CHAPTER 14

More About Classes

TOPICS

14.1 Instance and Static Members

CONCEPT: Each instance of a class has its own copies of the class's instance variables. If a member variable is declared `static`, however, all instances of that class have access to that variable. If a member function is declared `static`, it may be called without any instances of the class being defined.

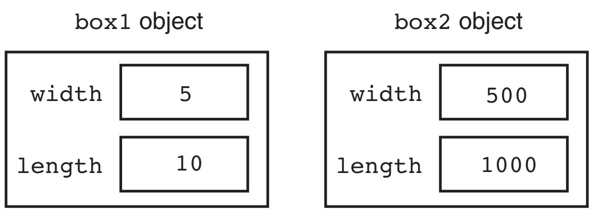
Instance Variables

Each class object (an instance of a class) has its own copy of the class's member variables. An object's member variables are separate and distinct from the member variables of other objects of the same class. For example, recall that the `Rectangle` class discussed in Chapter 13 has two member variables: `width` and `length`. Suppose that we define two objects of the `Rectangle` class and set their `width` and `length` member variables as shown in the following code.

```
Rectangle box1, box2;

// Set the width and length for box1.
box1.setWidth(5);
box1.setLength(10);

// Set the width and length for box2.
box2.setWidth(500);
box2.setLength(1000);
```

This code creates `box1` and `box2`, which are two distinct objects. Each has its own `width` and `length` member variables, as illustrated in Figure 14-1.

Figure 14-1



When the `getWidth` member function is called, it returns the value stored in the calling object's `width` member variable. For example, the following statement displays `5 500`.

```
cout << box1.getWidth() << " " << box2.getWidth() << endl;
```

In object-oriented programming, member variables such as the `Rectangle` class's `width` and `length` members are known as *instance variables*. They are called instance variables because each instance of the class has its own copies of the variables.

Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as a *static member variables* and *static member functions*. When a value is stored in a static member variable, it is not stored in an instance of the class. In fact, an instance of the class doesn't even have to exist in order for values to be stored in the class's static member variables. Likewise, static member functions do not operate on instance variables. Instead, they can operate only on static member variables. You can think of static member variables and static member functions as belonging to the class instead of to an instance of the class. In this section, we will take a closer look at static members. First we will examine static member variables.

Static Member Variables

When a member variable is declared with the key word `static`, there will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static member variable is shared by all instances of the class. For example, the following `Tree` class uses a static member variable to keep count of the number of instances of the class that are created.

Contents of `Tree.h`

```
// Tree class
class Tree
{
private:
     static int objectCount;     // Static member variable.
public:
     // Constructor
     Tree()
         { objectCount++; }

11       // Accessor function for objectCount
12       int getObjectCount() const
13          { return objectCount; }
14    };
15
16    // Definition of the static member variable, written
17    // outside the class.
18    int Tree::objectCount = 0;
```

First, notice in line 5 the declaration of the static member variable named `objectCount`: A static member variable is created by placing the key word `static` before the variable's data type. Also notice that in line 18 we have written a definition statement for the `objectCount` variable and that the statement is outside the class declaration. This external definition statement causes the variable to be created in memory and is required. In line 18 we have explicitly initialized the `objectCount` variable with the value 0. We could have left out the initialization because C++ automatically stores 0 in all uninitialized static member variables. It is a good practice to initialize the variable anyway, so it is clear to anyone reading the code that the variable starts out with the value 0.

Next, look at the constructor in lines 8 and 9. In line 9 the `++` operator is used to increment `objectCount`. Each time an instance of the `Tree` class is created, the constructor will be called, and the `objectCount` member variable will be incremented. As a result, the `objectCount` member variable will contain the number of instances of the `Tree` class that have been created. The `getObjectCount` function, in lines 12 and 13, returns the value in `objectCount`. Program 14-1 demonstrates this class.

Program 14-1

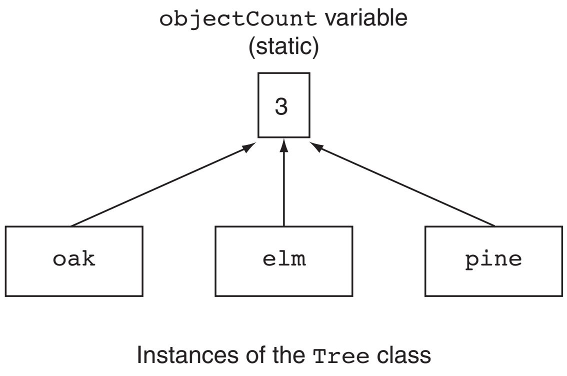
```
// This program demonstrates a static member variable.
#include <iostream>
#include "Tree.h"
using namespace std;

int main()
{
    // Define three Tree objects.
    Tree oak;
    Tree elm;
    Tree pine;

    // Display the number of Tree objects we have.
    cout << "We have " << pine.getObjectCount()
         << " trees in our program!\n";
    return 0;
}
```

Program Output

```
We have 3 trees in our program!
```

The program creates three instances of the `Tree` class, stored in the variables `oak`, `elm`, and `pine`. Although there are three instances of the class, there is only one copy of the static `objectCount` variable. This is illustrated in Figure 14-2.

Figure 14-2



In line 14 the program calls the `getObjectCount` member function to retrieve the number of instances that have been created. Although the program uses the `pine` object to call the member function, the same value would be returned if any of the objects had been used. For example, all three of the following `cout` statements would display the same thing.

```
cout << "We have " << oak.getObjectCount() << " trees\n";
cout << "We have " << elm.getObjectCount() << " trees\n";
cout << "We have " << pine.getObjectCount() << " trees\n";
```

A more practical use of a static member variable is demonstrated in Program 14-2. The `Budget` class is used to gather the budget requests for all the divisions of a company. The class uses a static member, `corpBudget`, to hold the amount of the overall corporate budget. When the member function `addBudget` is called, its argument is added to the current contents of `corpBudget`. By the time the program is finished, `corpBudget` will contain the total of all the values placed there by all the `Budget` class objects. (These files are stored in the Student Source Code Folder `Chapter 14\Budget Version 1`.)

Contents of `Budget.h` (Version 1)

```
#ifndef BUDGET_H
#define BUDGET_H

// Budget class declaration
class Budget
{
private:
     static double corpBudget;   // Static member
     double divisionBudget;      // Instance member
public:
     Budget()
        { divisionBudget = 0; }

     void addBudget(double b)
        { divisionBudget += b;
          corpBudget += b; }

     double getDivisionBudget() const
        { return divisionBudget; }

```

```
    double getCorpBudget() const
      { return corpBudget; }
};

// Definition of static member variable corpBudget
double Budget::corpBudget = 0;

#endif
```

Program 14-2

```
// This program demonstrates a static class member variable.
#include <iostream>
#include <iomanip>
#include "Budget.h"
using namespace std;

int main()
{
    int count;                          // Loop counter
    const int NUM_DIVISIONS = 4;        // Number of divisions
    Budget divisions[NUM_DIVISIONS];    // Array of Budget objects

    // Get the budget requests for each division.
    for (count = 0; count < NUM_DIVISIONS; count++)
    {
        double budgetAmount;
        cout << "Enter the budget request for division ";
        cout << (count + 1) << ": ";
        cin >> budgetAmount;
        divisions[count].addBudget(budgetAmount);
    }

    // Display the budget requests and the corporate budget.
    cout << fixed << showpoint << setprecision(2);
    cout << "\nHere are the division budget requests:\n";
    for (count = 0; count < NUM_DIVISIONS; count++)
    {
        cout << "\tDivision " << (count + 1) << "\t$ ";
        cout << divisions[count].getDivisionBudget() << endl;
    }
    cout << "\tTotal Budget Requests:\t$ ";
    cout << divisions[0].getCorpBudget() << endl;

    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the budget request for division 1: 100000 [Enter]
Enter the budget request for division 2: 200000 [Enter]
Enter the budget request for division 3: 300000 [Enter]
Enter the budget request for division 4: 400000 [Enter]
```

(program output continues)

Program 14-2 *(continued)*

```
Here are the division budget requests:
     Division 1      $ 100000.00
     Division 2      $ 200000.00
     Division 3      $ 300000.00
     Division 4      $ 400000.00
     Total Budget Requests:   $ 1000000.00
```

Static Member Functions

You declare a static member function by placing the `static` keyword in the function's prototype. Here is the general form:

```
static ReturnType FunctionName (ParameterTypeList);
```

A function that is a static member of a class cannot access any nonstatic member data in its class. With this limitation in mind, you might wonder what purpose static member functions serve. The following two points are important for understanding their usefulness:

- Even though static member variables are declared in a class, they are actually defined outside the class declaration. The lifetime of a class's static member variable is the lifetime of the program. This means that a class's static member variables come into existence before any instances of the class are created.
- A class's static member functions can be called before any instances of the class are created. This means that a class's static member functions can access the class's static member variables *before* any instances of the class are defined in memory. This gives you the ability to create very specialized setup routines for class objects.

Program 14-3, a modification of Program 14-2, demonstrates this feature. It asks the user to enter the main office's budget request before any division requests are entered. The `Budget` class has been modified to include a static member function named `mainOffice`. This function adds its argument to the static `corpBudget` variable and is called before any instances of the `Budget` class are defined. (These files are stored in the Student Source Code Folder `Chapter 14\Budget Version 2`.)

Contents of `Budget.h` (Version 2)

```
#ifndef BUDGET_H
#define BUDGET_H

// Budget class declaration
class Budget
{
private:
     static double corpBudget;   // Static member variable
     double divisionBudget;      // Instance member variable
public:
     Budget()
          { divisionBudget = 0; }

```

```
    void addBudget(double b)
        { divisionBudget += b;
          corpBudget += b; }

    double getDivisionBudget() const
        { return divisionBudget; }

    double getCorpBudget() const
        { return corpBudget; }

    static void mainOffice(double);  // Static member function
};

#endif
```

Contents of `Budget.cpp`

```
#include "Budget.h"

// Definition of corpBudget static member variable
double Budget::corpBudget = 0;

//**********************************************************
// Definition of static member function mainOffice.        *
// This function adds the main office's budget request to *
// the corpBudget variable.                                *
//**********************************************************

void Budget::mainOffice(double moffice)
{
    corpBudget += moffice;
}
```

Program 14-3

```
// This program demonstrates a static member function.
#include <iostream>
#include <iomanip>
#include "Budget.h"
using namespace std;

int main()
{
    int count;                      // Loop counter
    double mainOfficeRequest;       // Main office budget request
    const int NUM_DIVISIONS = 4;    // Number of divisions

    // Get the main office's budget request.
    // Note that no instances of the Budget class have been defined.
    cout << "Enter the main office's budget request: ";
    cin >> mainOfficeRequest;
    Budget::mainOffice(mainOfficeRequest);

```

(program continues)

Program 14-3 *(continued)*

```
19      Budget divisions[NUM_DIVISIONS]; // An array of Budget objects.
20
21      // Get the budget requests for each division.
22      for (count = 0; count < NUM_DIVISIONS; count++)
23      {
24          double budgetAmount;
25          cout << "Enter the budget request for division ";
26          cout << (count + 1) << ": ";
27          cin >> budgetAmount;
28          divisions[count].addBudget(budgetAmount);
29      }
30
31      // Display the budget requests and the corporate budget.
32      cout << fixed << showpoint << setprecision(2);
33      cout << "\nHere are the division budget requests:\n";
34      for (count = 0; count < NUM_DIVISIONS; count++)
35      {
36          cout << "\tDivision " << (count + 1) << "\t$ ";
37          cout << divisions[count].getDivisionBudget() << endl;
38      }
39      cout << "\tTotal Budget Requests:\t$ ";
40      cout << divisions[0].getCorpBudget() << endl;
41
42      return 0;
43  }
```

Program Output with Example Input Shown in Bold

```
Enter the main office's budget request: 100000 [Enter]
Enter the budget request for division 1: 200000 [Enter]
Enter the budget request for division 2: 200000 [Enter]
Enter the budget request for division 3: 300000 [Enter]
Enter the budget request for division 4: 400000 [Enter]

Here are the division budget requests:
    Division 1     $ 100000.00
    Division 2     $ 200000.00
    Division 3     $ 300000.00
    Division 4     $ 400000.00
    Total Requests (including main office): $ 1100000.00
```

Notice in line 17 the statement that calls the static function `mainOffice`:

```
Budget::mainOffice(amount);
```

Calls to static member functions do not use the regular notation of connecting the function name to an object name with the dot operator. Instead, static member functions are called by connecting the function name to the class name with the scope resolution operator.

NOTE: If an instance of a class with a static member function exists, the static member function can be called with the class object name and the dot operator, just like any other member function.

14.2 Friends of Classes

CONCEPT: A friend is a function or class that is not a member of a class, but has access to the private members of the class.

