations:

```
enqueue(3);
enqueue(6);
enqueue(9);
```

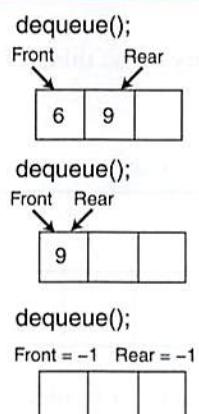
Figure 19-9 illustrates the state of the queue after each of these enqueue operations.

Figure 19-9 State of the queue



Notice in this example the front index (which is a variable holding a subscript or perhaps a pointer) always references the same physical element. The rear index moves forward in the array as items are enqueued. Now, let's see how dequeue operations are performed. Figure 19-10 illustrates the state of the queue after each of three consecutive dequeue operations.

Figure 19-10 State of the queue



In the dequeuing operation, the element at the front of the queue is removed. This is done by moving all the elements after it forward by one position. After the first dequeue operation, the value 3 is removed from the queue and the value 6 is at the front. After the second dequeue operation, the value 6 is removed and the value 9 is at the front. Notice when only one value is stored in the queue, that value is at both the front and the rear.

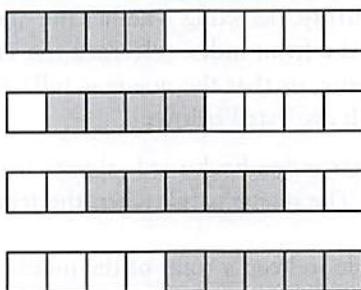
When the last dequeue operation is performed in Figure 19-10, the queue is empty. An empty queue can be signified by setting both front and rear indices to -1.

The problem with this algorithm is its inefficiency. Each time an item is dequeued, the remaining items in the queue are copied forward to their neighboring elements. The more items there are in the queue, the longer each successive dequeue operation will take.

Here is one way to overcome the problem: Make both the front and rear indices move in the array. As before, when an item is enqueued, the rear index is moved to make room for it. But in this design, when an item is dequeued, the front index moves by one element toward the rear of the queue. This logically removes the front item from the queue, and eliminates the need to copy the remaining items to their neighboring elements.

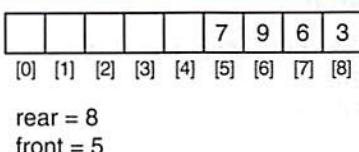
With this approach, as items are added and removed, the queue gradually “crawls” toward the end of the array. This is illustrated in Figure 19-11. The shaded squares represent the queue elements (between the front and rear).

Figure 19-11 The queue contents gradually crawl toward the end of the array

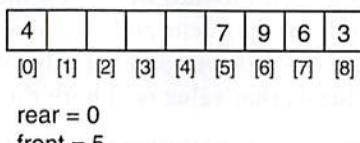


The problem with this approach is that the rear index cannot move beyond the last element in the array. The solution is to think of the array as circular instead of linear. When an item moves past the end of a circular array, it simply “wraps around” to the beginning. For example, consider the queue depicted in Figure 19-12.

Figure 19-12 A queue positioned at the end of an array



The value 3 is at the rear of the queue, and the value 7 is at the front of the queue. Now suppose an enqueue operation is performed, inserting the value 4 into the queue. Figure 19-13 shows how the rear of the queue wraps around to the beginning of the array.

Figure 19-13 The rear of the queue wraps around to the beginning

So, what is the code for wrapping the rear marker around to the opposite end of the array? One straightforward approach is to use an if statement such as

```
if (rear == queueSize - 1)
    rear = 0;
else
    rear++;
```

Another approach is with modular arithmetic:

```
rear = (rear + 1) % queueSize;
```

This statement uses the % operator to adjust the value in `rear` to the proper position. Although this approach appears more elegant, the choice of which code to use is yours.

Detecting Full and Empty Queues with Circular Arrays

One problem with the circular array algorithm is that, because both the front and rear indices move through the array, detecting whether the queue is full or empty is a challenge. When the rear index and the front index reference the same element, does it indicate that only one item is in the queue, or that the queue is full? A number of approaches are commonly taken, two of which are listed below:

- When moving the rear index backward, always leave one element empty between it and the front index. The queue is full when the rear index is within two positions of the front index.
- Use a counter variable to keep a total of the number of items in the queue.

Because it might be helpful to keep a count of items in the queue anyway, we will use the second method in our implementation.

A Static Queue Class

The declaration of the `IntQueue` class is as follows:

Contents of `IntQueue.h`

```
1 // Specification file for the IntQueue class
2 #ifndef INTQUEUE_H
3 #define INTQUEUE_H
4
```

```

5  class IntQueue
6  {
7  private:
8      int *queueArray; // Points to the queue array
9      int queueSize; // The queue size
10     int front; // Subscript of the queue front
11     int rear; // Subscript of the queue rear
12     int numItems; // Number of items in the queue
13 public:
14     // Constructor
15     IntQueue(int);
16
17     // Copy constructor
18     IntQueue(const IntQueue &);
19
20     // Destructor
21     ~IntQueue();
22
23     // Queue operations
24     void enqueue(int);
25     void dequeue(int &);
26     bool isEmpty() const;
27     bool isFull() const;
28     void clear();
29 };
30 #endif

```

Notice in addition to the operations discussed in this section, the class also declares a member function named `clear`. This function clears the queue by resetting the `front` and `rear` indices and setting the `numItems` member to 0. The member function definitions are listed below.

Contents of IntQueue.cpp

```

1 // Implementation file for the IntQueue class
2 #include <iostream>
3 #include "IntQueue.h"
4 using namespace std;
5
6 //*****
7 // This constructor creates an empty queue of a specified size. *
8 //*****
9
10 IntQueue::IntQueue(int s)
11 {
12     queueArray = new int[s];
13     queueSize = s;
14     front = -1;
15     rear = -1;
16     numItems = 0;
17 }
18

```

```
19 //*****
20 // Copy constructor
21 //*****
22
23 IntQueue::IntQueue(const IntQueue &obj)
24 {
25     // Allocate the queue array.
26     queueArray = new int[obj.queueSize];
27
28     // Copy the other object's attributes.
29     queueSize = obj.queueSize;
30     front = obj.front;
31     rear = obj.rear;
32     numItems = obj.numItems;
33
34     // Copy the other object's queue array.
35     for (int count = 0; count < obj.queueSize; count++)
36         queueArray[count] = obj.queueArray[count];
37 }
38
39 //*****
40 // Destructor
41 //*****
42
43 IntQueue::~IntQueue()
44 {
45     delete [] queueArray;
46 }
47
48 //*****
49 // Function enqueue inserts a value at the rear of the queue. *
50 //*****
51
52 void IntQueue::enqueue(int num)
53 {
54     if (isFull())
55         cout << "The queue is full.\n";
56     else
57     {
58         // Calculate the new rear position
59         rear = (rear + 1) % queueSize;
60         // Insert new item
61         queueArray[rear] = num;
62         // Update item count
63         numItems++;
64     }
65 }
66
67 //*****
68 // Function dequeue removes the value at the front of the queue *
69 // and copies it into num.
70 //*****
71
```

```
72 void IntQueue::dequeue(int &num)
73 {
74     if (isEmpty())
75         cout << "The queue is empty.\n";
76     else
77     {
78         // Move front
79         front = (front + 1) % queueSize;
80         // Retrieve the front item
81         num = queueArray[front];
82         // Update item count
83         numItems--;
84     }
85 }
86
87 //***** *****
88 // isEmpty returns true if the queue is empty, otherwise false. *
89 //***** *****
90
91 bool IntQueue::isEmpty() const
92 {
93     bool status;
94
95     if (numItems)
96         status = false;
97     else
98         status = true;
99
100    return status;
101 }
102
103 //***** *****
104 // isFull returns true if the queue is full, otherwise false. *
105 //***** *****
106
107 bool IntQueue::isFull() const
108 {
109     bool status;
110
111     if (numItems < queueSize)
112         status = false;
113     else
114         status = true;
115
116     return status;
117 }
118
119 //***** *****
120 // clear sets the front and rear indices, and sets numItems to 0. *
121 //***** *****
```

```

123 void IntQueue::clear()
124 {
125     front = queueSize - 1;
126     rear = queueSize - 1;
127     numItems = 0;
128 }
```

Program 19-7 is a driver that demonstrates the IntQueue class.

Program 19-7

```

1 // This program demonstrates the IntQueue class.
2 #include <iostream>
3 #include "IntQueue.h"
4 using namespace std;
5
6 int main()
7 {
8     const int MAX_VALUES = 5; // Max number of values
9
10    // Create an IntQueue to hold the values.
11    IntQueue iQueue(MAX_VALUES);
12
13    // Enqueue a series of items.
14    cout << "Enqueuing " << MAX_VALUES << " items...\n";
15    for (int x = 0; x < MAX_VALUES; x++)
16        iQueue.enqueue(x);
17
18    // Attempt to enqueue just one more item.
19    cout << "Now attempting to enqueue again...\n";
20    iQueue.enqueue(MAX_VALUES);
21
22    // Dequeue and retrieve all items in the queue
23    cout << "The values in the queue were:\n";
24    while (!iQueue.isEmpty())
25    {
26        int value;
27        iQueue.dequeue(value);
28        cout << value << endl;
29    }
30    return 0;
31 }
```

Program Output

```

Enqueuing 5 items...
Now attempting to enqueue again...
The queue is full.
The values in the queue were:
0
1
2
3
4
```

A Static Queue Template

The queue class shown previously works only with integers. A queue template can be easily designed to work with any data type, as shown by the following example:

Contents of Queue.h

```

1  #ifndef QUEUE_H
2  #define QUEUE_H
3  #include <iostream>
4  using namespace std;
5
6  // Stack template
7  template <class T>
8  class Queue
9  {
10 private:
11     T *queueArray;    // Points to the queue array
12     int queueSize;   // The queue size
13     int front;       // Subscript of the queue front
14     int rear;        // Subscript of the queue rear
15     int numItems;    // Number of items in the queue
16 public:
17     // Constructor
18     Queue(int);
19
20     // Copy constructor
21     Queue(const Queue &);
22
23     // Destructor
24     ~Queue();
25
26     // Queue operations
27     void enqueue(T);
28     void dequeue(T &);
29     bool isEmpty() const;
30     bool isFull() const;
31     void clear();
32 };
33
34 //*****
35 // This constructor creates an empty queue of a specified size. *
36 //*****
37 template <class T>
38 Queue<T>::Queue(int s)
39 {
40     queueArray = new T[s];
41     queueSize = s;
42     front = -1;
43     rear = -1;
44     numItems = 0;
45 }
46

```

```

47 // Copy constructor
48 // Copy constructor
49 // Copy constructor
50 template <Class T>
51 Queue<T>::Queue(const Queue &obj)
52 {
53 // Allocate the queue array.
54 queueArray = new T[obj].queueSize;
55 // Allocate the queue array.
56 // Copy the other object's attributes.
57 queueSize = obj.queueSize;
58 front = obj.front;
59 rear = obj.rear;
60 numItems = obj.numItems;
61 numItems = obj.numItems;
62 // Copy the other object's queue array.
63 for (int count = 0; count < obj.queueSize; count++)
64 queueArray[count] = obj.queueArray[count];
65 }
66 // Destructor
67 // Destructor
68 // Destructor
69 // Destructor
70 template <Class T>
71 Queue<T>::~Queue()
72 {
73 delete [] queueArray;
74 }
75 // Function enqueue inserts a value at the rear of the queue.
76 // Function enqueue inserts a value at the rear of the queue.
77 void Queue<T>::enqueue(T item)
78 {
79 template <Class T>
80 Queue<T>::enqueue(T item)
81 {
82 if (isFull())
83 cout << "The queue is full.\n";
84 else
85 {
86 // Calculate the new rear position
87 rear = (rear + 1) % queueSize;
88 // Insert new item
89 queueArray[rear] = item;
90 // Update item count
91 numItems++;
92 }
93 }
94 }
95 // Function dequeue removes the value at the front of the queue.
96 // And copies it into num.
97 // Function dequeue removes the value at the front of the queue.
98 // And copies it into num.
99 void Queue<T>::dequeue(T item)
100 {

```

```
101  {
102      if (isEmpty())
103          cout << "The queue is empty.\n";
104      else
105      {
106          // Move front
107          front = (front + 1) % queueSize;
108          // Retrieve the front item
109          item = queueArray[front];
110          // Update item count
111          numItems--;
112      }
113  }
114
115 //*****
116 // isEmpty returns true if the queue is empty, otherwise false. *
117 //*****
118 template <class T>
119 bool Queue<T>::isEmpty() const
120 {
121     bool status;
122
123     if (numItems)
124         status = false;
125     else
126         status = true;
127
128     return status;
129 }
130
131 //*****
132 // isFull returns true if the queue is full, otherwise false. *
133 //*****
134 template <class T>
135 bool Queue<T>::isFull() const
136 {
137     bool status;
138
139     if (numItems < queueSize)
140         status = false;
141     else
142         status = true;
143
144     return status;
145 }
146
147 //*****
148 // clear sets the front and rear indices, and sets numItems to 0. *
149 //*****
150 template <class T>
151 void Queue<T>::clear()
152 {
153     front = queueSize - 1;
154     rear = queueSize - 1;
155     numItems = 0;
156 }
157 #endif
```

Program 19-8 demonstrates the Queue template. It creates a queue that can hold strings, then prompts the user to enter a series of names that are enqueued. The program then dequeues all of the names and displays them.

Program 19-8

```

1 // This program demonstrates the Queue template.
2 #include <iostream>
3 #include <string>
4 #include "Queue.h"
5 using namespace std;
6
7 const int QUEUE_SIZE = 5;
8
9 int main()
10 {
11     string name;
12
13     // Create a Queue.
14     Queue<string> queue(QUEUE_SIZE);
15
16     // Enqueue some names.
17     for (int count = 0; count < QUEUE_SIZE; count++)
18     {
19         cout << "Enter a name: ";
20         getline(cin, name);
21         queue.enqueue(name);
22     }
23
24     // Dequeue the names and display them.
25     cout << "\nHere are the names you entered:\n";
26     for (int count = 0; count < QUEUE_SIZE; count++)
27     {
28         queue.dequeue(name);
29         cout << name << endl;
30     }
31     return 0;
32 }
```

Program Output with Example Input Shown in Bold

```

Enter a name: Chris Enter
Enter a name: Kathryn Enter
Enter a name: Alfredo Enter
Enter a name: Lori Enter
Enter a name: Kelly Enter
```

Here are the names you entered:

```

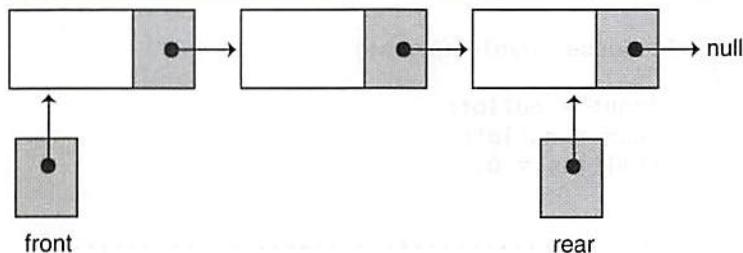
Chris
Kathryn
Alfredo
Lori
Kelly
```

19.5 Dynamic Queues

CONCEPT: A queue may be implemented as a linked list and expand or shrink with each enqueue or dequeue operation.

Dynamic queues, which are built around linked lists, are much more intuitive to understand than static queues. A dynamic queue starts as an empty linked list. With the first enqueue operation, a node is added, which is pointed to by the **front** and **rear** pointers. As each new item is added to the queue, a new node is added to the rear of the list, and the **rear** pointer is updated to point to the new node. As each item is dequeued, the node pointed to by the **front** pointer is deleted, and **front** is made to point to the next node in the list. Figure 19-14 shows the structure of a dynamic queue.

Figure 19-14 A dynamic queue



A dynamic integer queue class is listed here:

Contents of DynIntQueue.h

```

1  #ifndef DYNINTQUEUE_H
2  #define DYNINTQUEUE_H
3
4  class DynIntQueue
5  {
6  private:
7      // Structure for the queue nodes
8      struct QueueNode
9      {
10         int value;           // Value in a node
11         QueueNode *next;   // Pointer to the next node
12     };
13
14     QueueNode *front; // The front of the queue
15     QueueNode *rear;  // The rear of the queue
16     int numItems;     // Number of items in the queue
17 public:
18     // Constructor
19     DynIntQueue();
20
21     // Destructor
22     ~DynIntQueue();
  
```

```

23
24     // Queue operations
25     void enqueue(int);
26     void dequeue(int &);
27     bool isEmpty() const;
28     bool isFull() const;
29     void clear();
30 };
31 #endif

```

Contents of DynIntQueue.cpp

```

1  #include <iostream>
2  #include "DynIntQueue.h"
3  using namespace std;
4
5  //*****
6  // The constructor creates an empty queue. *
7  //*****
8
9  DynIntQueue::DynIntQueue()
10 {
11     front = nullptr;
12     rear = nullptr;
13     numItems = 0;
14 }
15
16 //*****
17 // Destructor
18 //*****
19
20 DynIntQueue::~DynIntQueue()
21 {
22     clear();
23 }
24
25 //*****
26 // Function enqueue inserts the value in num *
27 // at the rear of the queue.
28 //*****
29
30 void DynIntQueue::enqueue(int num)
31 {
32     QueueNode *newNode = nullptr;
33
34     // Create a new node and store num there.
35     newNode = new QueueNode;
36     newNode->value = num;
37     newNode->next = nullptr;
38
39     // Adjust front and rear as necessary.
40     if (isEmpty())
41     {
42         front = newNode;
43         rear = newNode;
44     }

```

```
45     else
46     {
47         rear->next = newNode;
48         rear = newNode;
49     }
50
51     // Update numItems.
52     numItems++;
53 }
54
55 //*****
56 // Function dequeue removes the value at the *
57 // front of the queue, and copies it into num. *
58 //*****
59
60 void DynIntQueue::dequeue(int &num)
61 {
62     QueueNode *temp = nullptr;
63
64     if (isEmpty())
65     {
66         cout << "The queue is empty.\n";
67     }
68     else
69     {
70         // Save the front node value in num.
71         num = front->value;
72
73         // Remove the front node and delete it.
74         temp = front;
75         front = front->next;
76         delete temp;
77
78         // Update numItems.
79         numItems--;
80     }
81 }
82
83 //*****
84 // Function isEmpty returns true if the queue *
85 // is empty, and false otherwise. *
86 //*****
87
88 bool DynIntQueue::isEmpty() const
89 {
90     bool status;
91
92     if (numItems > 0)
93         status = false;
94     else
95         status = true;
96     return status;
97 }
98 }
```

```

99 //*****
100 // Function clear dequeues all the elements *
101 // in the queue. *
102 //*****
103
104 void DynIntQueue::clear()
105 {
106     int value; // Dummy variable for dequeue
107
108     while(!isEmpty())
109         dequeue(value);
110 }

```

Program 19-9 is a driver that demonstrates the `DynIntQueue` class.

Program 19-9

```

1 // This program demonstrates the DynIntQueue class.
2 #include <iostream>
3 #include "DynIntQueue.h"
4 using namespace std;
5
6 int main()
7 {
8     const int MAX_VALUES = 5;
9
10    // Create a DynIntQueue object.
11    DynIntQueue iQueue;
12
13    // Enqueue a series of numbers.
14    cout << "Enqueuing " << MAX_VALUES << " items...\n";
15    for (int x = 0; x < 5; x++)
16        iQueue.enqueue(x);
17
18    // Dequeue and retrieve all numbers in the queue
19    cout << "The values in the queue were:\n";
20    while (!iQueue.isEmpty())
21    {
22        int value;
23        iQueue.dequeue(value);
24        cout << value << endl;
25    }
26    return 0;
27 }

```

Program Output

```

Enqueuing 5 items...
The values in the queue were:
0
1
2
3
4

```

A Dynamic Queue Template

The dynamic queue class shown previously in this chapter works only with integers. A dynamic queue template can be easily designed to work with any data type, as shown by the following example:

Contents of DynamicQueue.h

```

1  #ifndef DYNAMICQUEUE_H
2  #define DYNAMICQUEUE_H
3  #include <iostream>
4  using namespace std;
5
6  // DynamicQueue template
7  template <class T>
8  class DynamicQueue
9  {
10 private:
11     // Structure for the queue nodes
12     struct QueueNode
13     {
14         T value;           // Value in a node
15         QueueNode *next; // Pointer to the next node
16     };
17
18     QueueNode *front;    // The front of the queue
19     QueueNode *rear;    // The rear of the queue
20     int numItems;       // Number of items in the queue
21 public:
22     // Constructor
23     DynamicQueue();
24
25     // Destructor
26     ~DynamicQueue();
27
28     // Queue operations
29     void enqueue(T);
30     void dequeue(T &);
31     bool isEmpty() const;
32     bool isFull() const;
33     void clear();
34 };
35
36 //*****
37 // The constructor creates an empty queue. *
38 //*****
39 template <class T>
40 DynamicQueue<T>::DynamicQueue()
41 {
42     front = nullptr;
43     rear = nullptr;
44     numItems = 0;
45 }
46

```

```

47 //*****
48 // Destructor
49 //*****
50 template <class T>
51 DynamicQueue<T>::~DynamicQueue()
52 {
53     clear();
54 }
55 //*****
56 // Function enqueue inserts the value in num *
57 // at the rear of the queue.
58 //*****
59 template <class T>
60 void DynamicQueue<T>::enqueue(T item)
61 {
62     QueueNode *newNode = nullptr;
63
64     // Create a new node and store num there.
65     newNode = new QueueNode;
66     newNode->value = item;
67     newNode->next = nullptr;
68
69     // Adjust front and rear as necessary.
70     if (isEmpty())
71     {
72         front = newNode;
73         rear = newNode;
74     }
75     else
76     {
77         rear->next = newNode;
78         rear = newNode;
79     }
80
81     // Update numItems.
82     numItems++;
83 }
84
85 //*****
86 // Function dequeue removes the value at the *
87 // front of the queue, and copies it into num. *
88 //*****
89 template <class T>
90 void DynamicQueue<T>::dequeue(T &item)
91 {
92     QueueNode *temp = nullptr;
93
94     if (isEmpty())
95     {
96         cout << "The queue is empty.\n";
97     }
98     else
99     {
100         // Save the front node value in num.
101         item = front->value;
102     }
103 }
```

```

103         // Remove the front node and delete it.
104         temp = front;
105         front = front->next;
106         delete temp;
107
108         // Update numItems.
109         numItems--;
110     }
111 }
113
114 //*****
115 // Function isEmpty returns true if the queue *
116 // is empty, and false otherwise. *
117 //*****
118 template <class T>
119 bool DynamicQueue<T>::isEmpty() const
120 {
121     bool status;
122
123     if (numItems > 0)
124         status = false;
125     else
126         status = true;
127     return status;
128 }
129
130 //*****
131 // Function clear dequeues all the elements *
132 // in the queue. *
133 //*****
134 template <class T>
135 void DynamicQueue<T>::clear()
136 {
137     T value; // Dummy variable for dequeue
138
139     while(!isEmpty())
140         dequeue(value);
141 }
142 #endif

```

Program 19-10 demonstrates the `DynamicQueue` template. This program is a modification of Program 19-8. It creates a queue that can hold strings then prompts the user to enter a series of names that are enqueued. The program then dequeues all of the names and displays them. (The program's output is the same as that of Program 19-8.)

Program 19-10

```

1 // This program demonstrates the DynamicQueue template.
2 #include <iostream>
3 #include <string>
4 #include "DynamicQueue.h"
5 using namespace std;
6

```

(program continues)

Program 19-10 (continued)

```

7  const int QUEUE_SIZE = 5;
8
9  int main()
10 {
11     string name;
12
13     // Create a Queue.
14     DynamicQueue<string> queue;
15
16     // Enqueue some names.
17     for (int count = 0; count < QUEUE_SIZE; count++)
18     {
19         cout << "Enter a name: ";
20         getline(cin, name);
21         queue.enqueue(name);
22     }
23
24     // Dequeue the names and display them.
25     cout << "\nHere are the names you entered:\n";
26     for (int count = 0; count < QUEUE_SIZE; count++)
27     {
28         queue.dequeue(name);
29         cout << name << endl;
30     }
31     return 0;
32 }
```

Program Output*(Same as Program 19-8's output.)***19.6****The STL deque and queue Containers**

CONCEPT: The Standard Template Library provides two containers, `deque` and `queue`, for implementing queue-like data structures.

In this section, we will examine two ADTs offered by the Standard Template Library: `deque` and `queue`. A `deque` (pronounced “deck” or “deek”) is a double-ended queue. It is similar to a `vector`, but allows efficient access to values at both the front and the rear. The `queue` ADT is a container adapter that uses a `deque` as its default container.

The deque Container

Think of the `deque` container as a `vector` that provides quick access to the element at its front as well as at the back. (Like `vector`, `deque` also provides access to its elements with the `[]` operator.)

Programs that use the `deque` ADT must include the `deque` header. Because we are concentrating on its queue-like characteristics, we will focus our attention on the `push_back`, `pop_front`, and `front` member functions. Table 19-4 describes them.

Table 19-4 Some of the deque Template Member Functions

Member Function	Examples and Description
<code>push_back</code>	<code>iDeque.push_back();</code> Accepts as an argument a value to be inserted into the <code>deque</code> . The argument is inserted after the last element. (Pushed onto the back of the <code>deque</code> .)
<code>pop_front</code>	<code>iDeque.pop_front();</code> Removes the first element of the <code>deque</code> .
<code>front</code>	<code>cout << iDeque.front() << endl;</code> <code>front</code> returns a reference to the first element of the <code>deque</code> .

Program 19-11 demonstrates the `deque` container.

Program 19-11

```

1 // This program demonstrates the STL deque container.
2 #include <iostream>
3 #include <deque>
4 using namespace std;
5
6 int main()
7 {
8     const int MAX = 8;    // Max value
9     int count;           // Loop counter
10
11    // Create a deque object.
12    deque<int> iDeque;
13
14    // Enqueue a series of numbers.
15    cout << "I will now enqueue items...\n";
16    for (count = 2; count < MAX; count += 2)
17    {
18        cout << "Pushing " << count << endl;
19        iDeque.push_back(count);
20    }
21
22    // Dequeue and display the numbers.
23    cout << "I will now dequeue items...\n";
24    for (count = 2; count < MAX; count += 2)
25    {
26        cout << "Popping " << iDeque.front() << endl;
27        iDeque.pop_front();
28    }
29    return 0;
30 }
```

(program output continues)

Program 19-11 (continued)**Program Output**

```
I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6
```

**The queue Container Adapter**

The queue container adapter can be built upon vectors, lists, or deques. By default, it uses deque as its base.

