

Friends, Overloaded Operators, and Arrays in Classes

11

11.1 FRIEND FUNCTIONS 620

Programming Example: An Equality Function 620

Friend Functions 624

Programming Tip: Define Both Accessor Functions and Friend Functions 626

Programming Tip: Use Both Member and Nonmember Functions 628

Programming Example: Money Class (Version 1) 628

Implementation of `digit_to_int` (Optional) 635

Pitfall: Leading Zeros in Number Constants 636

The `const` Parameter Modifier 638

Pitfall: Inconsistent Use of `const` 639

11.2 OVERLOADING OPERATORS 643

Overloading Operators 644

Constructors for Automatic Type Conversion 647

Overloading Unary Operators 649

Overloading `>>` and `<<` 650

11.3 ARRAYS AND CLASSES 660

Arrays of Classes 660

Arrays as Class Members 664

Programming Example: A Class for a Partially Filled Array 665

11.4 CLASSES AND DYNAMIC ARRAYS 667


Programming Example: A String Variable Class 668

Destructors 671

Pitfall: Pointers as Call-by-Value Parameters 674

Copy Constructors 675

Overloading the Assignment Operator 680



Give us the tools, and we'll finish the job.

WINSTON CHURCHILL, *Radio Broadcast, February 9, 1941*

INTRODUCTION

This chapter teaches you more techniques for defining functions and operators for classes, including overloading common operators such as `+`, `*`, and `/` so that they can be used with the classes you define in the same way that they are used with the predefined types such as `int` and `double`.

PREREQUISITES

This chapter uses material from Chapters 2 through 10.

11.1 FRIEND FUNCTIONS

Trust your friends.

COMMON ADVICE

Until now we have implemented class operations, such as input, output, accessor functions, and so forth, as member functions of the class, but for some operations, it is more natural to implement the operations as ordinary (nonmember) functions. In this section, we discuss techniques for defining operations on objects as nonmember functions. We begin with a simple example.

PROGRAMMING EXAMPLE

An Equality Function

In Chapter 10, we developed a class called `DayOfYear` that records a date, such as January 1 or July 4, that might be a holiday or birthday or some other annual event. We gave progressively better versions of the class. The final version was produced in Self-Test Exercise 23 of Chapter 10. In Display 11.1, we repeat this final version of the class `DayOfYear` and have enhanced the class one more time by adding a function called `equal` that can test two objects of type `DayOfYear` to see if their values represent the same date.

DISPLAY 11.1 Equality Function (part 1 of 3)

```

1  //Program to demonstrate the function equal. The class DayOfYear
2  //is the same as in Self-Test Exercises 23-24 in Chapter 10.
3  #include <iostream>
4  using namespace std;

5  class DayOfYear
6  {
7  public:
8      DayOfYear(int the_month, int the_day);
9      //Precondition: the_month and the_day form a
10     //possible date. Initializes the date according
11     //to the arguments.

12     DayOfYear( );
13     //Initializes the date to January first.

14     void input( );

15     void output( );

16     int get_month( );
17     //Returns the month, 1 for January, 2 for February, etc.

18     int get_day( );
19     //Returns the day of the month.
20 private:
21     void check_date( );
22     int month;
23     int day;
24 };

25
26 bool equal(DayOfYear date1, DayOfYear date2);
27 //Precondition: date1 and date2 have values.
28 //Returns true if date1 and date2 represent the same date;
29 //otherwise, returns false.

30
31 int main( )
32 {
33     DayOfYear today, bach_birthday(3, 21);
34
35     cout << "Enter today's date:\n";
36     today.input( );
37     cout << "Today's date is ";
38     today.output( );
39
40     cout << "J. S. Bach's birthday is ";

```

(continued)

DISPLAY 11.1 Equality Function (part 2 of 3)

```

41     bach_birthday.output( );
42
43     if (equal(today, bach_birthday))
44         cout << "Happy Birthday Johann Sebastian!\n";
45     else
46         cout << "Happy Unbirthday Johann Sebastian!\n";
47     return 0;
48 }
49
50 bool equal(DayOfYear date1, DayOfYear date2)
51 {
52     return ( date1.get_month( ) == date2.get_month( ) &&
53             date1.get_day( ) == date2.get_day( ) );
54 }
55
56 DayOfYear::DayOfYear(int the_month, int the_day)
57     : month(the_month), day(the_day)
58 {
59     check_date();
60 }
61
62 int DayOfYear::get_month( )
63 {
64     return month;
65 }
66
67 int DayOfYear::get_day( )
68 {
69     return day;
70 }
71
72 //Uses iostream:
73 void DayOfYear::input( )
74 {
75     cout << "Enter the month as a number: ";
76     cin >> month;
77     cout << "Enter the day of the month: ";
78     cin >> day;
79 }
80
81 //Uses iostream:
82 void DayOfYear::output( )
83 {
84     cout << "month = " << month
85         << ", day = " << day << endl;
86 }

```

Omitted function and constructor definitions are as in Chapter 10, Self-Test Exercises 14 and 24, but those details are not needed for what we are doing here.

(continued)

DISPLAY 11.1 Equality Function (*part 3 of 3*)**Sample Dialogue**

```
Enter today's date:  
Enter the month as a number: 3  
Enter the day of the month: 21  
Today's date is month = 3, day = 21  
J. S. Bach's birthday is month = 3, day = 21  
Happy Birthday Johann Sebastian!
```

Suppose `today` and `bach_birthday` are two objects of type `DayOfYear` that have been given values representing some dates. You can test to see if they represent the same date with the following Boolean expression:

```
equal(today, bach_birthday)
```

This call to the function `equal` returns `true` if `today` and `bach_birthday` represent the same date. In Display 11.1 this Boolean expression is used to control an *if-else* statement.

The definition of the function `equal` is straight forward. Two dates are equal if they represent the same month and the same day of the month. The definition of `equal` uses accessor functions `get_month` and `get_day` to compare the months and the days represented by the two objects.

Notice that we did not make the function `equal` a member function. It would be possible to make `equal` a member function of the class `DayOfYear`, but `equal` compares *two* objects of type `DayOfYear`. If you make `equal` a member function, you must decide whether the calling object should be the first date or the second date. Rather than arbitrarily choosing one of the two dates as the calling object, we instead treated the two dates in the same way. We made `equal` an ordinary (nonmember) function that takes two dates as its arguments.

SELF-TEST EXERCISE

1. Write a function definition for a function called `before` that takes two arguments of the type `DayOfYear`, which is defined in Display 11.1. The function returns a `bool` value and returns `true` if the first argument represents a date that comes before the date represented by the second argument; otherwise, the function returns `false`; for example, January 5 comes before February 2.

Friend Functions

If your class has a full set of accessor functions, you can use the accessor functions to define a function to test for equality or to do any other kind of computing that depends on the private member variables. However, although this may give you access to the private member variables, it may not give you efficient access to them. Look again at the definition of the function `equal` given in Display 11.1. To read the month, it must make a call to the accessor function `get_month`. To read the day, it must make a call to the accessor function `get_day`. This works, but the code would be simpler and more efficient if we could just access the member variables.