Private members are hidden from all parts of the program outside the class, and accessing them requires a call to a public member function. Sometimes you will want to create an exception to that rule. A *friend* function is a function that is not part of a class, but that has access to the class's private members. In other words, a friend function is treated as if it were a member of the class. A friend function can be a regular stand-alone function, or it can be a member of another class. (In fact, an entire class can be declared a friend of another class.)

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. Classes keep a "list" of their friends, and only the external functions or classes whose names appear in the list are granted access. A function is declared a friend by placing the key word `friend` in front of a prototype of the function. Here is the general format:

```
friend ReturnType FunctionName (ParameterTypeList)
```

In the following declaration of the `Budget` class, the `addBudget` function of another class, `AuxiliaryOffice` has been declared a friend. (This file is stored in the Student Source Code Folder `Chapter 14\Budget Version 3`.)

Contents of `Budget.h` (Version 3)

```
#ifndef BUDGET_H
#define BUDGET_H
#include "Auxil.h"

// Budget class declaration
class Budget
{
private:
     static double corpBudget;   // Static member variable
     double divisionBudget;      // Instance member variable
public:
     Budget()
          { divisionBudget = 0; }

     void addBudget(double b)
          { divisionBudget += b;
            corpBudget += b; }

     double getDivisionBudget() const
          { return divisionBudget; }

     double getCorpBudget() const
          { return corpBudget; }

```

```
    // Static member function
    static void mainOffice(double);

    // Friend function
    friend void AuxiliaryOffice::addBudget(double, Budget &);
};

#endif
```

Let's assume another class, AuxiliaryOffice, represents a division's auxiliary office, perhaps in another country. The auxiliary office makes a separate budget request, which must be added to the overall corporate budget. The friend declaration of the AuxiliaryOffice::addBudget function tells the compiler that the function is to be granted access to Budget's private members. Notice the function takes two arguments: a double and a reference object of the Budget class. The Budget class object that is to be modified by the function is passed to it, by reference, as an argument. The following code shows the declaration of the AuxillaryOffice class. (This file is stored in the Student Source Code Folder Chapter 14\Budget Version 3.)

Contents of Auxil.h

```
#ifndef AUXIL_H
#define AUXIL_H

class Budget; // Forward declaration of Budget class

// Aux class declaration

class AuxiliaryOffice
{
private:
    double auxBudget;
public:
    AuxiliaryOffice()
        { auxBudget = 0; }

    double getDivisionBudget() const
        { return auxBudget; }

    void addBudget(double, Budget &);
};

#endif
```

Contents of Auxil.cpp

```
#include "Auxil.h"
#include "Budget.h"

//*************************************************************
// Definition of member function mainOffice.                  *
// This function is declared a friend by the Budget class.   *
// It adds the value of argument b to the static corpBudget *
// member variable of the Budget class.                       *
//*************************************************************
```

```
10
11  void AuxiliaryOffice::addBudget(double b, Budget &div)
12  {
13      auxBudget += b;
14      div.corpBudget += b;
15  }
```

Notice the `Auxil.h` file contains the following statement in line 4:

```
class Budget; // Forward declaration of Budget class
```

This is a *forward declaration* of the `Budget` class. It simply tells the compiler that a class named `Budget` will be declared later in the program. This is necessary because the compiler will process the `Auxil.h` file before it processes the `Budget` class declaration. When it is processing the `Auxil.h` file it will see the following function declaration in line 19:

```
void addBudget(double, Budget &);
```

The `addBudget` function's second parameter is a `Budget` reference variable. At this point, the compiler has not processed the `Budget` class declaration, so, without the forward declaration, it wouldn't know what a `Budget` reference variable is.

The following code shows the definition of the `addBudget` function. (This file is also stored in the Student Source Code Folder `Chapter 14\Budget Version 3`.)

Contents of `Auxil.cpp`

```
 1  #include "Auxil.h"
 2  #include "Budget.h"
 3
 4  //*************************************************************
 5  // Definition of member function mainOffice.                  *
 6  // This function is declared a friend by the Budget class.   *
 7  // It adds the value of argument b to the static corpBudget *
 8  // member variable of the Budget class.                       *
 9  //*************************************************************
10
11  void AuxiliaryOffice::addBudget(double b, Budget &div)
12  {
13      auxBudget += b;
14      div.corpBudget += b;
15  }
```

The parameter `div`, a reference to a `Budget` class object, is used in line 14. This statement adds the parameter `b` to `div.corpBudget`. Program 14-4 demonstrates the classes.

Program 14-4

```
// This program demonstrates a static member function.
#include <iostream>
#include <iomanip>
#include "Budget.h"
using namespace std;

```

(program continues)

Program 14-4 *(continued)*

```
int main()
{
    int count;                        // Loop counter
    double mainOfficeRequest;         // Main office budget request
    const int NUM_DIVISIONS = 4;      // Number of divisions

    // Get the main office's budget request.
    cout << "Enter the main office's budget request: ";
    cin >> mainOfficeRequest;
    Budget::mainOffice(mainOfficeRequest);

    Budget divisions[NUM_DIVISIONS];  // Array of Budget objects
    AuxiliaryOffice auxOffices[4];    // Array of AuxiliaryOffice

    // Get the budget requests for each division
    // and their auxiliary offices.
    for (count = 0; count < NUM_DIVISIONS; count++)
    {
        double budgetAmount; // To hold input

        // Get the request for the division office.
        cout << "Enter the budget request for division ";
        cout << (count + 1) << ": ";
        cin >> budgetAmount;
        divisions[count].addBudget(budgetAmount);

        // Get the request for the auxiliary office.
        cout << "Enter the budget request for that division's\n";
        cout << "auxiliary office: ";
        cin >> budgetAmount;
        auxOffices[count].addBudget(budgetAmount, divisions[count]);
    }

    // Display the budget requests and the corporate budget.
    cout << fixed << showpoint << setprecision(2);
    cout << "\nHere are the division budget requests:\n";
    for (count = 0; count < NUM_DIVISIONS; count++)
    {
        cout << "\tDivision " << (count + 1) << "\t\t$";
        cout << divisions[count].getDivisionBudget() << endl;
        cout << "\tAuxiliary office:\t$";
        cout << auxOffices[count].getDivisionBudget() << endl << endl;
    }
    cout << "Total Budget Requests:\t$ ";
    cout << divisions[0].getCorpBudget() << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the main office's budget request: **100000 [Enter]**
Enter the budget request for division 1: **100000 [Enter]**
Enter the budget request for that division's
auxiliary office: **50000 [Enter]**
Enter the budget request for division 2: **200000 [Enter]**
Enter the budget request for that division's
auxiliary office: **40000 [Enter]**
Enter the budget request for division 3: **300000 [Enter]**
Enter the budget request for that division's
auxiliary office: **70000 [Enter]**
Enter the budget request for division 4: **400000 [Enter]**
Enter the budget request for that division's
auxiliary office: **65000 [Enter]**

Here are the division budget requests:
    Division 1                $100000.00
    Auxiliary office:         $50000.00

    Division 2                $200000.00
    Auxiliary office:         $40000.00

    Division 3                $300000.00
    Auxiliary office:         $70000.00

    Division 4                $400000.00
    Auxiliary office:         $65000.00

Total Budget Requests:  $ 1325000.00
```

As mentioned before, it is possible to make an entire class a friend of another class. The `Budget` class could make the `AuxiliaryOffice` class its friend with the following declaration:

```
friend class AuxiliaryOffice;
```

This may not be a good idea, however. Every member function of `AuxiliaryOffice` (including ones that may be added later) would have access to the private members of `Budget`. The best practice is to declare as friends only those functions that must have access to the private members of the class.

Checkpoint

14.1 What is the difference between an instance member variable and a static member variable?

14.2 Static member variables are declared inside the class declaration. Where are static member variables defined?

14.3 Does a static member variable come into existence in memory before, at the same time as, or after any instances of its class?

14.4 What limitation does a static member function have?

14.5 What action is possible with a static member function that isn't possible with an instance member function?

14.6 If class `X` declares function `f` as a friend, does function `f` become a member of class `X`?

14.7 Class `Y` is a friend of class `X`, which means the member functions of class `Y` have access to the private members of class `X`. Does the friend key word appear in class `Y`'s declaration or in class `X`'s declaration?

14.3 Memberwise Assignment

CONCEPT: The = operator may be used to assign one object's data to another object, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.

Like other variables (except arrays), objects may be assigned to one another using the = operator. As an example, consider Program 14-5, which uses the `Rectangle` class (version 4) that we discussed in Chapter 13. Recall that the `Rectangle` class has two member variables: `width` and `length`. The constructor accepts two arguments, one for `width` and one for `length`.

Program 14-5

```
// This program demonstrates memberwise assignment.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
    // Define two Rectangle objects.
    Rectangle box1(10.0, 10.0);   // width = 10.0, length = 10.0
    Rectangle box2 (20.0, 20.0);  // width = 20.0, length = 20.0

    // Display each object's width and length.
    cout << "box1's width and length: " << box1.getWidth()
         << " " << box1.getLength() << endl;
    cout << "box2's width and length: " << box2.getWidth()
         << " " << box2.getLength() << endl << endl;

    // Assign the members of box1 to box2.
    box2 = box1;

    // Display each object's width and length again.
    cout << "box1's width and length: " << box1.getWidth()
         << " " << box1.getLength() << endl;
    cout << "box2's width and length: " << box2.getWidth()
         << " " << box2.getLength() << endl;

    return 0;
}
```

Program Output

```
box1's width and length: 10 10
box2's width and length: 20 20

box1's width and length: 10 10
box2's width and length: 10 10
```

The following statement, which appears in line 19, copies the `width` and `length` member variables of `box1` directly into the `width` and `length` member variables of `box2`:

```
box2 = box1;
```

Memberwise assignment also occurs when one object is initialized with another object's values. Remember the difference between assignment and initialization: assignment occurs between two objects that already exist, and initialization happens to an object being created. Consider the following code:

```
// Define box1.
Rectangle box1(100.0, 50.0);

// Define box2, initialize with box1's values
Rectangle box2 = box1;
```

The last statement defines a `Rectangle` object, `box2`, and initializes it to the values stored in `box1`. Because memberwise assignment takes place, the `box2` object will contain the exact same values as the `box1` object.

14.4 Copy Constructors

CONCEPT: A copy constructor is a special constructor that is called whenever a new object is created and initialized with another object's data.

Most of the time, the default memberwise assignment behavior in C++ is perfectly acceptable. There are instances, however, where memberwise assignment cannot be used. For example, consider the following class. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 1`.)

Contents of `StudentTestScores.h` (Version 1)

```
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
    string studentName; // The student's name
    double *testScores; // Points to array of test scores
    int numTestScores;  // Number of test scores

    // Private member function to create an
    // array of test scores.
    void createTestScoresArray(int size)
    { numTestScores = size;
      testScores = new double[size];
      for (int i = 0; i < size; i++)
          testScores[i] = DEFAULT_SCORE;}

23  public:
24      // Constructor
25      StudentTestScores(string name, int numScores)
26      { studentName = name;
27        createTestScoresArray(numScores); }
28
29      // Destructor
30      ~StudentTestScores()
31      { delete [] testScores; }
32
33      // The setTestScore function sets a specific
34      // test score's value.
35      void setTestScore(double score, int index)
36      { testScores[index] = score; }
37
38      // Set the student's name.
39      void setStudentName(string name)
40      { studentName = name; }
41
42      // Get the student's name.
43      string getStudentName() const
44      { return studentName; }
45
46      // Get the number of test scores.
47      int getNumTestScores() const
48      { return numTestScores; }
49
50      // Get a specific test score.
51      double getTestScore(int index) const
52      { return testScores[index]; }
53  };
54  #endif
```

This class stores a student's name and a set of test scores. Let's take a closer look at the code:

- Lines 11 through 13 declare the class's attributes. The `studentName` attribute is a `string` object that holds a student's name. The `testScores` attribute is an `int` pointer. Its purpose is to point to a dynamically allocated `int` array that holds the student's test score. The `numTestScore` attribute is an `int` that holds the number of test scores.
- The `createTestScoresArray` private member function, in lines 17 through 21, creates an array to hold the student's test scores. It accepts an argument for the number of test scores, assigns this value to the `numTestScores` attribute (line 18), and then dynamically allocates an `int` array for the `testScores` attribute (line 19). The `for` loop in lines 20 through 21 initializes each element of the array to the default value 0.0.
- The constructor, in lines 25 through 27, accepts the student's name and the number of test scores as arguments. In line 26 the name is assigned to the `studentName` attribute, and in line 27 the number of test scores is passed to the `createTestScoresArray` member function.
- The destructor, in lines 30 through 31, deallocates the test score array.
- The `setTestScore` member function, in lines 35 through 36, sets a specific score in the `testScores` attribute. The function accepts arguments for the score and the index where the score should be stored in the `testScores` array.