The insertion and removal operations supported by queue are the same as those supported by the stack ADT: push, pop, and front. There are differences in their behavior, however. The queue version of push always inserts an element at the rear of the queue. The queue version of pop always removes an element from the structure's front. The front function returns the value of the element at the front of the queue.

Program 19-12 demonstrates a queue. Because the definition of the queue does not specify which type of container is being adapted, the queue will be built on a deque.

Program 19-12

```
1 // This program demonstrates the STL queue container adapter.
2 #include <iostream>
3 #include <queue>
4 using namespace std;
5
6 int main()
7 {
8     const int MAX = 8;    // Max value
9     int count;           // Loop counter
10
11    // Define a queue object.
12    queue<int> iQueue;
13
14    // Enqueue a series of numbers.
15    cout << "I will now enqueue items...\n";
16    for (count = 2; count < MAX; count += 2)
17    {
18        cout << "Pushing " << count << endl;
19        iQueue.push(count);
20    }
21
22    // Dequeue and display the numbers.
23    cout << "I will now dequeue items...\n";
24    for (count = 2; count < MAX; count += 2)
```

```

25     {
26         cout << "Popping " << iQueue.front() << endl;
27         iQueue.pop();
28     }
29     return 0;
30 }
```

Program Output

I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6

Review Questions and Exercises**Short Answer**

1. What does LIFO mean?
2. What element is always retrieved from a stack?
3. What is the difference between a static stack and a dynamic stack?
4. Describe two operations that all stacks perform.
5. Describe two operations that static stacks must perform.
6. The STL stack is considered a container adapter. What does that mean?
7. What types may the STL stack be based on? By default, what type is an STL stack based on?
8. What does FIFO mean?
9. When an element is added to a queue, where is it added?
10. When an element is removed from a queue, where is it removed from?
11. Describe two operations that all queues perform.
12. What two queue-like containers does the STL offer?

Fill-in-the-Blank

13. The _____ element saved onto a stack is the first one retrieved.
14. The two primary stack operations are _____ and _____.
15. _____ stacks and queues are implemented as arrays.
16. _____ stacks and queues are implemented as linked lists.
17. The STL stack container is an adapter for the _____, _____, and _____ STL containers.
18. The _____ element saved in a queue is the first one retrieved.

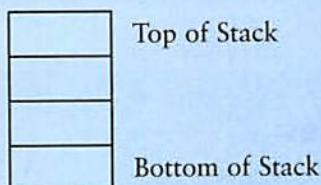
19. The two primary queue operations are _____ and _____.
20. The two ADTs in the Standard Template Library that exhibit queue-like behavior are _____ and _____.
21. The queue ADT, by default, adapts the _____ container.

Algorithm Workbench

22. Suppose the following operations are performed on an empty stack:

```
push(0);
push(9);
push(12);
push(1);
```

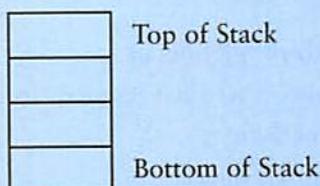
Insert numbers in the following diagram to show what will be stored in the static stack after the operations above have executed.



23. Suppose the following operations are performed on an empty stack:

```
push(8);
push(7);
pop();
push(19);
push(21);
pop();
```

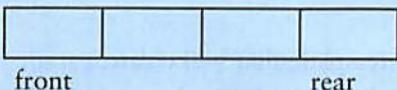
Insert numbers in the following diagram to show what will be stored in the static stack after the operations above have executed.



24. Suppose the following operations are performed on an empty queue:

```
enqueue(5);
enqueue(7);
enqueue(9);
enqueue(12);
```

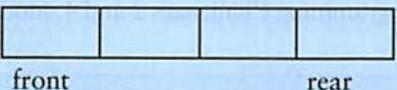
Insert numbers in the following diagram to show what will be stored in the static queue after the operations above have executed.



25. Suppose the following operations are performed on an empty queue:

```
enqueue(5);
enqueue(7);
dequeue();
enqueue(9);
enqueue(12);
dequeue();
enqueue(10);
```

Insert numbers in the following diagram to show what will be stored in the static queue after the operations above have executed.



26. What problem is overcome by using a circular array for a static queue?
 27. Write two different code segments that may be used to wrap an index back around to the beginning of an array when it moves past the end of the array. Use an *if/else* statement in one segment, and modular arithmetic in the other.

True or False

28. T F A static stack or queue is built around an array.
 29. T F The size of a dynamic stack or queue must be known in advance.
 30. T F The push operation inserts an element at the end of a stack.
 31. T F The pop operation retrieves an element from the top of a stack.
 32. T F The STL stack container's pop operation does not retrieve the top element of the stack, it just removes it.

Programming Challenges

1. Static Stack Template

Write your own version of a class template that will create a static stack of any data type. Demonstrate the class with a driver program.

2. Dynamic Stack Template

Write your own version of a class template that will create a dynamic stack of any data type. Demonstrate the class with a driver program.

3. Static Queue Template

Write your own version of a class template that will create a static queue of any data type. Demonstrate the class with a driver program.

4. Dynamic Queue Template

Write your own version of a class template that will create a dynamic queue of any data type. Demonstrate the class with a driver program.

5. Error Testing

The `DynIntStack` and `DynIntQueue` classes shown in this chapter are abstract data types using a dynamic stack and dynamic queue, respectively. The classes do not currently test for memory allocation errors. Modify the classes so they determine whether new nodes cannot be created by handling the `bad_alloc` exception.



NOTE: If you have already done Programming Challenges 2 and 4, modify the templates you created.

6. Dynamic String Stack

Design a class that stores strings on a dynamic stack. The strings should not be fixed in length. Demonstrate the class with a driver program.

7. Dynamic MathStack

The `MathStack` class shown in this chapter has only two member functions: `add` and `sub`. Write the following additional member functions:

Function	Description
<code>mult</code>	Pops the top two values off the stack, multiplies them, and pushes their product onto the stack.
<code>div</code>	Pops the top two values off the stack, divides the second value by the first, and pushes the quotient onto the stack.
<code>addAll</code>	Pops all values off the stack, adds them, and pushes their sum onto the stack.
<code>multAll</code>	Pops all values off the stack, multiplies them, and pushes their product onto the stack.

Demonstrate the class with a driver program.

8. Dynamic MathStack Template

Currently, the `MathStack` class is derived from the `IntStack` class. Modify it so it is a template, derived from the template you created in Programming Challenge 2 (Dynamic Stack Template).

9. File Reverser

Write a program that opens a text file and reads its contents into a stack of characters. The program should then pop the characters from the stack and save them in a second text file. The order of the characters saved in the second file should be the reverse of their order in the first file.

10. File Filter

Write a program that opens a text file and reads its contents into a queue of characters. The program should then dequeue each character, convert it to uppercase, and store it in a second file.

11. File Compare



Write a program that opens two text files and reads their contents into two separate queues. The program should then determine whether the files are identical by comparing the characters in the queues. When two nonidentical characters are encountered, the program should display a message indicating that the files are not the same. If both queues contain the same set of characters, a message should be displayed indicating that the files are identical.

12. Inventory Bin Stack

Design an inventory class that stores the following members:

<code>serialNum:</code>	An integer that holds a part's serial number.
<code>manufactDate:</code>	A member that holds the date the part was manufactured.
<code>lotNum:</code>	An integer that holds the part's lot number.

The class should have appropriate member functions for storing data into, and retrieving data from, these members.

Next, design a stack class that can hold objects of the class described above. If you wish, you may use the template you designed in Programming Challenge 1 or 2.

Last, design a program that uses the stack class described above. The program should have a loop that asks the user if he or she wishes to add a part to inventory, or take a part from inventory. The loop should repeat until the user is finished.

If the user wishes to add a part to inventory, the program should ask for the serial number, date of manufacture, and lot number. The data should be stored in an inventory object, and pushed onto the stack.

If the user wishes to take a part from inventory, the program should pop the top-most part from the stack and display the contents of its member variables.

When the user finishes the program, it should display the contents of the member values of all the objects that remain on the stack.

13. Inventory Bin Queue

Modify the program you wrote for Programming Challenge 12 (Inventory Bin Stack) so it uses a queue instead of a stack. Compare the order in which the parts are removed from the bin for each program.

14. Balanced Parentheses

A string of characters has balanced parentheses if each right parenthesis occurring in the string is matched with a preceding left parenthesis, in the same way that each right brace in a C++ program is matched with a preceding left brace. Write a program that uses a stack to determine whether a string entered at the keyboard has balanced parentheses.

15. Balanced Multiple Delimiters

A string may use more than one type of delimiter to bracket information into “blocks.” For example, a string may use braces { }, parentheses (), and brackets [] as delimiters. A string is properly delimited if each right delimiter is matched with a preceding left delimiter of the same type in such a way that either the resulting blocks of information are disjoint, or one of them is completely nested within the other. Write a program that uses a single stack to check whether a string containing braces, parentheses, and brackets is properly delimited.

20 Recursion

TOPICS

- | | | | |
|------|---|-------|---|
| 20.1 | Introduction to Recursion | 20.6 | Focus on Problem Solving and Program Design: A Recursive Binary Search Function |
| 20.2 | Solving Problems with Recursion | 20.7 | The Towers of Hanoi |
| 20.3 | Focus on Problem Solving and Program Design: The Recursive gcd Function | 20.8 | Focus on Problem Solving and Program Design: The QuickSort Algorithm |
| 20.4 | Focus on Problem Solving and Program Design: Solving Recursively Defined Problems | 20.9 | Exhaustive Algorithms |
| 20.5 | Focus on Problem Solving and Program Design: Recursive Linked List Operations | 20.10 | Focus on Software Engineering: Recursion versus Iteration |

20.1 Introduction to Recursion

CONCEPT: A recursive function is one that calls itself.

You have seen instances of functions calling other functions. Function A can call function B, which can then call function C. It's also possible for a function to call itself. A function that calls itself is a *recursive function*. Look at this `message` function:

```
void message()
{
    cout << "This is a recursive function.\n";
    message();
}
```

This function displays the string “This is a recursive function.\n”, then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the function? There's no way to stop the recursive calls. This function is like an infinite loop because there is no code to stop it from repeating.



NOTE: The function example `message` will eventually cause the program to crash. Do you remember learning in Chapter 19 that the system stores temporary data on a stack each time a function is called? Eventually, these recursive function calls will use up all available stack memory and cause it to overflow.

Like a loop, a recursive function must have some method to control the number of times it repeats. The following is a modification of the `message` function. It passes an integer argument that holds the number of times the function is to call itself.

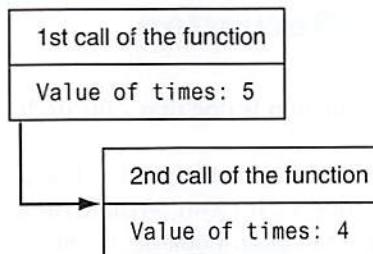
```
void message(int times)
{
    if (times > 0)
    {
        cout << "This is a recursive function.\n";
        message(times - 1);
    }
}
```

This function contains an `if` statement that controls the repetition. As long as the `times` argument is greater than zero, it will display the message and call itself again. Each time it calls itself, it passes `times - 1` as the argument. For example, let's say a program calls the function with the following statement:

```
message(5);
```

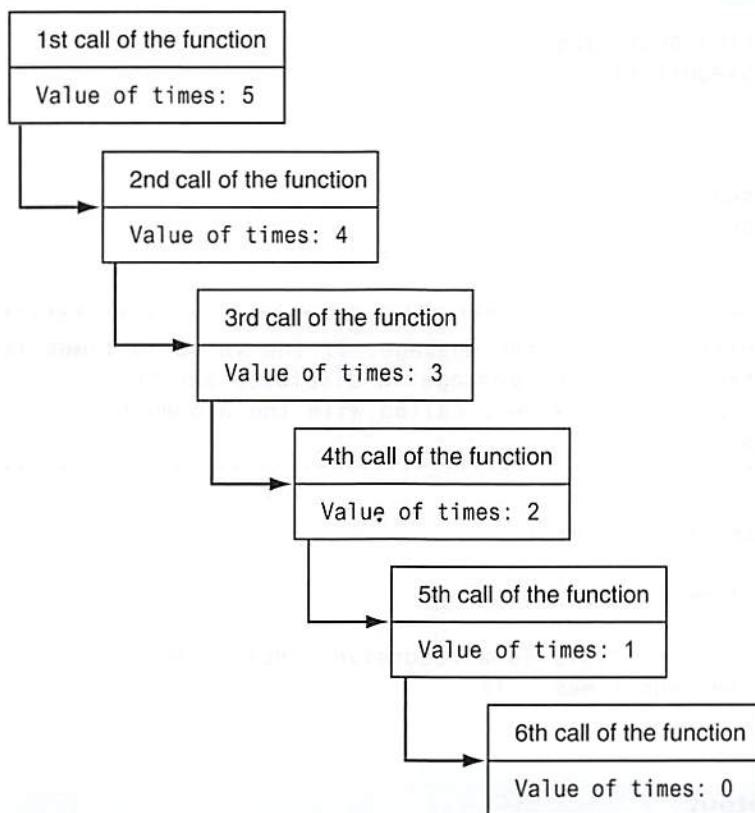
The argument, 5, will cause the function to call itself five times. The first time the function is called, the `if` statement will display the message then call itself with 4 as the argument. Figure 20-1 illustrates this.

Figure 20-1 Recursive function call



The diagram in Figure 20-1 illustrates two separate calls of the `message` function. Each time the function is called, a new instance of the `times` parameter is created in memory. The first time the function is called, the `times` parameter is set to 5. When the function calls itself, a new instance of `times` is created, and the value 4 is passed into it. This cycle repeats until, finally, zero is passed to the function. This is illustrated in Figure 20-2.

As you can see from Figure 20-2, the function is called a total of six times. The first time it is called from `main`, and the other five times it calls itself, so the *depth of recursion* is five. When the function reaches its sixth call, the `times` parameter will be set to 0. At that point, the `if` statement's conditional expression will be false, so the function will return. Control

Figure 20-2 Recursive function calls

of the program will return from the sixth instance of the function to the point in the fifth instance directly after the recursive function call:

```

void message (int times)
{
    if (times > 0
    {
        cout << "This is a recursive function.\n"
        message (times - 1); ← Recursive call
    }
} ← Control returns here from the recursive call,
      causing the function to return.
  
```

Because there are no more statements to be executed after the function call, the fifth instance of the function returns control of the program back to the fourth instance. This repeats until all instances of the function return. Program 20-1 demonstrates the recursive message function.

Program 20-1

```

1 // This program demonstrates a simple recursive function.
2 #include <iostream>
3 using namespace std;
4
  
```

(program continues)

Program 20-1

(continued)

```

5 // Function prototype
6 void message(int);
7
8 int main()
9 {
10     message(5);
11     return 0;
12 }
13
14 //*****
15 // Definition of function message. If the value in times is *
16 // greater than 0, the message is displayed and the           *
17 // function is recursively called with the argument          *
18 // times - 1.                                              *
19 //*****
20
21 void message(int times)
22 {
23     if (times > 0)
24     {
25         cout << "This is a recursive function.\n";
26         message(times - 1);
27     }
28 }
```

Program Output

This is a recursive function.
 This is a recursive function.

To further illustrate the inner workings of this recursive function, let's look at another version of the program. In Program 20-2, a message is displayed each time the function is entered, and another message is displayed just before the function returns.

Program 20-2

```

1 // This program demonstrates a simple recursive function.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototype
6 void message(int);
7
8 int main()
9 {
10     message(5);
11     return 0;
12 }
13
```

```

14 //*****
15 // Definition of function message. If the value in times is *
16 // greater than 0, the message is displayed and the function *
17 // is recursively called with the argument times - 1. *
18 //*****
19
20 void message(int times)
21 {
22     cout << "message called with " << times << " in times.\n";
23
24     if (times > 0)
25     {
26         cout << "This is a recursive function.\n";
27         message(times - 1);
28     }
29
30     cout << "message returning with " << times;
31     cout << " in times.\n";
32 }
```

Program Output

```

message called with 5 in times.
This is a recursive function.
message called with 4 in times.
This is a recursive function.
message called with 3 in times.
This is a recursive function.
message called with 2 in times.
This is a recursive function.
message called with 1 in times.
This is a recursive function.
message called with 0 in times.
message returning with 0 in times.
message returning with 1 in times.
message returning with 2 in times.
message returning with 3 in times.
message returning with 4 in times.
message returning with 5 in times.
```

20.2 Solving Problems with Recursion

CONCEPT: A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

Programs 20-1 and 20-2 in the previous section show simple demonstrations of *how* a recursive function works. But these examples don't show us *why* we would want to write a recursive function. Recursion can be a powerful tool for solving repetitive problems and is an important topic in upper-level computer science courses. What might not be clear to you yet is how to use recursion to solve a problem.

First, it should be noted that recursion is never absolutely required to solve a problem. Any problem that can be solved recursively can also be solved iteratively, with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because a function call requires several actions to be performed by the C++ runtime system. These actions include allocating memory for parameters and local variables and storing the address of the program location where control returns after the function terminates. These actions, which are sometimes referred to as *overhead*, take place with each function call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with iteration. Where an iterative algorithm might result in faster execution time, the programmer might be able to design a recursive algorithm faster.

In general, a recursive function works like this:

- If the problem can be solved now, without recursion, then the function solves it and returns.
- If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem.



In order to apply this approach, we first identify at least one case in which the problem can be solved without recursion. This is known as the *base case*. Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case*. In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached, and the recursion will stop.

Example: Using Recursion to Calculate the Factorial of a Number

Let's take an example from mathematics to examine an application of recursion. In mathematics, the notation $n!$ represents the factorial of the number n . The factorial of a non-negative number can be defined by the following rules:

$$\text{If } n = 0 \text{ then } n! = 1$$

$$\text{If } n > 0 \text{ then } n! = 1 \times 2 \times 3 \times \dots \times n$$

Let's replace the notation $n!$ with $\text{factorial}(n)$, which looks a bit more like computer code, and rewrite these rules as

$$\text{If } n = 0 \text{ then } \text{factorial}(n) = 1$$

$$\text{If } n > 0 \text{ then } \text{factorial}(n) = 1 \times 2 \times 3 \times \dots \times n$$

These rules state that when n is 0, its factorial is 1. When n is greater than 0, its factorial is the product of all the positive integers from 1 up to n . For instance, $\text{factorial}(6)$ is calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$.

When designing a recursive algorithm to calculate the factorial of any number, we first identify the base case, which is the part of the calculation we can solve without recursion. That is the case where n is equal to 0.

$$\text{If } n = 0 \text{ then } \text{factorial}(n) = 1$$

This tells us how to solve the problem when n is equal to 0, but what do we do when n is greater than 0? That is the recursive case, or the part of the problem we use recursion to solve. This is how we express it:

If $n > 0$ then $\text{factorial}(n) = n \times \text{factorial}(n - 1)$

This states if n is greater than 0, the factorial of n is n times the factorial of $n - 1$. Notice how the recursive call works on a reduced version of the problem, $n - 1$. So, our recursive rule for calculating the factorial of a number might look like this:

If $n = 0$ then $\text{factorial}(n) = 1$

If $n > 0$ then $\text{factorial}(n) = n \times \text{factorial}(n - 1)$

The following pseudocode shows how we might implement the factorial algorithm as a recursive function:

```
factorial(n)
  If n is 0 then
    return 1
  else
    return n times the factorial of n - 1
end factorial
```

Here is the C++ code for such a function:

```
int factorial(int n)
{
    if (n == 0)
        return 1; // Base case
    else
        return n * factorial(n - 1); // Recursive case
}
```

Program 20-3 demonstrates the recursive factorial function.

Program 20-3

```
1 // This program demonstrates a recursive function to
2 // calculate the factorial of a number.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype
7 int factorial(int);
8
9 int main()
10 {
11     int number;
12 }
```

(program continues)

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our factorial function, the value of the parameter *n* gets closer to 0 with each recursive call. When the parameter reaches 0, the function returns a value of 0. Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our factorial function, the value of the parameter *n* gets closer to 0 with each recursive call. The base case does not require recursion, so it stops the chain of recursive calls. This diagram illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually the recursion has to stop in order for a solution to be reached. If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

The diagram in Figure 20-3 illustrates the value of *n* and the return value during each call of the function.

Although this is a return statement, it does not immediately return. Before the return value can be determined, the value of *factorial(n - 1)* must be determined. The factorial function is called recursively until the fifth call, in which the *n* parameter will be set to zero.

The diagram in Figure 20-3 illustrates the value of *n* and the return value during each call of the function.

In the example run of the program, the factorial function is called with the argument 4 passed into *n*. Because *n* is not equal to 0, the if statement's else clause executes the following statement, in line 34:

```
return n * factorial(n - 1);
```

In the example run of the program, the factorial function is called with the argument 4

Program Output with Example Input shown in Bold

Enter an integer value and I will display its factorial: **4** Enter

The factorial of 4 is 24

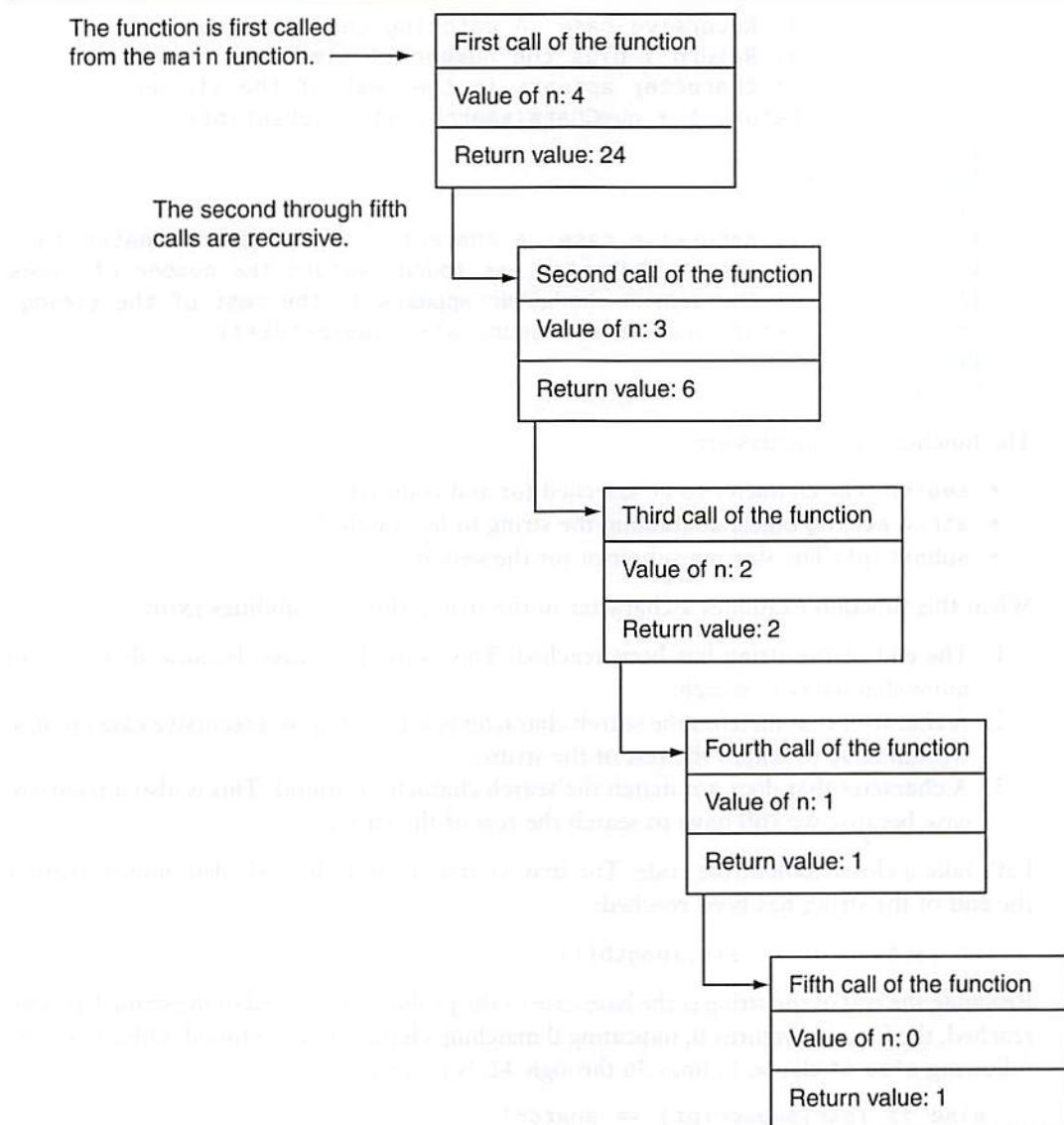
```

35 }
34 return n * factorial(n - 1); // Recursive case
33 else
32 return 1; // Base case
31 if (n == 0)
30 {
29 int factorial(int n)
28
27 // *****
26 // the factorial of the parameter n.
25 // Definition of factorial. A recursive function to calculate *
24 // *****
23 }
22 {
21 cout << "The factorial of " << number << " is ";
20 cout << "Enter an integer value and I will display\n";
19 cout << "Display the factorial of the number.
18 // Display the factorial of the number.
17 cin >> number;
16 cout << "its factorial: ";
15 cout << "Enter an integer value and I will display\n";
14 // Get a number from the user.

```

(continued)

Program 20-3

Figure 20-3 Recursive function calls

Example: Using Recursion to Count Characters

Let's look at another simple example of recursion. The following function counts the number of times a specific character appears in a string. The line numbers are from Program 20-4, which we will examine momentarily.

```

29 int numChars(char search, string str, int subscript)
30 {
31     if (subscript >= str.length())
32     {
33         // Base case: The end of the string is reached.
34         return 0;
35     }

```

```

36     else if (str[subscript] == search)
37     {
38         // Recursive case: A matching character was found.
39         // Return 1 plus the number of times the search
40         // character appears in the rest of the string.
41         return 1 + numChars(search, str, subscript+1);
42     }
43     else
44     {
45         // Recursive case: A character that does not match the
46         // search character was found. Return the number of times
47         // the search character appears in the rest of the string.
48         return numChars(search, str, subscript+1);
49     }
50 }
```

The function's parameters are

- **search:** The character to be searched for and counted
- **str:** a **String** object containing the string to be searched
- **subscript:** The starting subscript for the search

When this function examines a character in the string, three possibilities exist:

1. The end of the string has been reached. This is the base case, because there are no more characters to search.
2. A character that matches the search character is found. This is a recursive case because we still have to search the rest of the string.
3. A character that does not match the search character is found. This is also a recursive case because we still have to search the rest of the string.