A simpler and more efficient definition of the function `equal` given in Display 11.1 would be as follows:

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return (date1.month == date2.month &&
           date1.day == date2.day);
}
```

There is just one problem with this definition: It's illegal! It's illegal because the member variables `month` and `day` are private members of the class `DayOfYear`. Private member variables (and private member functions) cannot normally be referenced in the body of a function unless the function is a member function, and `equal` is not a member function of the class `DayOfYear`. But there is a way to give a nonmember function the same access privileges as a member function. If we make the function `equal` a *friend* of the class `DayOfYear`, then the previous definition of `equal` will be legal.

Friends can access
private members

A **friend function** of a class is not a member function of the class, but a friend function has access to the private members of that class just as a member function does. A friend function can directly read the value of a member variable and can even directly change the value of a member variable, for example, with an assignment statement that has a private member variable on one side of the assignment operator. To make a function a friend function, you must name it as a friend in the class definition. For example, in Display 11.2 we have rewritten the definition of the class `DayOfYear` so that the function `equal` is a friend of the class. You make a function a friend of a class by listing the function declaration in the definition of the class and placing the keyword `friend` in front of the function declaration.

A friend is not a
member

A friend function is added to a class definition by listing its function declaration, just as you would list the declaration of a member function, except that you precede the function declaration by the keyword `friend`. However, a friend is not a member function; rather, it really is an ordinary function with extraordinary access to the data members of the class. The friend is defined and called exactly like the ordinary function it is. In particular, the function definition for `equal` shown in Display 11.2 does

DISPLAY 11.2 Equality Function as a Friend

```

1  //Demonstrates the function equal.
2  //In this version equal is a friend of the class DayOfYear.
3  #include <iostream>
4  using namespace std;
5
6  class DayOfYear
7  {
8  public:
9      friend bool equal(DayOfYear date1, DayOfYear date2);
10     //Precondition: date1 and date2 have values.
11     //Returns true if date1 and date2 represent the same date;
12     //otherwise, returns false.
13
14     DayOfYear(int the_month, int the_day);
15     //Precondition: the_month and the_day form a
16     //possible date. Initializes the date according
17     //to the arguments.
18
19     DayOfYear( );
20     //Initializes the date to January first.
21
22     void input( );
23     void output( );
24
25     int get_month( );
26     //Returns the month, 1 for January, 2 for February, etc.
27
28     int get_day( );
29     //Returns the day of the month.
30
31 private:
32     void check_date( );
33     int month;
34     int day;
35 };
36
37 int main( )
38 {

```

<The main part of the program is the same as in Display 11.1.>

```

33 }
34
35 bool equal(DayOfYear date1, DayOfYear date2)
36 {
37     return (date1.month == date2.month &&
38            date1.day == date2.day);
39 }
40

```

Note that the private
member variables
month and day can
be accessed by name.

<The rest of this display, including the Sample Dialogue, is the same as in Display 11.1.>

not include the qualifier `DayOfYear::` in the function heading. Also, the `equal` function is not called by using the dot operator. The function `equal` takes objects of type `DayOfYear` as arguments the same way that any other nonmember function would take arguments of any other type. However, a friend function definition can access the private member variables and private member functions of the class by name, so it has the same access privileges as a member function.

■ PROGRAMMING TIP Define Both Accessor Functions and Friend Functions

It may seem that if you make all your basic functions friends of a class, then there is no need to include accessor and mutator functions in the class. After all, friend functions have access to the private member variables and so do not need accessor or mutator functions. This is not entirely wrong. It is true that if you made all the functions in the world friends of a class, you would not need accessor or mutator functions. However, making all functions friends is not practical.

In order to see why you still need accessor functions, consider the example of the class `DayOfYear` given in Display 11.2. You might use this class in another program, and that other program might very well want to do something with the month part of a `DayOfYear` object. For example, the program might want to calculate how many months there are remaining in the year. Specifically, the `main` part of the program might contain the following:

```
DayOfYear today;
cout << "enter today's date: \n";
today.input();
cout << "There are " << (12 - today.get_month())
    << " months left in this year.\n";
```

You cannot replace `today.get_month()` with `today.month` because `month` is a private member of the class. You need the accessor function `get_month`.

You have just seen that you definitely need to include accessor functions in your class. Other cases require mutator functions. You may think that, because you usually need accessor and mutator functions, you do not need friends. In a sense, that is true. Notice that you could define the function `equal` either as a friend without using accessor functions (Display 11.2) or not as a friend and use accessor functions (as in Display 11.1). In most situations, the only reason to make a function a friend is to make the definition of the function simpler and more efficient; but sometimes, that is reason enough.

Friend Functions

A **friend function** of a class is an ordinary function except that it has access to the private members of objects of that class. To make a function a friend of a class, you must list the function declaration for the friend function in the class definition. The function declaration is preceded by the keyword *friend*. The function declaration may be placed in either the private section or the public section, but it will be a public function in either case, so it is clearer to list it in the public section.

SYNTAX (of a class definition with friend functions)

```
class Class_Name
{
public:
    friend Declaration_for_Friend_Function_1
    friend Declaration_for_Friend_Function_2
        .
        .
        .
    Member_Function_Declarations
private:
    Private_Member_Declarations
};
```

You need not list the friend functions first. You can intermix the order of these function declarations.

EXAMPLE