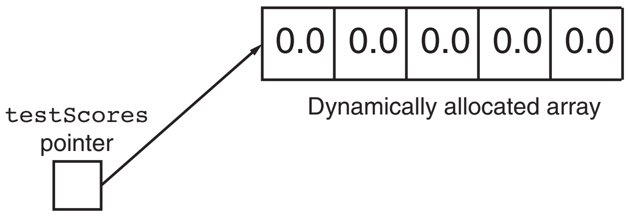
- The `setStudentName` member function, in lines 39 through 40, accepts an argument that is assigned to the `studentName` attribute.
- The `getStudentName` member function, in lines 43 through 44, returns the value of the `studentName` attribute.
- The `getNumTestScores` member function, in lines 47 through 48, returns the number of test scores stored in the object.
- The `getTestScore` member function, in lines 51 through 52, returns a specific score (specified by the index parameter) from the `testScores` attribute.

A potential problem with this class lies in the fact that one of its members, `testScores`, is a pointer. The `createTestScoresArray` member function (called by the constructor) performs a critical operation with the pointer: it dynamically allocates a section of memory for the `testScores` array and assigns default values to each of its element. For instance, the following statement creates a `StudentTestScores` object named `student1`, whose `test-Scores` member references dynamically allocated memory holding an array of 5 `double`'s:

```
StudentTestScores("Maria Jones Tucker", 5);
```

This is depicted in Figure 14-3.

Figure 14-3

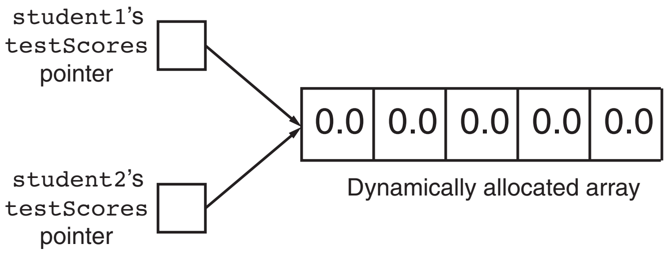


Consider what happens when another `StudentTestScores` object is created and initialized with the `student1` object, as in the following statement:

```
StudentTestScores student2 = student1;
```

In the statement above, `student2`'s constructor isn't called. Instead, memberwise assignment takes place, copying each of `student1`'s member variables into `student2`. This means that a separate section of memory is not allocated for `student2`'s `testScores` member. It simply gets a copy of the address stored in `student1`'s `testScores` member. Both pointers will point to the same address, as depicted in Figure 14-4.

Figure 14-4



In this situation, either object can manipulate the values stored in the array, causing the changes to show up in the other object. Likewise, one object can be destroyed, causing its destructor to be called, which frees the allocated memory. The remaining object's `testScores` pointer would still reference this section of memory, although it should no longer be used.

The solution to this problem is to create a *copy constructor* for the object. A copy constructor is a special constructor that's called when an object is initialized with another object's data. It has the same form as other constructors, except it has a reference parameter of the same class type as the object itself. For example, here is a copy constructor for the `StudentTestScores` class:

```
StudentTestScores(StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < length; i++)
       testScores[i] = obj.testScores[i]; }
```

When the = operator is used to initialize a `StudentTestScores` object with the contents of another `StudentTestScores` object, the copy constructor is called. The `StudentTestScores` object that appears on the right side of the = operator is passed as an argument to the copy constructor. For example, look at the following statement:

```
StudentTestScores student1 ("Molly McBride", 8);
StudentTestScores student2 = student1;
```

In this code, the `student1` object is passed as an argument to the `student2` object's copy constructor.

NOTE: C++ requires that a copy constructor's parameter be a reference object.

As you can see from studying the copy constructor's code, `student2`'s `testScores` member will properly reference its own dynamically allocated memory. There will be no danger of `student1` inadvertently destroying or corrupting `student2`'s data.

Using `const` Parameters in Copy Constructors

Because copy constructors are required to use reference parameters, they have access to their argument's data. Since the purpose of a copy constructor is to make a copy of the argument, there is no reason the constructor should modify the argument's data. With this in mind, it's a good idea to make copy constructors' parameters constant by specifying the `const` key word in the parameter list. Here is an example:

```
StudentTestScores(const StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
       testScores[i] = obj.testScores[i]; }
```

The const key word ensures that the function cannot change the contents of the parameter. This will prevent you from inadvertently writing code that corrupts data.

The complete listing for the revised StudentTestScores class is shown here. (This file is stored in the Student Source Code Folder Chapter 14\StudentTestScores Version 2.)

Contents of StudentTestScores.h (Version 2)

```
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
    string studentName;  // The student's name
    double *testScores;  // Points to array of test scores
    int numTestScores;   // Number of test scores

     // Private member function to create an
     // array of test scores.
    void createTestScoresArray(int size)
    { numTestScores = size;
      testScores = new double[size];
      for (int i = 0; i < size; i++)
          testScores[i] = DEFAULT_SCORE; }

public:
    // Constructor
    StudentTestScores(string name, int numScores)
    { studentName = name;
      createTestScoresArray(numScores); }

    // Copy constructor
    StudentTestScores(const StudentTestScores &obj)
    { studentName = obj.studentName;
      numTestScores = obj.numTestScores;
      testScores = new double[numTestScores];
      for (int i = 0; i < numTestScores; i++)
          testScores[i] = obj.testScores[i]; }

    // Destructor
    ~StudentTestScores()
    { delete [] testScores; }

    // The setTestScore function sets a specific
    // test score's value.
    void setTestScore(double score, int index)
    { testScores[index] = score; }

```

```
          // Set the student's name.
          void setStudentName(string name)
          { studentName = name; }

          // Get the student's name.
          string getStudentName() const
          { return studentName; }

          // Get the number of test scores.
          int getNumTestScores() const
          { return numTestScores; }

          // Get a specific test score.
          double getTestScore(int index) const
          { return testScores[index]; }
      };
      #endif
```

Copy Constructors and Function Parameters

When a class object is passed by value as an argument to a function, it is passed to a parameter that is also a class object, and the copy constructor of the function's parameter is called. Remember that when a nonreference class object is used as a function parameter, it is created when the function is called, and it is initialized with the argument's value.

This is why C++ requires the parameter of a copy constructor to be a reference object. If an object were passed to the copy constructor by value, the copy constructor would create a copy of the argument and store it in the parameter object. When the parameter object is created, its copy constructor will be called, thus causing another parameter object to be created. This process will continue indefinitely (or at least until the available memory fills up, causing the program to halt).

To prevent the copy constructor from calling itself an infinite number of times, C++ requires its parameter to be a reference object.

The Default Copy Constructor

Although you may not realize it, you have seen the action of a copy constructor before. If a class doesn't have a copy constructor, C++ creates a *default copy constructor* for it. The default copy constructor performs the memberwise assignment discussed in the previous section.

Checkpoint

14.8 Briefly describe what is meant by memberwise assignment.

14.9 Describe two instances when memberwise assignment occurs.

14.10 Describe a situation in which memberwise assignment should not be used.

14.11 When is a copy constructor called?

14.12 How does the compiler know that a member function is a copy constructor?

14.13 What action is performed by a class's default copy constructor?

14.5 Operator Overloading

CONCEPT: C++ allows you to redefine how standard operators work when used with class objects.

VideoNote
Operator Overloading

C++ provides many operators to manipulate data of the primitive data types. However, what if you wish to use an operator to manipulate class objects? For example, assume that a class named `Date` exists, and objects of the `Date` class hold the month, day, and year in member variables. Suppose the `Date` class has a member function named `add`. The `add` member function adds a number of days to the date and adjusts the member variables if the date goes to another month or year. For example, the following statement adds five days to the date stored in the `today` object:

```
today.add(5);
```

Although it might be obvious that the statement is adding five days to the date stored in `today`, the use of an operator might be more intuitive. For example, look at the following statement:

```
today += 5;
```

This statement uses the standard `+=` operator to add 5 to `today`. This behavior does not happen automatically, however. The `+=` operator must be *overloaded* for this action to occur. In this section, you will learn to overload many of C++'s operators to perform specialized operations on class objects.

NOTE: You have already experienced the behavior of an overloaded operator. The / operator performs two types of division: floating point and integer. If one of the / operator's operands is a floating point type, the result will be a floating point value. If both of the / operator's operands are integers, however, a different behavior occurs: the result is an integer and any fractional part is thrown away.

Overloading the = Operator

Although copy constructors solve the initialization problems inherent with objects containing pointer members, they do not work with simple assignment statements. Copy constructors are just that—constructors. They are only invoked when an object is created. Statements like the following still perform memberwise assignment:

```
student2 = student1;
```

In order to change the way the assignment operator works, it must be overloaded. Operator overloading permits you to redefine an existing operator's behavior when used with a class object.

C++ allows a class to have special member functions called *operator functions*. If you wish to redefine the way a particular operator works with an object, you define a function for that operator. The Operator function is then executed any time the operator is used with an object of that class. For example, the following version of the `StudentTestScores` class overloads the `=` operator. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 3`.)

Contents of `StudentTestScores` (Version 3)

```
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
    string studentName;  // The student's name
    double *testScores;  // Points to array of test scores
    int numTestScores;   // Number of test scores

    // Private member function to create an
    // array of test scores.
    void createTestScoresArray(int size)
    { numTestScores = size;
      testScores = new double[size];
      for (int i = 0; i < size; i++)
          testScores[i] = DEFAULT_SCORE; }

public:
    // Constructor
    StudentTestScores(string name, int numScores)
    { studentName = name;
     createTestScoresArray(numScores); }

    // Copy constructor
    StudentTestScores(const StudentTestScores &obj)
    { studentName = obj.studentName;
      numTestScores = obj.numTestScores;
      testScores = new double[numTestScores];
      for (int i = 0; i < numTestScores; i++)
         testScores[i] = obj.testScores[i]; }

    // Destructor
    ~StudentTestScores()
    { delete [] testScores; }

    // The setTestScore function sets a specific
    // test score's value.
    void setTestScore(double score, int index)
    { testScores[index] = score; }

    // Set the student's name.
    void setStudentName(string name)
    { studentName = name; }

    // Get the student's name.
    string getStudentName() const
    { return studentName; }

```

```
      // Get the number of test scores.
      int getNumTestScores()
      { return numTestScores; }

      // Get a specific test score.
      double getTestScore(int index) const
      { return testScores[index]; }

      // Overloaded = operator
      void operator=(const StudentTestScores &right)
      { delete [] testScores;
        studentName = right.studentName;
        numTestScores = right.numTestScores;
        testScores = new double[numTestScores];
        for (int i = 0; i < numTestScores; i++)
          testScores[i] = right.testScores[i]; }
};
#endif
```

Let's examine the operator function to understand how it works. First look at the function header:

```
Return   Function       Parameter for object
type     name           on the right side of operator
  ↓        ↓                     ↓
void operator = (const studnetTestScores &right)
```

The name of the function is `operator=`. This specifies that the function overloads the = operator. Because it is a member of the `StudentTestScores` class, this function will be called only when an assignment statement executes where the object on the left side of the = operator is a `StudentTestScores` object.

NOTE: You can, if you choose, put spaces around the operator symbol. For instance, the function header above could also read:

```
void operator = (const StudentTestScores &right)
```

The function has one parameter: a constant reference object named `right`. This parameter references the object on the right side of the operator. For example, when the following statement is executed, `right` will reference the `student1` object:

```
student2 = student1;
```

It is not required that the parameter of an operator function be a reference object. The `StudentTestScores` example declares `right` as a `const` reference for the following reasons:

- It was declared as a reference for efficiency purposes. This prevents the compiler from making a copy of the object being passed into the function.
- It was declared constant so the function will not accidentally change the contents of the argument.

NOTE: In the example, the parameter was named `right` simply to illustrate that it references the object on the right side of the operator. You can name the parameter anything you wish. It will always take the object on the operator's right as its argument.

In learning the mechanics of operator overloading, it is helpful to know that the following two statements do the same thing:

```
student2 = student1;            // Call operator= function
student2.operator=(student1);  // Call operator= function
```

In the last statement you can see exactly what is going on in the function call. The `student1` object is being passed to the function's parameter, `right`. Inside the function, the values in `right`'s members are used to initialize `student2`. Notice that the `operator=` function has access to the `right` parameter's private members. Because the `operator=` function is a member of the `StudentTestScores` class, it has access to the private members of any `StudentTestScores` object that is passed into it.

NOTE: C++ allows operator functions to be called with regular function call notation, or by using the operator symbol.

Program 14-6 demonstrates the `StudentTestScores` class with its overloaded assignment operator. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 3`.)