Let's take a closer look at the code. The first **if** statement, in line 31, determines whether the end of the string has been reached:

```
if (subscript >= str.length())
```

Reaching the end of the string is the base case of the problem. If the end of the string has been reached, the function returns 0, indicating 0 matching characters were found. Otherwise, the following **else if** clause, in lines 36 through 42, is executed:

```

else if (str[subscript] == search)
{
    // Recursive case: A matching character was found.
    // Return 1 plus the number of times the search
    // character appears in the rest of the string.
    return 1 + numChars(search, str, subscript+1);
}
```

If **str[subscript]** contains the search character, then we have found one matching character. But because we have not reached the end of the string, we must continue to search the rest of the string for more matching characters. So, at this point, the function performs a recursive call. The **return** statement returns 1 plus the number of times the search character appears in the string, starting at **subscript+1**. In essence, this statement returns 1 plus the number of times the search character appears in the rest of the string.

Finally, if `str[subscript]` does not contain the search character, the following `else` clause in lines 43 through 49 is executed:

```

    else
    {
        // Recursive case: A character that does not match the
        // search character was found. Return the number of times
        // the search character appears in the rest of the string.
        return numChars(search, str, subscript+1);
    }
}

```

The `return` statement in line 48 makes a recursive call to search the remainder of the string. In essence, this code returns the number of times the search character appears in the rest of the string. Program 20-4 demonstrates the function.

Program 20-4

```

1 // This program demonstrates a recursive function for counting
2 // the number of times a character appears in a string.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 // Function prototype
8 int numChars(char, string, int);
9
10 int main()
11 {
12     string str = "abcddef";
13
14     // Display the number of times the letter
15     // 'd' appears in the string.
16     cout << "The letter d appears "
17         << numChars('d', str, 0) << " times.\n";
18
19     return 0;
20 }
21
22 //*****
23 // Function numChars. This recursive function      *
24 // counts the number of times the character      *
25 // search appears in the string str. The search   *
26 // begins at the subscript stored in subscript.   *
27 //*****
28
29 int numChars(char search, string str, int subscript)
30 {
31     if (subscript >= str.length())
32     {
33         // Base case: The end of the string is reached.
34         return 0;
35     }
36     else if (str[subscript] == search)

```

(program continues)

Program 20-4

(continued)

```

37      {
38          // Recursive case: A matching character was found.
39          // Return 1 plus the number of times the search
40          // character appears in the rest of the string.
41          return 1 + numChars(search, str, subscript+1);
42      }
43      else
44      {
45          // Recursive case: A character that does not match the
46          // search character was found. Return the number of times
47          // the search character appears in the rest of the string.
48          return numChars(search, str, subscript+1);
49      }
50  }
```

Program Output

The letter d appears 4 times.

Direct and Indirect Recursion

The examples we have discussed so far show recursive functions that directly call themselves. This is known as *direct recursion*. There is also the possibility of creating *indirect recursion* in a program. This occurs when function A calls function B, which in turn calls function A. There can even be several functions involved in the recursion. For example, function A could call function B, which could call function C, which calls function A.

**Checkpoint**

- 20.1 What happens if a recursive function never returns?
- 20.2 What is a recursive function's base case?
- 20.3 What will the following program display?

```

#include <iostream>
using namespace std;

// Function prototype
void showMe(int arg);

int main()
{
    int num = 0;
    showMe(num);
    return 0;
}

void showMe(int arg)
{
    if (arg < 10)
        showMe(++arg);
    else
        cout << arg << endl;
}
```

- 20.4 What is the difference between direct and indirect recursion?

20.3**Focus on Problem Solving and Program Design:
The Recursive gcd Function**

CONCEPT: The `gcd` function uses recursion to find the greatest common divisor (GCD) of two numbers.

Our next example of recursion is the calculation of the greatest common divisor, or GCD, of two numbers. Using Euclid's algorithm, the GCD of two positive integers, x and y , is:

$$\begin{aligned} \text{gcd}(x, y) &= y; && \text{if } y \text{ divides } x \text{ evenly} \\ \text{gcd}(y, \text{remainder of } x/y); && \text{otherwise} \end{aligned}$$

The definition above states the GCD of x and y is y if x/y has no remainder. Otherwise, the answer is the GCD of y and the remainder of x/y . Program 20-5 shows the recursive C++ implementation.

Program 20-5

```

1 // This program demonstrates a recursive function to calculate
2 // the greatest common divisor (gcd) of two numbers.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype
7 int gcd(int, int);
8
9 int main()
10 {
11     int num1, num2;
12
13     // Get two numbers.
14     cout << "Enter two integers: ";
15     cin >> num1 >> num2;
16
17     // Display the GCD of the numbers.
18     cout << "The greatest common divisor of " << num1;
19     cout << " and " << num2 << " is ";
20     cout << gcd(num1, num2) << endl;
21
22     return 0;
23 }
24 //*****
25 // Definition of gcd. This function uses recursion to
26 // calculate the greatest common divisor of two integers,
27 // passed into the parameters x and y.
28 //*****
29
30 int gcd(int x, int y)

```

(program continues)

Program 20-5 (continued)

```

31  {
32      if (x % y == 0)
33          return y;           // Base case
34      else
35          return gcd(y, x % y); // Recursive case
36  }

```

Program Output with Example Input Shown in BoldEnter two integers: **49 28** **Enter**

The greatest common divisor of 49 and 28 is 7

20.4 Focus on Problem Solving and Program Design: Solving Recursively Defined Problems

CONCEPT: Some mathematical problems are designed for a recursive solution.

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of *Fibonacci numbers*. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Notice after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as

$$F_0 = 0$$

$$F_1 = 1$$

$$F_N = F_{N-1} + F_{N-2} \text{ for } N \geq 2.$$

A recursive C++ function to calculate the n th number in the Fibonacci series is shown here:

```

int fib(int n)
{
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}

```

The function is demonstrated in Program 20-6, which displays the first 10 numbers in the Fibonacci series.

Program 20-6

```

1 // This program demonstrates a recursive function
2 // that calculates Fibonacci numbers.
3 #include <iostream>
4 using namespace std;
5

```

```

6 // Function prototype
7 int fib(int);
8
9 int main()
10 {
11     cout << "The first 10 Fibonacci numbers are:\n";
12     for (int x = 0; x < 10; x++)
13         cout << fib(x) << " ";
14     cout << endl;
15     return 0;
16 }
17
18 //*****
19 // Function fib. Accepts an int argument *
20 // in n. This function returns the nth   *
21 // Fibonacci number.                      *
22 //*****
23
24 int fib(int n)
25 {
26     if (n <= 0)
27         return 0;           // Base case
28     else if (n == 1)
29         return 1;           // Base case
30     else
31         return fib(n - 1) + fib(n - 2); // Recursive case
32 }

```

Program Output

The first 10 Fibonacci numbers are:
0 1 1 2 3 5 8 13 21 34

Another such example is Ackermann's function. A Programming Challenge at the end of this chapter will ask you to write a recursive function that calculates Ackermann's function.

20.5

Focus on Problem Solving and Program Design: Recursive Linked List Operations

CONCEPT: Recursion can be used to traverse the nodes in a linked list.

Recall in Chapter 18, we discussed a class named `NumberList` that holds a linked list of `double` values. In this section, we will modify the class by adding recursive member functions. The functions will use recursion to traverse the linked list and perform the following operations:

- Count the number of nodes in the list.

To count the number of nodes in the list by recursion, we introduce two new member functions: `numNodes` and `countNodes`. `countNodes` is a private member function that uses recursion, and `numNodes` is the public interface that calls it.

- Display the value of the list nodes in reverse order.

To display the nodes in the list in reverse order, we introduce two new member functions: `displayBackwards` and `showReverse`. `showReverse` is a private member function that uses recursion, and `displayBackwards` is the public interface that calls it.

The class declaration, which is saved in `NumberList.h`, is shown here:

```

1 // Specification file for the NumberList class
2 #ifndef NUMBERLIST_H
3 #define NUMBERLIST_H
4
5 class NumberList
6 {
7 private:
8     // Declare a structure for the list
9     struct ListNode
10    {
11        double value;
12        struct ListNode *next;
13    };
14
15    ListNode *head;      // List head pointer
16
17    // Private member functions
18    int countNodes(ListNode *) const;
19    void showReverse(ListNode *) const;
20
21 public:
22    // Constructor
23    NumberList()
24        { head = nullptr; }
25
26    // Destructor
27    ~NumberList();
28
29    // Linked List Operations
30    void appendNode(double);
31    void insertNode(double);
32    void deleteNode(double);
33    void displayList() const;
34    int numNodes() const
35        { return countNodes(head); }
36    void displayBackwards() const
37        { showReverse(head); }
38    };
39 #endif

```

Counting the Nodes in the List

The `numNodes` function is declared inline. It simply calls the `countNodes` function and passes the `head` pointer as an argument. (Because the `head` pointer, which is private, must be passed to `countNodes`, the `numNodes` function is needed as an interface.)

The function definition for `countNodes` is shown here:

```

173 int NumberList::countNodes(ListNode *nodePtr) const
174 {
175     if (nodePtr != nullptr)
176         return 1 + countNodes(nodePtr->next);
177     else
178         return 0;
179 }
```

The function's recursive logic can be expressed as:

```

If the current node has a value
    Return 1 + the number of the remaining nodes
Else
    Return 0
End If
```

Program 20-7 demonstrates the function.

Program 20-7

```

1 // This program counts the nodes in a list.
2 #include <iostream>
3 #include "NumberList.h"
4 using namespace std;
5
6 int main()
7 {
8     const int MAX = 10; // Maximum number of values
9
10    // Define a NumberList object.
11    NumberList list;
12
13    // Build the list with a series of numbers.
14    for (int x = 0; x < MAX; x++)
15        list.insertNode(x);
16
17    // Display the number of nodes in the list.
18    cout << "The number of nodes is "
19        << list.numNodes() << endl;
20
21 }
```

Program Output

The number of nodes is 10

Displaying List Nodes in Reverse Order

The technique for displaying the list nodes in reverse order is designed like the node counting procedure: A public member function, which serves as an interface, passes the head pointer to a private member function. The public `displayBackwards` function, declared

inline, is the interface. It calls the `showReverse` function and passes the `head` pointer as an argument. The function definition for `showReverse` is shown here:

```

187 void NumberList::showReverse(ListNode *nodePtr) const
188 {
189     if (nodePtr != nullptr)
190     {
191         showReverse(nodePtr->next);
192         cout << nodePtr->value << " ";
193     }
194 }
```

The base case for the function is `nodePtr` being set to `nullptr`. When this is true, the function has reached the last node in the list, so it returns. It is not until this happens that any instances of the `cout` statement execute. The instance of the function whose `nodePtr` variable points to the last node in the list will be the first to execute the `cout` statement. It will then return, and the previous instance of the function will execute its `cout` statement. This repeats until all the instances of the function have returned.

The modified class declaration is stored in `NumberList.h`, and its member function implementation is in `NumberList.cpp`. The remainder of the class implementation is unchanged from Chapter 18, so it is not shown here. Program 20-8 demonstrates the function.

Program 20-8

```

1 // This program demonstrates the recursive function
2 // for displaying the list's nodes in reverse.
3 #include <iostream>
4 #include "NumberList.h"
5 using namespace std;
6
7 int main()
8 {
9     const double MAX = 10.0; // Upper limit of values
10
11    // Create a NumberList object.
12    NumberList list;
13
14    // Add a series of numbers to the list.
15    for (double x = 1.5; x < MAX; x += 1.1)
16        list.appendNode(x);
17
18    // Display the values in the list.
19    cout << "Here are the values in the list:\n";
20    list.displayList();
21
22    // Display the values in reverse order.
23    cout << "Here are the values in reverse order:\n";
24    list.displayBackwards();
25
26 }
```

Program Output

Here are the values in the list:

1.5
2.6
3.7
4.8
5.9
7
8.1
9.2

Here are the values in reverse order:

9.2 8.1 7 5.9 4.8 3.7 2.6 1.5

20.6 Focus on Problem Solving and Program Design: A Recursive Binary Search Function

CONCEPT: The binary search algorithm can be defined as a recursive function.

In Chapter 8, you learned about the binary search algorithm and saw an iterative example written in C++. The binary search algorithm can also be implemented recursively. For example, the procedure can be expressed as

If array[middle] equals the search value, then the value is found.

Else, if array[middle] is less than the search value, perform a binary search on the upper half of the array.

Else, if array[middle] is greater than the search value, perform a binary search on the lower half of the array.

The recursive binary search algorithm is an example of breaking a problem down into smaller pieces until it is solved. A recursive binary search function is shown here:

```
int binarySearch(int array[], int first, int last, int value)
{
    int middle;      // Midpoint of search
    if (first > last)
        return -1;
    middle = (first + last) / 2;
    if (array[middle] == value)
        return middle;
    if (array[middle] < value)
        return binarySearch(array, middle+1, last, value);
    else
        return binarySearch(array, first, middle-1, value);
}
```

The first parameter, `array`, is the array to be searched. The next parameter, `first`, holds the subscript of the first element in the search range (the portion of the array to be searched). The next parameter, `last`, holds the subscript of the last element in the search range. The last parameter, `value`, holds the value to be searched for. Like the iterative version, this function returns the subscript of the value if it is found, or `-1` if the value is not found. Program 20-9 demonstrates the function.

Program 20-9

```

1 // This program demonstrates the recursive binarySearch function.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototype
6 int binarySearch(int [], int, int, int);
7
8 const int SIZE = 20; // Array size
9
10 int main()
11 {
12     // Define an array of employee ID numbers
13     int tests[SIZE] = {101, 142, 147, 189, 199, 207, 222,
14         234, 289, 296, 310, 319, 388, 394,
15         417, 429, 447, 521, 536, 600};
16     int empID;    // To hold an ID number
17     int results; // To hold the search results
18
19     // Get an employee ID number to search for.
20     cout << "Enter the Employee ID you wish to search for: ";
21     cin >> empID;
22
23     // Search for the ID number in the array.
24     results = binarySearch(tests, 0, SIZE - 1, empID);
25
26     // Display the results of the search.
27     if (results == -1)
28         cout << "That number does not exist in the array.\n";
29     else
30     {
31         cout << "That ID is found at element " << results;
32         cout << " in the array\n";
33     }
34     return 0;
35 }
36
37 //*****
38 // The binarySearch function performs a recursive binary search *
39 // on a range of elements of an integer array passed into the   *
40 // parameter array. The parameter first holds the subscript of   *
41 // the range's starting element, and last holds the subscript   *
42 // of the range's last element. The parameter value holds the   *
43 // search value. If the search value is found, its array       *
44 // subscript is returned. Otherwise, -1 is returned indicating *
45 // the value was not in the array.                            *
46 //*****
47
48 int binarySearch(int array[], int first, int last, int value)
49 {
50     int middle; // Midpoint of search
51
52     if (first > last)
53         return -1;
54     middle = (first + last)/2;

```

```

55     if (array[middle]==value)
56         return middle;
57     if (array[middle]<value)
58         return binarySearch(array, middle+1, last,value);
59     else
60         return binarySearch(array, first,middle-1,value);
61 }

```

Program Output with Example Input Shown in Bold

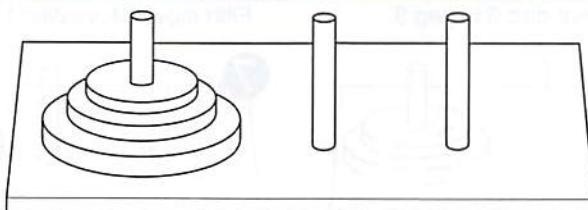
Enter the Employee ID you wish to search for: **521** That ID is found at element 17 in the array

20.7 The Towers of Hanoi

CONCEPT: The repetitive steps involved in solving the Towers of Hanoi game can be easily implemented in a recursive algorithm.

The Towers of Hanoi is a mathematical game often used in computer science textbooks to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in Figure 20-4.

Figure 20-4 The pegs and discs in the Towers of Hanoi game



Notice the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend in which a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The job of the monks is to move the discs from the first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disc may be moved at a time.
- A disc cannot be placed on top of a smaller disc.
- All discs must be stored on a peg except while being moved.

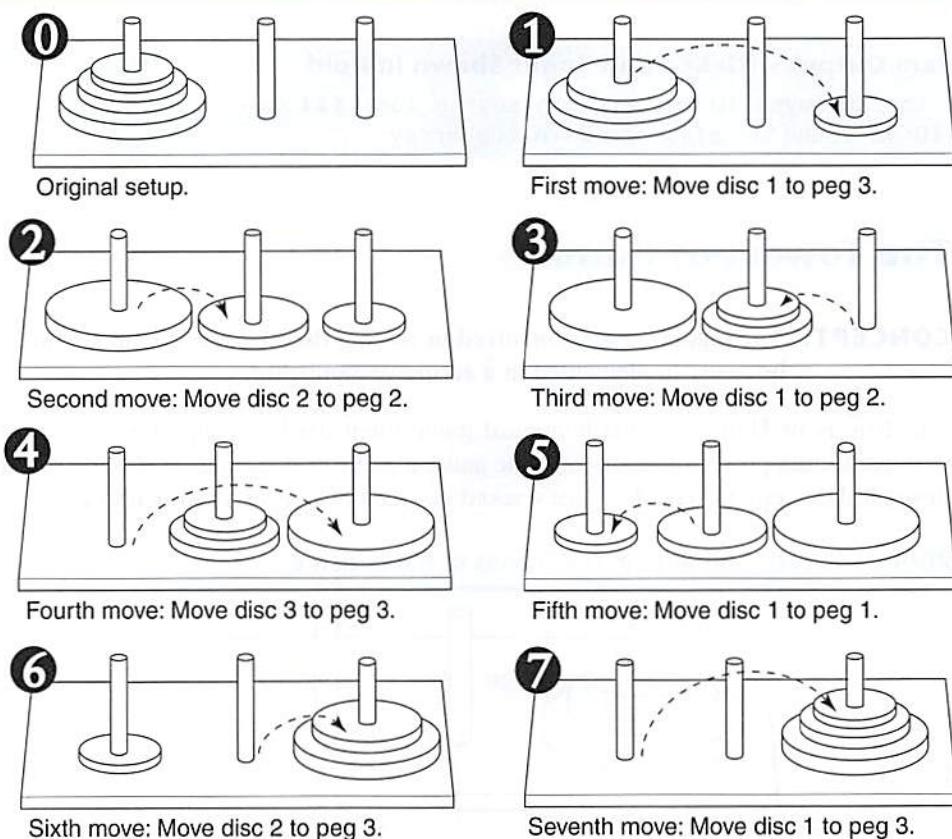
According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you only have one disc, the solution to the game is simple: move the disc from peg 1 to peg 3. If you have two discs, the solution requires three moves:

- Move disc 1 to peg 2.
- Move disc 2 to peg 3.
- Move disc 1 to peg 3.

Notice this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increases. To move three discs requires the seven moves shown in Figure 20-5.

Figure 20-5 Moving three discs



The following statement describes the overall solution to the problem:

Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.

The following pseudocode algorithm can be used as the basis of a recursive function that simulates the solution to the game. Notice in this algorithm we use the variables A, B, and C to hold peg numbers.

To move n discs from peg A to peg C, using peg B as a temporary peg

If n > 0 Then

Move n - 1 discs from peg A to peg B, using peg C as a temporary peg

Move the remaining disc from the peg A to peg C

Move n - 1 discs from peg B to peg C, using peg A as a temporary peg

End If

The base case for the algorithm is reached when there are no more discs to move. The following code is for a function that implements this algorithm. Note the function does not actually move anything, but displays instructions indicating all of the disc moves to make.

```

void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
    if (num > 0)
    {
        moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
        cout << "Move a disc from peg " << fromPeg
             << " to peg " << toPeg << endl;
        moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
    }
}

```

This function accepts arguments into the following three parameters:

- | | |
|---------|---------------------------------------|
| num | The number of discs to move. |
| fromPeg | The peg from which to move the discs. |
| toPeg | The peg to which to move the discs. |
| tempPeg | The peg to use as a temporary peg. |

If num is greater than 0, then there are discs to move. The first recursive call is

```
moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
```

This statement is an instruction to move all but one disc from fromPeg to tempPeg, using toPeg as a temporary peg. The next statement is

```
cout << "Move a disc from peg " << fromPeg
     << " to peg " << toPeg << endl;
```

This simply displays a message indicating that a disc should be moved from fromPeg to toPeg. Next, another recursive call is executed:

```
moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
```

This statement is an instruction to move all but one disc from tempPeg to toPeg, using fromPeg as a temporary peg. Program 20-10 demonstrates this function.

Program 20-10

```

1 // This program displays a solution to the Towers of
2 // Hanoi game.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype
7 void moveDiscs(int, int, int, int);
8
9 int main()
10 {
11     const int NUM_DISCS = 3; // Number of discs to move
12     const int FROM_PEG = 1; // Initial "from" peg
13     const int TO_PEG = 3; // Initial "to" peg
14     const int TEMP_PEG = 2; // Initial "temp" peg
15
16     // Play the game.
17     moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
18     cout << "All the pegs are moved!\n";

```

(program continues)

Program 20-10 (continued)

```

19     return 0;
20 }
21
22 //***** The moveDiscs function displays a disc move in ****
23 // The Towers of Hanoi game. *
24 // The parameters are: *
25 // num: The number of discs to move. *
26 // fromPeg: The peg to move from. *
27 // toPeg: The peg to move to. *
28 // tempPeg: The temporary peg. *
29 //***** *
30 //*****
31
32 void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
33 {
34     if (num > 0)
35     {
36         moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
37         cout << "Move a disc from peg " << fromPeg
38             << " to peg " << toPeg << endl;
39         moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
40     }
41 }
```

Program Output

```

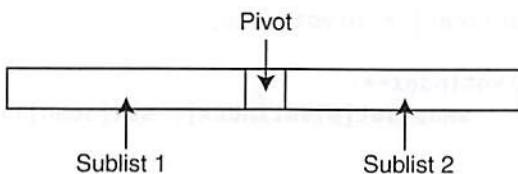
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```

20.8**Focus on Problem Solving and Program Design:
The QuickSort Algorithm**

CONCEPT: The QuickSort algorithm uses recursion to efficiently sort a list.

The QuickSort algorithm is a popular general-purpose sorting routine developed in 1960 by C.A.R. Hoare. It can be used to sort lists stored in arrays or linear linked lists. It sorts a list by dividing it into two sublists. Between the sublists is a selected value known as the *pivot*. This is illustrated in Figure 20-6.

Notice in the figure sublist 1 is positioned to the left of (before) the pivot, and sublist 2 is positioned to the right of (after) the pivot. Once a pivot value has been selected, the algorithm exchanges the other values in the list until all the elements in sublist 1 are less than the pivot, and all the elements in sublist 2 are greater than the pivot.

Figure 20-6 A partitioned data set

Once this is done, the algorithm repeats the procedure on sublist 1, then on sublist 2. The recursion stops when there is only one element in a sublist. At that point, the original list is completely sorted.

The algorithm is coded primarily in two functions: `quickSort` and `partition`. `quickSort` is a recursive function. Its pseudocode is shown here:

```
quickSort:
  If Starting Index < Ending Index
    Partition the List around a Pivot
    quickSort Sublist 1
    quickSort Sublist 2
  End If
```

Here is the C++ code for the `quickSort` function:

```
void quickSort(int set[], int start, int end)
{
    int pivotPoint;
    if (start < end)
    {
        // Get the pivot point.
        pivotPoint = partition(set, start, end);
        // Sort the first sublist.
        quickSort(set, start, pivotPoint - 1);
        // Sort the second sublist.
        quickSort(set, pivotPoint + 1, end);
    }
}
```

This version of `quickSort` works with an array of integers. Its first argument is the array holding the list that is to be sorted. The second and third arguments are the starting and ending subscripts of the list.

The subscript of the pivot element is returned by the `partition` function. `partition` not only determines which element will be the pivot, but also controls the rearranging of the other values in the list. Our version of this function selects the element in the middle of the list as the pivot, then scans the remainder of the list searching for values less than the pivot.

The code for the `partition` function is shown here:

```
int partition(int set[], int start, int end)
{
    int pivotValue, pivotIndex, mid;
    mid = (start + end) / 2;
    swap(set[start], set[mid]);
    pivotIndex = start;
    pivotValue = set[start];
```

```

        for (int scan = start + 1; scan <= end; scan++)
        {
            if (set[scan] < pivotValue)
            {
                pivotIndex++;
                swap(set[pivotIndex], set[scan]);
            }
        }
        swap(set[start], set[pivotIndex]);
        return pivotIndex;
    }
}

```



NOTE: The `partition` function does not initially sort the values into their final order. Its job is only to move the values that are less than the pivot to the pivot's left, and move the values that are greater than the pivot to the pivot's right. As long as that condition is met, they may appear in any order. The ultimate sorting order of the entire list is achieved cumulatively, though the recursive calls to `quickSort`.

There are many different ways of partitioning the list. As previously stated, the method shown in the function above selects the middle value as the pivot. That value is then moved to the beginning of the list (by exchanging it with the value stored there). This simplifies the next step, which is to scan the list.

A `for` loop scans the remainder of the list, and when an element is found whose value is less than the pivot, that value is moved to a location left of the pivot point.

A third function, `swap`, is used to swap the values found in any two elements of the list. The function is shown below:

```

void swap(int &value1, int &value2)
{
    int temp = value1;
    value1 = value2;
    value2 = temp;
}

```

Program 20-11 demonstrates the QuickSort algorithm.

Program 20-11

```

1 // This program demonstrates the QuickSort Algorithm.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototypes
6 void quickSort(int [], int, int);
7 int partition(int [], int, int);
8 void swap(int &, int &);
9
10 int main()
11 {
12     const int SIZE = 10; // Array size
13     int count;           // Loop counter
14     int array[SIZE] = {7, 3, 9, 2, 0, 1, 8, 4, 6, 5};

```