```
class FuelTank
{
public:
    friend double need_to_fill(FuelTank tank);
    //Precondition: Member variables of tank have values.
    //Returns the number of liters needed to fill tank.
    FuelTank(double the_capacity, double the_level);
    FuelTank();
    void input();
    void output();
private:
    double capacity;//in liters
    double level;
};
```

A friend function is *not* a member function. A friend function is defined and called the same way as an ordinary function. You do not use the dot operator in a call to a friend function and you do not use a type qualifier in the definition of a friend function.

■ PROGRAMMING TIP Use Both Member and Nonmember Functions

Member functions and friend functions serve a very similar role. In fact, sometimes it is not clear whether you should make a particular function a friend of your class or a member function of the class. In most cases, you can make a function either a member function or a friend and have it perform the same task in the same way. There are, however, places where it is better to use a member function and places where it is better to use a friend function (or even a plain old function that isn't a friend, like the version of `equal` in Display 11.1). A simple rule to help you decide between member functions and nonmember functions is the following:

- Use a member function if the task being performed by the function involves only one object.
- Use a nonmember function if the task being performed involves more than one object. For example, the function `equal` in Display 11.1 (and Display 11.2) involves two objects, so we made it a nonmember (friend) function.

Whether you make a nonmember function a friend function or use accessor and mutator functions is a matter of efficiency and personal taste. As long as you have enough accessor and mutator functions, either approach will work.

The choice of whether to use a member or nonmember function is not always as simple as the above two rules. With more experience, you will discover situations in which it pays to violate those rules. A more accurate but harder to understand rule is to use member functions if the task is intimately related to a single object; use a nonmember function when the task involves more than one object and the objects are used symmetrically. However, this more accurate rule is not clear-cut, and the two simple rules given above will serve as a reliable guide until you become more sophisticated in handling objects. ■

PROGRAMMING EXAMPLE

Money Class (Version 1)

Display 11.3 contains the definition of a class called `Money`, which represents amounts of U.S. currency. The value is implemented as a single integer value that represents the amount of money as if it were converted to all pennies. For example, \$9.95 would be stored as the value 995. Since we use an integer to represent the amount of money, the amount is represented as an exact quantity. We did not use a value of type `double` because values of type `double` are stored as approximate values and we want our money amounts to be exact quantities.

This integer for the amount of money (expressed as all cents) is stored in a member variable named `all_cents`. We could use `int` for the type of the

member variable `all_cents`, but with some compilers that would severely limit the amounts of money we could represent. In some implementations of C++, only 2 bytes are used to store the `int` type.¹ The result of the 2-byte implementation is that the largest value of type `int` is only slightly larger than 32000, but 32000 cents represents only \$320, which is a fairly small amount of money. Since we may want to deal with amounts of money much larger than \$320, we have used `long` for the type of the member variable `all_cents`. C++ compilers that implement the `int` type in 2 bytes usually implement the type `long` in 4 bytes. Values of type `long` are integers just like the values of the type `int`, except that the 4-byte `long` implementation enables the largest allowable value of type `long` to be much larger than the largest allowable value of type `int`. On most systems the largest allowable value of type `long` is 2 billion or larger. (The type `long` is also called `long int`. The two names `long` and `long int` refer to the same type.)

The class `Money` has two operations that are friend functions: `equal` and `add` (which are defined in Display 11.3). The function `add` returns a `Money` object whose value is the sum of the values of its two arguments. A function call of the form `equal(amount1, amount2)` returns `true` if the two objects `amount1` and `amount2` have values that represent equal amounts of money.

Notice that the class `Money` reads and writes amounts of money as we normally write amounts of money, such as \$9.95 or -\$9.95. First, consider the member function `input` (also defined in Display 11.3). That function first reads a single character, which should be either the dollar sign ('\$') or the minus sign ('-'). If this first character is the minus sign, then the function remembers that the amount is negative by setting the value of the variable `negative` to `true`. It then reads an additional character, which should be the dollar sign. On the other hand, if the first symbol is not '-', then `negative` is set equal to `false`. At this point the negative sign (if any) and the dollar sign have been read. The function `input` then reads the number of dollars as a value of type `long` and places the number of dollars in the local variable named `dollars`. After reading the dollars part of the input, the function `input` reads the remainder of the input as values of type `char`; it reads in three characters, which should be a decimal point and two digits.

(You might be tempted to define the member function `input` so that it reads the decimal point as a value of type `char` and then reads the number of cents as a value of type `int`. This is not done because of the way that some C++ compilers treat leading zeros. As explained in the Pitfall section entitled "Leading Zeros in Number Constants," many compilers still in use do not read numbers with leading zeros as you would like them to, so an amount like \$7.09 may be read incorrectly if your C++ code were to read the 09 as a value of type `int`.)

¹ See Chapter 2 for details. Display 2.2 has a description of data types as most recent compilers implement them.

DISPLAY 11.3 Money Class—Version 1 (part 1 of 4)

```

1  //Program to demonstrate the class Money.
2  #include <iostream>
3  #include <cstdlib>
4  #include <cctype>
5  using namespace std;

6  //Class for amounts of money in U.S. currency.
7  class Money
8  {
9  public:
10     friend Money add(Money amount1, Money amount2);
11     //Precondition: amount1 and amount2 have been given values.
12     //Returns the sum of the values of amount1 and amount2.

13     friend bool equal(Money amount1, Money amount2);
14     //Precondition: amount1 and amount2 have been given values.
15     //Returns true if the amount1 and amount2 have the same value;
16     //otherwise, returns false.

17     Money(long dollars, int cents);
18     //Initializes the object so its value represents an amount with the
19     //dollars and cents given by the arguments. If the amount is negative,
20     //then both dollars and cents must be negative.

21     Money(long dollars);
22     //Initializes the object so its value represents $dollars.00.

23     Money( );
24     //Initializes the object so its value represents $0.00.

25     double get_value( );
26     //Precondition: The calling object has been given a value.
27     //Returns the amount of money recorded in the data of the calling object.

28     void input(istream& ins);
29     //Precondition: If ins is a file input stream, then ins has already been
30     //connected to a file. An amount of money, including a dollar sign, has been
31     //entered in the input stream ins. Notation for negative amounts is -$100.00.
32     //Postcondition: The value of the calling object has been set to
33     //the amount of money read from the input stream ins.

34     void output(ostream& outs);
35     //Precondition: If outs is a file output stream, then outs has already been
36     //connected to a file.
37     //Postcondition: A dollar sign and the amount of money recorded
38     //in the calling object have been sent to the output stream outs.
39 private:
40     long all_cents;
41 };

```

(continued)

DISPLAY 11.3 Money Class—Version 1 (part 2 of 4)

```

42  int digit_to_int(char c);
43  //Function declaration for function used in the definition of Money::input:
44  //Precondition: c is one of the digits '0' through '9'.
45  //Returns the integer for the digit; for example, digit_to_int ('3') returns 3.

46  int main( )
47  {
48      Money your_amount, my_amount(10, 9), our_amount;
49      cout << "Enter an amount of money: ";
50      your_amount.input(cin);
51      cout << "Your amount is ";
52      your_amount.output(cout);
53      cout << endl;
54      cout << "My amount is ";
55      my_amount.output(cout);
56      cout << endl;

57      if (equal(your_amount, my_amount))
58          cout << "We have the same amounts.\n";
59      else
60          cout << "One of us is richer.\n";
61      our_amount = add(your_amount, my_amount);
62      your_amount.output(cout);
63      cout << " + ";
64      my_amount.output(cout);
65      cout << " equals ";
66      our_amount.output(cout);
67      cout << endl;
68      return 0;
69  }

70  Money add(Money amount1, Money amount2)
71  {
72      Money temp;
73
74      temp.all_cents = amount1.all_cents + amount2.all_cents;
75      return temp;
76  }

77
78  bool equal(Money amount1, Money amount2)
79  {
80      return (amount1.all_cents == amount2.all_cents);
81  }
82
83  Money::Money(long dollars, int cents)
84  {
85      if (dollars * cents < 0) //If one is negative and one is positive

```

(continued)

DISPLAY 11.3 Money Class—Version 1 (part 3 of 4)