Program 14-6

```
// This program demonstrates the overloaded = operator
#include <iostream>
#include "StudentTestScores.h"
using namespace std;

// Function prototype
void displayStudent(StudentTestScores);

int main()
{
    // Create a StudentTestScores object and
    // assign test scores.
    StudentTestScores student1("Kelly Thorton", 3);
    student1.setTestScore(100.0, 0);
    student1.setTestScore(95.0, 1);
    student1.setTestScore(80, 2);

    // Create another StudentTestScore object
    // with default test scores.
    StudentTestScores student2("Jimmy Griffin", 5);

    // Assign the student1 object to student2
    student2 = student1;

    // Display both objects. They should
    // contain the same data.
    displayStudent(student1);
    displayStudent(student2);
    return 0;
}

```

```
// The displayStudent function accepts a
// StudentTestScores object's data.
void displayStudent(StudentTestScores s)
{
    cout << "Name: " << s.getStudentName() << endl;
    cout << "Test Scores: ";
    for (int i = 0; i < s.getNumTestScores(); i++)
        cout << s.getTestScore(i) << " ";
    cout << endl;
}
```

Program Output

```
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
```

The = Operator's Return Value

There is only one problem with the overloaded = operator shown in Program 14-6: it has a `void` return type. C++'s built-in = operator allows multiple assignment statements such as:

```
a = b = c;
```

In this statement, the expression `b = c` causes `c` to be assigned to `b` and then returns the value of `c`. The return value is then stored in `a`. If a class object's overloaded = operator is to function this way, it too must have a valid return type.

For example, the `StudentTestScores` class's `operator=` function could be written as:

```
const StudentTestScores operator=(const StudentTestScores &right)
{ delete [] testScores;
  studentName = right.studentName;
  numTestScores = right.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
      testScores[i] = right.testScores[i];
  return *this; }
```

The data type of the operator function specifies that a `const StudentTestScores` object is returned. Look at the last statement in the function:

```
return *this;
```

This statement returns the value of a dereferenced pointer: `this`. But what is `this`? Read on.

The `this` Pointer

The `this` pointer is a special built-in pointer that is available to a class's member functions. It always points to the instance of the class making the function call. For example, if `student1` and `student2` are both `StudentTestScores` objects, the following statement causes the `getStudentName` function to operate on `student1`:

```
cout << student1.getStudentName() << endl;
```

Likewise, the following statement causes `getStudentName` to operate on `student2`:

```
cout << student2.getStudentName() << endl;
```

When `getStudentName` is operating on `student1`, the `this` pointer is pointing to `student1`. When `getStudentName` is operating on `student2`, `this` is pointing to `student2`. The `this` pointer always points to the object that is being used to call the member function.

NOTE: The `this` pointer is passed as a hidden argument to all nonstatic member functions.

The overloaded = operator function is demonstrated in Program 14-7. The multiple assignment statement in line 22 causes the `operator=` function to execute. (This file and the revised version of the `StudentTestScores` class is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 4`.)

Program 14-7

```
 1  // This program demonstrates the overloaded = operator returning a value.
 2  #include <iostream>
 3  #include "StudentTestScores.h"
 4  using namespace std;
 5
 6  // Function prototype
 7  void displayStudent(StudentTestScores);
 8
 9  int main()
10  {
11      // Create a StudentTestScores object.
12      StudentTestScores student1("Kelly Thorton", 3);
13      student1.setTestScore(100.0, 0);
14      student1.setTestScore(95.0, 1);
15      student1.setTestScore(80, 2);
16
17      // Create two more StudentTestScores objects.
18      StudentTestScores student2("Jimmy Griffin", 5);
19      StudentTestScores student3("Kristen Lee", 10);
20
21      // Assign student1 to student2 and student3.
22      student3 = student2 = student1;
23
24      // Display the objects.
25      displayStudent(student1);
26      displayStudent(student2);
27      displayStudent(student3);
28      return 0;
29  }
30
31  // displayStudent function
32  void displayStudent(StudentTestScores s)
```

```
{
    cout << "Name: " << s.getStudentName() << endl;
    cout << "Test Scores: ";
    for (int i = 0; i < s.getNumTestScores(); i++)
        cout << s.getTestScore(i) << " ";
    cout << endl;
}
```

Program Output

```
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
```

Some General Issues of Operator Overloading

Now that you have had a taste of operator overloading, let's look at some of the general issues involved in this programming technique.

Although it is not a good programming practice, you can change an operator's entire meaning if that's what you wish to do. There is nothing to prevent you from changing the = symbol from an assignment operator to a "display" operator. For instance, the following class does just that:

```
class Weird
{
private:
   int value;
public:
   Weird(int v)
      { value = v; }
   void operator=(const weird &right)
      { cout << right.value << endl; }
};
```

Although the `operator=` function in the `Weird` class overloads the assignment operator, the function doesn't perform an assignment. Instead, it displays the contents of `right.value`. Consider the following program segment:

```
Weird a(5), b(10);
a = b;
```

Although the statement `a = b` looks like an assignment statement, it actually causes the contents of `b`'s `value` member to be displayed on the screen:

```
10
```

Another operator overloading issue is that you cannot change the number of operands taken by an operator. The = symbol must always be a binary operator. Likewise, ++ and -- must always be unary operators.

The last issue is that although you may overload most of the C++ operators, you cannot overload all of them. Table 14-1 shows all of the C++ operators that may be overloaded.

Table 14-1

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`	`!`	`=`	`<`
`>`	`+=`	`-=`	`*=`	`/=`	`%=`	`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`	`--`	`->*`	`,`	`->`
`[]`	`()`	`new`	`delete`								

NOTE: Some of the operators in Table 14-1 are beyond the scope of this book and are not covered.

The only operators that cannot be overloaded are

```
?:    .    .*    ::    sizeof
```

Overloading Math Operators

Many classes would benefit not only from an overloaded assignment operator, but also from overloaded math operators. To illustrate this, consider the `FeetInches` class shown in the following two files. (These files are stored in the Student Source Code Folder `Chapter 14\FeetInches Version 1`.)

Contents of `FeetInches.h` (Version 1)

```
#ifndef FEETINCHES_H
#define FEETINCHES_H

// The FeetInches class holds distances or measurements
// expressed in feet and inches.

class FeetInches
{
private:
    int feet;          // To hold a number of feet
    int inches;        // To hold a number of inches
    void simplify();   // Defined in FeetInches.cpp
public:
    // Constructor
    FeetInches(int f = 0, int i = 0)
         { feet = f;
          inches = i;
          simplify(); }

    // Mutator functions
    void setFeet(int f)
       { feet = f; }

```

```
     void setInches(int i)
        { inches = i;
          simplify(); }

     // Accessor functions
     int getFeet() const
        { return feet; }

     int getInches() const
        { return inches; }

     // Overloaded operator functions
     FeetInches operator + (const FeetInches &); // Overloaded +
     FeetInches operator - (const FeetInches &); // Overloaded -
};

#endif
```

Contents of `FeetInches.cpp` (Version 1)

```
// Implementation file for the FeetInches class
#include <cstdlib>  // Needed for abs()
#include "FeetInches.h"

//**************************************************************
// Definition of member function simplify. This function      *
// checks for values in the inches member greater than        *
// twelve or less than zero. If such a value is found,        *
// the numbers in feet and inches are adjusted to conform     *
// to a standard feet & inches expression. For example,       *
// 3 feet 14 inches would be adjusted to 4 feet 2 inches and  *
// 5 feet -2 inches would be adjusted to 4 feet 10 inches.    *
//**************************************************************

void FeetInches::simplify()
{
     if (inches >= 12)
     {
          feet += (inches / 12);
          inches = inches % 12;
     }
     else if (inches < 0)
     {
          feet -= ((abs(inches) / 12) + 1);
          inches = 12 - (abs(inches) % 12);
     }
}

//*********************************************
// Overloaded binary + operator.              *
//*********************************************

```

```
FeetInches FeetInches::operator + (const FeetInches &right)
{
     FeetInches temp;

     temp.inches = inches + right.inches;
     temp.feet = feet + right.feet;
     temp.simplify();
     return temp;
}

//*********************************************
// Overloaded binary - operator.              *
//*********************************************

FeetInches FeetInches::operator - (const FeetInches &right)
{
     FeetInches temp;

     temp.inches = inches - right.inches;
     temp.feet = feet - right.feet;
     temp.simplify();
     return temp;
}
```

The `FeetInches` class is designed to hold distances or measurements expressed in feet and inches. It consists of eight member functions:

- A constructor that allows the `feet` and `inches` members to be set. The default values for these members is zero.
- A `setFeet` function for storing a value in the `feet` member.
- A `setInches` function for storing a value in the `inches` member.
- A `getFeet` function for returning the value in the `feet` member.
- A `getInches` function for returning the value in the `inches` member.
- A `simplify` function for normalizing the values held in `feet` and `inches`. This function adjusts any set of values where the `inches` member is greater than 12 or less than 0.
- An `operator +` function that overloads the standard + math operator.
- An `operator -` function that overloads the standard – math operator.

NOTE: The `simplify` function uses the standard library function `abs()` to get the absolute value of the `inches` member. The `abs()` function requires that `cstdlib` be included.

The overloaded + and – operators allow one `FeetInches` object to be added to or subtracted from another. For example, assume the `length1` and `length2` objects are defined and initialized as follows:

```
FeetInches length1(3, 5), length2(6, 3);
```

The `length1` object is holding the value 3 feet 5 inches, and the `length2` object is holding the value 6 feet 3 inches. Because the + operator is overloaded, we can add these two objects in a statement such as:

```
length3 = length1 + length2;
```

This statement will add the values of the `length1` and `length2` objects and store the result in the `length3` object. After the statement executes, the `length3` object will be set to 9 feet 8 inches.

The member function that overloads the + operator appears in lines 33 through 41 of the `FeetInches.cpp` file.

This function is called anytime the + operator is used with two `FeetInches` objects. Just like the overloaded = operator we defined in the previous section, this function has one parameter: a constant reference object named `right`. This parameter references the object on the right side of the operator. For example, when the following statement is executed, `right` will reference the `length2` object:

```
length3 = length1 + length2;
```

As before, it might be helpful to think of the statement above as the following function call:

```
length3 = length1.operator+(length2);
```

The `length2` object is being passed to the function's parameter, `right`. When the function finishes, it will return a `FeetInches` object to `length3`. Now let's see what is happening inside the function. First, notice that a `FeetInches` object named `temp` is defined locally in line 35:

```
FeetInches temp;
```

This object is a temporary location for holding the results of the addition. Next, line 37 adds `inches` to `right.inches` and stores the result in `temp.inches`:

```
temp.inches = inches + right.inches;
```

The `inches` variable is a member of `length1`, the object making the function call. It is the object on the left side of the operator. `right.inches` references the `inches` member of `length2`. The next statement, in line 38, is very similar. It adds `feet` to `right.feet` and stores the result in `temp.feet`:

```
temp.feet = feet + right.feet;
```

At this point in the function, `temp` contains the sum of the `feet` and `inches` members of both objects in the expression. The next step is to adjust the values so they conform to a normal value expressed in feet and inches. This is accomplished in line 39 by calling `temp.simplify()`:

```
temp.simplify();
```

The last step, in line 40, is to return the value stored in `temp`:

```
return temp;
```

In the statement `length3 = length1 + length2`, the `return` statement in the operator function causes the values stored in `temp` to be returned to the `length3` object.

Program 14-8 demonstrates the overloaded operators. (This file is stored in the student source code folder `Chapter 14\FeetInches Version 1`.)