```

15     // Display the array contents.
16     for (count = 0; count < SIZE; count++)
17         cout << array[count] << " ";
18     cout << endl;
19
20
21     // Sort the array.
22     quickSort(array, 0, SIZE - 1);
23
24     // Display the array contents.
25     for (count = 0; count < SIZE; count++)
26         cout << array[count] << " ";
27     cout << endl;
28     return 0;
29 }
30
31 //*****
32 // quickSort uses the quicksort algorithm to      *
33 // sort set, from set[start] through set[end].   *
34 //*****
35
36 void quickSort(int set[], int start, int end)
37 {
38     int pivotPoint;
39
40     if (start < end)
41     {
42         // Get the pivot point.
43         pivotPoint = partition(set, start, end);
44         // Sort the first sublist.
45         quickSort(set, start, pivotPoint - 1);
46         // Sort the second sublist.
47         quickSort(set, pivotPoint + 1, end);
48     }
49 }
50
51 //*****
52 // partition selects the value in the middle of the      *
53 // array set as the pivot. The list is rearranged so      *
54 // all the values less than the pivot are on its left      *
55 // and all the values greater than pivot are on its right. *
56 //*****
57
58 int partition(int set[], int start, int end)
59 {
60     int pivotValue, pivotIndex, mid;
61
62     mid = (start + end) / 2;
63     swap(set[start], set[mid]);
64     pivotIndex = start;
65     pivotValue = set[start];
66     for (int scan = start + 1; scan <= end; scan++)
67     {
68         if (set[scan] < pivotValue)
69         {

```

(program continues)

Program 20-11 (continued)

```

70         pivotIndex++;
71         swap(set[pivotIndex], set[scan]);
72     }
73 }
74 swap(set[start], set[pivotIndex]);
75 return pivotIndex;
76 }
77
78 //*****
79 // swap simply exchanges the contents of *
80 // value1 and value2. *
81 //*****
82
83 void swap(int &value1, int &value2)
84 {
85     int temp = value1;
86
87     value1 = value2;
88     value2 = temp;
89 }
```

Program Output

```

7 3 9 2 0 1 8 4 6 5
0 1 2 3 4 5 6 7 8 9
```

20.9**Exhaustive Algorithms**

CONCEPT: An exhaustive algorithm is one that finds a best combination of items by looking at all the possible combinations.

Recursion is helpful if you need to examine many possible combinations and identify the best combination. For example, consider all the different ways you can make change for \$1.00 using our system of coins:

- 1 dollar piece, or
- 2 fifty-cent pieces, or
- 4 quarters, or
- 1 fifty-cent piece and 2 quarters, or
- 3 quarters, 2 dimes, and 1 nickel, or
- ... *there are many more possibilities.*

Although there are many ways to make change for \$1.00, some ways are better than others. For example, you would probably rather give a single dollar piece instead of 100 pennies.

An algorithm that looks at all the possible combinations of items in order to find the best combination of items is called an *exhaustive algorithm*. Program 20-12 presents a recursive function that exhaustively tries all the possible combinations of coins. The program then displays the total number of combinations that can be used to make the specified change, and the best combination of coins.

Program 20-12

```

1 // This program demonstrates a recursive function that exhaustively
2 // searches through all possible combinations of coin values to find
3 // the best way to make change for a specified amount.
4 #include <iostream>
5 using namespace std;
6
7 // Constants
8 const int MAX_COINS_CHANGE = 100; // Max number of coins to give in change
9 const int MAX_COIN_VALUES = 6; // Max number of coin values
10 const int NO_SOLUTION = INT_MAX; // Indicates no solution
11
12 // Function prototype
13 void makeChange(int, int[], int);
14
15 // coinValues - global array of coin values to choose from
16 int coinValues[MAX_COIN_VALUES] = {100, 50, 25, 10, 5, 1 };
17
18 // bestCoins - global array of best coins to make change with
19 int bestCoins[MAX_COINS_CHANGE];
20
21 // Global variables
22 int numBestCoins = NO_SOLUTION, // Number of coins in bestCoins
23     numSolutions = 0,           // Number of ways to make change
24     numCoins;                 // Number of allowable coins
25
26
27 int main()
28 {
29     int coinsUsed[MAX_COINS_CHANGE], // List of coins used
30     numCoinsUsed = 0,              // The number of coins used
31     amount;                      // The amount to make change for
32
33     // Display the possible coin values.
34     cout << "Here are the valid coin values, in cents: ";
35     for (int index = 0; index < 5; index++)
36         cout << coinValues[index] << " ";
37     cout << endl;
38
39     // Get input from the user.
40     cout << "Enter the amount of cents (as an integer) "
41         << "to make change for: ";
42     cin >> amount;
43     cout << "What is the maximum number of coins to give as change? ";
44     cin >> numCoins;
45
46     // Call the recursive function.
47     makeChange(numCoins, amount, coinsUsed, numCoinsUsed);
48
49     // Display the results.
50     cout << "Number of possible combinations: " << numSolutions << endl;
51     cout << "Best combination of coins:\n";
52     if (numBestCoins == NO_SOLUTION)
53         cout << "\tNo solution\n";

```

(program continues)

Program 20-12 (continued)

```

54     else
55     {
56         for (int count = 0; count < numBestCoins; count++)
57             cout << bestCoins[count] << " ";
58     }
59     cout << endl;
60     return 0;
61 }
62 ****
63 // Function makeChange. This function uses the following parameters: *
64 // coinsLeft - The number of coins left to choose from. *
65 // amount - The amount to make change for. *
66 // coinsUsed - An array that contains the coin values used so far. *
67 // numCoinsUsed - The number of values in the coinsUsed array. *
68 // *
69 //
70 // This recursive function finds all the possible ways to make change *
71 // for the value in amount. The best combination of coins is stored in *
72 // the array bestCoins. *
73 ****
74
75 void makeChange(int coinsLeft, int amount, int coinsUsed[],
76                 int numCoinsUsed)
77 {
78     int coinPos, // To calculate array position of coin being used
79     count; // Loop counter
80
81     if (coinsLeft == 0) // If no more coins are left
82         return;
83     else if (amount < 0) // If amount to make change for is negative
84         return;
85     else if (amount == 0) // If solution is found
86     {
87         // Store as bestCoins if best
88         if (numCoinsUsed < numBestCoins)
89         {
90             for (count = 0; count < numCoinsUsed; count++)
91                 bestCoins[count] = coinsUsed[count];
92             numBestCoins = numCoinsUsed;
93         }
94         numSolutions++;
95         return;
96     }
97
98     // Find the other combinations using the coin
99     coinPos = numCoins - coinsLeft;
100    coinsUsed[numCoinsUsed] = coinValues[coinPos];
101    numCoinsUsed++;
102    makeChange(coinsLeft, amount - coinValues[coinPos],
103               coinsUsed, numCoinsUsed);
104
105    // Find the other combinations not using the coin.
106    numCoinsUsed--;
107    makeChange(coinsLeft - 1, amount, coinsUsed, numCoinsUsed);
108 }

```

Program Output with Example Input Shown in Bold

Here are the valid coin values, in cents: 100 50 25 10 5 1
 Enter the amount of cents (as an integer) to make change for: **62** **Enter**
 What is the maximum number of coins to give as change? **6** **Enter**
 Number of possible combinations: 77
 Best combination of coins:
50 10 1 1

20.10 Focus on Software Engineering: Recursion versus Iteration

CONCEPT: Recursive algorithms can also be coded with iterative control structures. There are advantages and disadvantages to each approach.

Any algorithm that can be coded with recursion can also be coded with an iterative control structure, such as a `while` loop. Both approaches achieve repetition, but which is best to use?

There are several reasons not to use recursion. Recursive algorithms are certainly less efficient than iterative algorithms. Each time a function is called, the system incurs overhead that is not necessary with a loop. Also, in many cases an iterative solution may be more evident than a recursive one. In fact, the majority of repetitive programming tasks are best done with loops.

Some problems, however, are more easily solved with recursion than with iteration. For example, the mathematical definition of the GCD formula is well-suited for a recursive approach. The QuickSort algorithm is also an example of a function that is easier to code with recursion than iteration.

The speed and amount of memory available to modern computers diminishes the performance impact of recursion so much that inefficiency is no longer a strong argument against it. Today, the choice of recursion or iteration is primarily a design decision. If a problem is more easily solved with a loop, that should be the approach you take. If recursion results in a better design, that is the choice you should make.

Review Questions and Exercises

Short Answer

- What is the base case of each of the recursive functions listed in Questions 12, 13, and 14?
- What type of recursive function do you think would be more difficult to debug, one that uses direct recursion, or one that uses indirect recursion? Why?
- Which repetition approach is less efficient, a loop or a recursive function? Why?
- When should you choose a recursive algorithm over an iterative algorithm?
- Explain what is likely to happen when a recursive function that has no way of stopping executes.

Fill-in-the-Blank

- The _____ of recursion is the number of times a function calls itself.
- A recursive function's solvable problem is known as its _____. This causes the recursion to stop.

8. _____ recursion is when a function explicitly calls itself.
 9. _____ recursion is when function A calls function B, which in turns calls function A.

Algorithm Workbench

10. Write a recursive function to return the number of times a specified number occurs in an array.
 11. Write a recursive function to return the largest value in an array.

Predict the Output

What is the output of the following programs?

```
12. #include <iostream>
using namespace std;

int function(int);

int main()
{
    int x = 10;

    cout << function(x) << endl;
    return 0;
}

int function(int num)
{
    if (num <= 0)
        return 0;
    else
        return function(num - 1) + num;
}

13. #include <iostream>
using namespace std;

void function(int);

int main()
{
    int x = 10;

    function(x);
    return 0;
}

void function(int num)
{
    if (num > 0)
    {
        for (int x = 0; x < num; x++)
            cout << '*';
        cout << endl;
        function(num - 1);
    }
}
```

```

14. #include <iostream>
    #include <string>
    using namespace std;

    void function(string, int, int);

    int main()
    {
        string mystr = "Hello";
        cout << mystr << endl;
        function(mystr, 0, mystr.size());
        return 0;
    }
    void function(string str, int pos, int size)
    {
        if (pos < size)
        {
            function(str, pos + 1, size);
            cout << str[pos];
        }
    }
}

```

Programming Challenges

1. Iterative Factorial

Write an iterative version (using a loop instead of recursion) of the factorial function shown in this chapter. Test it with a driver program.

2. Recursive Conversion

Convert the following function to one that uses recursion.

```

void sign(int n)
{
    while (n > 0)
        cout << "No Parking\n";
    n--;
}

```

Demonstrate the function with a driver program.

3. QuickSort Template

Create a template version of the QuickSort algorithm that will work with any data type. Demonstrate the template with a driver function.

4. Recursive Array Sum

Write a function that accepts an array of integers and a number indicating the number of elements as arguments. The function should recursively calculate the sum of all the numbers in the array. Demonstrate the function in a driver program.

5. Recursive Multiplication`

Write a recursive function that accepts two arguments into the parameters x and y . The function should return the value of x times y . Remember, multiplication can be performed as repeated addition:

$$7 * 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4$$

6. Recursive Power Function

Write a function that uses recursion to raise a number to a power. The function should accept two arguments: the number to be raised and the exponent. Assume that the exponent is a nonnegative integer. Demonstrate the function in a program.

7. Sum of Numbers

Write a function that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the function will return the sum of 1, 2, 3, 4, ... 50. Use recursion to calculate the sum. Demonstrate the function in a program.

8. `isMember` Function

Write a recursive Boolean function named `isMember`. The function should accept two arguments: an array and a value. The function should return true if the value is found in the array, or false if the value is not found in the array. Demonstrate the function in a driver program.

9. String Reverser

Write a recursive function that accepts a `string` object as its argument and prints the string in reverse order. Demonstrate the function in a driver program.

10. `maxNode` Function

Add a member function named `maxNode` to the `NumberList` class discussed in this chapter. The function should return the largest value stored in the list. Use recursion in the function to traverse the list. Demonstrate the function in a driver program.

11. Palindrome Detector

A palindrome is any word, phrase, or sentence that reads the same forward and backward. Here are some well-known palindromes:

Able was I, ere I saw Elba
 A man, a plan, a canal, Panama
 Desserts, I stressed
 Kayak

Write a `bool` function that uses recursion to determine if a string argument is a palindrome. The function should return `true` if the argument reads the same forward and backward. Demonstrate the function in a program.

12. Ackermann's Function

Ackermann's Function is a recursive mathematical algorithm that can be used to test how well a computer performs recursion. Write a function `A(m, n)` that solves Ackermann's Function. Use the following logic in your function:

```
If m = 0 then return n + 1
If n = 0 then return A(m-1, 1)
Otherwise,      return A(m-1, A(m, n-1))
```

Test your function in a driver program that displays the following values:

`A(0, 0) A(0, 1) A(1, 1) A(1, 2) A(1, 3) A(2, 2) A(3, 2)`

TOPICS

- | | |
|---|---|
| 21.1 Definition and Applications of
Binary Trees | 21.3 Template Considerations for Binary
Search Trees |
| 21.2 Binary Search Tree Operations | |

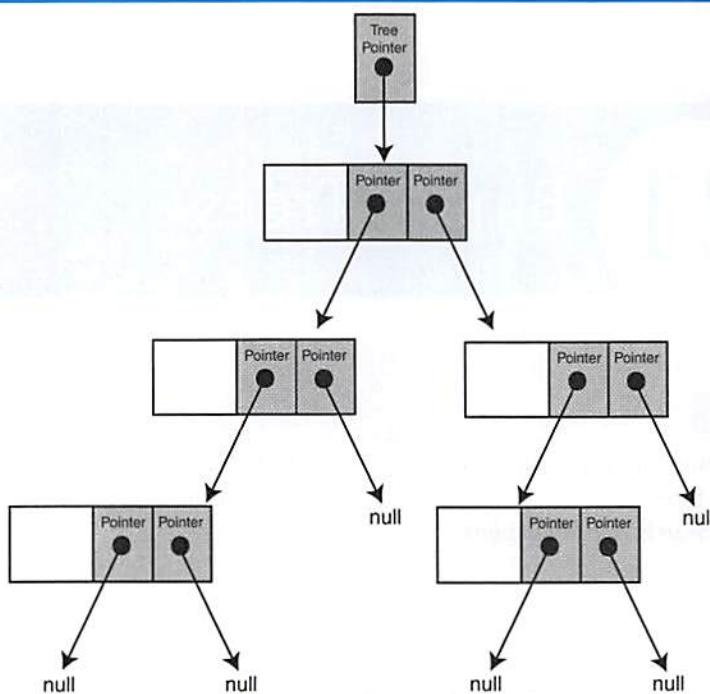
21.1

Definition and Applications of Binary Trees

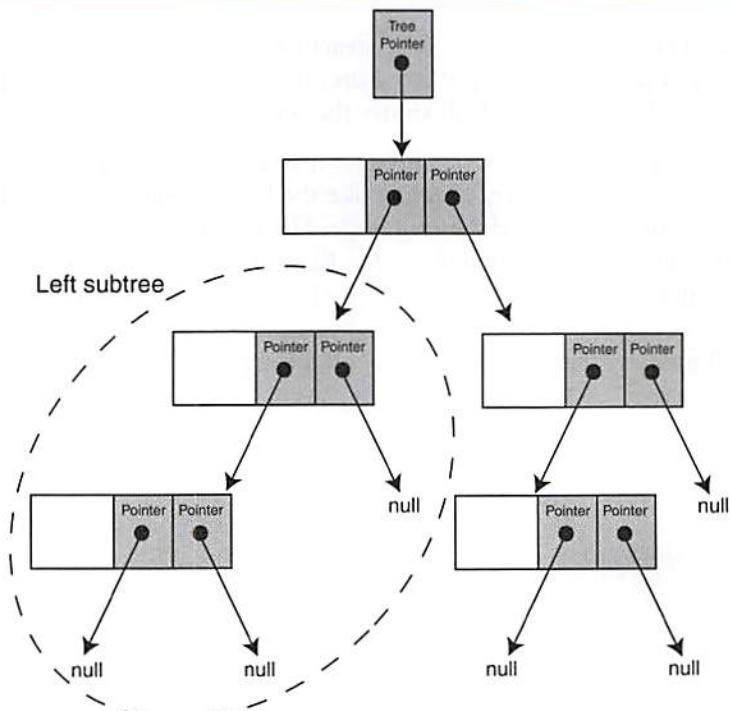
CONCEPT: A binary tree is a nonlinear linked structure in which each node may point to two other nodes, and every node but the root node has a single predecessor. Binary trees expedite the process of searching large sets of data.

A standard linked list is a linear data structure in which one node is linked to the next. A *binary tree* is a nonlinear linked structure. It is nonlinear because each node can point to two other nodes. Figure 21-1 illustrates the organization of a binary tree.

The data structure is called a tree because it resembles an upside-down tree. It is anchored at the top by a *tree pointer*, which is like the head pointer in a standard linked list. The first node in the list is called the *root node*. The root node has pointers to two other nodes, which are called *children*, or *child nodes*. Each of the children has its own set of two pointers, and can have its own children. Notice not all nodes have two children. Some point to only one node, and some point to no other nodes. A node with no children is called a *leaf node*. All pointers that do not point to a node are set to `nullptr`.

Figure 21-1 A binary tree

Binary trees can be divided into *subtrees*. A subtree is an entire branch of the tree, from one particular node down. For example, Figure 21-2 shows the left subtree from the root node of the tree shown in Figure 21-1.

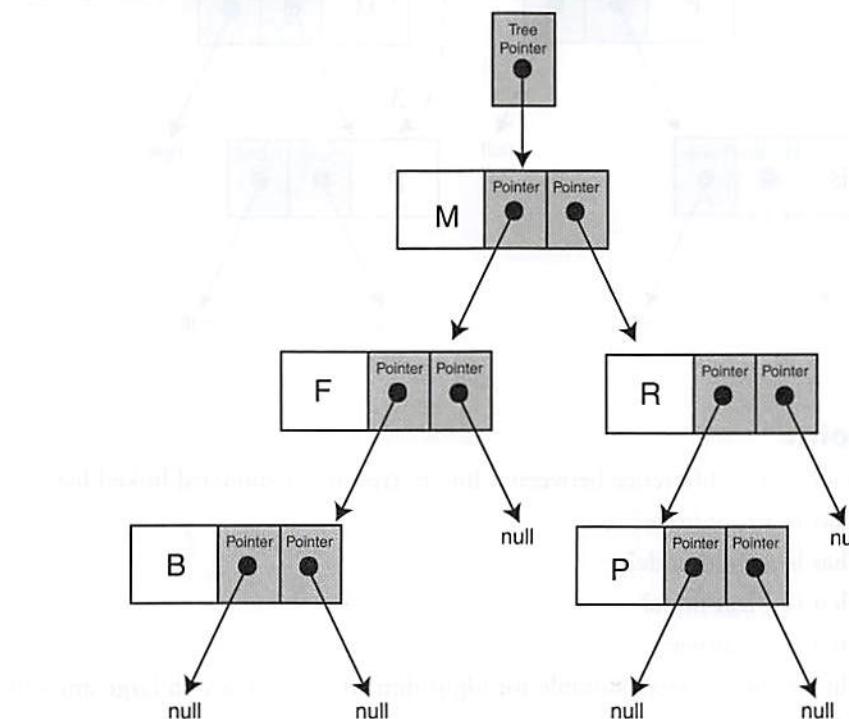
Figure 21-2 Left subtree

Applications of Binary Trees

Searching any linear data structure, such as an array or a standard linked list, is slow when the structure holds a large amount of data. This is because of the sequential nature of linear data structures. Binary trees are excellent data structures for searching large amounts of data. They are commonly used in database applications to organize key values that index database records. When used to facilitate searches, a binary tree is called a *binary search tree*. Binary search trees are the primary focus of this chapter.

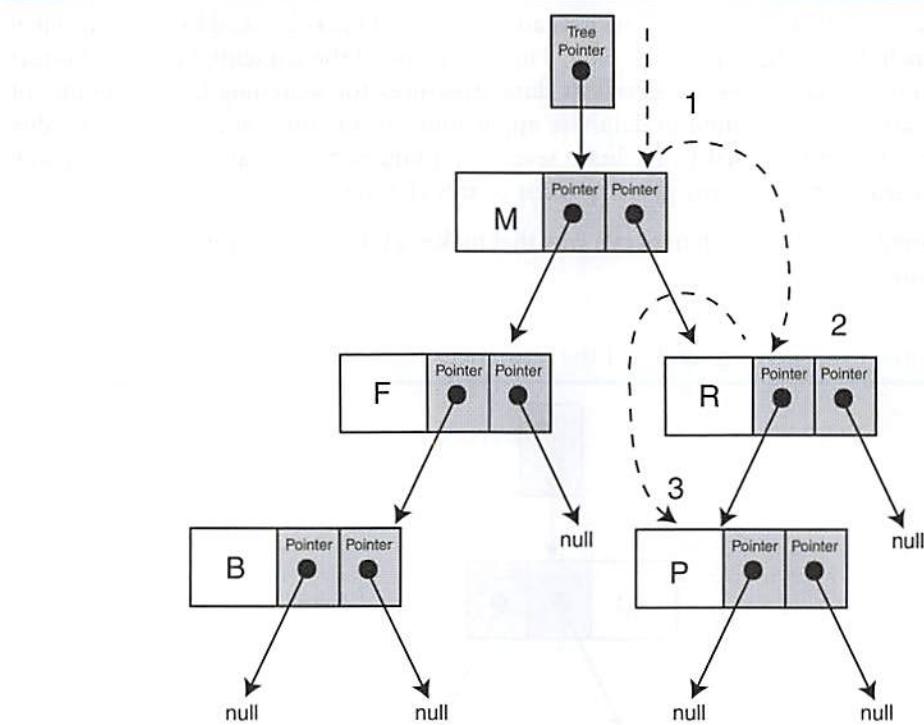
Data are stored in binary search trees in a way that makes a binary search simple. For example, look at Figure 21-3.

Figure 21-3 A binary tree storing letters of the alphabet



The figure depicts a binary search tree where each node stores a letter of the alphabet. Notice the root node holds the letter M. The left child of the root node holds the letter F, and the right child holds R. Values are stored in a binary search tree so a node's left child holds data whose value is less than the node's data, and the node's right child holds data whose value is greater than the node's data. This is true for all nodes in the tree that have children.

It is also true that *all* the nodes to the left of a node hold values less than the node's value. Likewise, all the nodes to the right of a node hold values that are greater than the node's data. When an application is searching a binary tree, it starts at the root node. If the root node does not hold the search value, the application branches either to the left or right child, depending on whether the search value is less than or greater than the value at the root node. This process continues until the value is found. Figure 21-4 illustrates the search pattern for finding the letter P in the binary tree shown.

Figure 21-4 Search pattern for finding the letter P

Checkpoint

- 21.1 Describe the difference between a binary tree and a standard linked list.
- 21.2 What is a root node?
- 21.3 What is a child node?
- 21.4 What is a leaf node?
- 21.5 What is a subtree?
- 21.6 Why are binary trees suitable for algorithms that must search large amounts of data?

21.2

Binary Search Tree Operations

CONCEPT: Many operations may be performed on a binary search tree. In this section, we will discuss creating a binary search tree and inserting, finding, and deleting nodes.

In this section, you will learn some basic operations that may be performed on a binary search tree. We will study a simple class that implements a binary tree for storing integer values.

Creating a Binary Trees

We will demonstrate the fundamental binary tree operations using a simple ADT: the `IntBinaryTree` class. The basis of our binary tree node is the following struct declaration:

```
struct TreeNode
{
    int value;
    TreeNode *left;
    TreeNode *right;
};
```

Each node has a `value` member for storing its integer data, as well as `left` and `right` pointers. The struct is implemented in the class declaration shown here:

Contents of IntBinaryTree.h

```
1 // Specification file for the IntBinaryTree class
2 #ifndef INTBINARYTREE_H
3 #define INTBINARYTREE_H
4
5 class IntBinaryTree
6 {
7 private:
8     struct TreeNode
9     {
10         int value;          // The value in the node
11         TreeNode *left;    // Pointer to left child node
12         TreeNode *right;   // Pointer to right child node
13     };
14
15     TreeNode *root;      // Pointer to the root node
16
17     // Private member functions
18     void insert(TreeNode *&, TreeNode *&);
19     void destroySubTree(TreeNode *);
20     void deleteNode(int, TreeNode *&);
21     void makeDeletion(TreeNode *&);
22     void displayInOrder(TreeNode *) const;
23     void displayPreOrder(TreeNode *) const;
24     void displayPostOrder(TreeNode *) const;
25
26 public:
27     // Constructor
28     IntBinaryTree()
29     { root = nullptr; }
30
31     // Destructor
32     ~IntBinaryTree()
33     { destroySubTree(root); }
34
35     // Binary tree operations
36     void insertNode(int);
37     bool searchNode(int);
38     void remove(int);
39 }
```

```

40     void displayInOrder() const
41         { displayInOrder(root); }
42
43     void displayPreOrder() const
44         { displayPreOrder(root); }
45
46     void displayPostOrder() const
47         { displayPostOrder(root); }
48 };
49 #endif

```

The root pointer will be used as the tree pointer. Similar to the head pointer in a linked list, root will point to the first node in the tree, or to `nullptr` if the tree is empty. It is initialized in the constructor, which is declared inline. The destructor calls `destroySubTree`, a private member function that recursively deletes all the nodes in the tree.

Inserting a Node

The code to insert a node into the tree is fairly straightforward. The public member function `insertNode` is called with the number to be inserted passed as an argument. The code for the function, which is in `IntBinaryTree.cpp`, is shown here:



```

27 void IntBinaryTree::insertNode(int num)
28 {
29     TreeNode *newNode = nullptr;      // Pointer to a new node.
30
31     // Create a new node and store num in it.
32     newNode = new TreeNode;
33     newNode->value = num;
34     newNode->left = newNode->right = nullptr;
35
36     // Insert the node.
37     insert(root, newNode);
38 }

```

First, a new node is allocated in line 32 and its address stored in the local pointer variable `newNode`. The value passed as an argument is stored in the node's `value` member in line 33. The node's `left` and `right` child pointers are set to `nullptr` in line 34 because all nodes must be inserted as leaf nodes. Next, the private member function `insert` is called in line 37. Notice the `root` pointer and the `newNode` pointer are passed as arguments. The code for the `insert` function is shown here:

```

12 void IntBinaryTree::insert(TreeNode *&nodePtr, TreeNode *&newNode)
13 {
14     if (nodePtr == nullptr)
15         nodePtr = newNode;          // Insert the node.
16     else if (newNode->value < nodePtr->value)
17         insert(nodePtr->left, newNode); // Search the left branch.
18     else
19         insert(nodePtr->right, newNode); // Search the right branch.
20 }

```

In general, this function takes a pointer to a subtree, and a pointer to a new node as arguments. It searches for the appropriate location in the subtree to insert the node, then makes the insertion. Notice the declaration of the first parameter, `nodePtr`:

```
TreeNode *&nodePtr
```

The `nodePtr` parameter is not simply a pointer to a `TreeNode` structure, but a *reference* to a pointer to a `TreeNode` structure. This means that any action performed on `nodePtr` is actually performed on the argument that was passed into `nodePtr`. The reason for this will be explained momentarily.

The `if` statement in line 14 determines whether `nodePtr` is set to `nullptr`:

```
if (nodePtr == nullptr)
    nodePtr = newNode; // Insert the node.
```

If `nodePtr` is set to `nullptr`, it is at the end of a branch, and the insertion point has been found. `nodePtr` is then made to point to `newNode`, which inserts `newNode` into the tree. This is why the `nodePtr` parameter is a reference. If it weren't a reference, this function would be making a copy of a node point to the new node, not the actual node in the tree.

If `nodePtr` is not set to `nullptr`, the following `else if` statement in line 16 executes:

```
else if (newNode->value < nodePtr->value)
    insert(nodePtr->left, newNode); // Search the left branch.
```

If the new node's value is less than the value pointed to by `nodePtr`, the insertion point is somewhere in the left subtree. If this is the case, the `insert` function is recursively called in line 17 with `nodePtr->left` passed as the subtree argument.

If the new node's value is not less than the value pointed to by `nodePtr`, the `else` statement in line 18 executes:

```
else
    insert(nodePtr->right, newNode); // Search the right branch.
```

The `else` statement recursively calls the `insert` function called with `nodePtr->right` passed as the subtree argument.

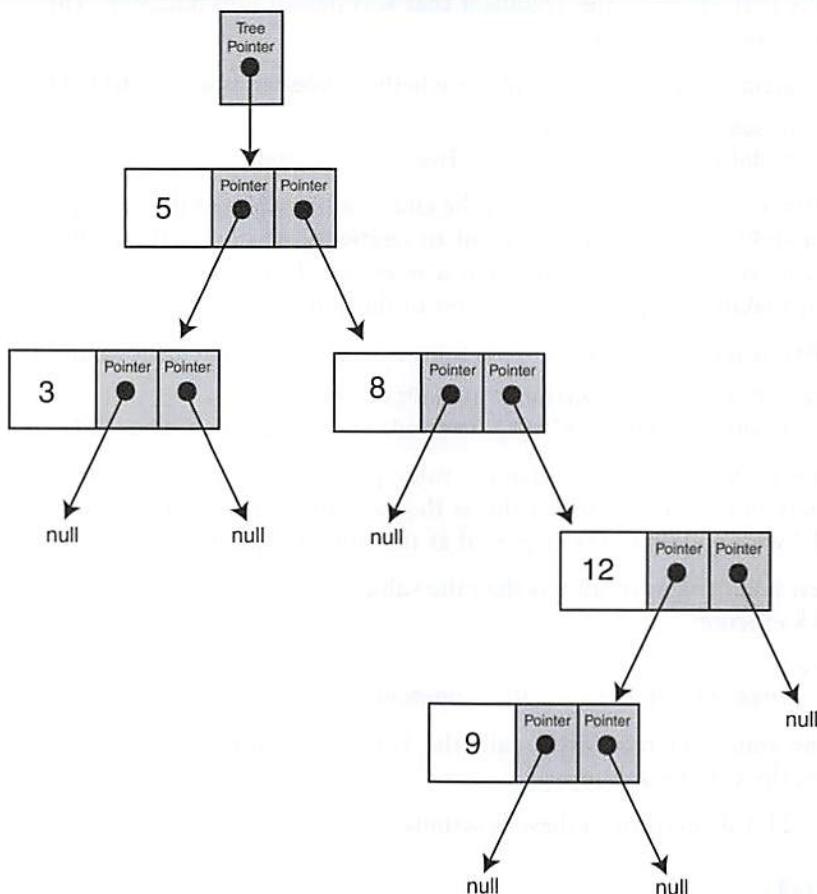
Program 21-1 demonstrates these functions.

Program 21-1

```
1 // This program builds a binary tree with 5 nodes.
2 #include <iostream>
3 #include "IntBinaryTree.h"
4 using namespace std;
5
6 int main()
7 {
8     IntBinaryTree tree;
9
10    cout << "Inserting nodes. ";
11    tree.insertNode(5);
12    tree.insertNode(8);
13    tree.insertNode(3);
14    tree.insertNode(12);
15    tree.insertNode(9);
16    cout << "Done.\n";
17
18    return 0;
19 }
```

Figure 21-5 shows the structure of the binary tree built by Program 21-1.

Figure 21-5 Binary tree built by Program 21-1



NOTE: The shape of the tree is determined by the order in which the values are inserted. The root node in the diagram above holds the value 5 because that was the first value inserted. By stepping through the function, you can see how the other nodes came to appear in their depicted positions.



NOTE: If the new value being inserted into the tree is equal to an existing value, the insertion algorithm inserts it to the right of the existing value.

Traversing the Tree

There are three common methods for traversing a binary tree and processing the value of each node: *inorder*, *preorder*, and *postorder*. Each of these methods is best implemented as a recursive function. The algorithms are described as follows:

- **Inorder traversal**
 1. The current node's left subtree is traversed.
 2. The current node's data is processed.
 3. The current node's right subtree is traversed.
- **Preorder traversal**
 1. The current node's data is processed
 2. The current node's left subtree is traversed.
 3. The current node's right subtree is traversed.
- **Postorder traversal**
 1. The current node's left subtree is traversed.
 2. The current node's right subtree is traversed.
 3. The current node's data is processed.

The `IntBinaryTree` class can display all the values in the tree using all three of these algorithms. The algorithms are initiated by the following inline public member functions:

```
void displayInOrder() const
    { displayInOrder(root); }

void displayPreOrder() const
    { displayPreOrder(root); }

void displayPostOrder() const
    { displayPostOrder(root); }
```

Each of the public member functions calls an overloaded recursive private member function and passes the root pointer as an argument. The recursive functions, which are very simple and straightforward, are shown here:

```
149 //*****
150 // The displayInOrder member function displays the values      *
151 // in the subtree pointed to by nodePtr, via inorder traversal. *
152 //*****
153
154 void IntBinaryTree::displayInOrder(TreeNode *nodePtr) const
155 {
156     if (nodePtr)
157     {
158         displayInOrder(nodePtr->left);
159         cout << nodePtr->value << endl;
160         displayInOrder(nodePtr->right);
161     }
162 }
163 //*****
164 // The displayPreOrder member function displays the values      *
165 // in the subtree pointed to by nodePtr, via preorder traversal. *
166 //*****
167
168
169 void IntBinaryTree::displayPreOrder(TreeNode *nodePtr) const
170 {
171     if (nodePtr)
172     {
173         cout << nodePtr->value << endl;
174         displayPreOrder(nodePtr->left);
```

```

175         displayPreOrder(nodePtr->right);
176     }
177 }
178
179 //*****
180 // The displayPostOrder member function displays the values *
181 // in the subtree pointed to by nodePtr, via postorder traversal. *
182 //*****
183
184 void IntBinaryTree::displayPostOrder(TreeNode *nodePtr) const
185 {
186     if (nodePtr)
187     {
188         displayPostOrder(nodePtr->left);
189         displayPostOrder(nodePtr->right);
190         cout << nodePtr->value << endl;
191     }
192 }
```

Program 21-2, which is a modification of Program 21-1, demonstrates each of the traversal methods.

Program 21-2

```

1 // This program builds a binary tree with 5 nodes.
2 // The nodes are displayed with inorder, preorder,
3 // and postorder algorithms.
4 #include <iostream>
5 #include "IntBinaryTree.h"
6 using namespace std;
7
8 int main()
9 {
10     IntBinaryTree tree;
11
12     // Insert some nodes.
13     cout << "Inserting nodes.\n";
14     tree.insertNode(5);
15     tree.insertNode(8);
16     tree.insertNode(3);
17     tree.insertNode(12);
18     tree.insertNode(9);
19
20     // Display inorder.
21     cout << "Inorder traversal:\n";
22     tree.displayInOrder();
23
24     // Display preorder.
25     cout << "\nPreorder traversal:\n";
26     tree.displayPreOrder();
27
28     // Display postorder.
29     cout << "\nPostorder traversal:\n";
30     tree.displayPostOrder();
31
32 }
```

Program Output

```

Inserting nodes.
Inorder traversal:
3
5
8
9
12

Preorder traversal:
5
3
8
12
9

Postorder traversal:
3
9
12
8
5

```

Searching the Tree

The IntBinaryTree class has a public member function, `searchNode`, that returns true if a value is found in the tree, or false otherwise. The function simply starts at the root node and traverses the tree until it finds the search value or runs out of nodes. The code is shown here:

```

63 bool IntBinaryTree::searchNode(int num)
64 {
65     TreeNode *nodePtr = root;
66
67     while (nodePtr)
68     {
69         if (nodePtr->value == num)
70             return true;
71         else if (num < nodePtr->value)
72             nodePtr = nodePtr->left;
73         else
74             nodePtr = nodePtr->right;
75     }
76     return false;
77 }
```

Program 21-3 demonstrates this function.

Program 21-3

```

1 // This program builds a binary tree with 5 nodes.
2 // The SearchNode function is demonstrated.
3 #include <iostream>
4 #include "IntBinaryTree.h"
5 using namespace std;
6
7 int main()
8 {
9     IntBinaryTree tree;
10
11    // Insert some nodes in the tree.
12    cout << "Inserting nodes.\n";
13    tree.insertNode(5);
14    tree.insertNode(8);
15    tree.insertNode(3);
16    tree.insertNode(12);
17    tree.insertNode(9);
18
19    // Search for the value 3.
20    if (tree.searchNode(3))
21        cout << "3 is found in the tree.\n";
22    else
23        cout << "3 was not found in the tree.\n";
24
25    // Search for the value 100.
26    if (tree.searchNode(100))
27        cout << "100 is found in the tree.\n";
28    else
29        cout << "100 was not found in the tree.\n";
30
31 }

```

Program Output

Inserting nodes.
3 is found in the tree.
100 was not found in the tree.

Deleting a Node

Deleting a leaf node is not difficult. We simply find its parent and set the child pointer that links to it to `nullptr`, then free the node's memory. But what if we want to delete a node that has child nodes? We must delete the node while at the same time preserving the subtrees to which the node links.

There are two possible situations to face when deleting a nonleaf node: the node has one child, or the node has two children. Figure 21-6 illustrates a tree in which we are about to delete a node with one subtree.

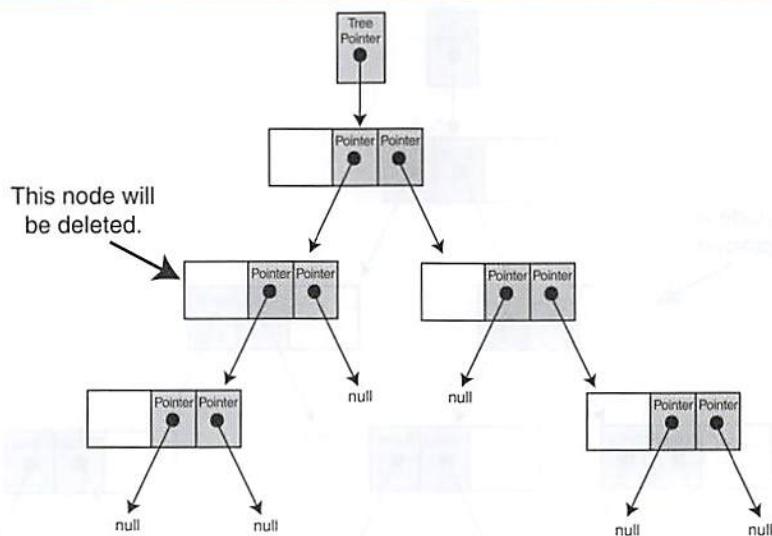
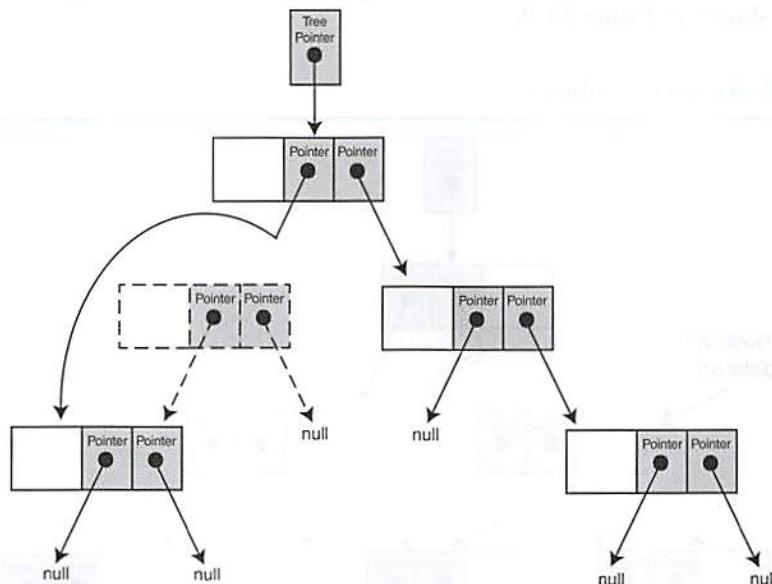
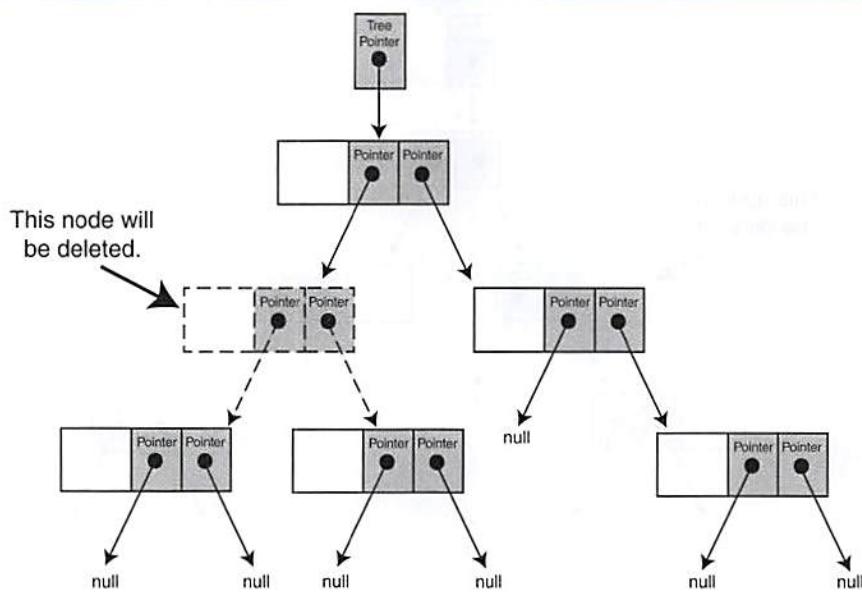
Figure 21-6 A node with one subtree is about to be deleted

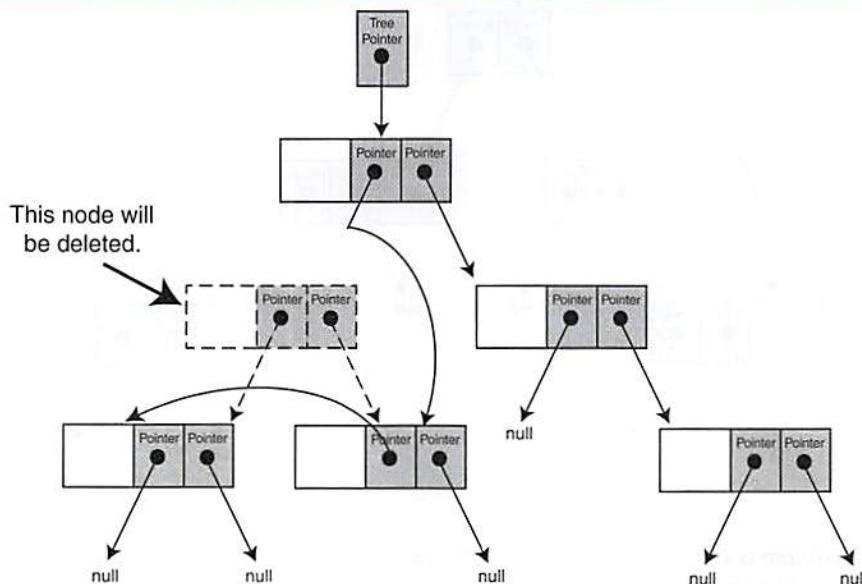
Figure 21-7 shows how we will link the node's subtree with its parent.

Figure 21-7 Linking the parent node with the subtree

The problem is not as easily solved, however, when the node we are about to delete has two subtrees. For example, look at Figure 21-8:

Figure 21-8 A node with two subtrees is about to be deleted

Obviously, we cannot attach both of the node's subtrees to its parent, so there must be an alternative solution. One way of addressing this problem is to attach the node's right subtree to the parent, then find a position in the right subtree to attach the left subtree. The result is shown in Figure 21-9.

Figure 21-9 Relinking the subtrees

Now we will see how this action is implemented in code. To delete a node from the `IntBinaryTree`, call the public member `remove`. The argument passed to the function is the value of the node you wish to delete. The `remove` member function is shown here:

```
84 void IntBinaryTree::remove(int num)
85 {
86     deleteNode(num, root);
87 }
```

The `remove` member function calls the `deleteNode` member function. It passes the value of the node to delete and the `root` pointer. The `deleteNode` member function is shown here:

```
95 void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
96 {
97     if (num < nodePtr->value)
98         deleteNode(num, nodePtr->left);
99     else if (num > nodePtr->value)
100        deleteNode(num, nodePtr->right);
101    else
102        makeDeletion(nodePtr);
103 }
```

Notice this function's arguments are references to pointers. Like the `insert` function, the `deleteNode` function must have access to an actual pointer in the tree. You will see why momentarily.

The `deleteNode` function uses an `if/else if` statement. The first part of the statement is in lines 97 and 98:

```
if (num < nodePtr->value)
    deleteNode(num, nodePtr->left);
```

This code compares the parameter `num` with the `value` member of the node to which `nodePtr` points. If `num` is less, then the value being searched for will appear somewhere in `nodePtr`'s left subtree (if it appears in the tree at all). In this case, the `deleteNode` function is recursively called with `num` as the first argument, and `nodePtr->left` as the second argument.

If `num` is not less than `nodePtr->value`, the `else if` in lines 99 and 100 statement is executed:

```
else if (num > nodePtr->value)
    deleteNode(num, nodePtr->right);
```

If `num` is greater than `nodePtr->value`, then the value being searched for will appear somewhere in `nodePtr`'s right subtree (if it appears in the tree at all). So, the `deleteNode` function is recursively called with `num` as the first argument, and `nodePtr->right` as the second argument.

If `num` is equal to `nodePtr->value`, then neither of the `if` statements will find a true condition. In this case, `nodePtr` points to the node that is to be deleted, and the trailing `else` in lines 101 and 102 will execute:

```
else
    makeDeletion(nodePtr);
```

The trailing `else` statement calls the `makeDeletion` function and passes `nodePtr` as its argument. The `makeDeletion` function actually deletes the node from the tree and must reattach the deleted node's subtrees as shown in Figure 21-9. Therefore, it must have access to the actual pointer, in the binary tree, to the node that is being deleted (not just a copy of the pointer). This is why the `nodePtr` parameter in the `deleteNode` function is a reference. It must pass to `makeDeletion` the actual pointer, in the binary tree, to the node that is to be deleted. The `makeDeletion` function's code is as follows:

```

112 void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
113 {
114     // Define a temporary pointer to use in reattaching
115     // the left subtree.
116     TreeNode *tempNodePtr = nullptr;
117
118     if (nodePtr == nullptr)
119         cout << "Cannot delete empty node.\n";
120     else if (nodePtr->right == nullptr)
121     {
122         tempNodePtr = nodePtr;
123         nodePtr = nodePtr->left; // Reattach the left child.
124         delete tempNodePtr;
125     }
126     else if (nodePtr->left == nullptr)
127     {
128         tempNodePtr = nodePtr;
129         nodePtr = nodePtr->right; // Reattach the right child.
130         delete tempNodePtr;
131     }
132     // If the node has two children.
133     else
134     {
135         // Move one node to the right.
136         tempNodePtr = nodePtr->right;
137         // Go to the end left node.
138         while (tempNodePtr->left)
139             tempNodePtr = tempNodePtr->left;
140         // Reattach the left subtree.
141         tempNodePtr->left = nodePtr->left;
142         tempNodePtr = nodePtr;
143         // Reattach the right subtree.
144         nodePtr = nodePtr->right;
145         delete tempNodePtr;
146     }
147 }
```

Program 21-4 demonstrates these functions.

Program 21-4

```

1 // This program builds a binary tree with 5 nodes.
2 // The deleteNode function is used to remove two of them.
3 #include <iostream>
4 #include "IntBinaryTree.h"
5 using namespace std;
6
7 int main()
8 {
9     IntBinaryTree tree;
10
11    // Insert some values into the tree.
12    cout << "Inserting nodes.\n";
13    tree.insertNode(5);
14    tree.insertNode(8);
15    tree.insertNode(3);
16    tree.insertNode(12);
17    tree.insertNode(9);
18
19    // Display the values.
20    cout << "Here are the values in the tree:\n";
21    tree.displayInOrder();
22
23    // Delete the value 8.
24    cout << "Deleting 8... \n";
25    tree.remove(8);
26
27    // Delete the value 12.
28    cout << "Deleting 12... \n";
29    tree.remove(12);
30
31    // Display the values.
32    cout << "Now, here are the nodes:\n";
33    tree.displayInOrder();
34
35 }

```

Program Output

```

Inserting nodes.
Here are the values in the tree:
3
5
8
9
12
Deleting 8...
Deleting 12...
Now, here are the nodes:
3
5
9

```

For your reference, the entire contents of the IntBinaryTree.cpp file are shown below:

Contents of IntBinaryTree.cpp

```
1 // Implementation file for the IntBinaryTree class
2 #include <iostream>
3 #include "IntBinaryTree.h"
4 using namespace std;
5
6 //*****insert*****
7 // insert accepts a TreeNode pointer and a pointer to a node. *
8 // The function inserts the node into the tree pointed to by *
9 // the TreeNode pointer. This function is called recursively. *
10 //*****
11
12 void IntBinaryTree::insert(TreeNode *&nodePtr, TreeNode *&newNode)
13 {
14     if (nodePtr == nullptr)
15         nodePtr = newNode;           // Insert the node.
16     else if (newNode->value < nodePtr->value)
17         insert(nodePtr->left, newNode); // Search the left branch.
18     else
19         insert(nodePtr->right, newNode); // Search the right branch.
20 }
21
22 //*****insertNode*****
23 // insertNode creates a new node to hold num as its value, *
24 // and passes it to the insert function.
25 //*****
26
27 void IntBinaryTree::insertNode(int num)
28 {
29     TreeNode *newNode = nullptr;    // Pointer to a new node.
30
31     // Create a new node and store num in it.
32     newNode = new TreeNode;
33     newNode->value = num;
34     newNode->left = newNode->right = nullptr;
35
36     // Insert the node.
37     insert(root, newNode);
38 }
39
40 //*****destroySubTree*****
41 // destroySubTree is called by the destructor. It *
42 // deletes all nodes in the tree.
43 //*****
44
45 void IntBinaryTree::destroySubTree(TreeNode *nodePtr)
46 {
47     if (nodePtr)
48     {
49         if (nodePtr->left)
50             destroySubTree(nodePtr->left);
```

```
51         if (nodePtr->right)
52             destroySubTree(nodePtr->right);
53             delete nodePtr;
54     }
55 }
56
57 //*****
58 // searchNode determines whether a value is present in *
59 // the tree. If so, the function returns true.          *
60 // Otherwise, it returns false.                      *
61 //*****
62
63 bool IntBinaryTree::searchNode(int num)
64 {
65     TreeNode *nodePtr = root;
66
67     while (nodePtr)
68     {
69         if (nodePtr->value == num)
70             return true;
71         else if (num < nodePtr->value)
72             nodePtr = nodePtr->left;
73         else
74             nodePtr = nodePtr->right;
75     }
76     return false;
77 }
78
79 //*****
80 // remove calls deleteNode to delete the           *
81 // node whose value member is the same as num.   *
82 //*****
83
84 void IntBinaryTree::remove(int num)
85 {
86     deleteNode(num, root);
87 }
88
89
90 //*****
91 // deleteNode deletes the node whose value      *
92 // member is the same as num.                   *
93 //*****
94
95 void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
96 {
97     if (num < nodePtr->value)
98         deleteNode(num, nodePtr->left);
99     else if (num > nodePtr->value)
100        deleteNode(num, nodePtr->right);
101    else
102        makeDeletion(nodePtr);
103 }
104
105
```

```
106 //*****
107 // makeDeletion takes a reference to a pointer to the node *
108 // that is to be deleted. The node is removed and the *
109 // branches of the tree below the node are reattached. *
110 //*****
111
112 void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
113 {
114     // Define a temporary pointer to use in reattaching
115     // the left subtree.
116     TreeNode *tempNodePtr = nullptr;
117
118     if (nodePtr == nullptr)
119         cout << "Cannot delete empty node.\n";
120     else if (nodePtr->right == nullptr)
121     {
122         tempNodePtr = nodePtr;
123         nodePtr = nodePtr->left; // Reattach the left child.
124         delete tempNodePtr;
125     }
126     else if (nodePtr->left == nullptr)
127     {
128         tempNodePtr = nodePtr;
129         nodePtr = nodePtr->right; // Reattach the right child.
130         delete tempNodePtr;
131     }
132     // If the node has two children.
133     else
134     {
135         // Move one node to the right.
136         tempNodePtr = nodePtr->right;
137         // Go to the end left node.
138         while (tempNodePtr->left)
139             tempNodePtr = tempNodePtr->left;
140         // Reattach the left subtree.
141         tempNodePtr->left = nodePtr->left;
142         tempNodePtr = nodePtr;
143         // Reattach the right subtree.
144         nodePtr = nodePtr->right;
145         delete tempNodePtr;
146     }
147 }
148
149 //*****
150 // The displayInOrder member function displays the values *
151 // in the subtree pointed to by nodePtr, via inorder traversal. *
152 //*****
153
154 void IntBinaryTree::displayInOrder(TreeNode *nodePtr) const
155 {
156     if (nodePtr)
157     {
158         displayInOrder(nodePtr->left);
159         cout << nodePtr->value << endl;
160         displayInOrder(nodePtr->right);
161     }
162 }
```