Program 14-8

```
// This program demonstrates the FeetInches class's overloaded
// + and - operators.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
    int feet, inches; // To hold input for feet and inches

    // Create three FeetInches objects. The default arguments
    // for the constructor will be used.
    FeetInches first, second, third;

    // Get a distance from the user.
    cout << "Enter a distance in feet and inches: ";
    cin >> feet >> inches;

    // Store the distance in the first object.
    first.setFeet(feet);
    first.setInches(inches);

    // Get another distance from the user.
    cout << "Enter another distance in feet and inches: ";
    cin >> feet >> inches;

    // Store the distance in second.
    second.setFeet(feet);
    second.setInches(inches);

    // Assign first + second to third.
    third = first + second;

    // Display the result.
    cout << "first + second = ";
    cout << third.getFeet() << " feet, ";
    cout << third.getInches() << " inches.\n";

    // Assign first - second to third.
    third = first - second;

    // Display the result.
    cout << "first - second = ";
    cout << third.getFeet() << " feet, ";
    cout << third.getInches() << " inches.\n";

    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a distance in feet and inches: 6 5 [Enter]
Enter another distance in feet and inches: 3 10 [Enter]
first + second = 10 feet, 3 inches.
first - second = 2 feet, 7 inches.
```

Overloading the Prefix ++ Operator

Unary operators, such as ++ and – –, are overloaded in a fashion similar to the way binary operators are implemented. Because unary operators only affect the object making the operator function call, however, there is no need for a parameter. For example, let's say you wish to have a prefix increment operator for the `FeetInches` class. Assume the `FeetInches` object `distance` is set to the values 7 feet and 5 inches. A `++` operator function could be designed to increment the object's `inches` member. The following statement would cause `distance` to have the value 7 feet 6 inches:

```
++distance;
```

The following function overloads the prefix `++` operator to work in this fashion:

```
FeetInches FeetInches::operator++()
{
   ++inches;
   simplify();
   return *this;
}
```

This function first increments the object's `inches` member. The `simplify()` function is called and then the dereferenced `this` pointer is returned. This allows the operator to perform properly in statements like this:

```
distance2 = ++distance1;
```

Remember, the statement above is equivalent to

```
distance2 = distance1.operator++();
```

Overloading the Postfix ++ Operator

Overloading the postfix `++` operator is only slightly different than overloading the prefix version. Here is the function that overloads the postfix operator with the `FeetInches` class:

```
FeetInches FeetInches::operator++(int)
{
   FeetInches temp(feet, inches);
   inches++;
   simplify();
   return temp;
}
```

The first difference you will notice is the use of a *dummy parameter*. The word `int` in the function's parentheses establishes a nameless integer parameter. When C++ sees this parameter in an operator function, it knows the function is designed to be used in postfix mode. The second difference is the use of a temporary local variable, the `temp` object. `temp` is initialized with the `feet` and `inches` values of the object making the function call. `temp`, therefore, is a copy of the object being incremented, but before the increment takes place. After `inches` is incremented and the `simplify` function is called, the contents of `temp` is returned. This causes the postfix operator to behave correctly in a statement like this:

```
distance2 = distance1++;
```

You will find a version of the `FeetInches` class with the overloaded prefix and postfix `++` operators stored in the Student Source Code Folder `Chapter 14\FeetInches Version 2`. In that folder you will also find Program 14-9, which demonstrates these overloaded operators.

Program 14-9

```
// This program demonstrates the FeetInches class's overloaded
// prefix and postfix ++ operators.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
    int count;  // Loop counter

    // Define a FeetInches object with the default
    // value of 0 feet, 0 inches.
    FeetInches first;

    // Define a FeetInches object with 1 foot 5 inches.
    FeetInches second(1, 5);

    // Use the prefix ++ operator.
    cout << "Demonstrating prefix ++ operator.\n";
    for (count = 0; count < 12; count++)
    {
        first = ++second;
        cout << "first: " << first.getFeet() << " feet, ";
        cout << first.getInches() << " inches. ";
        cout << "second: " << second.getFeet() << " feet, ";
        cout << second.getInches() << " inches.\n";
    }

    // Use the postfix ++ operator.
    cout << "\nDemonstrating postfix ++ operator.\n";
    for (count = 0; count < 12; count++)
    {
        first = second++;
        cout << "first: " << first.getFeet() << " feet, ";
        cout << first.getInches() << " inches. ";
        cout << "second: " << second.getFeet() << " feet, ";
        cout << second.getInches() << " inches.\n";
    }

    return 0;
}
```

Program Output

```
Demonstrating prefix ++ operator.
first: 1 feet 6 inches. second: 1 feet 6 inches.
first: 1 feet 7 inches. second: 1 feet 7 inches.
first: 1 feet 8 inches. second: 1 feet 8 inches.
first: 1 feet 9 inches. second: 1 feet 9 inches.
first: 1 feet 10 inches. second: 1 feet 10 inches.
first: 1 feet 11 inches. second: 1 feet 11 inches.
first: 2 feet 0 inches. second: 2 feet 0 inches.
first: 2 feet 1 inches. second: 2 feet 1 inches.
first: 2 feet 2 inches. second: 2 feet 2 inches.
first: 2 feet 3 inches. second: 2 feet 3 inches.
first: 2 feet 4 inches. second: 2 feet 4 inches.
first: 2 feet 5 inches. second: 2 feet 5 inches.

Demonstrating postfix ++ operator.
first: 2 feet 5 inches. second: 2 feet 6 inches.
first: 2 feet 6 inches. second: 2 feet 7 inches.
first: 2 feet 7 inches. second: 2 feet 8 inches.
first: 2 feet 8 inches. second: 2 feet 9 inches.
first: 2 feet 9 inches. second: 2 feet 10 inches.
first: 2 feet 10 inches. second: 2 feet 11 inches.
first: 2 feet 11 inches. second: 3 feet 0 inches.
first: 3 feet 0 inches. second: 3 feet 1 inches.
first: 3 feet 1 inches. second: 3 feet 2 inches.
first: 3 feet 2 inches. second: 3 feet 3 inches.
first: 3 feet 3 inches. second: 3 feet 4 inches.
first: 3 feet 4 inches. second: 3 feet 5 inches.
```

Checkpoint

14.14 Assume there is a class named `Pet`. Write the prototype for a member function of `Pet` that overloads the = operator.

14.15 Assume that `dog` and `cat` are instances of the `Pet` class, which has overloaded the = operator. Rewrite the following statement so it appears in function call notation instead of operator notation:

```
dog = cat;
```

14.16 What is the disadvantage of an overloaded = operator returning `void`?

14.17 Describe the purpose of the `this` pointer.

14.18 The `this` pointer is automatically passed to what type of functions?

14.19 Assume there is a class named `Animal` that overloads the = and + operators. In the following statement, assume `cat`, `tiger`, and `wildcat` are all instances of the `Animal` class:

```
wildcat = cat + tiger;
```

Of the three objects, `wildcat`, `cat`, or `tiger`, which is calling the `operator+` function? Which object is passed as an argument into the function?

14.20 What does the use of a dummy parameter in a unary operator function indicate to the compiler?

Overloading Relational Operators

In addition to the assignment and math operators, relational operators may be overloaded. This capability allows classes to be compared in statements that use relational expressions such as:

```
if (distance1 < distance2)
{
   ... code ...
}
```

Overloaded relational operators are implemented like other binary operators. The only difference is that a relational operator function should always return a `true` or `false` value. The `FeetInches` class in the Student Source Code Folder `Chapter 14\ FeetInches Version 3` contains functions to overload the >, <, and == relational operators. Here is the function for overloading the > operator:

```
bool FeetInches::operator > (const FeetInches &right)
{
   bool status;

   if (feet > right.feet)
      status = true;
   else if (feet == right.feet &&  inches > right.inches)
      status = true;
   else
      status = false;

   return status;
}
```

As you can see, the function compares the `feet` member (and if necessary, the `inches` member) with that of the parameter. If the calling object contains a value greater than that of the parameter, `true` is returned. Otherwise, `false` is returned.

Here is the code that overloads the < operator:

```
bool FeetInches::operator < (const FeetInches &right)
{
   bool status;

   if (feet < right.feet)
      status = true;
   else if (feet == right.feet &&  inches < right.inches)
      status = true;
   else
      status = false;

   return status;
}
```

Here is the code that overloads the == operator:

```
bool FeetInches::operator == (const FeetInches &right)
{
   bool status;

   if (feet == right.feet &&  inches == right.inches)
      status = true;
   else
      status = false;
```

```
        return status;
    }
```

Program 14-10 demonstrates these overloaded operators. (This file is also stored in the Student Source Code Folder `Chapter 14\FeetInches Version 3`.)

Program 14-10

```
// This program demonstrates the FeetInches class's overloaded
// relational operators.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
    int feet, inches; // To hold input for feet and inches

    // Create two FeetInches objects. The default arguments
    // for the constructor will be used.
    FeetInches first, second;

    // Get a distance from the user.
    cout << "Enter a distance in feet and inches: ";
    cin >> feet >> inches;

    // Store the distance in first.
    first.setFeet(feet);
    first.setInches(inches);

    // Get another distance.
    cout << "Enter another distance in feet and inches: ";
    cin >> feet >> inches;

    // Store the distance in second.
    second.setFeet(feet);
    second.setInches(inches);

    // Compare the two objects.
    if (first == second)
        cout << "first is equal to second.\n";
    if (first > second)
        cout << "first is greater than second.\n";
    if (first < second)
        cout << "first is less than second.\n";

    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a distance in feet and inches: 6 5 [Enter]
Enter another distance in feet and inches: 3 10 [Enter]
first is greater than second.
```

(program output continues)

Program 14-10 *(continued)*

Program Output with Different Example Input Shown in Bold

```
Enter a distance in feet and inches: 5 5 [Enter]
Enter another distance in feet and inches: 5 5 [Enter]
first is equal to second.
```

Program Output with Different Example Input Shown in Bold

```
Enter a distance in feet and inches: 3 4 [Enter]
Enter another distance in feet and inches: 3 7 [Enter]
first is less than second.
```

Overloading the << and >> Operators

Overloading the math and relational operators gives you the ability to write those types of expressions with class objects just as naturally as with integers, floats, and other built-in data types. If an object's primary data members are private, however, you still have to make explicit member function calls to send their values to `cout`. For example, assume `distance` is a `FeetInches` object. The following statements display its internal values:

```
cout << distance.getFeet() << " feet, ";
cout << distance.getInches() << "inches";
```

It is also necessary to explicitly call member functions to set a `FeetInches` object's data. For instance, the following statements set the `distance` object to user-specified values:

```
cout << "Enter a value in feet: ";
cin >> f;
distance.setFeet(f);
cout << "Enter a value in inches: ";
cin >> i;
distance.setInches(i);
```

By overloading the stream insertion operator (<<), you could send the `distance` object to `cout`, as shown in the following code, and have the screen output automatically formatted in the correct way.

```
cout << distance;
```

Likewise, by overloading the stream extraction operator (>>), the `distance` object could take values directly from `cin`, as shown here.

```
cin >> distance;
```

Overloading these operators is done in a slightly different way, however, than overloading other operators. These operators are actually part of the `ostream` and `istream` classes defined in the C++ runtime library. (The `cout` and `cin` objects are instances of `ostream` and `istream`.) You must write operator functions to overload the `ostream` version of << and the `istream` version of >>, so they work directly with a class such as `FeetInches`. The `FeetInches` class in the Student Source Code Folder `Chapter 14\FeetInches Version 4` contains functions to overload the << and >> operators. Here is the function that overloads the << operator:

```
ostream &operator << (ostream &strm, const FeetInches &obj)
{
   strm << obj.feet << " feet, " << obj.inches << " inches";
   return strm;
}
```

Notice the function has two parameters: an `ostream` reference object and a `const FeetInches` reference object. The `ostream` parameter will be a reference to the actual `ostream` object on the left side of the << operator. The second parameter is a reference to a `FeetInches` object. This parameter will reference the object on the right side of the << operator. This function tells C++ how to handle any expression that has the following form:

```
ostreamObject << FeetInchesObject
```

So, when C++ encounters the following statement, it will call the overloaded `operator<<` function:

```
cout << distance;
```

Notice that the function's return type is `ostream &`. This means that the function returns a reference to an `ostream` object. When the `return strm;` statement executes, it doesn't return a copy of `strm`, but a reference to it. This allows you to chain together several expressions using the overloaded << operator, such as:

```
cout << distance1 << " " << distance2 << endl;
```

Here is the function that overloads the stream extraction operator to work with the `FeetInches` class:

```
istream &operator >> (istream &strm, FeetInches &obj)
{
   // Prompt the user for the feet.
   cout << "Feet: ";
   strm >> obj.feet;

   // Prompt the user for the inches.
   cout << "Inches: ";
   strm >> obj.inches;

   // Normalize the values.
   obj.simplify();

   return strm;
}
```

The same principles hold true for this operator. It tells C++ how to handle any expression in the following form:

```
istreamObject >> FeetInchesObject
```

Once again, the function returns a reference to an `istream` object, so several of these expressions may be chained together.

You have probably realized that neither of these functions is quite ready to work, though. Both functions attempt to directly access the `FeetInches` object's private members. Because the functions aren't themselves members of the `FeetInches` class, they don't have this type of access. The next step is to make the operator functions friends of `FeetInches`. This is shown in the following listing of the `FeetInches` class declaration. (This file is stored in the Student Source Code Folder `Chapter 14\FeetInches Version 4`.)

NOTE: Some compilers require you to prototype the >> and << operator functions outside the class. For this reason, we have added the following statements to the `FeetInches.h` class specification file.

```
class FeetInches;                          // Forward Declaration

// Function Prototypes for Overloaded Stream Operators
ostream &operator << (ostream &, const FeetInches &);
istream &operator >> (istream &, FeetInches &);
```

Contents of `FeetInches.h` (Version 4)

```
#ifndef FEETINCHES_H
#define FEETINCHES_H

#include <iostream>
using namespace std;

class FeetInches; // Forward Declaration

// Function Prototypes for Overloaded Stream Operators
ostream &operator << (ostream &, const FeetInches &);
istream &operator >> (istream &, FeetInches &);

// The FeetInches class holds distances or measurements
// expressed in feet and inches.

class FeetInches
{
private:
    int feet;          // To hold a number of feet
    int inches;        // To hold a number of inches
    void simplify();   // Defined in FeetInches.cpp
public:
    // Constructor
    FeetInches(int f = 0, int i = 0)
        { feet = f;
          inches = i;
          simplify(); }

    // Mutator functions
    void setFeet(int f)
        { feet = f; }

    void setInches(int i)
        { inches = i;
          simplify(); }

    // Accessor functions
38        int getFeet() const
39           { return feet; }
40
41        int getInches() const
42            { return inches; }
43
44        // Overloaded operator functions
45        FeetInches operator + (const FeetInches &); // Overloaded +
46        FeetInches operator - (const FeetInches &); // Overloaded -
47        FeetInches operator ++ ();                 // Prefix ++
48        FeetInches operator ++ (int);              // Postfix ++
49        bool operator > (const FeetInches &);      // Overloaded >
50        bool operator < (const FeetInches &);      // Overloaded <
51        bool operator == (const FeetInches &);     // Overloaded ==
52
53        // Friends
54        friend ostream &operator << (ostream &, const FeetInches &);
55        friend istream &operator >> (istream &, FeetInches &);
56    };
57
58    #endif
```

Lines 54 and 55 in the class declaration tell C++ to make the overloaded << and >> operator functions friends of the `FeetInches` class:

```
friend ostream &operator<<(ostream &, const FeetInches &);
friend istream &operator>>(istream &, FeetInches &);
```

These statements give the operator functions direct access to the `FeetInches` class's private members. Program 14-11 demonstrates how the overloaded operators work. (This file is also stored in the Student Source Code Folder `Chapter 14\FeetInches Version 4`.)

Program 14-11

```
// This program demonstrates the << and >> operators,
// overloaded to work with the FeetInches class.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
    FeetInches first, second; // Define two objects.

    // Get a distance for the first object.
    cout << "Enter a distance in feet and inches.\n";
    cin >> first;

    // Get a distance for the second object.
    cout << "Enter another distance in feet and inches.\n";
    cin >> second;

    // Display the values in the objects.
    cout << "The values you entered are:\n";
```

(program continues)

Program 14-11 *(continued)*

```
      cout << first << " and " << second << endl;
      return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a distance in feet and inches.
Feet: 6 [Enter]
Inches: 5 [Enter]
Enter another distance in feet and inches.
Feet: 3 [Enter]
Inches: 10 [Enter]
The values you entered are:
6 feet, 5 inches and 3 feet, 10 inches
```

Overloading the [ ] Operator

In addition to the traditional operators, C++ allows you to change the way the `[]` symbols work. This gives you the ability to write classes that have array-like behaviors. For example, the `string` class overloads the `[]` operator so you can access the individual characters stored in `string` class objects. Assume the following definition exists in a program:

```
string name = "William";
```

The first character in the string, ‘W,’ is stored at `name[0]`, so the following statement will display W on the screen.

```
cout << name[0];
```

You can use the overloaded `[]` operator to create an array class, like the following one. The class behaves like a regular array, but performs the bounds-checking that C++ lacks.

Contents of `IntArray.h`

```
// Specification file for the IntArray class
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
private:
     int *aptr;                     // Pointer to the array
     int arraySize;                 // Holds the array size
     void subscriptError();         // Handles invalid subscripts
public:
     IntArray(int);                 // Constructor
     IntArray(const IntArray &);    // Copy constructor
     ~IntArray();                   // Destructor

     int size() const               // Returns the array size
         { return arraySize;}

     int &operator[](const int &);  // Overloaded [] operator
};
#endif
```

Contents of `IntArray.cpp`

```
// Implementation file for the IntArray class
#include <iostream>
#include <cstdlib>   // For the exit function
#include "IntArray.h"
using namespace std;

//*******************************************************
// Constructor for IntArray class. Sets the size of the *
// array and allocates memory for it.                   *
//*******************************************************

IntArray::IntArray(int s)
{
    arraySize = s;
    aptr = new int [s];
    for (int count = 0; count < arraySize; count++)
        *(aptr + count) = 0;
}

//******************************************************
// Copy Constructor for IntArray class.               *
//******************************************************

IntArray::IntArray(const IntArray &obj)
{
    arraySize = obj.arraySize;
    aptr = new int [arraySize];
    for(int count = 0; count < arraySize; count++)
        *(aptr + count) = *(obj.aptr + count);
}

//******************************************************
// Destructor for IntArray class.                     *
//******************************************************

IntArray::~IntArray()
{
    if (arraySize > 0)
        delete [] aptr;
}

//************************************************************
// subscriptError function. Displays an error message and   *
// terminates the program when a subscript is out of range. *
//************************************************************

void IntArray::subscriptError()
{
    cout << "ERROR: Subscript out of range.\n";
    exit(0);
}

```

```
53  //*****************************************************
54  // Overloaded [] operator. The argument is a subscript.*
55  // This function returns a reference to the element    *
56  // in the array indexed by the subscript.              *
57  //*****************************************************
58
59  int &IntArray::operator[](const int &sub)
60  {
61      if (sub < 0 || sub >= arraySize)
62          subscriptError();
63      return aptr[sub];
64  }
```

Before focusing on the overloaded operator, let's look at the constructors and the destructor. The code for the first constructor in lines 12 through 18 of the `IntArray.cpp` file follows:

```
IntArray::IntArray(int s)
{
   arraySize = s;
   aptr = new int [s];
   for (int count = 0; count < arraySize; count++)
      *(aptr + count) = 0;
}
```

When an instance of the class is defined, the number of elements the array is to have is passed into the constructor's parameter, `s`. This value is copied to the `arraySize` member, and then used to dynamically allocate enough memory for the array. The constructor's final step is to store zeros in all of the array's elements:

```
for (int count = 0; count < arraySize; count++)
   *(aptr + count) = 0;
```

The class also has a copy constructor in lines 24 through 30, which is used when a class object is initialized with another object's data:

```
IntArray::IntArray(const IntArray &obj)
{
   arraySize = obj.arraySize;
   aptr = new int [arraySize];
   for(int count = 0; count < arraySize; count++)
      *(aptr + count) = *(obj.aptr + count);
}
```

A reference to the initializing object is passed into the parameter `obj`. Once the memory is successfully allocated for the array, the constructor copies all the values in `obj`'s array into the calling object's array.

The destructor, in lines 36 through 40, simply frees the memory allocated by the class's constructors. First, however, it checks the value in `arraySize` to be sure the array has at least one element:

```
IntArray::~IntArray()
{
   if (arraySize > 0)
      delete [] aptr;
}
```

The `[]` operator is overloaded similarly to other operators. The definition of the `operator[]` function appears in lines 59 through 64:

```
int &IntArray::operator[](const int &sub)
{
   if (sub < 0 || sub >= arraySize)
      subscriptError();
   return aptr[sub];
}
```

The `operator[]` function can have only a single parameter. The one shown uses a constant reference to an integer. This parameter holds the value placed inside the brackets in an expression. For example, if `table` is an `IntArray` object, the number 12 will be passed into the `sub` parameter in the following statement:

```
cout << table[12];
```

Inside the function, the value in the `sub` parameter is tested by the following `if` statement:

```
if (sub < 0 || sub >= arraySize)
   subscriptError();
```

This statement determines whether `sub` is within the range of the array's subscripts. If `sub` is less than 0 or greater than or equal to `arraySize`, it's not a valid subscript, so the `subscriptError` function is called. If `sub` is within range, the function uses it as an offset into the array and returns a reference to the value stored at that location.

One critically important aspect of the function above is its return type. It's crucial that the function return not simply an integer, but a *reference* to an integer. The reason for this is that expressions such as the following must be possible:

```
table[5] = 27;
```

Remember, the built-in = operator requires the object on its left to be an lvalue. An lvalue must represent a modifiable memory location, such as a variable. The integer return value of a function is not an lvalue. If the `operator[]` function merely returns an integer, it cannot be used to create expressions placed on the left side of an assignment operator.

A reference to an integer, however, is an lvalue. If the `operator[]` function returns a reference, it can be used to create expressions like the following:

```
table[7] = 52;
```

In this statement, the `operator[]` function is called with 7 passed as its argument. Assuming 7 is within range, the function returns a reference to the integer stored at `(aptr + 7)`. In essence, the statement above is equivalent to:

```
*(aptr + 7) = 52;
```

Because the `operator[]` function returns actual integers stored in the array, it is not necessary for math or relational operators to be overloaded. Even the stream operators << and >> will work just as they are with the `IntArray` class.

Program 14-12 demonstrates how the class works.

Program 14-12

```
// This program demonstrates an overloaded [] operator.
#include <iostream>
#include "IntArray.h"
using namespace std;

int main()
{
    const int SIZE = 10;  // Array size

    // Define an IntArray with 10 elements.
    IntArray table(SIZE);

    // Store values in the array.
    for (int x = 0; x < SIZE; x++)
        table[x] = (x * 2);

    // Display the values in the array.
    for (int x = 0; x < SIZE; x++)
        cout << table[x] << " ";
    cout << endl;

    // Use the standard + operator on array elements.
    for (int x = 0; x < SIZE; x++)
        table[x] = table[x] + 5;

    // Display the values in the array.
    for (int x = 0; x < SIZE; x++)
        cout << table[x] << " ";
    cout << endl;

    // Use the standard ++ operator on array elements.
    for (int x = 0; x < SIZE; x++)
        table[x]++;

    // Display the values in the array.
    for (int x = 0; x < SIZE; x++)
        cout << table[x] << " ";
    cout << endl;

    return 0;
}
```

Program Output

```
0 2 4 6 8 10 12 14 16 18
5 7 9 11 13 15 17 19 21 23
6 8 10 12 14 16 18 20 22 24
```

Program 14-13 demonstrates the `IntArray` class's bounds-checking capability.

Program 14-13

```
// This program demonstrates the IntArray class's bounds-checking ability.
#include <iostream>
#include "IntArray.h"
using namespace std;

int main()
{
    const int SIZE = 10;  // Array size

    // Define an IntArray with 10 elements.
    IntArray table(SIZE);

    // Store values in the array.
    for (int x = 0; x < SIZE; x++)
        table[x] = x;

    // Display the values in the array.
    for (int x = 0; x < SIZE; x++)
        cout << table[x] << " ";
    cout << endl;

    // Attempt to use an invalid subscript.
    cout << "Now attempting to use an invalid subscript.\n";
    table[SIZE + 1] = 0;
    return 0;
}
```

Program Output

```
0 1 2 3 4 5 6 7 8 9
Now attempting to use an invalid subscript.
ERROR: Subscript out of range.
```

Checkpoint

14.21 Describe the values that should be returned from functions that overload relational operators.

14.22 What is the advantage of overloading the << and >> operators?

14.23 What type of object should an overloaded << operator function return?

14.24 What type of object should an overloaded >> operator function return?

14.25 If an overloaded << or >> operator accesses a private member of a class, what must be done in that class's declaration?

14.26 Assume the class `NumList` has overloaded the `[]` operator. In the expression below, `list1` is an instance of the `NumList` class:

```
list1[25]
```

Rewrite the expression above to explicitly call the function that overloads the `[]` operator.

14.6 Object Conversion

CONCEPT: Special operator functions may be written to convert a class object to any other type.

As you've already seen, operator functions allow classes to work more like built-in data types. Another capability that operator functions can give classes is automatic type conversion.

Data type conversion happens "behind the scenes" with the built-in data types. For instance, suppose a program uses the following variables:

```
int i;
double d;
```

The statement below automatically converts the value in `i` to a floating-point number and stores it in `d`:

```
d = i;
```

Likewise, the following statement converts the value in `d` to an integer (truncating the fractional part) and stores it in `i`:

```
i = d;
```

The same functionality can also be given to class objects. For example, assuming `distance` is a `FeetInches` object and `d` is a `double`, the following statement would conveniently convert `distance`'s value into a floating-point number and store it in `d`, if `FeetInches` is properly written:

```
d = distance;
```

To be able to use a statement such as this, an operator function must be written to perform the conversion. The Student Source Code Folder `Chapter 14\FeetInches Version 5` contains a version of the `FeetInches` class with such an operator function. Here is the code for the operator function that converts a `FeetInches` object to a `double`:

```
FeetInches::operator double()
{
   double temp = feet;

   temp += (inches / 12.0);
   return temp;
}
```

This function contains an algorithm that will calculate the decimal equivalent of a feet and inches measurement. For example, the value 4 feet 6 inches will be converted to 4.5. This value is stored in the local variable `temp`. The `temp` variable is then returned.

NOTE: No return type is specified in the function header. Because the function is a `FeetInches-to-double` conversion function, it will always return a `double`. Also, because the function takes no arguments, there are no parameters.