```

163
164 //*****
165 // The displayPreOrder member function displays the values   *
166 // in the subtree pointed to by nodePtr, via preorder traversal. *
167 //*****
168
169 void IntBinaryTree::displayPreOrder(TreeNode *nodePtr) const
170 {
171     if (nodePtr)
172     {
173         cout << nodePtr->value << endl;
174         displayPreOrder(nodePtr->left);
175         displayPreOrder(nodePtr->right);
176     }
177 }
178
179 //*****
180 // The displayPostOrder member function displays the values   *
181 // in the subtree pointed to by nodePtr, via postorder traversal. *
182 //*****
183
184 void IntBinaryTree::displayPostOrder(TreeNode *nodePtr) const
185 {
186     if (nodePtr)
187     {
188         displayPostOrder(nodePtr->left);
189         displayPostOrder(nodePtr->right);
190         cout << nodePtr->value << endl;
191     }
192 }

```



Checkpoint

- 21.7 Describe the sequence of events in an inorder traversal.
- 21.8 Describe the sequence of events in a preorder traversal.
- 21.9 Describe the sequence of events in a postorder traversal.
- 21.10 Describe the steps taken in deleting a leaf node.
- 21.11 Describe the steps taken in deleting a node with one child.
- 21.12 Describe the steps taken in deleting a node with two children.

21.3

Template Considerations for Binary Search Trees

CONCEPT: Binary search trees may be implemented as templates, but any data types used with them must support the `<`, `>`, and `==` operators.

When designing a binary tree template, remember any data types stored in the binary tree must support the `<`, `>`, and `==` operators. If you use the tree to store class objects, these operators must be overridden.

The following code shows an example of a binary tree template. Program 21-5 demonstrates the template. It creates a binary tree that can hold strings, then prompts the user to enter a series of names that are inserted into the tree. The program then displays the contents of the tree using inorder traversal.

Contents of BinaryTree.h

```

1  #ifndef BINARYTREE_H
2  #define BINARYTREE_H
3  #include <iostream>
4  using namespace std;
5
6  // BinaryTree template
7  template <class T>
8  class BinaryTree
9  {
10 private:
11     struct TreeNode
12     {
13         T value;           // The value in the node
14         TreeNode *left;    // Pointer to left child node
15         TreeNode *right;   // Pointer to right child node
16     };
17
18     TreeNode *root; // Pointer to the root node
19
20     // Private member functions
21     void insert(TreeNode *&, TreeNode *&);
22     void destroySubTree(TreeNode *);
23     void deleteNode(T, TreeNode *&);
24     void makeDeletion(TreeNode *&);
25     void displayInOrder(TreeNode *) const;
26     void displayPreOrder(TreeNode *) const;
27     void displayPostOrder(TreeNode *) const;
28
29 public:
30     // Constructor
31     BinaryTree()
32     { root = nullptr; }
33
34     // Destructor
35     ~BinaryTree()
36     { destroySubTree(root); }
37
38     // Binary tree operations
39     void insertNode(T);
40     bool searchNode(T);
41     void remove(T);
42
43     void displayInOrder() const
44     { displayInOrder(root); }
45
46     void displayPreOrder() const
47     { displayPreOrder(root); }
48

```

```
49     void displayPostOrder() const
50     { displayPostOrder(root); }
51 };
52
53 //*****insert*****
54 // insert accepts a TreeNode pointer and a pointer to a node. *
55 // The function inserts the node into the tree pointed to by *
56 // the TreeNode pointer. This function is called recursively.*  

57 //*****insert*****
58 template <class T>
59 void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode)
60 {
61     if (nodePtr == nullptr)
62         nodePtr = newNode;           // Insert the node
63     else if (newNode->value < nodePtr->value)
64         insert(nodePtr->left, newNode); // Search the left branch
65     else
66         insert(nodePtr->right, newNode); // Search the right branch
67 }
68
69 //*****insertNode*****
70 // insertNode creates a new node to hold num as its value, *
71 // and passes it to the insert function. *
72 //*****insertNode*****
73 template <class T>
74 void BinaryTree<T>::insertNode(T item)
75 {
76     TreeNode *newNode = nullptr; // Pointer to a new node
77
78     // Create a new node and store num in it.
79     newNode = new TreeNode;
80     newNode->value = item;
81     newNode->left = newNode->right = nullptr;
82
83     // Insert the node.
84     insert(root, newNode);
85 }
86
87 //*****destroySubTree*****
88 // destroySubTree is called by the destructor. It *
89 // deletes all nodes in the tree. *
90 //*****destroySubTree*****
91 template <class T>
92 void BinaryTree<T>::destroySubTree(TreeNode *nodePtr)
93 {
94     if (nodePtr)
95     {
96         if (nodePtr->left)
97             destroySubTree(nodePtr->left);
98         if (nodePtr->right)
99             destroySubTree(nodePtr->right);
100        delete nodePtr;
101    }
102 }
103 }
```

```

104 // ****
105 // searchNode determines if a value is present in *
106 // the tree. If so, the function returns true. *
107 // Otherwise, it returns false.
108 // ****
109 template <class T>
110 bool BinaryTree<T>::searchNode(T item)
111 {
112     TreeNode *nodePtr = root;
113
114     while (nodePtr)
115     {
116         if (nodePtr->value == item)
117             return true;
118         else if (item < nodePtr->value)
119             nodePtr = nodePtr->left;
120         else
121             nodePtr = nodePtr->right;
122     }
123     return false;
124 }
125
126 // ****
127 // remove calls deleteNode to delete the
128 // node whose value is the same as num.
129 // ****
130 template <class T>
131 void BinaryTree<T>::remove(T item)
132 {
133     deleteNode(item, root);
134 }
135
136 // ****
137 // deleteNode deletes the node whose value
138 // member is the same as num.
139 // ****
140 template <class T>
141 void BinaryTree<T>::deleteNode(T item, TreeNode *nodePtr)
142 {
143     if (item < nodePtr->value)
144         deleteNode(item, nodePtr->left);
145     else if (item > nodePtr->value)
146         deleteNode(item, nodePtr->right);
147     else
148         makeDeleteion(nodePtr);
149 }
150
151 // ****
152 // makeDeleteion takes a reference to a pointer to the node
153 // that is to be deleted. The node is removed and the
154 // branches of the tree below the node are reattached.
155 // ****
156 template <class T>
157 void BinaryTree<T>::makeDeleteion(TreeNode *nodePtr)
158 {

```

```
159     // Define a temporary pointer to use in reattaching
160     // the left subtree.
161     TreeNode *tempNodePtr = nullptr;
162
163     if (nodePtr == nullptr)
164         cout << "Cannot delete empty node.\n";
165     else if (nodePtr->right == nullptr)
166     {
167         tempNodePtr = nodePtr;
168         nodePtr = nodePtr->left; // Reattach the left child
169         delete tempNodePtr;
170     }
171     else if (nodePtr->left == nullptr)
172     {
173         tempNodePtr = nodePtr;
174         nodePtr = nodePtr->right; // Reattach the right child
175         delete tempNodePtr;
176     }
177     // If the node has two children.
178     else
179     {
180         // Move one node to the right.
181         tempNodePtr = nodePtr->right;
182         // Go to the end left node.
183         while (tempNodePtr->left)
184             tempNodePtr = tempNodePtr->left;
185         // Reattach the left subtree.
186         tempNodePtr->left = nodePtr->left;
187         tempNodePtr = nodePtr;
188         // Reattach the right subtree.
189         nodePtr = nodePtr->right;
190         delete tempNodePtr;
191     }
192 }
193
194 //*****
195 // The displayInOrder member function displays the values *
196 // in the subtree pointed to by nodePtr, via inorder traversal. *
197 //*****
198 template <class T>
199 void BinaryTree<T>::displayInOrder(TreeNode *nodePtr) const
200 {
201     if (nodePtr)
202     {
203         displayInOrder(nodePtr->left);
204         cout << nodePtr->value << endl;
205         displayInOrder(nodePtr->right);
206     }
207 }
208
209 //*****
210 // The displayPreOrder member function displays the values *
211 // in the subtree pointed to by nodePtr, via preorder traversal. *
212 //*****
213 template <class T>
214 void BinaryTree<T>::displayPreOrder(TreeNode *nodePtr) const
```

```

215  {
216      if (nodePtr)
217      {
218          cout << nodePtr->value << endl;
219          displayPreOrder(nodePtr->left);
220          displayPreOrder(nodePtr->right);
221      }
222  }
223
224 //*****
225 // The displayPostOrder member function displays the values *
226 // in the subtree pointed to by nodePtr, via postorder traversal.* 
227 //*****
228 template <class T>
229 void BinaryTree<T>::displayPostOrder(TreeNode *nodePtr) const
230 {
231     if (nodePtr)
232     {
233         displayPostOrder(nodePtr->left);
234         displayPostOrder(nodePtr->right);
235         cout << nodePtr->value << endl;
236     }
237 }
238 #endif

```

Program 21-5

```

1 // This program demonstrates the BinaryTree class template.
2 // It builds a binary tree with 5 nodes.
3 #include <iostream>
4 #include "BinaryTree.h"
5 using namespace std;
6
7 const int NUM_NODES = 5;
8
9 int main()
10 {
11     string name;
12
13     // Create the binary tree.
14     BinaryTree<string> tree;
15
16     // Insert some names.
17     for (int count = 0; count < NUM_NODES; count++)
18     {
19         cout << "Enter a name: ";
20         getline(cin, name);
21         tree.insertNode(name);
22     }
23
24     // Display the values.
25     cout << "\nHere are the values in the tree:\n";
26     tree.displayInOrder();
27     return 0;
28 }

```

Program Output with Example Input Shown in Bold

```
Enter a name: David Enter
Enter a name: Geri Enter
Enter a name: Chris Enter
Enter a name: Samantha Enter
Enter a name: Anthony Enter
```

Here are the values in the tree:

```
Anthony
Chris
David
Geri
Samantha
```

Review Questions and Exercises**Short Answer**

1. Each node in a binary tree may point to how many other nodes?
2. How many predecessors may each node other than the root node have?
3. What is a leaf node?
4. What is a subtree?
5. What initially determines the shape of a binary tree?
6. What are the three methods of traversing a binary tree? What is the difference between these methods?

Fill-in-the-Blank

7. The first node in a binary tree is called the _____.
8. A binary tree node's left and right pointers point to the node's _____.
9. A node with no children is called a(n) _____.
10. A(n) _____ is an entire branch of the tree, from one particular node down.
11. The three common types of traversal with a binary tree are _____, _____, and _____.

Algorithm Workbench

12. Write a pseudocode algorithm for inserting a node in a tree.
13. Write a pseudocode algorithm for the inorder traversal.
14. Write a pseudocode algorithm for the preorder traversal.
15. Write a pseudocode algorithm for the postorder traversal.
16. Write a pseudocode algorithm for searching a tree for a specified value.

17. Suppose the following values are inserted into a binary tree, in the order given:

12, 7, 9, 10, 22, 24, 30, 18, 3, 14, 20

Draw a diagram of the resulting binary tree.

18. How would the values in the tree you sketched for Question 17 be displayed in an inorder traversal?
19. How would the values in the tree you sketched for Question 17 be displayed in a preorder traversal?
20. How would the values in the tree you sketched for Question 17 be displayed in a postorder traversal?

True or False

21. T F Each node in a binary tree must have at least two children.
22. T F When a node is inserted into a tree, it must be inserted as a leaf node.
23. T F Values stored in the current node's left subtree are less than the value stored in the current node.
24. T F The shape of a binary tree is determined by the order in which values are inserted.
25. T F In inorder traversal, the node's data is processed first, then the left and right nodes are visited.

Programming Challenges

1. Binary Tree Template

Write your own version of a class template that will create a binary tree that can hold values of any data type. Demonstrate the class with a driver program.

2. Node Counter

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that counts and returns the number of nodes in the tree. Demonstrate the function in a driver program.

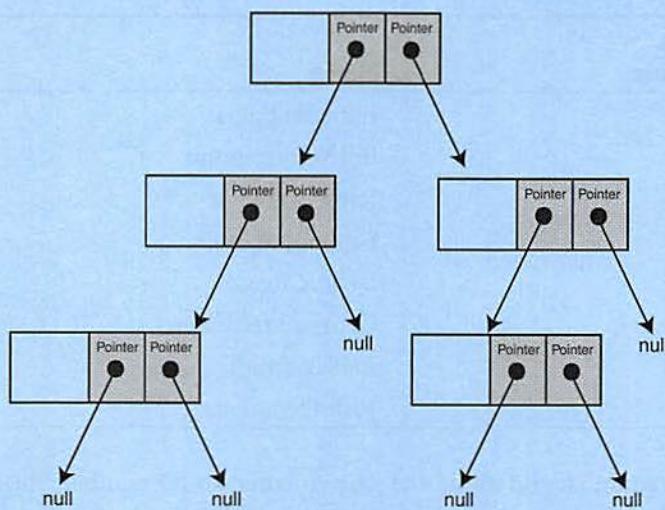
3. Leaf Counter

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that counts and returns the number of leaf nodes in the tree. Demonstrate the function in a driver program.

4. Tree Height

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that returns the height of the tree. The height of the tree is the number of levels it contains. For example, the tree shown in Figure 21-10 has three levels.



Figure 21-10 A tree with three levels

Demonstrate the function in a driver program.

5. Tree Width

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that returns the width of the tree. The width of the tree is the largest number of nodes in the same level. Demonstrate the function in a driver program.

6. Tree Assignment Operators, Copy Constructors, and Move Constructors

Design an overloaded copy assignment operator, a move assignment operator, a copy constructor, and a move constructor for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class. Demonstrate them in a driver program.

7. Queue Converter

Write a program that stores a series of numbers in a binary tree. Then have the program insert the values into a queue in ascending order. Dequeue the values and display them on the screen to confirm that they were stored in the proper order.

8. Employee Tree

Design an `EmployeeInfo` class that holds the following employee information:

Employee ID Number:	an integer
Employee Name:	a string

Next, use the template you designed in Programming Challenge 1 (Binary Tree Template) to implement a binary tree whose nodes hold an instance of the `EmployeeInfo` class. The nodes should be sorted on the Employee ID number.

Test the binary tree by inserting nodes with the following information.

Employee ID Number	Name
1021	John Williams
1057	Bill Witherspoon
2487	Jennifer Twain
3769	Sophia Lancaster
1017	Debbie Reece
1275	George McMullen
1899	Ashley Smith
4218	Josh Plemmons

Your program should allow the user to enter an ID number, then search the tree for the number. If the number is found, it should display the employee's name. If the node is not found, it should display a message indicating so.

A

The ASCII Character Set

Nonprintable ASCII Characters				Printable ASCII Characters			
Dec	Hex	Oct	Name of Character	Dec	Hex	Oct	Character
0	0	0	NULL	32	20	40	(Space)
1	1	1	SOTT	33	21	41	!
2	2	2	STX	34	22	42	"
3	3	3	ETY	35	23	43	#
4	4	4	EOT	36	24	44	\$
5	5	5	ENQ	37	25	45	%
6	6	6	ACK	38	26	46	&
7	7	7	BELL	39	27	47	'
8	8	10	BKSPC	40	28	50	(
9	9	11	HZTAB	41	29	51)
10	a	12	NEWLN	42	2a	52	*
11	b	13	VTAB	43	2b	53	+
12	c	14	FF	44	2c	54	,
13	d	15	CR	45	2d	55	-
14	e	16	SO	46	2e	56	.
15	f	17	SI	47	2f	57	/
16	10	20	DLE	48	30	60	0
17	11	21	DC1	49	31	61	1
18	12	22	DC2	50	32	62	2
19	13	23	DC3	51	33	63	3
20	14	24	DC4	52	34	64	4
21	15	25	NAK	53	35	65	5
22	16	26	SYN	54	36	66	6
23	17	27	ETB	55	37	67	7
24	18	30	CAN	56	38	70	8
25	19	31	EM	57	39	71	9
26	1a	32	SUB	58	3a	72	:
27	1b	33	ESC	59	3b	73	;
28	1c	34	FS	60	3c	74	<
29	1d	35	GS	61	3d	75	=
30	1e	36	RS	62	3e	76	>
31	1f	37	US	63	3f	77	?
127	7f	177	DEL	64	40	100	@

Printable ASCII Characters				Printable ASCII Characters			
Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
65	41	101	A	96	60	140	'
66	42	102	B	97	61	141	a
67	43	103	C	98	62	142	b
68	44	104	D	99	63	143	c
69	45	105	E	100	64	144	d
70	46	106	F	101	65	145	e
71	47	107	G	102	66	146	f
72	48	110	H	103	67	147	g
73	49	111	I	104	68	150	h
74	4a	112	J	105	69	151	i
75	4b	113	K	106	6a	152	j
76	4c	114	L	107	6b	153	k
77	4d	115	M	108	6c	154	l
78	4e	116	N	109	6d	155	m
79	4f	117	O	110	6e	156	n
80	50	120	P	111	6f	157	o
81	51	121	Q	112	70	160	p
82	52	122	R	113	71	161	q
83	53	123	S	114	72	162	r
84	54	124	T	115	73	163	s
85	55	125	U	116	74	164	t
86	56	126	V	117	75	165	u
87	57	127	W	118	76	166	v
88	58	130	X	119	77	167	w
89	59	131	Y	120	78	170	x
90	5a	132	Z	121	79	171	y
91	5b	133	[122	7a	172	z
92	5c	134	\	123	7b	173	{
93	5d	135		124	7c	174	-
94	5e	136	^	125	7d	175	}
95	5f	137	-	126	7e	176	~

Operator Precedence and Associativity

The operators are shown in order of precedence, from highest to lowest.

Operator	Associativity
::	unary: left to right binary: right to left
() [] -> .	left to right
++ - + - ! ~ (type) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	right to left
= += -= *= /= %= &= ^= = <<= >>=	right to left
,	left to right

C++ Quick Reference

Commonly Used C++ Data Types

Data Type	Description
char	Character
unsigned char	Unsigned Character
int	Integer
short int	Short integer
unsigned short int	Unsigned short integer
unsigned int	Unsigned integer
long int	Long integer
unsigned long int	Unsigned long integer
long long int	Very large integer
unsigned long long int	Very large unsigned integer
float	Single precision floating point
double	double precision floating point
long double	Long double precision floating point

Forms of the if Statement

Simple if

```
if (expression)
    statement;
```

Example

```
if (x < y)
    x++;
```

if/else

```
if (expression)
    statement;
else
    statement;
```

Example

```
if (x < y)
    x++;
else
    x--;
```

if/else if

```
if (expression)
    statement;
else if (expression)
    statement;
else
    statement;
```

Example

```
if (x < y)
    x++;
else if (x < z)
    x--;
else
    y++;
```

To conditionally-execute more than one statement, enclose the statements in braces:

Form

```
if (expression)
{
    statement;
    statement;
}
```

Example

```
if (x < y)
{
    x++;
    cout << x;
```

Web Sites

For the *Starting Out with C++ Companion* website
www.pearsonhighered.com/gaddis
 For Pearson Higher Ed Computer Science
www.pearsonhighered.com/cs

Commonly Used Operators

Assignment Operators

=	Assignment
+=	Combined addition/assignment
-=	Combined subtraction/assignment
*=	Combined multiplication/assignment
/=	Combined division/assignment
%=	Combined modulus/assignment

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

Relational Operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Logical Operators

&&	AND
	OR
!	NOT

Increment/Decrement

++	Increment
--	Decrement

Conditional Operator ?:

Form:

```
expression ? expression : expression
```

Example:

```
x = a < b ? a : b;
```

The statement above works like:

```
if (a < b)
    x = a;
else
    x = b;
```

The while Loop

Form:

while (expression) Example:	
statement;	cout << x++ << endl;

while (expression) while (x < 100)	
---	--

{	{
statement;	cout << x << endl;
statement;	x++;
}	}

The do-while Loop

Form:

do	Example:
statement;	do
while (expression);	cout << x++ << endl;
do	while (x < 100);
{	
statement;	do
statement;	cout << x << endl;
}	x++;
while (expression);	}
}	while (x < 100);

C++ Quick Reference (continued)**The for Loop****Form:**

```
for (initialization; test; update)
    statement;
```

```
for (initialization; test; update)
{
    statement;
    statement;
```

Example:

```
for (count = 0; count < 10; count++)
    cout << count << endl;
```

```
for (count = 0; count < 10; count++)
{
    cout << "The value of count is ";
    cout << count << endl;
}
```

The switch/case Construct**Form:**

```
switch (integer-expression)
{
    case integer-constant:
        statement(s);
        break;
    case integer-constant:
        statement(s);
        break;
    default:
        statement;
```

Example:

```
switch (choice)
{
    case 0 :
        cout << "You selected 0.\n";
        break;
    case 1 :
        cout << "You selected 1.\n";
        break;
    default :
        cout << "You did not select 0 or 1.\n";
}
```

Using coutRequires `<iostream>` header file.**Commonly used stream manipulators**

Name	Description
<code>endl</code>	Advances output to the beginning of the next line
<code>fixed</code>	Sets fixed point notation
<code>left</code>	Sets left justification
<code>right</code>	Sets right justification
<code>setprecision</code>	Sets the number of significant digits
<code>setw</code>	Sets field width
<code>showpoint</code>	Forces decimal point & trailing zeros to display

Example:

```
cout << setprecision(2) << fixed
    << left << x << endl;
```

Member functions for output formatting

Name	Description
<code>precision</code>	Sets the number of significant digits
<code>setf</code>	Sets one or more <code>ios</code> flags
<code>unsetf</code>	Clears one or more <code>ios</code> flags
<code>width</code>	Sets field width

Example:

```
cout.precision(2);
```

Using cinRequires `<iostream>` header file**Commonly used stream manipulators**

Name	Description
<code>setw</code>	Sets field width
Member functions for specialized input	
<code>getline</code>	Reads a line input as a C-string
<code>get</code>	Reads a character
<code>ignore</code>	Ignores the last character entered
<code>width</code>	Sets field width

Some Commonly Used Library Functions**(The following require `<string>`)**

<code>stod</code>	Converts a string to a double
<code>stoi</code>	Converts a string to an int
<code>stol</code>	Converts a string to a long int

(The following require `<cstdlib>`)

<code>rand</code>	Generates a pseudo-random number
<code> srand</code>	Sets seed value for random numbers

(The following require `<cctype>`)

<code>islower</code>	Returns true if char argument is lowercase
<code>isupper</code>	Returns true if char argument is uppercase
<code>tolower</code>	Returns the lowercase equivalent of the char argument
<code>toupper</code>	Returns the uppercase equivalent of the char argument

(The following require `<cmath>`)

<code>pow</code>	Raises a number to a power
<code>sqrt</code>	Returns square root of a number

(The following require `<cstring>`)

<code>strcat</code>	Appends a C-string to another C-string
<code>strcpy</code>	Copies a C-string
<code>strlen</code>	Returns the length of a C-string

Index

Symbols

- (negation operator), 63, 93, 94
- (subtraction operator), 64
--, 231–235
-=, 107–109
-> (object pointer), 739
-> (structure pointer), 637–638, 639–641
!, 184, 189–190
!=, 152–154
", 30
#, 30
%, 64
%-, 107–109
& (address operator), 503–505
& (reference variables), 355
&&, 184–186, 190
(), 30
* (indirection operator), 509
* (multiplication operator), 64
* (pointer variable declaration), 640–641
*=, 107–109
. (dot operator), 618, 732
/, 64
/* */, 72
//, 30, 72
/=, 107–109
::, 730, 766
;, 30
?:, 201
\\", 35
\', 35
\", 35
\a, 35
\b, 35
\n, 35
\r, 35
\t, 35
{, 30, 166, 240–241
||, 184, 186–189, 190
}, 30, 166, 240–241
~ (destructor), 767
+, 64, 596

A

abstract base classes, 963–967
abstract data type (ADT) definition, 1165–1166
and structures, 613–615
abstraction, 613
accessors, 730
access specifiers
base class, 936–940
in class declarations, 726
accumulator, 261
actual argument, 317
actual parameters, 317
adapter classes, containers, 1030–1031
addition operator (+), 614
address, memory, 6
address operator (&), 503–505
ADT, *see abstract data type*
aggregate classes, member initialization list usage with, 870
aggregation, 866–871
“has a” relationship, 868
in UML diagrams, 870–871
algebraic expressions, 94–96
algorithms, 8, 1029, 1086
categories of, 1086–1087
exhaustive, 1250–1253
factorial, 1228–1231
for permutations detection, 1090–1092
plugging functions into, 1092–1094
QuickSort, 1246–1249
search, 463–469, 1087–1090
selection sort, 482–486
sorting, 476–486, 1087–1090
STL, to perform set operations, 1094–1103
ALU (arithmetic and logic unit), 5
American Standard Code for Information Interchange (ASCII), 50
anonymous enumerated type, 645
append member function, *string* class, 598
application software, 7–8
arguments, 95–96, 317–321
actual, 317
arrays as, 413–423
formal, 317
passing, with pointers, 526–527, 588–590
structures as, 631–633
arithmetic and logic unit (ALU), 5
arithmetic assignment operators, 107
arithmetic expressions, 91–100
arithmetic operators, 63–71, 94
arrays, 381–449
accessing elements, 383–384

arrays (*continued*)
 assigning one, to another, 404–405
 averaging values in, 406–407
 binary search, 466–469
BookInfo structure, 625
 bounds checking, 395–396
 bubble sort, 476–479
 comparing, 409–410
const key word, 418–419
 contents of, 394
 described, 381–382
 elements, swapping, 479–481
enum with, 645–647
 from file to, 392–393
 as function arguments, 413–423
 highest and lowest values, 407
 initialization, 388–391, 627, 777
 inputting and outputting, 384–388
 linear search, 463–466
 linked lists *vs.*, 1123
 memory requirements of, 382–383
 National Commerce Bank case study, 433–435
 of objects, 777–778
 off-by-one error, 396–398
 parallel, 410–413
 partial initialization, 391–392
 partially filled, 407–409
 and pointers, 512–516
 printing contents of, 405–406
 processing array contents, 402–410
 range-based for loop, 398–402
 search algorithms, 463–469
 selection sort, 482–486
sizeof operator, 689
 sizing, 392
 sorting algorithms, 476–486
 and STL vectors, 435–448, 495–497
 structure, 625–627
 summing values in, 406
 three or more dimensions, 431–433
 two-dimensional, 424–431
array class, 1031–1033
 member functions, 1033
array subscript operator ([]), 435, 437–439, 596, 1043–1044
arrow operator (→), 637, 640, 738
 ascending order, sorting in, 476
 ASCII, 50, 51, 197
assign member function, *string* class, 598
 assignment
 combined, 107–109
 memberwise, 830–831
 multiple, 106
 assignment operator (=), 39, 60–61, 106, 163–164, 596
 assignment statement, 60
 associative containers, 1030
 associativity, 93–94, 190–191
at() member function
 array class, 1033
 map class, 1059

at member function
 string class, 598
 vector, 448
atof library function, 579
atoi library function, 579
atol library function, 579
 attributes, 720
auto key word, 61–62
 to define iterator, 1037
 averages
 in arrays, 406–407
 calculating, 97–98

B

back() member function, *array* class, 1033
bad_alloc exception, 1006–1007
bad member function, file stream, 676
 base case, recursion, 1228
 base class, 908
 abstract, 963–967
 multiple, 970–975
 base class access specification, 912–913, 920–921
 base class functions, redefining, 936–940
 base class pointers, 955–957
 BASIC, 11
begin member function
 string class, 598
begin() member function
 array class, 1033, 1035
 map class, 1060–1061
 binary digit (bit), 6
 binary files, 270, 688–689
 binary numbers, 9
 binary operators, 63
 binary predicate, 1111
 binary search, 466–469
 efficiency, 469
 recursive version, 1223–1226
binary_search() function, 1087–1090
 binary trees, 1078, 1257–1258
 applications of, 1257–1258
 child nodes, 1257
 creating, 1261–1262
 deleting a node, 1268–1277
 described, 1257–1258
 inorder traversal, 1265
 inserting a node, 1262–1264
 leaf nodes, 1257
 NULL address, 1257
 operations, 1260–1277
 postorder traversal, 1265
 preorder traversal, 1265
 root node, 1257
 searching, 1267–1268
 searching for a value in, 1247
 search trees, 1259
 subtrees, 1258
 templates for, 1277–1282

traversing, 1264–1266
tree pointer, 1257
binding, 948
 dynamic, 949
 static, 949
bit, 6
blocks, of code, 213–216, 240–241
block scope, 214
blueprints, classes as, 723
body
 of function, 307, 309
 of loop, 237
bool
 data type, 58
 flag, 183
 returning from function, 338–340
Boolean expressions, 58, 152, 393
bounds checking, for arrays, 395–396
braces, 240–241
break statement, 206, 207, 288–290
bubble sort, 476–479
byte, 5–6

C

C, 11
C#, 11
C++, 11
C++ 11, 437
 range-based **for** loop with **vector** in, 439
calling a function, 308–315
capacity member function
 string class, 598
 vector, 448
capacity() member function, **vector** class, 1052–1053
capture list, 1111
capitalization, 43
case conversion, character, 561–563
case statement, 205–206
case study
 Demetris Leadership Center, 469–476, 486–494
 dollarFormat function, 603–604
 General Crates, Inc., 134–137
 Home Software Company, 603–604,
 788–791
 National Commerce Bank, 433–435
 United Cause, 544–549
catch block, 989
catch key word, 989
cbegin() member function, **array** class, 1033, 1038
CD (compact disc), 6
cend() member function, **array** class, 1033, 1038
central processing unit (CPU), 3–5
character case conversion, 561–564
 tolower function, 562
 toupper function, 562
character literals, 49, 50–53
characters, 17, 557–563
 character literals, 49, 50–53
 comparing, 197–200

converting cases, 561–563
and **string** objects, 120–126
character testing, 557–561
functions, 559–560
isalnum function, 558
isalpha function, 558
isdigit function, 558
islower function, 558
isprint function, 558
ispunct function, 558
isspace function, 558
isupper function, 558
char data type, 49–53
cin, 18, 85–90
entering multiple values, 87–89
getline member function, 121, 567, 582–583
get member function, 122–124
ignore member function, 125, 583
inputting characters, 122
keyboard buffer, 89
Circle pointer, 639
circularly linked list, 1153
class constructors, 923–927
classes, 719–816, 817–906
 abstract base, 963–967
 accessors, 730
 access specifiers, 726–727
 aggregation, 866–871
 arguments to constructors, 759–764
 arrays of class objects, 777–778
 base, 908
 as blueprint, 723
 collaborations, 871–874
 const member functions, 728, 731
 constructor overloading, 771–773
 constructors, 754–766, 922–923
 conversion of class objects, 864–866
 copy constructor, 831–834
 and data hiding, 731
 declaration statements, 726
 default constructor, 759, 766
 defining class objects, 731–743
 derived, 908–909
 described, 726
 destructors, 767–770, 922–926
 dot operator (.), 732
 dynamically allocated objects, 739–740, 769–770
 finding, 794–802
 forward declaration, 827
 friend functions, 825–826
 getter function, 730
 “has a” relationship, 868
 hierarchies of, 941–946
 Home Software Company case study, 782–783
 implementation file, 746
 include guard, 746–747
 inheritance, 907–916
 inline member functions, 751–753
 instance members, 817–824
 “is a” relationship, 908–915, 947

classes (continued)

- member functions, defining, 729–730
- memberwise assignment, 830–831
- multiple inheritance, 970–977
- mutators, 730
- objects *vs.*, 719–722
- operator overloading, 837–863
- overloading member functions, 775
- PassFailExam*, 944
- placement of public and private members, 728–730
- pointers, 738–741
- polymorphism, 947–963
- private member functions, 775–777
- private members, 726, 728–729, 744–745
- problem domain, 795
- and procedural/object-oriented programming, 719–725
- protected members and class access, 916–921
- public member functions, 728–729
- public members, 726, 728–729
- redefining base class functions, 936–940
- responsibilities of, 794–816
- scope resolution operator (::), 730, 766
- setter function, 730
- specification and implementation, 745–751
- stale data, avoiding, 738
- static members, 817–824
- templates, 1008–1013
- and UML, 792–794
- virtual functions, 949–953
- whole-part relationship, 868
- class implementation file, 746
- class specification file, 745
- class template objects, 1018
- class templates, 1014–1017
 - defining objects of, 1018–1019
 - and inheritance, 1020–1022
 - linked list, 1131–1133
 - type parameter, 1008
- clear member function**
 - file stream objects, 676
 - string* class, 598
 - vector*, 445, 448, 1014–1015
- C++ library, 557, 561, 564, 722
- close member function**, file stream objects, 273
- closing of file, 273
- cmath header file, 96, 126
- COBOL, 11
- code reuse, 306
- collaborations, class, 871–876
- combined assignment operators, 107–109
- comments, 27, 71–73
 - /* */, 72
 - //, 27
 - multi-line, 72–73
 - single-line, 71–72
- compact disc (CD), 6
- compare member function**, *string* class, 597–599
- compilers, 12, 665
 - default operations provided by, 894–895
- compound operators, 107
- concatenation, 569–570
- conditional expression, 201–204
 - value of, using, 202–203
- conditional loop, 251
- conditional operator, 201–204
- console, 31
- console output, 31
- const**
 - as array parameter, 418–419
 - copy constructors, 831–834
 - member functions, 728, 731
- const_iterator**, and mutable iterator, 1038
- constants, 41
 - arguments passed to, 759–766
 - in base and derived classes, 922–933
- constructors, 754–757
 - copy, 831–836
 - default, 759, 764–766
 - default arguments with, 764–766
 - global, 344–346
 - named, 73–75
 - overloading, 771–773
 - pointers to, 525–527
- constructors
 - associative, 1030
 - class, 923–927
 - delegation, 774–775
 - derived classes, 922
 - inheritance, 933–934
 - sequence, 1030
 - STL, 435
- containers
 - adapter class, 1030–1031
 - array class, 1031–1033
 - associative, 1030
 - definition, 1029
 - iterators and, 1034
 - sequence, 1030
- continue statement**, 290–292
- control unit, 5
- control variable, 239
- conversion
 - object, 864–866
 - string/numeric*, 579–585
- copy assignment operator, 841–843
- copy constructors, 831–836
 - const parameters in, 834–836
 - default, 836
 - and function parameters, 836
- list* class, 1155
- map* class, 1055
- set* class, 1079
- vector* class, 1041
- copy member function, *string* class, 599
- count-controlled loops, 251, 259–260
- count() member function
 - multimap* class, 1075–1076
 - multiset* class, 1085
 - set* class, 1082
- count_if() function, 1093–1094

counters, 245–246
cout, 18, 31–36, 155
 fixed manipulator with, 116–117
 left manipulator with, 118–119
 right manipulator with, 118–119
 setprecision manipulator with, 113–115
 setw manipulator with, 111–112
 showpoint manipulator with, 117–118
cout statement, 233
C programming language, 10–11, 566
C++ programming language, 10, 11, 22
 arithmetic operators, 63–71
 assignment operation, 60–61, 62
 auto key word, 61–62
 bool data type, 58
 char data type, 49–53
 comments, 71–73
 cout object, 31–36
 floating-point data types, 55–58
 identifiers, 42–43
 #include directive, 36–38
 initialization, 60–61, 62
 integer data types, 43–48
 named constants, 73–75
 parts of program, 27–31
 programming style, 75–76
 scope, 62–63
 sizeof operator, 59
 string class, 53–54
 variables and literals, 38–41
CPU (central processing unit), 3–5
crbegin() member function, array class, 1033
CRC cards, 875–876
crend() member function, array class, 1033
C++ runtime library, 126
cstdint header file, 579
cstring header file, 568, 573, 577
C-strings, 564–576
 in arrays, 566–568
 comparing, 574–576
 concatenation, 569–570
 copying, 570–571
 described, 564–565
 filenames as, 288
 handling functions, 585–590
 length of, 568–569
 library functions, 568–578
 and null terminators, 564–565
 numeric conversion functions, 579–585
 searching within, 570–574
 strcmp function, 593
 and string literals, 565–566
c_str member function, 288
ctime header file, 129

D

database management systems (DBMS), 665
data hiding, 731

data() member function, array class, 1033
data types
 abstract, 613–614
 bool, 58
 char, 49–53
 coercion, 100
 conversion, 100–101
 demotion, 100
 double, 55
 enumerated, 642–653
 float, 55
 floating-point, 55–58
 generic, 1008
 int, 44, 45
 integer, 43–48
 long, 45
 long double, 45
 numeric, 44
 primitive, 614
 promotion, 100
 ranking, 100–101
 short, 44, 45
 size of, determining, 59
 string class, 53–54
 type casting, 103–106
 unsigned int, 44, 45
 unsigned long, 44, 45
 unsigned short, 44, 45
debugging
 desk-checking, 21
 hand-tracing, 132–134
 stubs and drivers, 367–368
decision making, 151–230
 blocks and scope, 213–216
 checking numeric ranges, 191
 comparing characters and strings, 197–200
 conditional execution, 164–166
 conditional operator, 201–204
 flags, 183–184
 if/else if statements, 178–180, 182–183
 if/else statements, 168–170
 if statement, 156–164
 logical operators, 184–191
 menus, 192–195
 nested if statements, 171–174
 relational operators, 151–155
 relationship, value of, 152–154
 semicolons, 160
 switch statement, 204–213
 truth, 154–156
 validating user input, 195–196
decode, 5
decrement operator (--), 231–236
 in mathematical expressions, 235
 postfix and prefix modes, 233–235
 in relational expressions, 235–236
default arguments, 351–354, 764–766
default constructors, 759, 766
1st class, 1154

default constructors (*continued*)
 map class, 1055
 set class, 1079
 vector class, 1040
default copy constructor, 836
default statement, 205–206
#define directive, 747
delete operator, 531
Demetris Leadership Center case study,
 469–476, 486–494
depth of recursion, 1224
deque (STL type), 1214–1216
 front member function, 1215
 pop_front member function, 1215
 push_back member function, 1215
dequeue operation, 1196–1197
dereferencing, of pointers, 509
derived class, 908
descending order, sorting in, 476
designing programs, 18–22
desk-checking, 21
destructors, 767–769, 922
 base and derived classes, in, 920–921
 virtual, 947–963
digital versatile disc (DVD), 6
direct access file, 271
direct recursion, 1234
disk drive, 6
division by zero, 169, 989–990
dollarFormat function case study, 603–604
dot operator (.), 618, 637, 732
double data type, 55
double literals, 57
double precision, 55
doubly linked list, 1153
do-while loops, 246–250, 266
 with menus, 248–251
 as posttest loop, 247
drivers, 367–369
dummy parameter, 849
DVD (digital versatile disc), 6
dynamic binding, 949
dynamic memory allocation, 530–534
 bad_alloc exception, 1006–1007
 objects, 738–741, 759, 770
 for structures, 639–640
dynamic queues, 1195, 1207–1214
dynamic stacks, 1166, 1182–1192

E

efficiency
 binary search, 469
 linear search, 465–466

elements (array)
 accessing, 383–384
 described, 382
 processing, 402–410
 removing, from vectors, 444–445

emplace() member function
 map class, 1058

multimap class, 1075
set class, 1081
vector class, 1050–1052

emplace_back() member function, vector class,
 1050–1052

empty member function
 string class, 597
 vector, 446–447, 448

empty() member function, array class, 1033

encapsulation, 720

#endif directive, 746–747

endl, 33

end member function
 string class, 597

end() member function
 array class, 1033, 1035
 map class, 1060–1061

end of file, detecting, 283–285

end-of-file marker, 668

enqueue operation, 1195–1198

enum, 642–653
 anonymous, 645
 with arrays, 645–647
 assigning, to int variables, 643
 assigning integers to enum variables, 643
 comparing enumerators, 643–644
 declaration and definition, 651
 defining, 643
 math operators with, 645
 outputting values with, 647–649
 and scope, 650–651
 specifying values, 649–650
 strongly-typed, 651–653
 enumerated data types, 642–653, *see also enum*
 enumerators, 643–644, 650–651
eof member function, file stream, 676

equal-to operator (==), 152–153, 163–164

equal_range() member function, multiset
 class, 1085

erase member function
 string class, 597

erase() member function
 map class, 1059
 multimap class, 1078

errors
 logical, 21
 off-by-one, 396–398
 recovering from, 999–1001
 syntax, 12

error testing, files, 675–678

escape sequences, 34–36
 \\", 35
 \\, 35
 \\, 35
 \\a, 35
 \\b, 35
 \\n, 35
 \\r, 35
 \\t, 35
 newline, 35

- exception handler, 990
- exceptions, 989–1007
 - `bad_alloc`, 1006–1007
 - catch block, 990
 - dynamic memory allocation, 531
 - exceptions handler, 990
 - extracting data from, 1001–1005
 - handling, 990–993
 - memory allocation error, 1006–1007
 - multiple, handling, 996–1001
 - `new operator`, 531
 - not catching, 993
 - object-oriented handling, 993–996
 - recovering from errors, 999–1001
 - rethrowing, 1006
 - throwing, 990
 - `throw` key word, 990
 - throw point, 990
 - try block, 990
 - try/catch construct, 990–993
 - unwinding the stack, 1005
- executable file, 12
- execute, 5
- exhaustive algorithms, 1250–1253
- `exit()` function, 364–366
- exponents, 95–97
- expressions, 85–137
 - algebraic, 94–96
 - arithmetic, 91–100
 - case study, 134–137
 - characters and `string` objects, 120–126
 - `cin` object, 85–90
 - conditional, 201–204
 - C-style and prestandard C++ forms, 107–108
 - formatting output, 110–120
 - and hand tracing programs, 132–134
 - initialization, 252, 256
 - mathematical, 91–100, 235
 - and mathematical library functions, 126–132
 - multiple and combined assignment, 106–110
 - overflow and underflow, 102–103
 - relational, 235–236
 - type casting, 103–106
 - type conversion, 100–101
- F**
- factorial algorithm, 1228–1231
- `fail` member function, file stream, 286, 675, 676
- false values, 152, 154–155, 162–163
- fetch, 5
- fetch/decode/execute cycle, 5
- Fibonacci numbers, 1236–1237
- field width, 111, 113
- FIFO (first-in first out), 1195
- file access flags, 666
 - `ios::app`, 666
 - `ios::ate`, 666
 - `ios::badbit`, 676
 - `ios::binary`, 666
 - `ios::eofbit`, 676
 - `ios::failbit`, 676
 - `ios::goodbit`, 676
 - `ios::hardfail`, 676
 - `ios::in`, 666
 - `ios::out`, 666
 - `ios::trunc`, 666
- file access methods, 271
- file buffer, 273
- filename extensions, 271
- filenames
 - and file stream objects, 271–272
 - user-specified, 286–287
- file open errors, 285–286
- file operations, 665–717
 - append mode, 666
 - binary files, 688–693
 - described, 666
 - end-of-file marker, 668
 - `fstream` data type, 666, 667, 705
 - member functions for reading and writing, 678–685
 - opening files for input and output, 705–709
 - opening files with definition statements, 670–671
 - opening multiple files, 686–688
 - open modes of `ifstream` and `ofstream`, 669–670
 - random-access files, 697–705
 - reading a character, 678
 - reading a line, 680
 - records with structures, 693–697
 - rewinding, 704–705
 - writing a character, 684–685
- files
 - closing of, 273
 - for data storage, 269–288
 - detecting end of, 283–285
 - file open errors, 285–286
 - input/output program, 272
 - opening, and creating file objects, 272–273
 - processing, with loops, 282–283
 - reading from, 278, 280–281
 - read position, 279–280
 - types of, 270–271
 - user-specified filenames, 286–287
 - writing to, 274
- file stream objects, 271
 - closing, 273
 - creating, 272–273
 - member functions, 678–685
 - passing to functions, 673–675
- fill constructor
 - `vector` class, 1040
- `fill(value)` member function, `array` class, 1033
- filters, 686
- final, 961–963
- `find()` member function
 - `map` class, 1062
 - `multimap` class, 1076–1077
 - `set` class, 1082–1083
- finding classes, 794–802
- `find` member function, `string` class, 599

- first-in first-out (FIFO), 1195
 fixed manipulator, 116–117, 671
 flags, 183–184
 integer, 184
 flash memory, 6
 float data type, 55
 floating-point data types, 55–58
 comparing, 161–162
 and integer variables, 57–58
 floating-point literals, 56–57
 floating point numbers, 44
 float literals, 57
 floppy disk drive, 6
 flowcharts, 20
for loops, 251–261, 266
 counter variable, 255
 initialization expression, 252, 256
 omitting expressions of, 258–260
 as pretest loop, 255
 test expression, 252
 update expression, 252, 256, 257–258
 user-controlled, 256–257
 while and do-while *vs.*, 254–255
for_each() function, 1092–1093
 formal argument, 317
 formal parameters, 317
 formatting output, 110–120, 671–673
 fixed manipulator, 116–117
 left manipulators, 118–119
 right manipulators, 118–119
 setprecision manipulator, 113–115
 showpoint manipulator, 117–118
 FORTRAN, 11
 forward declaration, 827
forward_list class, 1158
 friend class, 829
 friend functions, 825–829
 friend key word, 825
 front member function
 deque, 1214
 front() member function, array class, 1033
 fstream header file, 272
 fstream objects, 666
 function arguments
 file stream objects as, 673–675
 structures as, 631–633
 function call operator, 1107
 function call statements, 308
 function declarations, 315
 function header, 307
 function object
 anonymous, constructing, 1110
 definition, 1107
 function parameters
 pointers as, 521–529
 reference variables as, 354–359
 function pointer, 1092
 function prototypes, 315–316
 functions, 28
 bool value, returning, 338–340
 calling, 308–315
 default arguments, 351–354, 764
 defining, 306–307
 exit(), 364–366
 friend, 825–829
 generic, 1008
 inline, 751–753
 local and global variables, 340–348
 main, 28–29, 308, 545
 member, 720
 menu-driven programs, 324–328
 modular programming, 305–306
 overloading, 360–364
 overriding, 958
 passing data by value, 322–323
 pointers, returning, 534–536
 prototypes, 315–316
 pure virtual, 963–967
 recursive, 1223–1227
 redefining base class, 936–940
 reference variables as parameters, 354–359
 return statement, 328–329
 sending data into, 317–321
 static local variables, 348–351
 static member, 822–825
 string handling, 585–590
 structures, returning, 634–636
 stubs and drivers, 367–369
 value-returning, 330–338
 virtual, 949–952, 963–967
 void, 307
 function signature, 361
 function templates, 1008–1013
 with multiple types, 1012–1013
 overloading with, 1013
 using operators in, 1012
- C**
- games, 269
 GCD (greatest common divisor), 1235–1246
 General Crates, Inc. case study, 134–137
 generalization, inheritance and, 907–908
 generic data types, 1008
 generic functions, 1008
 getArea member function, 724
 getCircleData function, 635
 getItem function, 633
 getLine member function, 121
 cin, 567, 582–583
 file streams, 680–683
 get member function, 122–124
 file streams, 683–684
 getter functions, 730
 global constants, 344–346
 global variables, 342–344
 good member function, file stream, 676
 greatest common divisor (GCD), 1235–1236

H

handler, exception, 990
 hand tracing programs, 132–134
 Hanoi, Towers of, 1243–1246
 hardware, 3–7
 CPU, 3–5
 input devices, 7
 main memory, 5–6
 output devices, 7
 secondary storage, 6
 “has a” relationship, 868
 header
 function, 307
 loop, 237, 252
 header file, 28
 cmath, 96, 126
 cstdlib, 365, 579
 cstring, 568
 ctime, 129
 fstream, 272
 iomanip, 112, 115
 iostream, 28, 36–37
 string, 53, 591
 STL, 1030
 hexadecimal literals, 48
 hiding data, 720, 731
 hierarchies, class, 941–946
 hierarchy chart, 20
 high-level languages, 10
 Hoare, C.A.R., 1246
 Home Software Company case study, 603–604

I

identifiers, 42–43
 capitalization, 43
 legal, 43
if/else if statements, 178–180
 nested decision structures *vs.*, 182–183
 trailing else, 181
if/else statements, 168–170
#ifndef directive, 746–747
if statement
 conditionally executed code, 158, 164–166
 expanding, 164–167
 floating-point comparisons, 161–162
 nested, 171–174
 programming style, 161
 semicolon in, 160
ifstream objects, 272, 669
 close member function, 273
 open member function, 273
 >> with, 278
ignore member function, **cin**, 125, 582
 image editors, 269
 implementation file, class, 745
#include directive, 28, 36–38, 725, 746, 750
 in-place initialization, 750
 include file directory, 749

includes() function, 1095

include guard, 746–747
 increment operator (++), 231–236
 in mathematical expressions, 235
 postfix and prefix modes, 233–235
 in relational expressions, 235–236
 indirection operator (*), 509
 indirect recursion, 1234
 infinite loops, 240
 inheritance, 907–915
 base class, 908
 class hierarchies, 941–946
 and class templates, 1018–1022
 constructor, 933–934
 constructors and destructors, 922–927
 derived class, 908–909
 “is a” relationship, 908–915
 multiple, 970–977
 redefining functions, 936–940
 initialization, 60–61, 62
 array, 388–391, 427–428, 627
 for loops, 252, 256
 pointers, 518–519
 structure, 622–624
 structure array, 627
 inline expansion, 753
 inline member functions, 751–753
 inorder traversal, binary trees, 1265
 input, 17–18
 array contents, 384–388
 with **cin**, 85–89
 reading, into **string** objects, 591
 input devices, 7
 input file, 269, 272
 input–output stream library, 36
 input validation
 and decision making, 195–196
 and while loops, 243–245
insert member function
 string class, 603
insert() member function
 map class, 1058
 multimap class, 1075
 set class, 1081
 vector class, 1045–1047
 instances
 of class, 723
 of classes, 731–741
 of structures, 618
 variables, 817–818
 instantiation, 731
int, 44, 45, 643
 integer data types, 43–48
 integer division, 64, 66, 101
 integer flags, 184
 integer literals, 47
 integer variables, 57–58
 integrated development environments (IDE), 12
iomanip header file, 112, 115

i
ios::app access flag, 666
ios::ate access flag, 666
ios::badbit status flag, 676
ios::binary access flag, 666, 689
ios::eofbit status flag, 676
ios::failbit status flag, 676
ios::goodbit status flag, 676
ios::hardfail status flag, 676
ios::in access flag, 666
ios::out access flag, 666
iostream header file, 28, 36–37
ios::trunc access flag, 666
isalnum library function, 558
isalpha library function, 558
“is a” relationship, 908–915, 947