The revised `FeetInches` class also has an operator function that converts a `FeetInches` object to an `int`. The function, shown here, simply returns the `feet` member, thus truncating the `inches` value:

```
FeetInches:: operator int()
{
   return feet;
}
```

Program 14-14 demonstrates both of these conversion functions. (This file is also stored in the Student Source Code Folder `Chapter 14\FeetInches Version 5`.)

Program 14-14

```
// This program demonstrates the FeetInches class's
// conversion functions.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
    double d;  // To hold double input
    int i;     // To hold int input

    // Define a FeetInches object.
    FeetInches distance;

    // Get a distance from the user.
    cout << "Enter a distance in feet and inches:\n";
    cin >> distance;

    // Convert the distance object to a double.
    d = distance;

    // Convert the distance object to an int.
    i = distance;

    // Display the values.
    cout << "The value " << distance;
    cout << " is equivalent to " << d << " feet\n";
    cout << "or " << i << " feet, rounded down.\n";
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a distance in feet and inches:
Feet: 8 [Enter]
Inches: 6 [Enter]
The value 8 feet, 6 inches is equivalent to 8.5 feet
or 8 feet, rounded down.
```

See the Case Study on Creating a String Class for another example. You can download the case study from the book's companion Web site at www.pearsonhighered.com/gaddis.

Checkpoint

14.27 When overloading a binary operator such as + or –, what object is passed into the operator function's parameter?

14.28 Explain why overloaded prefix and postfix ++ and –– operator functions should return a value.

14.29 How does C++ tell the difference between an overloaded prefix and postfix ++ or –– operator function?

14.30 Write member functions of the `FeetInches` class that overload the prefix and postfix –– operators. Demonstrate the functions in a simple program similar to Program 14-14.

14.7 Aggregation

CONCEPT: Aggregation occurs when a class contains an instance of another class.

VideoNote
Class Aggregation

In real life, objects are frequently made of other objects. A house, for example, is made of door objects, window objects, wall objects, and much more. It is the combination of all these objects that makes a house object.

When designing software, it sometimes makes sense to create an object from other objects. For example, suppose you need an object to represent a course that you are taking in college. You decide to create a `Course` class, which will hold the following information:

- The course name
- The instructor's last name, first name, and office number
- The textbook's title, author, and publisher

In addition to the course name, the class will hold items related to the instructor and the textbook. You could put attributes for each of these items in the `Course` class. However, a good design principle is to separate related items into their own classes. In this example, an `Instructor` class could be created to hold the instructor-related data and a `TextBook` class could be created to hold the textbook-related data. Instances of these classes could then be used as attributes in the `Course` class.

Let's take a closer look at how this might be done. To keep things simple, the `Instructor` class will have only the following functions:

- A default constructor that assigns empty strings to the instructor's last name, first name, and office number.
- A constructor that accepts arguments for the instructor's last name, first name, and office number
- A `set` function that can be used to set all of the class's attributes
- A `print` function that displays the object's attribute values

The code for the `Instructor` class is shown here:

Contents of `Instructor.h`

```
#ifndef INSTRUCTOR
#define INSTRUCTOR
#include <iostream>
#include <string>
using namespace std;

// Instructor class
class Instructor
{
private:
    string lastName;     // Last name
    string firstName;    // First name
    string officeNumber; // Office number
public:
    // The default constructor stores empty strings
    // in the string objects.
    Instructor()
       { set("", "", ""); }

    // Constructor
    Instructor(string lname, string fname, string office)
       { set(lname, fname, office); }

    // set function
    void set(string lname, string fname, string office)
       { lastName = lname;
         firstName = fname;
         officeNumber = office; }

     // print function
     void print() const
        { cout << "Last name: " << lastName << endl;
          cout << "First name: " << firstName << endl;
          cout << "Office number: " << officeNumber << endl; }
};
#endif
```

The code for the `TextBook` class is shown next. As before, we want to keep the class simple. The only functions it has are a default constructor, a constructor that accepts arguments, a `set` function, and a `print` function.

Contents of `TextBook.h`

```
#ifndef TEXTBOOK
#define TEXTBOOK
#include <iostream>
#include <string>
using namespace std;

// TextBook class
class TextBook
```

```
{
private:
    string title;     // Book title
    string author;    // Author name
    string publisher; // Publisher name
public:
    // The default constructor stores empty strings
    // in the string objects.
    TextBook()
       { set("", "", ""); }

    // Constructor
    TextBook(string textTitle, string auth, string pub)
       { set(textTitle, auth, pub); }

    // set function
    void set(string textTitle, string auth, string pub)
       { title = textTitle;
         author = auth;
         publisher = pub; }

    // print function
    void print() const
       { cout << "Title: " << title << endl;
         cout << "Author: " << author << endl;
         cout << "Publisher: " << publisher << endl; }
};
#endif
```

The `Course` class is shown next. Notice that the `Course` class has an `Instructor` object and a `TextBook` object as member variables. Those objects are used as attributes of the `Course` object. Making an instance of one class an attribute of another class is called *object aggregation*. The word *aggregate* means "a whole that is made of constituent parts." In this example, the `Course` class is an aggregate class because an instance of it is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a "has a" relationship between the classes. For example, the relationships that exist among the `Course`, `Instructor`, and `TextBook` classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

The "has a" relationship is sometimes called a *whole–part relationship* because one object is part of a greater whole.

Contents of `Course.h`

```
#ifndef COURSE
#define COURSE
#include <iostream>
#include <string>
#include "Instructor.h"
#include "TextBook.h"
using namespace std;

```

```
class Course
{
private:
    string courseName;     // Course name
    Instructor instructor; // Instructor
    TextBook textbook;     // Textbook
public:
    // Constructor
    Course(string course, string instrLastName,
           string instrFirstName, string instrOffice,
           string textTitle, string author,
           string publisher)
        { // Assign the course name.
          courseName = course;

        // Assign the instructor.
        instructor.set(instrLastName, instrFirstName, instrOffice);

        // Assign the textbook.
        textbook.set(textTitle, author, publisher); }

    // print function
    void print() const
    { cout << "Course name: " << courseName << endl << endl;
      cout << "Instructor Information:\n";
      instructor.print();
      cout << "\nTextbook Information:\n";
      textbook.print();
      cout << endl;}
};
#endif
```

Program 14-15 demonstrates the `Course` class.

Program 14-15

```
// This program demonstrates the Course class.
#include "Course.h"

int main()
{
    // Create a Course object.
    Course myCourse("Intro to Computer Science", // Course name
         "Kramer", "Shawn", "RH3010",       // Instructor info
         "Starting Out with C++", "Gaddis", // Textbook title and author
         "Addison-Wesley");                 // Textbook publisher

    // Display the course info.
    myCourse.print();
    return 0;
}
```

(program output continues)

Program 14-15 *(continued)*

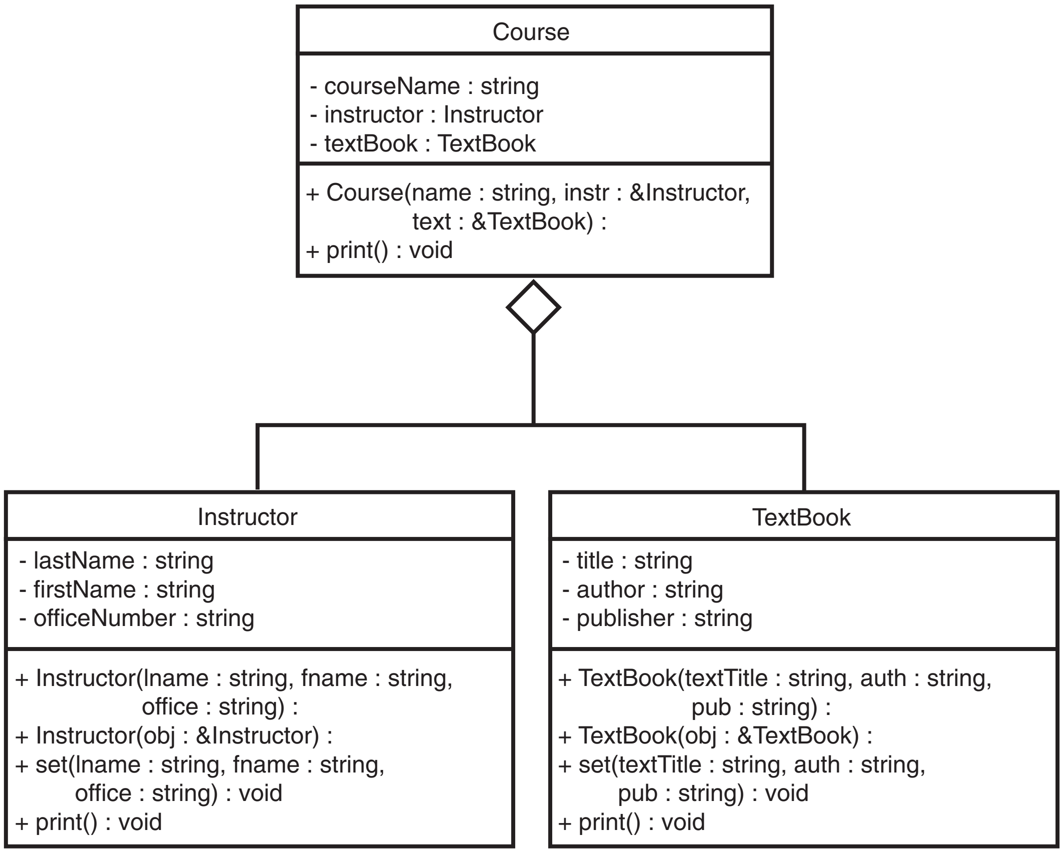
Program Output

```
Course name: Intro to Computer Science

Instructor Information:
Last name: Kramer
First name: Shawn
Office number: RH3010

Textbook Information:
Title: Starting Out with C++
Author: Gaddis
Publisher: Addison-Wesley
```

Aggregation in UML Diagrams

In Chapter 13 you were introduced to the Unified Modeling Language (UML) as a tool for designing classes. You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that is the aggregate. Figure 14-5 shows a UML diagram depicting the relationship between the `Course`, `Instructor`, and `TextBook` classes. The open diamond is closest to the `Course` class because it is the aggregate (the whole).

Figure 14-5


Course
- courseName : string
- instructor : Instructor
- textBook : TextBook
+ Course(name : string, instr : &Instructor, text : &TextBook) :
+ print() : void
Instructor
- lastName : string
- firstName : string
- officeNumber : string
+ Instructor(lname : string, fname : string, office : string) :
+ Instructor(obj : &Instructor) :
+ set(lname : string, fname : string, office : string) : void
+ print() : void
TextBook
- title : string
- author : string
- publisher : string
+ TextBook(textTitle : string, auth : string, pub : string) :
+ TextBook(obj : &TextBook) :
+ set(textTitle : string, auth : string, pub : string) : void
+ print() : void

14.8 Focus on Object-Oriented Design: Class Collaborations

CONCEPT: It is common for classes to interact, or collaborate, with one another to perform their operations. Part of the object-oriented design process is identifying the collaborations between classes.

In an object-oriented application it is common for objects of different classes to collaborate. This simply means that objects interact with each other. Sometimes one object will need the services of another object in order to fulfill its responsibilities. For example, let's say an object needs to read a number from the keyboard and then format the number to appear as a dollar amount. The object might use the services of the `cin` object to read the number from the keyboard and then use the services of another object that is designed to format the number.

If one object is to collaborate with another object, then it must know something about the other object's member functions and how to call them. Let's look at an example.

The following code shows a class named `Stock`. An object of this class holds data about a company's stock. This class has two attributes: `symbol` and `sharePrice`. The `symbol` attribute holds the trading symbol for the company's stock. This is a short series of characters that are used to identify the stock on the stock exchange. For example, the XYZ Company's stock might have the trading symbol XYZ. The `sharePrice` attribute holds the current price per share of the stock. The class also has the following member functions:

- A default constructor that initializes `symbol` to an empty string and `sharePrice` to 0.0.
- A constructor that accepts arguments for the symbol and share price.
- A copy constructor
- A `set` function that accepts arguments for the symbol and share price.
- A `getSymbol` function that returns the stock's trading symbol.
- A `getSharePrice` function that returns the current price of the stock.

Contents of `Stock.h`

```
#ifndef STOCK
#define STOCK
#include <string>
using namespace std;

class Stock
{
private:
    string symbol;      // Trading symbol of the stock
    double sharePrice;  // Current price per share
public:
    // Default constructor
    Stock()
        { set("", 0.0);}

    // Constructor
    Stock(const string sym, double price)
        { set(sym, price); }

    // Copy constructor
    Stock(const Stock &obj)
        { set(obj.symbol, obj.sharePrice); }

    // Mutator function
    void set(string sym, double price)
        { symbol = sym;
          sharePrice = price; }

    // Accessor functions
    string getSymbol() const
        { return symbol; }

    double getSharePrice() const
        { return sharePrice; }
};
#endif
```

The following code shows another class named `StockPurchase` that uses an object of the `Stock` class to simulate the purchase of a stock. The `StockPurchase` class is responsible for calculating the cost of the stock purchase. To do that, the `StockPurchase` class must know how to call the `Stock` class's `getSharePrice` function to get the price per share of the stock.

Contents of `StockPurchase.h`

```
#ifndef STOCK_PURCHASE
#define STOCK_PURCHASE
#include "Stock.h"

class StockPurchase
{
private:
    Stock stock;  // The stock that was purchased
    int shares;   // The number of shares
public:
    // The default constructor sets shares to 0. The stock
    // object is initialized by its default constructor.
    StockPurchase()
        { shares = 0;}

    // Constructor
    StockPurchase(const Stock &stockObject, int numShares)
        { stock = stockObject;
          shares = numShares; }

    // Accessor function
    double getCost() const
        { return shares * stock.getSharePrice(); }
};
#endif
```

The second constructor for the `StockPurchase` class accepts a `Stock` object representing the stock that is being purchased and an `int` representing the number of shares to purchase. In line 18 we see the first collaboration: the `StockPurchase` constructor makes a copy of the `Stock` object by using the `Stock` class's copy constructor. The next collaboration takes place in the `getCost` function. This function calculates and returns the cost of the stock purchase. In line 23 it calls the `Stock` class's `getSharePrice` function to determine the stock's price per share. Program 14-16 demonstrates this class.

Program 14-16

```
// Stock trader program
#include <iostream>
#include <iomanip>
#include "Stock.h"
#include "StockPurchase.h"
using namespace std;

int main()
{
    int sharesToBuy;  // Number of shares to buy

    // Create a Stock object for the company stock. The
    // trading symbol is XYZ and the stock is currently
    // priced at $9.62 per share.
    Stock xyzCompany("XYZ", 9.62);

    // Display the symbol and current share price.
    cout << setprecision(2) << fixed << showpoint;
    cout << "XYZ Company's trading symbol is "
         << xyzCompany.getSymbol() << endl;
    cout << "The stock is currently $"
         << xyzCompany.getSharePrice()
         << " per share.\n";

    // Get the number of shares to purchase.
    cout << "How many shares do you want to buy? ";
    cin >> sharesToBuy;

    // Create a StockPurchase object for the transaction.
    StockPurchase buy(xyzCompany, sharesToBuy);

    // Display the cost of the transaction.
    cout << "The cost of the transaction is $"
         << buy.getCost() << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
XYZ Company's trading symbol is XYZ
The stock is currently $9.62 per share.
How many shares do you want to buy? 100 [Enter]
The cost of the transaction is $962.00
```

Determining Class Collaborations with CRC Cards

During the object-oriented design process, you can determine many of the collaborations that will be necessary between classes by examining the responsibilities of the classes. In Chapter 13 we discussed the process of finding the classes and their responsibilities. Recall from that section that a class's responsibilities are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

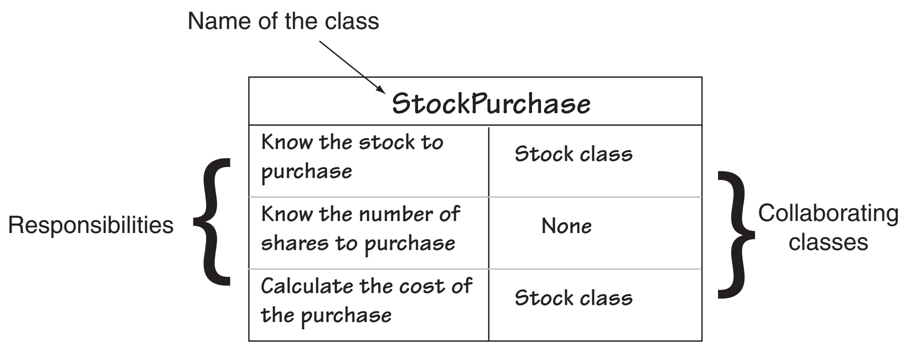
Often you will determine that the class must collaborate with another class in order to fulfill one or more of its responsibilities. One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards. *CRC* stands for class, responsibilities, and collaborations.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which is discussed in Chapter 13), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as

- Will an object of this class need to get data from another object in order to fulfill this responsibility?
- Will an object of this class need to request another object to perform an operation in order to fulfill this responsibility?

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. Figure 14-6 shows an example CRC card for the `StockPurchase` class.

Figure 14-6


Name of the class
StockPurchase
Responsibilities
Know the stock to purchase
Stock class
Know the number of shares to purchase
None
Calculate the cost of the purchase
Stock class
Collaborating classes

From the CRC card shown in the figure, we can see that the `StockPurchase` class has the following responsibilities and collaborations:

- Responsibility: To know the stock to purchase
  Collaboration: The `Stock` class
- Responsibility: To know the number of shares to purchase
  Collaboration: None
- Responsibility: To calculate the cost of the purchase
  Collaboration: The `Stock` class

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 that was 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`

```
// Specification file for the Dealer class
#ifndef DEALER_H
#define DEALER_H
#include <string>
#include "Die.h"
using namespace std;

```

```
class Dealer
{
private:
    Die die1;               // Object for die #1
    Die die2;               // Object for die #2
    int die1Value;          // Value of die #1
    int die2Value;          // Value of die #2

public:
    Dealer();               // Constructor
    void rollDice();        // To roll the dice
    string getChoOrHan();   // To get the result (Cho or Han)
    int getDie1Value();     // To get the value of die #1
    int getDie2Value();     // To get the value of die #2
};
#endif
```

Contents of `Dealer.cpp`

```
// Implementation file for the Dealer class
#include "Dealer.h"
#include "Die.h"
#include <string>
using namespace std;

//******************************************
// Constructor                             *
//******************************************
Dealer::Dealer()
{
    // Set the intial dice values to 0.
    // (We will not use these values.)
    die1Value = 0;
    die2Value = 0;
}

//******************************************
// The rollDice member function rolls the  *
// dice and saves their values.            *
//******************************************
void Dealer::rollDice()
{
    // Roll the dice.
    die1.roll();
    die2.roll();

    // Save the dice values.
    die1Value = die1.getValue();
    die2Value = die2.getValue();
}

```

```
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 #1 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 first die (stored in the `die2Value` member variable).

Now, let's look at the `Player` class.

Contents of `Player.h`

```
// Specification file for the Player class
#ifndef PLAYER_H
#define PLAYER_H
#include <string>
using namespace std;

class Player
{
private:
    string name;           // The player's name
    string guess;          // The player's guess
    int points;            // The player's points

public:
    Player(string);        // Constructor
    void makeGuess();      // Causes player to make a guess
    void addPoints(int);   // Adds points to the player
    string getName();      // Returns the player's name
    string getGuess();     // Returns the player's guess
    int getPoints();       // Returns the player's points
};
#endif
```

Contents of `Player.cpp`

```
// Implementation file for the Player class
#include "Player.h"
#include <cstdlib>
#include <ctime>
#include <string>
using namespace std;

//**********************************************
// Constructor                                 *
//**********************************************
Player::Player(string playerName)
```

```
{
    // Seed the random number generator.
    srand(time(0));

    name = playerName;
    guess = "";
    points = 0;
}

//**********************************************
// The makeGuess member function causes the    *
// player to make a guess, either "Cho (even)" *
// or "Han (odd)".                              *
//**********************************************
void Player::makeGuess()
{
    const int MIN_VALUE = 0;
    const int MAX_VALUE = 1;

    int guessNumber; // For the user's guess

    // Get a random number, either 0 or 1.
    guessNumber = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;

    // Convert the random number to Cho or Han.
    if (guessNumber == 0)
        guess = "Cho (even)";
    else
        guess = "Han (odd)";
}

//**********************************************
// The addPoints member function adds a        *
// specified number of points to the player's *
// current balance.                             *
//**********************************************
void Player::addPoints(int newPoints)
{
    points += newPoints;
}

//**********************************************
// The getName member function returns a       *
// player's name.                               *
//**********************************************
string Player::getName()
{
    return name;
}

//**********************************************
// The getGuess member function returns a      *
// player's guess.                              *
//**********************************************
```

```
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, and then displays the overall game results.

Program 14-17

```
// This program simulates the game of Cho-Han.
#include <iostream>
#include <string>
#include "Dealer.h"
#include "Player.h"
using namespace std;

```

```
// Function prototypes
void roundResults(Dealer &, Player &, Player &);
void checkGuess(Player &, Dealer &);
void displayGrandWinner(Player, Player);

int main()
{
    const int MAX_ROUNDS = 5;  // Number of rounds
    string player1Name;        // First player's name
    string player2Name;        // Second player's name

    // Get the player's names.
    cout << "Enter the first player's name: ";
    cin >> player1Name;
    cout << "Enter the second player's name: ";
    cin >> player2Name;

    // Create the dealer.
    Dealer dealer;

    // Create the two players.
    Player player1(player1Name);
    Player player2(player2Name);

    // Play the rounds.
    for (int round = 0; round < MAX_ROUNDS; round++)
    {
        cout << "----------------------------\n";
        cout << "Now playing round " << (round + 1)
             << endl;

        // Roll the dice.
        dealer.rollDice();

        // The players make their guesses.
        player1.makeGuess();
        player2.makeGuess();

        // Determine the winner of this round.
        roundResults(dealer, player1, player2);
    }

    // Display the grand winner.
    displayGrandWinner(player1, player2);
    return 0;
}

//***************************************************
// The roundResults function detremines the results *
// of the current round.                            *
//***************************************************
void roundResults(Dealer &dealer, Player &player1, Player &player2)
```

(program continues)

Program 14-17 *(continued)*

```
{
    // Show the dice values.
    cout << "The dealer rolled " << dealer.getDie1Value()
         << " and " << dealer.getDie2Value() << endl;

    // Show the result (Cho or Han).
    cout << "Result: " << dealer.getChoOrHan() << endl;

    // Check each player's guess and award points.
    checkGuess(player1, dealer);
    checkGuess(player2, dealer);
}

//****************************************************
// The checkGuess function checks a player's guess   *
// against the dealer's result.                       *
//****************************************************
void checkGuess(Player &player, Dealer &dealer)
{
    const int POINTS_TO_ADD = 1; // Points to award winner

    // Get the player's guess
    string guess = player.getGuess();

    // Get the result (Cho or Han).
    string choHanResult = dealer.getChoOrHan();

    // Display the player's guess.
    cout << "The player " << player.getName()
         << " guessed " << player.getGuess() << endl;

    // Award points if the player guessed correctly.
    if (guess == choHanResult)
    {
        player.addPoints(POINTS_TO_ADD);
        cout << "Awarding " << POINTS_TO_ADD
             << " point(s) to " << player.getName()
             << endl;
    }
}

//****************************************************
// The displayGrandWinner function displays the      *
// game's grand winner.                              *
//****************************************************
void displayGrandWinner(Player player1, Player player2)
{
    cout << "----------------------------\n";
    cout << "Game over. Here are the results:\n";

    // Display player #1's results.
    cout << player1.getName() << ": "
         << player1.getPoints() << " points\n";

    // Display player #2's results.
    cout << player2.getName() << ": "
         << player2.getPoints() << " points\n";

    // Determine the grand winner.
    if (player1.getPoints() > player2.getPoints())
    {
        cout << player1.getName()
             << " is the grand winner!\n";
    }
    else if (player2.getPoints() > player1.getPoints())
    {
        cout << player2.getName()
             << " is the grand winner!\n";
    }
    else
    {
        cout << "Both players are tied!\n";
    }
}
```

Program Output with Example Input Shown in Bold

```
Enter the first player's name: Bill [Enter]
Enter the second player's name: Jill [Enter]
----------------------------
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
----------------------------
```

(program output continues)

Program 14-17 *(continued)*

```
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 that `player1Name` is passed as an argument to the constructor.
- Line 30 creates another instance of the `Player` class to represent player #2. Notice that `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 through 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.

Review Questions and Exercises

Short Answer

1. Describe the difference between an instance member variable and a static member variable.
2. Assume that 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 that `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 Error

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 and then prints out its English description.

2. **Day of the Year**

   Assuming that 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 that 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 that 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:

`maxSickDays`	A `NumDays` object that records the maximum number of days of sick leave the employee may take.
`sickTaken`	A `NumDays` object that records the number of days of sick leave the employee has already taken.
`maxVacation`	A `NumDays` object that records the maximum number of days of paid vacation the employee may take.
`vacTaken`	A `NumDays` 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," etc.
- `monthNumber` An integer variable that holds the number of the month. For example, January would be 1, February would be 2, etc. 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, 2014`
`>>`	`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 13/45/2014 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 that 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, and 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.