is_permutation() function, 1090–1092
isdigit library function, 558
islower library function, 558
isprint library function, 558
ispunct library function, 558
isspace library function, 558
isupper library function, 558
iteration, 238
iterators, 1034–1039
 association with containers, 1034
 auto to define, 1037
 categories of, 1034
 definition, 1034–1035
 and **map** class, 1060–1062
 mutable, 1038
 reverse, 1038–1039
 and **set** class, 1082
 usage in **vector** class, 1045
itoa library function, 584

J
Java, 11
JavaScript, 11

K
keyboard buffer, 89
key-value pairs, 1054
key words, 14

L
lambda expression, 1107, 1111–1112
language elements, 14
last-in-first-out (LIFO), 1165
left manipulator, 118–119
legacy code, 566
legal identifiers, 43
length, of C-strings, 568–569
length member function, **string** class, 597, 603
library functions, 95
 atof, 579
 atoi, 579
 atol, 579
 for C-strings, 578–578
isalnum, 558
isalpha, 558
isdigit, 558
islower, 558
isprint, 558
ispunct, 558
isspace, 558
isupper, 558
itoa, 584
strcat, 569–570, 577
strcmp, 574–576
strcpy, 570–571, 578
strlen, 568–569, 577
strncat, 571
strncpy, 571
strstr, 572–574, 578
tolower, 561
toupper, 561
lifetime, of variables, 342
LIFO (last-in, first-out), 1165
linear search
 algorithm for, 463–465
 efficiency, 465–466
lines, 16
LinkedList class template, 1147
linked lists
 appending a node, 1126–1131
 arrays and vectors *vs.*, 1123–1158
 circularly, 1153
 circularly linked, 1153
 class as node type, 1147–1153
 composition of, 1124
 counting nodes, 1238–1239
 declarations, 1124–1125
 deleting a node, 1137–1140
 described, 1123
 destroying, 1139–1140
 displaying nodes in reverse, 1239–1241
 doubly linked, 1153
 inserting a node, 1133–1137
 list head, 1124
 NULL address, 1124
 operations, 1125–1141
 recursion with, 1237–1241
 self-referential data structure, 1125
 singly linked, 1153
 template for, 1141–1153
 traversing, 1131–1133
 variations of, 1153
linker, 12
list class, 1154–1158
 definition statements, 1154–1155
 member functions, 1155–1157
list head, linked list, 1124
literals, 40–41
 character, 49, 50–53
 double, 57
 float, 57
 floating-point, 56–57
 hexadecimal, 48
 integer, 47

- long integer, 47
 - octal, 48
 - string, 41, 50–53, 565–566
 - local scope, 214
 - local variables, 340–342
 - initializing, with parameter values, 342
 - lifetime of, 342
 - with same name as global, 347–348
 - static, 348–351
 - logical errors, 21
 - logical operators, 184–191
 - && (AND), 184–186
 - ! (NOT), 189–190, 576
 - || (OR), 186–189
 - associativity, 190–191
 - and numeric ranges, 191
 - precedence, 190–191
 - short-circuit evaluation, 185, 187
 - long data types, 44, 45
 - long double data types, 44, 45
 - long double precision, 55
 - long integer literals, 47
 - long long integer literal, 48
 - loop header, 237, 252
 - loops, 231–292
 - breaking and continuing, 288–292
 - conditional, 251
 - control variable, 239
 - count-controlled, 251, 259–260
 - counters, 245–246
 - described, 236
 - do-while, 246–250, 266
 - and files, 269–288
 - for, 251–261, 266
 - and increment/decrement operators, 231–236
 - infinite, 240
 - input validation with, 243–245
 - nested, 266–268
 - posttest, 246–248
 - pretest, 239, 255
 - processing files with, 282–283
 - programming style, 241–242
 - running total, 261–263
 - selecting, 265–266
 - sentinels, 264–265
 - user-controlled, 248, 256–257
 - while, 236–242, 265
 - lowercase conversion, character, 561–563
 - low-level languages, 10
 - lvalues, 886–888
 - references, 888
- M**
- machine language, 9
 - main function, 28–29, 545
 - main memory, 5–6
 - manipulators
 - fixed, 116–117
 - left, 118–119
 - right, 118–119
 - setprecision, 113–115
 - showpoint, 117–118
 - stream, 33, 119
 - mantissa, 55
 - map class, 1054
 - add new elements to, 1057
 - at() member function, 1059
 - definition statements, 1055
 - emplace() member function, 1058
 - erase() member function, 1059
 - initializing, 1056–1057
 - insert member function, 1058
 - iterators and, 1060–1062
 - keys in, 1069–1071
 - member functions, 1055–1056
 - range-based for loop, 1060
 - storing own classes objects as values in, 1065–1069
 - storing vectors as values in, 1062–1064
 - mathematical expressions, 91–100, 235
 - algebraic, to programming statements, 94–95
 - associativity, 93–94
 - exponents, 95–97
 - grouping with parentheses, 94
 - operator precedence, 92–93
 - mathematical library functions, 126–132
 - random numbers, 128–131
 - mathematical operators, with enum, 645
 - max_size() member function, array class, 1033
 - max_size() member function, vector class, 1052–1053
 - member access specification
 - defined, 920
 - inherited, 920
 - member functions, 720, 945
 - array class, 1033
 - binding, 948
 - dynamic binding, 949
 - getLength, 744
 - getObjectCount, 820
 - getTaxRate, 763
 - list class, 1155–1157
 - map class, 1055–1056
 - multimap class, 1072–1073
 - other overloaded, 775
 - overriding, 958
 - private, 775–777, 781
 - public, 781–782
 - redefining, 936–940, 958
 - set class, 1080
 - setLength, 744, 745
 - static, 822–824
 - static binding, 949
 - static stack class, 1167
 - vector class, 1041–1043
 - virtual, 949–952
 - withdraw, 783
 - member initialization list, 758
 - usage with aggregate classes, 870
 - members, of structures, 615, 618–621, 640–641

member variable
 static stack class, 1167
 memberwise assignment, 830–831
 memory
 flash, 6
 main, 5–6
 random-access, 5
 memory address, 6, 404
 memory allocation, dynamic, *see*
 dynamic memory allocation
 memory leak, 532
 avoiding, 541–544
 memory requirements of arrays, 382–383
 menu-driven programs, 192, 324–328
 menus, 192–195, 209–211
 message, 23
 methods, 719
 microprocessors, 4
 modular program, 324
 modular programming, 305–306
 move semantics, 889–894
 implementation in class, 894
 multi-line comments, 72–73
multimap class, 1072
 adding elements to, 1075
 count() member function, 1075–1076
 emplace() member function, 1075
 erase() member function, 1078
 find() member function, 1076–1077
 insert member function, 1075
 member functions, 1072–1073
 multiple assignment, 106
 multiple inheritance, 970–977
multiset class, 1085
 mutable iterators, 1038
 mutators, 730

N

named constants, 73–75
 names
 of functions, 307
 of variables, 43, 215–216, 347–348
nameSlice functions, 586
 namespaces, 28
 National Commerce Bank, case study, 433–435
 negation operator (2), 63, 93, 94
 nested if statements, 171–174
 nested loops, 266–268
 break statement in, 290
 nested structures, 627–630
 newline escape sequence, 35
 new operator, 530–532
 nodes, 1123
 appending, 1126–1131
 of binary trees, 1257, 1262–1264, 1268–1277
 class as node type, 1147–1153
 counting, 1238–1239
 deleting, 1137–1140, 1268–1277
 displaying, in reverse, 1239–1241

inserting, 1133–1137, 1262–1264
 of linked lists, 1123, 1126–1137,
 1147–1153, 1238–1241
 NOT (!) operator, 576
 null character, 51, 564
 null pointer, 508
nullptr, 508
 null statement, 160
 null terminator, 51, 564–565
 userName[count], 587, 588
 numbers, 17
 numbers, random, 128–131
 numeric data
 checking ranges of, 191
 integer data types for, 44
 from text files, 280–281

O

object aggregation, 866–871
 object code, 12
 object conversion, 864–866
 object file, 12
 object-oriented design
 aggregation, 866–871
 class collaborations, 871–874
 classes, finding, 794–803
 CRC cards, 875–876
 generalization and specialization, 907–908
 inheritance, 907–916
 problem domain, 795
 responsibilities, identifying, 800–802
 UML, 792–794, 870–871
 object-oriented programming (OOP), 22, 23, 719–725
 game simulation, 876–882
 problem solving, 781–788
 Unified Modeling Language, 792–794
 object reusability, 722
 objects, 719, 722–724, *class vs.*
 array of, 777–778
 attributes, 720
 data hiding, 720–721
 dynamically allocated, 739–741, 759
 encapsulation, 720
 methods, 721
 pointers, 738–741
 state, 735
 octal literals, 48
 off-by-one error, 396–398
 off position, 6
ofstream objects, 272, 669–670
 close member function, 273
 open member function, 273
 <<, used with, 274
 one-dimensional arrays, 425
 on position, 6
 OOP, 22, 23, 719–725
 open member function, **file stream objects**, 273
 operands, 60, 100
 operating systems, 7

- operator functions, 838
 operator overloading, 837–863
 [] operator, 596, 858–861
 >> and << operators, 854–858
 = operator, 838–839
 general issues, 843–844
 math operators, 844–849
 postfix ++ operator, 849–850
 prefix ++ operator, 849
 relational operators, 852–854
 operators, 14, 15
 (-) negation, 63
 - (subtraction), 64
 --, 231–235
 -=, 107–109
 -> (object pointer), 739
 -> (structure pointer), 637–638, 639–641
 !, 184, 189–190, 576
 !=, 152–154
 %(modulus), 64, 93
 %-, 107–109
 &(address), 503–505
 &&, 184–186, 393
 • (indirection), 509
 • (multiplication), 64, 93–94
 • (pointer variable declaration), 640–641
 *=, 107–109
 . (dot operator), 618, 732
 / (division), 64, 93
 /=, 107–109
 ?:, 201
 || (OR), 184, 186–189
 +, 64, 93, 596
 ++, 231–235
 +=, 107–109, 596
 <, 152–154
 <<, 32, 274
 <=, 152–154
 =, 596
 ==, 152–154, 163–164, 241
 >, 152–154
 >=, 152–154
 >>, 86, 596
 [] operator, 435, 437–439, 596
 associativity, 93–94, 190–191
 binary, 63, 64
 copy assignment, 841–843
 overloading, 837–863
 precedence of, 92–93, 190–191
 relational, *see* relational operators
 scope resolution (: :), 730
 string class, 96
 ternary, 63
 unary, 63
OR
 with enum, 647–649
 formatting, 671–673
 || logical operator, 186–189
output, 18
 output devices, 7
 output file, 269, 272
 overflow, 102–103
 overhead, 1228
 overloading functions, 360–364
 constructors, 771–773
 member functions, 767
 templates, 1013
override, 961–963
 overriding, 958
- P**
- parallel arrays, 410–413
 parameters
 array, 417
 pointers as, 521–529
 reference variables as, 354–359
 parentheses, 94
 partially filled arrays, 407–409
 Pascal, 11
 passing by value, 322–323
 passing to functions
 with pointers, 588–590
 percentage discounts, 67–70
 permutations, algorithms for detection, 1090–1092
 pointers, 503–549
 address operator (&), 503–505
 arithmetic with, 516–517
 and arrays, 512–516
 base class, 955–957
 comparing, 519–521
 constant, 528–529
 constant, to constants, 529
 to constants, 525–527
 creating and using, 507–512
 dynamic memory allocation, 530–534
 as function parameters, 521–529
 initializing, 518–519
 to objects, 738–741
 passing C-string arguments with, 588–590
 returning, from a function, 534–536
 smart, 541–543
 structure pointer operator, 637–638, 640, 641
 to structures, 637–639
 structures containing, 697
 United Cause case study, 544–548
 pointer variables, *see* pointers
 polymorphism, 947–963
 abstract base classes, 963–967
 base class pointers, 955–957
 dynamic binding, 949
 overriding, 958
 pure virtual function, 963–967
 and references or pointers, 953–955
 references/pointers, 953–955
 static binding, 949
 virtual destructors, 947–963
 virtual functions, 949–952, 963–967

- pop_back** member function
 vector, 448
- pop_front** member function
 deque, 1214
- pop operation (stacks), 1166
- postfix mode, 233–235
- postorder traversal, binary trees, 1265–1266
- posttest loop, 246–248
- pow** function, 95–97, 128
- precedence, operator, 92–93, 190–191
- predicate**, 1111
- prefix, template, 1008, 1013
- prefix modes, 233–235
- preorder traversal, binary trees, 1265
- preprocessor, 11
- preprocessor directive, 28, 29
- prestandard C++
 standard *vs.*, 75–76
 type cast expressions, 107–108
- pretest loop, 239
- priming read, 244
- primitive data types, 614
- priority_queue** (adapter class), 1031
- private member functions, 775–777
- private members, class, 728–729, 744–745
- problem domain, 795
- procedural programming, 22–23, 719–722
- processing, 19
- processing array contents, 402–410
- programmability, of computers, 1–2
- programmer-defined data types, 435
- programmer-defined identifiers, 14, 15
- programmers, 2, 9
- programming, 1–23
 computer systems
 input, processing, and output, 17–18
 procedural and object-oriented, 22–23
 process of, 18–22
 programmability of computers, 1–2
 program elements, 14–17
 programs and programming languages, 8–13
 programming languages, 8–11, *see also* specific languages
 high-level, 10
 low-level, 10
- programming process, 18–22
- programming style, 75–76
 and if statements, 161
 and nested decision structures, 174–175
 and while loops, 241–242
- programs
 defined, 1
 described, 8–9
 designing/creating, 18–22
 elements, 14–17
 primary activities of, 17–18
- protected members, 916–921
- prototypes, function, 315–316
- pseudocode, 20–21, 64, 466
- public member functions, 728–729
- public members, class, 728–729
- punctuation, 14, 15–16
- pure virtual function, 963–967
- push_back** member function
 deque, 1214
 vector, 441–442, 448
- push operation (stacks), 1166
- put** member function, file streams, 684–686
- Python, 11
- Q**
- queue** (adapter class), 1031
- queue** (STL type), 1216–1217
- queue** container adapter, 1216–1217
- queues, 1195–1217
 applications of, 1195
 array-based, 1198
 crawling problem with array, 1197
 dequeueing, 1195
 described, 1195
 dynamic, 1195, 1207–1210, 1207–1214
 dynamic template, 1211–1214
 empty, detecting, 1198
 enqueueing, 1195–1198
 first-in, first-out, 1195
 full, detecting, 1198
 linked list-based, 1207–1210
 operations, 1195–1198
 static, 1195, 1198–1202
 static template, 1203–1206
 STL queue and **dequeue** containers, 1214–1217
- QuickSort algorithm, 1246–1250
- R**
- RAM, 5
- random-access files, 271, 697–705
- random-access memory (RAM), 5
- random file access, 697
- random numbers, 128–131
 limiting range of, 130
 seeding, 129
 time function with, 129
- range-based for loop, 398–402
 map class, 1060
 modifying array with, 400–402
 set class, 1081
 versus regular for loop, 402
- range constructor
 list class, 1155
 map class, 1055
 set class, 1079
 vector class, 1041
- range variable, 398
- rbegin()** member function, **array** class, 1033, 1038
- reading data, from file, 278, 280–281
- read** member function, file stream objects, 689–691
- read position, 279–280
- records, 693–697

- recursion, 1223–1253
 base case, 1228
 binary search, 1241–1243
 counting characters, 1231–1234
 depth of, 1224
 direct, 1234
 exhaustive algorithms, 1250–1253
 factorial algorithm, 1228–1231
 Fibonacci numbers, 1236–1237
 greatest common divisor (GCD), 1235–1236
 indirect, 1234
 infinite, 1223
 iteration *vs.*, 1253
 linked list operations, 1237–1241
 problem solving with, 1227–1234
 QuickSort algorithm, 1246–1250
 recursive functions, 1223–1227
 recursively defined problems, 1236–1237
 Towers of Hanoi, 1243–1246
 recursive case, 1228
 redefining base class functions, 936–940, 947
 reference parameters, constant, 648
 reference variables
 as parameters, 354–359
 pointers *vs.*, 521–522
reinterpret_cast, 691, 693
 relational expressions, 152, 235–236
 relational operators, 151–155
 and characters, 197–198
 and pointers, 519
 and **string** class, 198–200
 truth, 154–156
 value of relationship, 152–154
 relationships
 “has a,” 868
 “is a,” 908–915, 947
 whole-part, 868
rend() member function, **array** class, 1033, 1038
replace member function, **string** class, 603
reserve() member function, **vector** class, 1052–1053
 reserved words, 15
resize member function,
 string class, 598
 vector, 449
 responsibilities, identifying, 800–803
 rethrowing an exception, 1006
 returning
 bool value from functions, 338–340
 pointers from functions, 534–539
 structures from functions, 634–636
 values from functions, 330–338
 return statement, 328–329
 reusability, object, 722
 reverse iterator, 1038–1039
reverse member function
 vector, 448
 rewinding a file, 704–705
right manipulators, 118–119
- Ruby**, 11
 running, of programs, 5
 running total (for loops), 261–263
 run-time library, 12
rvalues, 60, 886–888
 references, 888–889
- S**
- scope, 62–63
scope resolution operator (:), 730, 766
search algorithm
 binary search, 466–469, 1241–1243
 Demetris Leadership Center case study, 469–476
 linear search, 463–466
 for **STL vector**, 495–497
search trees, binary, 1259
secondary storage, 6
seekg member function, file stream objects, 697–702
seekp member function, file stream objects, 697–702
selection sort algorithm, 482–486
self-referential data structure, 1125
 semicolons, 160
 sentinels, 264–265
sequence containers, 1030
sequence structure, 156
sequential file access, 271, 697
 seekg member function, 704–705
sequential search, 463
set class, 1079
 count() member function, 1082
 definition statements, 1079
 emplace() member function, 1081
 find() member function, 1082–1083
 insert() member function, 1081
 iterators and, 1082
 member functions, 1080
 range-based **for loop**, 1081
 store objects of class in, 1083–1085
 set_difference() function, 1095, 1099–1100
 set_intersection() function, 1095, 1098–1099
 set_symmetric_difference() function, 1095, 1101–1102
 set_union() function, 1095, 1096–1098
 setprecision manipulator, 113–115, 671
 setter functions, 730
 setw manipulator, 672–673
 shared_ptr, 541
 short, 44, 45
 short-circuit evaluation, 185, 187
 showItem function, 633
 showpoint manipulator, 117–118
 shrink_to_fit() member function, **vector** class, 1052–1053
 single-line comments, 71–72
 single precision, 55
 singly linked list, 1153

size declarators, of arrays, 382
size member function
 string class, 603
 vector, 443–444
size() member function, array class, 1033
sizeof operator, 59
 smart pointers, 541–544
software
 application, 7–8
 system, 7
software developers, 1
software development tools, 7
software engineering, 22, 530–536,
 1014, 1253
sort algorithm (STL), 1026
sort() function, 1087–1090
 sorting, of strings, 576–578, 592–594
sorting algorithm
 bubble sort, 476–479
 Demetris Leadership Center case study, 486–494
 QuickSort, 1246–1249
 selection sort, 482–486
 for STL vector, 495–497
source code, 11
source files, 11
 specialization, inheritance and, 907–908
specialized templates, 1023
specification file, class, 745
spreadsheets, 269, 665
stack (adapter class), 1031
stack (STL type), 1193–1194
stacks, 1165–1217
 applications of, 1166
 array-based, 1166
 cafeteria plates, 1165–1166
 described, 1165–1166
 dynamic, 1166, 1182–1192
 implementing, 1173–1175
isEmpty function, 1185
isEmpty operation, 1167
isFull operation, 1167
 LIFO, 1165
 linked list-based, 1182–1187
 for math, 1173–1175
 operations, 1166–1167, 1173–1175
 pop, 1166–1167
 push, 1166
 static, 1166–1167, 1176–1182
 STL stack container, 1193–1194
 unwinding, 1005
stale data, 738
Standard Template Library (STL), 435, 1029
 algorithms to perform set operations, 1094–1103
array class, 1031–1033
 associative containers, 1029
forward_list class, 1158
 functional classes in, 1113–1114
 header files, 1030
includes() function, 1095
 iterators (*see* iterators)
list class, 1154–1158
queue, 1216–1217
sequence containers, 1029
set_difference() function, 1095, 1099–1100
set_intersection() function, 1095, 1098–1099
set_symmetric_difference() function, 1095,
 1101–1102
set_union() function, 1095, 1096–1098
stack, 1193–1194
 storing objects, 1193–1194
 vector, 435–449, 495–497
state, object, 735
statements, 16
 static binding, 949
static key word, 818
 static local variables, 348–351
 static member functions, 822–825
 static member variables, 818–822
 static queues, 1195, 1198–1202
 template, 1203–1206
 static stacks, 1166–1173, 1176–1182
STL, *see* Standard Template Library
storage, secondary, 6
strcat library function, 569–570, 577
strcmp library function, 574–576
strcpy library function, 570, 578
stream extraction operator, 86
stream insertion operator, 32, 274, 596
stream manipulator, 33, 119
stream object, 31
string class, 591–599, 725
 append member function, 598
 assign member function, 598
 begin member function, 599
 capacity member function, 599
 clear member function, 599
 compare member function, 599
 comparing and sorting, 592–594
 constructors, 771
 copy member function, 599
 defining string objects, 591–592, 595–596
 described, 53–54
 empty member function, 599
 end member function, 599
 erase member function, 599
 find member function, 599
 Home Software Company case study, 603–604
 input, reading into a string object, 592
 insert member function, 603
 length member function, 597, 599
 at member function, 598
 member functions, 597–599
 operators, 596
 and relational operators, 198–200
 replace member function, 599
 resize member function, 599
 size member function, 599
 substr member function, 599

swap member function, 599
using, 53–54
string constant, 29
stringCopy function, 586
string header file, 53, 591
string literals, 29, 41, 50–53
string literals, 565–566
string objects
 and characters, 120–126
 comparing, 198–200
 defining, 591–592, 595–596
 functions for handling, 585–590
 member functions and operators, 126
 numeric conversion functions, 579–585
 reading input into, 592
 sorting, 576–579
string tokenizing, 600–601
strlen library function, 568–569, 577
strncat library function, 578
strcpy library function, 578
strongly typed enum, 651–653
strstr library function, 572–574, 578
struct, 615, *see also structures*
structure pointer operator (->), 637–640
structures, 613–653
 and abstract data types, 613–615
 accessing structure members, 618–621
 arrays of, 625–627
 combining data into, 615–618
 containing pointers, 697
 and enumerated data types, 642–653
 as function arguments, 631–634
 initializing, 622–624
 nested, 627–630
 pointers as members of, 640–641
 pointers to, 637–640
 records, creating with, 693–697
 returning, from a function, 634–636
 self-referential, 1125
structure variables, 617–618, 621
stubs, 367–369
subscript, of array element, 383
substr member function, **string class**, 599
swap member function, 449
string class, 603
sweep (second) member function, **array class**, 1033
switch statement, 204–213
break, 206, 207
case, 205–206
default, 205–206
 fallthrough capability, 207–209
 with menus, 209–211
syntax, 14
syntax errors, 12
system software
 operating system, 7
 software development tools, 7
 utility program, 7

T

tags, of structures, 616, 618
tellg member function, **file stream objects**, 702–704
tellp member function, **file stream objects**, 702–704
template function, 1008
template prefix, 1008, 1013, 1015
templates
 binary trees, 1277–1283
 class, 1014–1023
 defining, 1013, 1014
 dynamic queue, 1211–1214
 dynamic stack, 1187–1192
 function, 1008–1013
 linked list, 1141–1153
 prefixes, 1008, 1013
 specialized, 1023
 static queue, 1198–1202
 static queues, 1203–1206
 static stack, 1176–1181
 type parameter, 1008
ternary operators, 63, 201
text editor, 11
text files
 described, 270
 numeric data from, 280–281
this pointer, 837, 839–840
throwing an exception, 531, 990
throw key word, 990, 1005
throw point, 990
time library function, 129
tolower library function, 561
top-down design, 20
to_string function, 581
toupper library function, 561
Towers of Hanoi, 1243–1246
trailing else, 181
traversing
 binary tree, 1264–1267
 linked list, 1131–1133
true values, 152, 154–155, 162–163
truth, 154–156, 162–163
try block, 990
try/catch construct, 990–992
try key word, 990
two-dimensional arrays, 424–431
 initializing, 427–428
 passing, to functions, 428–429
 summing columns, 431
 summing elements of, 430
 summing rows, 430–431
type cast expression, 103
type casting, 103–106
type coercion, 100
type conversion, 100–101
type parameters, 1008–1010, 1012

U

UML, *see* Unified Modeling Language
 unary predicate, 1111
 unary operator, 63
 underflow, 102–103
 Unified Modeling Language (UML), 792–794, 870–871
 access specification, showing, 793
 aggregation, showing, 870–871
 class diagram, 792–794
 constructors, showing, 794
 destructors, showing, 794
 parameter notation, 793–794
 unique_ptr, 541–544
 United Cause, case study, 544–549
 unordered_multimap class, 1078
 unordered_multiset class, 1085
 unordered_set class, 1085
 unsigned int, 44, 45
 unsigned long, 44, 45
 unsigned short, 44, 45
 unwinding the stack, 1005
 update expression (for loop), 252, 256, 257–258
 multiple statements in, 257–258
 uppercase conversion, character, 561–563
 USB drives, 6
 user-controlled loops, 248, 256–257
 user-specified filenames, 286–287
 utility programs, 7

V

value, passing by, 647
 value-returning functions, 330–338
 calling, 332–338
 defining, 330–331
 variable declaration, 17
 variable definitions, 17, 38
 variables, 16–17
 control, 239
 flag, 183–184
 global, 342–344
 instance, 817–824
 integer, 44, 184
 local, 340–342, 347–348, 348–351
 loop control, 239
 names, 43
 overflow and underflow, 102–103
 with same name, 215–216, 347–348
 static member, 818–824
 structure, 617–618, 621
 capacity member function, 448
 clear member function, 445, 448
 compared to linked list, 1123
 defining, 436–437
 empty member function, 446–447, 448
 initialization list, with C++, 11, 437
 at member function, 448

pop_back member function, 448
 push_back member function, 441–442, 448
 range-based for loop, 439–441
 removing elements from, 444–445
 resize member function, 449
 reverse member function, 448
 searching and sorting, 495–497
 size member function, 443–444
 STL, 435–436
 storing and retrieving values in, 437–439
 swap member function, 449
 vector, 1040
 basic operations, 1043–1044
 capacity() member function, 1052–1053
 defining, 1043–1044
 definition statements, 1040–1041
 emplace() member function, 1050–1052
 emplace_back() member function, 1050–1052
 insert member function, 1045–1047
 iterators usage, 1045
 max_size() member function, 1052–1053
 member functions, 1041–1043
 reserve() member function, 1052–1053
 shrink_to_fit() member function, 1052–1053
 storing values in, 1048–1050
 virtual destructors, 947–963, 958–963
 virtual functions
 and polymorphism, 953–955
 pure, 963–967
 virtual key word, 949
 virtual member functions, 947–963
 Visual Basic, 11
 void function, 307
 volatile memory, RAM as, 5

W

weak_ptr, 541
 Web browsers, 269
 while loops, 236–242, 265
 input validation with, 243–245
 logic of, 237–239
 as pretest loop, 239
 programming style, 241–242
 whitespace characters, 120
 whole-part relationship, 868
 word processors, 269, 665
 write member function, file stream objects, 689

Z

zero(es)
 division by, 169, 989–990
 trailing, 117, 118

Credits

Cover Image © Roman Samokhin/123RF

Figure 1-2a pg. 3 © iko/Shutterstock

Figure 1-2b pg. 3 © Nikita Rogul/Shutterstock

Figure 1-2c pg. 3 © Feng Yu/Shutterstock

Figure 1-2d pg. 3 © Chiyacat/Shutterstock

Figure 1-2e pg. 3 © Eikostas/Shutterstock

Figure 1-2f pg. 3 © tkemot/Shutterstock

Figure 1-2g pg. 3 © Vitaly Korovin/Shutterstock

Figure 1-2h pg. 3 © Lusoimages/Shutterstock

Figure 1-2i pg. 3 © jocic/Shutterstock

Figure 1-2j pg. 3 © Best Pictures here/Shutterstock

Figure 1-2k pg. 3 © Peter Guess/Shutterstock

Figure 1-2l pg. 3 © Aquila/Shutterstock

Figure 1-2m pg. 3 © Andre Nitsievsky/Shutterstock

Figure 1-3 pg. 4 © U.S. Army Center of Military History

Figure 1-4 pg. 4 © Creativa/Shutterstock

Chapter 1 Microsoft screenshots - SEE MICROSOFT AGREEMENT FOR FULL CREDIT LINE.

Chapter 2 Microsoft screenshots - SEE MICROSOFT AGREEMENT FOR FULL CREDIT LINE.

Chapter 5 Microsoft screenshots - SEE MICROSOFT AGREEMENT FOR FULL CREDIT LINE.

Chapter 17 Microsoft screenshots - SEE MICROSOFT AGREEMENT FOR FULL CREDIT LINE.

MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE INFORMATION CONTAINED IN THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED AS PART OF THE SERVICES FOR ANY PURPOSE. ALL SUCH DOCUMENTS AND RELATED GRAPHICS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS HEREBY DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF INFORMATION AVAILABLE FROM THE SERVICES. THE DOCUMENTS AND RELATED GRAPHICS CONTAINED HEREIN COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED HEREIN AT ANY TIME. PARTIAL SCREEN SHOTS MAY BE VIEWED IN FULL WITHIN THE SOFTWARE VERSION SPECIFIED.