```

86     {
87         cout << "Illegal values for dollars and cents.\n";
88         exit(1);
89     }
90     all_cents = dollars * 100 + cents;
91 }
92
93 Money::Money(long dollars) : all_cents(dollars * 100)
94 {
95     //Body intentionally blank.
96 }
97
98 Money::Money( ) : all_cents(0)
99 {
100     //Body intentionally blank.
101 }
102
103 double Money::get_value( )
104 {
105     return (all_cents * 0.01);
106 }
107 //Uses iostream, ctype, cstdlib:
108 void Money::input(istream& ins)
109 {
110     char one_char, decimal_point, digit1, digit2;
111     //digits for the amount of cents
112     long dollars;
113     int cents;
114     bool negative;//set to true if input is negative.
115
116     ins >> one_char;
117     if (one_char == ' ')
118     {
119         negative = true;
120         ins >> one_char; //read '$'
121     }
122     else
123         negative = false;
124     //if input is legal, then one_char == '$'
125
126     ins >> dollars >> decimal_point >> digit1 >> digit2;
127
128     if (one_char != '$' || decimal_point != '.'
129         || !isdigit(digit1) || !isdigit(digit2))

```

(continued)

DISPLAY 11.3 Money Class—Version 1 (part 4 of 4)

```

130     {
131         cout << "Error illegal form for money input\n";
132         exit(1);
133     }
134     cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);
135
136     all_cents = dollars * 100 + cents;
137     if (negative)
138         all_cents = -all_cents;
139 }
140
141 //Uses cstdlib and iostream:
142 void Money::output(ostream& outs)
143 {
144     long positive_cents, dollars, cents;
145     positive_cents = labs(all_cents);
146     dollars = positive_cents / 100;
147     cents = positive_cents % 100;
148
149     if (all_cents < 0)
150         outs << "-$" << dollars << '.';
151     else
152         outs << "$" << dollars << '.';
153
154     if (cents < 10)
155         outs << '0';
156     outs << cents;
157 }
158
159 int digit_to_int(char c)
160 {
161     return (static_cast<int>(c) - static_cast<int>('0'));
162 }
163

```

Sample Dialogue

```

Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09
One of us is richer.
$123.45 + $10.09 equals $133.54

```

The following assignment statement converts the two digits that make up the cents part of the input amount to a single integer, which is stored in the local variable `cents`:

```
cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);
```

After this assignment statement is executed, the value of `cents` is the number of cents in the input amount.

The helping function `digit_to_int` takes an argument that is a digit, such as '3', and converts it to the corresponding *int* value, such as 3. We need this helping function because the member function `input` reads the two digits for the number of cents as two values of type *char*, which are stored in the local variables `digit1` and `digit2`. However, once the digits are read into the computer, we want to use them as numbers. Therefore, we use the function `digit_to_int` to convert a digit such as '3' to a number such as 3. The definition of the function `digit_to_int` is given in Display 11.3. You can simply take it on faith that this definition does what it is supposed to do and treat the function as a black box. All you need to know is that `digit_to_int('0')` returns 0, `digit_to_int('1')` returns 1, and so forth. However, it is not too difficult to see how this function works, so you may want to read the optional section that follows this one. It explains the implementation of `digit_to_int`.

Once the local variables `dollars` and `cents` are set to the number of dollars and the number of cents in the input amount, it is easy to set the member variable `all_cents`. The following assignment statement sets `all_cents` to the correct number of cents:

```
all_cents = dollars * 100 + cents;
```

However, this always sets `all_cents` to a positive amount. If the amount of money is negative, then the value of `all_cents` must be changed from positive to negative. This is done with the following statement:

```
if (negative)
    all_cents = -all_cents;
```

The member function `output` (Display 11.3) calculates the number of dollars and the number of cents from the value of the member variable `all_cents`. It computes the number of dollars and the number of cents using integer division by 100. For example, if `all_cents` has a value of 995 (cents), then the number of dollars is 995/100, which is 9, and the number of cents is 995%100, which is 95. Thus, \$9.95 would be the value output when the value of `all_cents` is 995 (cents).

The definition for the member function `output` needs to make special provisions for outputting negative amounts of money. The result of integer division with negative numbers does not have a standard definition and can vary from one implementation to another. To avoid this problem, we have taken the absolute value of the number in `all_cents` before performing

division. To compute the absolute value we use the predefined function `labs`. The function `labs` returns the absolute value of its argument, just like the function `abs`, but `labs` takes an argument of type *long* and returns a value of type *long*. The function `labs` is in the library with header file `cstdlib`, just like the function `abs`. (Some versions of C++ do not include `labs`. If your implementation of C++ does not include `labs`, you can easily define the function for yourself.)

Implementation of `digit_to_int` (Optional)

The definition of the function `digit_to_int` from Display 11.3 is reproduced here:

```
int digit_to_int(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0'));
}
```

At first glance, the formula for the value returned may seem a bit strange, but the details are not too complicated. The digit to be converted—for example, '3'—is the parameter `c`, and the returned value will turn out to be the corresponding *int* value—in this example, 3. As we pointed out in Chapters 2 and 6, values of type *char* are implemented as numbers. Unfortunately, the number implementing the digit '3', for example, is not the number 3. The type cast `static_cast<int>(c)` produces the number that implements the character `c` and converts this number to the type *int*. This changes `c` from the type *char* to a number of type *int* but, unfortunately, not to the number we want. For example, `static_cast<int>('3')` is not 3, but is some other number. We need to convert `static_cast<int>(c)` to the number corresponding to `c` (for example, '3' to 3). So let's see how we must adjust `static_cast<int>(c)` to get the number we want.

We know that the digits are in order. So `static_cast<int>('0') + 1` is equal to `static_cast<int>('1')`; `static_cast<int>('1') + 1` is equal to `static_cast<int>('2')`; `static_cast<int>('2') + 1` is equal to `static_cast<int>('3')`, and so forth. Knowing that the digits are in this order is all we need to know in order to see that `digit_to_int` returns the correct value. If `c` is '0', the value returned is

$$\text{static_cast<int>(c) - static_cast<int>('0')}$$

which is

$$\text{static_cast<int>('0') - static_cast<int>('0')}$$

So `digit_to_int('0')` returns 0.

Now let's consider what happens when `c` has the value '1'. The value returned is then `static_cast<int>(c) - static_cast<int>('0')`, which is `static_cast<int>('1') - static_cast<int>('0')`. That equals `(static_`

`cast<int>('0') + 1) - static_cast<int>('0')`, and that, in turn, equals `static_cast<int>('0') - static_cast<int>('0') + 1`. Since `static_cast<int>('0') - static_cast<int>('0')` is 0, this result is `0 + 1`, or 1. You can check the other digits, '2' through '9', for yourself; each digit produces a number that is 1 larger than the previous digit.

PITFALL Leading Zeros in Number Constants

The following are the object declarations given in the main part of the program in Display 11.3:

```
Money your_amount, my_amount(10, 9), our_amount;
```

The two arguments in `my_amount(10,9)` represent \$10.09. Since we normally write cents in the format “.09,” you might be tempted to write the object declaration as `my_amount(10,09)`. However, this will cause problems. In mathematics, the numerals 9 and 09 represent the same number. However, some C++ compilers use a leading zero to signal a different kind of numeral, so in C++ the constants 9 and 09 are not necessarily the same number. With some compilers, a leading zero means that the number is written in base 8 rather than base 10. Since base 8 numerals do not use the digit 9, the constant 09 does not make sense in C++. The constants 00 through 07 should work correctly, since they mean the same thing in base 8 and in base 10, but some systems in some contexts will have trouble even with 00 through 07.

The ANSI C++ standard provides that input should default to being interpreted as decimal, regardless of the leading 0. The GNU project C++ compiler, g++, and Microsoft’s VC++ compiler do comply with the standard, and so they do not have a problem with leading zeros. Most compiler vendors track the ANSI standard and thus should be compliant with the ANSI C++ standard, and so this problem with leading zeros should eventually go away. You should write a small program to test this on your compiler. ■

SELF-TEST EXERCISES

2. What is the difference between a friend function for a class and a member function for the class?
3. Suppose you wish to add a friend function to the class `DayOfYear` defined in Display 11.2. This friend function will be named after and will take two arguments of the type `DayOfYear`. The function returns *true* if the first argument represents a date that comes after the date represented by the second argument; otherwise, the function returns *false*. For example, February 2 comes after January 5. What do you need to add to the definition of the class `DayOfYear` in Display 11.2?

4. Suppose you wish to add a friend function for subtraction to the class `Money` defined in Display 11.3. What do you need to add to the description of the class `Money` that we gave in Display 11.3? The subtraction function should take two arguments of type `Money` and return a value of type `Money` whose value is the value of the first argument minus the value of the second argument.
5. Notice the member function `output` in the class definition of `Money` given in Display 11.3. In order to write a value of type `Money` to the screen, you call `output` with `cout` as an argument. For example, if `purse` is an object of type `Money`, then to output the amount of money in `purse` to the screen, you write the following in your program:

```
purse.output(cout);
```

It might be nicer not to have to list the stream `cout` when you send output to the screen.

Rewrite the class definition for the type `Money` given in Display 11.3. The only change is that this rewritten version overloads the function name `output` so that there are two versions of `output`. One version is just like the one shown in Display 11.3; the other version of `output` takes no arguments and sends its output to the screen. With this rewritten version of the type `Money`, the following two calls are equivalent:

```
purse.output(cout);
```

and

```
purse.output();
```

but the second is simpler. Note that since there will be two versions of the function `output`, you can still send output to a file. If `outs` is an output file stream that is connected to a file, then the following will output the money in the object `purse` to the file connected to `outs`:

```
purse.output(outs);
```

6. Notice the definition of the member function `input` of the class `Money` given in Display 11.3. If the user enters certain kinds of incorrect input, the function issues an error message and ends the program. For example, if the user omits a dollar sign, the function issues an error message. However, the checks given there do not catch all kinds of incorrect input. For example, negative amounts of money are supposed to be entered in the form `-$9.95`, but if the user mistakenly enters the amount in the form `$-9.95`, then the `input` will not issue an error message and the value of the `Money` object will be set to an incorrect value. What amount will the member function `input` read if the user mistakenly enters `$-9.95`? How might you add additional checks to catch most errors caused by such a misplaced minus sign?

7. The Pitfall section entitled “Leading Zeros in Number Constants” suggests that you write a short program to test whether a leading 0 will cause your compiler to interpret input numbers as base-8 numerals. Write such a program.

The *const* Parameter Modifier

A call-by-reference parameter is more efficient than a call-by-value parameter. A call-by-value parameter is a local variable that is initialized to the value of its argument, so when the function is called there are two copies of the argument. With a call-by-reference parameter, the parameter is just a placeholder that is replaced by the argument, so there is only one copy of the argument. For parameters of simple types, such as *int* or *double*, the difference in efficiency is negligible, but for class parameters the difference in efficiency can sometimes be important. Thus, it can make sense to use a call-by-reference parameter rather than a call-by-value parameter for a class, even if the function does not change the parameter.

If you are using a call-by-reference parameter and your function does not change the value of the parameter, you can mark the parameter so that the compiler knows that the parameter should not be changed. To do so, place the modifier *const* before the parameter type. The parameter is then called a **constant parameter**. For example, consider the class *Money* defined in Display 11.3. The *Money* parameters for the friend function *add* can be made into constant parameters as follows:

```
class Money
{
    public:
        friend Money add(const Money& amount1, const Money& amount2);
        //Precondition: amount1 and amount2 have been given values.
        //Returns the sum of the values of amount1 and amount2.
        ...
}
```

When you use constant parameters, the modifier *const* must be used in both the function declaration and in the heading of the function definition, so with the change in the class definition above, the function definition for *add* would begin as follows:

```
Money add(const Money& amount1, const Money& amount2)
{
    ...
}
```

The remainder of the function definition would be the same as in Display 11.3.

Constant parameters are a form of automatic error checking. If your function definition contains a mistake that causes an inadvertent change to the constant parameter, then the computer will issue an error message.

The parameter modifier *const* can be used with any kind of parameter; however, it is normally used only for call-by-reference parameters for classes (and occasionally for certain other parameters whose corresponding arguments are large).

Call-by-reference parameters are replaced with arguments when a function is called, and the function call may (or may not) change the value of the argument. When you have a call to a member function, the calling object behaves very much like a call-by-reference parameter. When you have a call to a member function, that function call can change the value of the calling object. For example, consider the following, where the class `Money` is as in Display 11.3:

*const with
member functions*

```
Money m;
m.input(cin);
```

When the object `m` is declared, the value of the member variable `all_cents` is initialized to 0. The call to the member function `input` changes the value of the member variable `all_cents` to a new value determined by what the user types in. Thus, the call `m.input(cin)` changes the value of `m`, just as if `m` were a call-by-reference argument.

The modifier `const` applies to calling objects in the same way that it applies to parameters. If you have a member function that should not change the value of a calling object, you can mark the function with the `const` modifier; the computer will then issue an error message if your function code inadvertently changes the value of the calling object. In the case of a member function, the `const` goes at the end of the function declaration, just before the final semicolon, as shown here:

```
class Money
{
public:
    ...
    void output(ostream& outs) const;
    ...
}
```

The modifier `const` should be used in both the function declaration and the function definition, so the function definition for `output` would begin as follows:

```
void Money::output(ostream& outs) const
{
    ...
}
```

The remainder of the function definition would be the same as in Display 11.3.

PITFALL Inconsistent Use of `const`

Use of the `const` modifier is an all-or-nothing proposition. If you use `const` for one parameter of a particular type, then you should use it for every other parameter that has that type and that is not changed by the function call; moreover, if the type is a class type, then you should also use the `const` modifier for every member function that does not change the value of its calling object. The reason has to do with function calls within function calls. For example, consider the following definition of the function `guarantee`:



```
void guarantee(const Money& price)
{
    cout << "If not satisfied, we will pay you\n"
          << "double your money back.\n"
          << "That's a refund of $"
          << (2 * price.get_value()) << endl;
}
```

If you do *not* add the *const* modifier to the function declaration for the member function `get_value`, then the function `guarantee` will give an error message on most compilers. The member function `get_value` does not change the calling object `price`. However, when the compiler processes the function definition for `guarantee`, it will think that `get_value` does (or at least might) change the value of `price`. This is because when it is translating the function definition for `guarantee`, all that the compiler knows about the member function `get_value` is the function declaration for `get_value`; if the function declaration does not contain a *const*, which tells the compiler that the calling object will not be changed, then the compiler assumes that the calling object will be changed. Thus, if you use the modifier *const* with parameters of type `Money`, then you should also use *const* with all `Money` member functions that do not change the value of their calling object. In particular, the function declaration for the member function `get_value` should include a *const*.

In Display 11.4 we have rewritten the definition of the class `Money` given in Display 11.3, but this time we have used the *const* modifier where appropriate. The definitions of the member and friend functions would be the same as they are in Display 11.3, except that the modifier *const* must be used in function headings so that the headings match the function declarations shown in Display 11.4. ■

***const* Parameter Modifier**

If you place the modifier *const* before the type for a call-by-reference parameter, the parameter is called a **constant parameter**. (The heading of the function definition should also have a *const* so that it matches the function declaration.) When you add the *const*, you are telling the compiler that this parameter should not be changed. If you make a mistake in your definition of the function so that it does change the constant parameter, then the computer will give an error message. Parameters of a class type that are not changed by the function ordinarily should be constant call-by-reference parameters, rather than call-by-value parameters.

(continued)

If a member function does not change the value of its calling object, then you can mark the function by adding the *const* modifier to the function declaration. If you make a mistake in your definition of the function so that it does change the calling object and the function is marked with *const*, then the computer will give an error message. The *const* is placed at the end of the function declaration, just before the final semicolon. The heading of the function definition should also have a *const* so that it matches the function declaration.

EXAMPLE

```
class Sample
{
public:
    Sample();
    friend int compare(const Sample& s1, const Sample& s2);
    void input();
    void output() const;
private:
    int stuff;
    double more_stuff;
};
```

Use of the *const* modifier is an all-or-nothing proposition. You should use the *const* modifier whenever it is appropriate for a class parameter and whenever it is appropriate for a member function of the class. If you do not use *const* every time that it is appropriate for a class, then you should never use it for that class.

DISPLAY 11.4 The Class Money with Constant Parameters (part 1 of 2)

```
1 //Class for amounts of money in U.S. currency.
2 class Money
3 {
4     public:
5         friend Money add(const Money& amount1, const Money& amount2);
6         //Precondition: amount1 and amount2 have been given values.
7         //Returns the sum of the values of amount1 and amount2.
8
9         friend bool equal(const Money& amount1, const Money& amount2);
10        //Precondition: amount1 and amount2 have been given values.
11        //Returns true if amount1 and amount2 have the same value;
12        //otherwise, returns false.
13
14        Money(long dollars, int cents);
```

(continued)

DISPLAY 11.4 The Class Money with Constant Parameters (part 2 of 2)

```

13      //Initializes the object so its value represents an amount with the
14      //dollars and cents given by the arguments. If the amount is negative,
15      //then both dollars and cents must be negative.
16      Money(long dollars);
17      //Initializes the object so its value represents $dollars.00.
18      Money( );
19      //Initializes the object so its value represents $0.00.
20      double get_value( ) const;
21      //Precondition: The calling object has been given a value.
22      //Returns the amount of money recorded in the data of the calling object.
23      void input(istream& ins);
24      //Precondition: If ins is a file input stream, then ins has already been
25      //connected to a file. An amount of money, including a dollar sign, has been
26      //entered in the input stream ins. Notation for negative amounts is -$100.00.
27      //Postcondition: The value of the calling object has been set to
28      //the amount of money read from the input stream ins.
29      void output(ostream& outs) const;
30      //Precondition: If outs is a file output stream, then outs has already been
31      //connected to a file.
32      //Postcondition: A dollar sign and the amount of money recorded
33      //in the calling object have been sent to the output stream outs.
34  private:
35      long all_cents;
36  };

```

SELF-TEST EXERCISES

8. Give the complete definition of the member function `get_value` that you would use with the definition of `Money` given in Display 11.4.
9. Why would it be incorrect to add the modifier `const`, as shown here, to the function declaration for the member function `input` of the class `Money` given in Display 11.4?

```

class Money
{
    ...
public:
    void input(istream& ins) const;
    ...

```

10. What are the differences and the similarities between a call-by-value parameter and a call-by-`const`-reference parameter? Function declarations that illustrate these are


```
void call_by_value(int x);
void call_by_const_reference(const int& x);
```

11. Given the following definitions:

```
const int x = 17;
class A
{
public:
    A( );
    A(int x);
    int f( ) const;
    int g(const A& x);
private:
    int i;
};
```

Each of the three *const* keywords is a promise to the compiler that the compiler will enforce. What is the promise in each case?

11.2 OVERLOADING OPERATORS

He's a smooth operator.

LINE FROM A SONG BY SADE (WRITTEN BY SADE ADU AND RAY ST. JOHN)

Earlier in this chapter, we showed you how to make the function `add` a friend of the class `Money` and use it to add two objects of type `Money` (Display 11.3). The function `add` is adequate for adding objects, but it would be nicer if you could simply use the usual `+` operator to add values of type `Money`, as in the last line of the following code:

```
Money total, cost, tax;
cout << "Enter cost and tax: ";
cost.input(cin);
tax.input(cin);
total = cost + tax;
```

instead of having to use the slightly more awkward

```
total = add(cost, tax);
```

Recall that an operator, such as `+`, is really just a function except that the syntax for how it is used is slightly different from that of an ordinary function. In an ordinary function call, the arguments are placed in parentheses after the function name, as in the following:

```
add(cost, tax)
```

With a (binary) operator, the arguments are placed on either side of the operator, as shown here:

```
cost + tax
```

A function can be overloaded to take arguments of different types. An operator is really a function, so an operator can be overloaded. The way you overload an operator, such as `+`, is basically the same as the way you overload a function name. In this section we show you how to overload operators in C++.

Overloading Operators

You can overload the operator `+` (and many other operators) so that it will accept arguments of a class type. The difference between overloading the `+` operator and defining the function `add` (given in Display 11.3) involves only a slight change in syntax. The definition of the overloaded operator `+` is basically the same as the definition of the function `add`. The only differences are that you use the name `+` instead of the name `add` and you precede the `+` with the keyword *operator*. In Display 11.5 we have rewritten the type `Money` to include the overloaded operator `+` and we have embedded the definition in a small demonstration program.

The class `Money`, as defined in Display 11.5, also overloads the `==` operator so that `==` can be used to compare two objects of type `Money`. If `amount1` and `amount2` are two objects of type `Money`, we want the expression

```
amount1 == amount2
```

to return the same value as the following Boolean expression:

```
amount1.all_cents == amount2.all_cents
```

As shown in Display 11.5, this is the value returned by the overloaded operator `==`.

You can overload most, but not all, operators. The operator need not be a friend of a class, but you will often want it to be a friend. Check the box entitled “Rules on Overloading Operators” for some technical details on when and how you can overload an operator.

Operator Overloading

A (binary) operator, such as `+`, `-`, `/`, `%`, and so forth, is simply a function that is called using a different syntax for listing its arguments. With an operator, the arguments are listed before and after the operator; with a function, the arguments are listed in parentheses after the function name. An operator definition is written similarly to a function definition, except that the operator definition includes the reserved word *operator* before the operator name. The predefined operators, such as `+` and so forth, can be overloaded by giving them a new definition for a class type.

An operator may be a friend of a class although this is not required. An example of overloading the `+` operator as a friend is given in Display 11.5.

DISPLAY 11.5 Overloading Operators (part 1 of 2)

```

1  //Program to demonstrate the class Money. (This is an improved version of
2  //the class Money that we gave in Display 11.3 and rewrote in Display 11.4.)
3  #include <iostream>
4  #include <cstdlib>
5  #include <cctype>
6  using namespace std;
7
8  //Class for amounts of money in U.S. currency.
9  class Money
10 {
11 public:
12     friend Money operator +(const Money& amount1, const Money& amount2);
13     //Precondition: amount1 and amount2 have been given values.
14     //Returns the sum of the values of amount1 and amount2.
15
16     friend bool operator ==(const Money& amount1, const Money& amount2);
17     //Precondition: amount1 and amount2 have been given values.
18     //Returns true if amount1 and amount2 have the same value;
19     //otherwise, returns false.
20
21     Money(long dollars, int cents);
22     Money(long dollars);
23     Money( );
24     double get_value( ) const;
25     void input(istream& ins);
26     void output(ostream& outs) const;
27 private:
28     long all_cents;
29 };

```

Some comments from Display 11.4 have been omitted to save space in this book, but they should be included in a real program.

<Any extra function declarations from Display 11.3 go here.>

```

28 int main( )
29 {
30     Money cost(1, 50), tax(0, 15), total;
31     total = cost + tax;
32     cout << "cost = ";
33     cost.output(cout);
34     cout << endl;
35     cout << "tax = ";
36     tax.output(cout);
37     cout << endl;
38     cout << "total bill = ";
39     total.output(cout);
40     cout << endl;

```

(continued)

DISPLAY 11.5 Overloading Operators (*part 2 of 2*)

```
41     if (cost == tax)
42         cout << "Move to another state.\n";
43     else
44         cout << "Things seem normal.\n";
45     return 0;
46 }
47
48 Money operator +(const Money& amount1, const Money& amount2)
49 {
50     Money temp;
51     temp.all_cents = amount1.all_cents + amount2.all_cents;
52     return temp;
53 }
54
55 bool operator ==(const Money& amount1, const Money& amount2)
56 {
57     return (amount1.all_cents == amount2.all_cents);
58 }
59
```

<The definitions of the member functions are the same as in Display 11.3 except that *const* is added to the function headings in various places so that the function headings match the function declarations in the preceding class definition. No other changes are needed in the member function definitions. The bodies of the member function definitions are identical to those in Display 11.3>

Output

```
cost = $1.50
tax = $0.15
total bill = $1.65
Things seem normal.
```

SELF-TEST EXERCISES

12. What is the difference between a (binary) operator and a function?
13. Suppose you wish to overload the operator < so that it applies to the type *Money* defined in Display 11.5. What do you need to add to the description of *Money* given in Display 11.5?

14. Suppose you wish to overload the operator `<=` so that it applies to the type `Money` defined in Display 11.5. What do you need to add to the description of `Money` given in Display 11.5?
15. Is it possible using operator overloading to change the behavior of `+` on integers? Why or why not?

Rules on Overloading Operators

- When overloading an operator, at least one argument of the resulting overloaded operator must be of a class type.
- An overloaded operator can be, but does not have to be, a friend of a class; the operator function may be a member of the class or an ordinary (non-friend) function. (Overloading an operator as a class member is discussed in Appendix 8.)
- You cannot create a new operator. All you can do is overload existing operators, such as `+`, `-`, `*`, `/`, `%`, and so forth.
- You cannot change the number of arguments that an operator takes. For example, you cannot change `%` from a binary to a unary operator when you overload `%`; you cannot change `++` from a unary to a binary operator when you overload it.
- You cannot change the precedence of an operator. An overloaded operator has the same precedence as the ordinary version of the operator. For example, `x*y+z` always means `(x*y)+z`, even if `x`, `y`, and `z` are objects and the operators `+` and `*` have been overloaded for the appropriate classes.
- The following operators cannot be overloaded: the dot operator (`.`), the scope resolution operator (`::`), and the operators `.*` and `?:`, which are not discussed in this book.
- Although the assignment operator `=` can be overloaded so that the default meaning of `=` is replaced by a new meaning, this must be done in a different way from what is described here. Overloading `=` is discussed in the section “Overloading the Assignment Operator” later in this chapter. Some other operators, including `[]` and `->`, also must be overloaded in a way that is different from what is described in this chapter. The operators `[]` and `->` are discussed later in this book.

Constructors for Automatic Type Conversion

If your class definition contains the appropriate constructors, the system will perform certain type conversions automatically. For example, if your program contains the definition of the class `Money` given in Display 11.5, you could use the following in your program:

```

Money base_amount(100, 60), full_amount;
full_amount = base_amount + 25;
full_amount.output(cout);

```

The output will be

\$125.60

The code above may look simple and natural enough, but there is one subtle point. The 25 (in the expression `base_amount + 25`) is not of the appropriate type. In Display 11.5 we only overloaded the operator `+` so that it could be used with two values of type `Money`. We did not overload `+` so that it could be used with a value of type `Money` and an integer. The constant 25 is an integer and is not of type `Money`. The constant 25 can be considered to be of type `int` or of type `long`, but 25 cannot be used as a value of type `Money` unless the class definition somehow tells the system how to convert an integer to a value of type `Money`. The only way that the system knows that 25 means \$25.00 is that we included a constructor that takes a single argument of type `long`. When the system sees the expression

```
base_amount + 25
```

it first checks to see if the operator `+` has been overloaded for the combination of a value of type `Money` and an integer. Since there is no such overloading, the system next looks to see if there is a constructor that takes a single argument that is an integer. If it finds a constructor that takes a single-integer argument, it uses that constructor to convert the integer 25 to a value of type `Money`. The constructor with one argument of type `long` tells the system how to convert an integer, such as 25, to a value of type `Money`. The one-argument constructor says that 25 should be converted to an object of type `Money` whose member variable `all_cents` is equal to 2500; in other words, the constructor converts 25 to an object of type `Money` that represents \$25.00. (The definition of the constructor is in Display 11.3.)

Note that this type conversion will not work unless there is a suitable constructor. For example, the type `Money` (Display 11.5) has no constructor that takes an argument of type `double`, so the following is illegal and would produce an error message if you were to put it in a program that declares `base_amount` and `full_amount` to be of type `Money`:

```
full_amount = base_amount + 25.67;
```

To make this use of `+` legal, you could change the definition of the class `Money` by adding another constructor. The function declaration for the constructor you need to add is the following:

```

class Money
{
public:
    . . .

```

```
Money(double amount);
//Initializes the object so its value represents $amount.
. . .
```

Writing the definition for this new constructor is Self-Test Exercise 16.

These automatic type conversions (produced by constructors) seem most common and compelling with overloaded numeric operators such as + and -. However, these automatic conversions apply in exactly the same way to arguments for ordinary functions, arguments for member functions, and arguments for other overloaded operators.

SELF-TEST EXERCISE

16. Give the definition for the constructor discussed at the end of the previous section. The constructor is to be added to the class Money in Display 11.5. The definition begins as follows:

```
Money::Money(double amount)
{
```

Overloading Unary Operators

In addition to the binary operators, such as + in $x+y$, there are also unary operators, such as the operator - when it is used to mean negation. In the following statement, the unary operator - is used to set the value of a variable x equal to the negative of the value of the variable y :

```
x = -y;
```

The increment and decrement operators ++ and -- are other examples of unary operators.

You can overload unary operators as well as binary operators. For example, you can redefine the type Money given in Display 11.5 so that it has both a unary and a binary operator version of the subtraction/negation operator -. The redone class definition is given in Display 11.6. Suppose your program contains this class definition and the following code:

```
Money amount1(10), amount2(6), amount3;
```

Then the following sets the value of amount3 to amount1 minus amount2:

```
amount3 = amount1 - amount2;
```

The following will, then, output \$4.00 to the screen:

```
amount3.output(cout);
```

On the other hand, the following will set amount3 equal to the negative of amount1:

```
amount3 = -amount1;
```

The following will, then, output -\$10.00 to the screen:

```
amount3.output(cout);
```

You can overload the ++ and -- operators in ways similar to the way we overloaded the negation operator in Display 11.6. The overloading definition will apply to the operator when it is used in prefix position, as in ++x and --x. The postfix versions of ++ and --, as in x++ and x--, are handled in a different manner, but we will not discuss these postfix versions. (Hey, you can't learn everything in a first course!)

Overloading >> and <<

<< is an operator

The insertion operator << that we used with cout is a binary operator like the binary operators + or -. For example, consider the following:

```
cout << "Hello out there.\n";
```

The operator is <<, the first operand is the output stream cout, and the second operand is the string value "Hello out there.\n". You can change either of these operands. If fout is an output stream of type ofstream and fout has been connected to a file with a call to open, then you can replace cout with fout and the string will instead be written to the file connected to fout. Of course, you can also replace the string "Hello out there.\n" with another string, a variable, or a number. Since the insertion operator << is an operator, you should be able to overload it just as you overload operators such as + and -. This is true, but there are a few more details to worry about when you overload the input and output operators >> and <<.

Overloading <<

In our previous definitions of the class Money, we used the member function output to output values of type Money (Displays 11.3 through 11.6). This is adequate, but it would be nicer if we could simply use the insertion operator << to output values of type Money as in the following:

```
Money amount(100);
cout << "I have " << amount << " in my purse.\n";
```

instead of having to use the member function output as shown here:

```
Money amount(100);
cout << "I have ";
amount.output(cout);
cout << " in my purse.\n";
```

One problem in overloading the operator << is deciding what value should be returned when << is used in an expression like the following:

```
cout << amount
```


DISPLAY 11.6 Overloading a Unary Operator

```

1  //Class for amounts of money in U.S. currency.
2  class Money
3  {
4  public:
5      friend Money operator +(const Money& amount1, const Money& amount2);

6      friend Money operator -(const Money& amount1, const Money& amount2);
7      //Precondition: amount1 and amount2 have been given values.
8      //Returns amount1 minus amount2.

9      friend Money operator -(const Money& amount);
10     //Precondition: amount has been given a value.
11     //Returns the negative of the value of amount.

12     friend bool operator ==(const Money& amount1, const Money& amount2);

13     Money(long dollars, int cents);
14     Money(long dollars);
15     Money( );

16     double get_value( ) const;

17     void input(istream& ins);
18     void output(ostream& outs) const;
19 private:
20     long all_cents;
21 };

```

*This is an improved version of the class **Money** given in Display 11.5.*

*We have omitted the **include** directives and some of the comments, but you should include them in your programs.*

<Any additional function declarations as well as the main part of the program go here.>

```

22  Money operator -(const Money& amount1, const Money& amount2)
23  {
24      Money temp;
25      temp.all_cents = amount1.all_cents - amount2.all_cents;
26      return temp;
27  }

28  Money operator -(const Money& amount)
29  {
30      Money temp;
31      temp.all_cents = -amount.all_cents;
32      return temp;
33  }

```

<The other function definitions are the same as in Display 11.5.>

The two operands in this expression are `cout` and `amount`, and evaluating the expression should cause the value of `amount` to be written to the screen. But if `<<` is an operator like `+` or `*`, then the expression above should also return some value. After all, expressions with other operands, such as `n1 + n2`, return values. But what does `cout << amount` return? To obtain the answer to that question, we need to look at a more complicated expression involving `<<`.

Chains of `<<`

Let's consider the following expression, which involves evaluating a chain of expressions using `<<`:

```
cout << "I have " << amount << " in my purse.\n";
```

If you think of the operator `<<` as being analogous to other operators, such as `+`, then the above should be (and in fact is) equivalent to the following:

```
((cout << "I have ") << amount) << " in my purse.\n";
```

What value should `<<` return in order to make sense of this expression? The first thing evaluated is the subexpression:

```
(cout << "I have ")
```

If things are to work out, then the subexpression had better return `cout` so that the computation can continue as follows:

```
(cout << amount) << " in my purse.\n";
```

And if things are to continue to work out, `(cout << amount)` had better also return `cout` so that the computation can continue as follows:

```
cout << " in my purse.\n";
```

`<<` returns a stream

This is illustrated in Display 11.7. The operator `<<` should return its first argument, which is a stream of type `ostream`.

Thus, the declaration for the overloaded operator `<<` (to use with the class `Money`) should be as follows:

```
class Money
{
public:
    . . .
    friend ostream& operator <<(ostream& outs, const
                               Money& amount);
    //Precondition: If outs is a file output stream, then outs
    //has already been connected to a file.
    //Postcondition: A dollar sign and the amount of money
    //recorded in the calling object have been sent to the output
    //stream outs.
    . . .
}
```

Once we have overloaded the insertion (output) operator `<<`, we will no longer need the member function `output` and thus can delete `output` from

DISPLAY 11.7 << as an Operator

```

1  cout << "I have " << amount << " in my purse.\n";
2
3  means the same as
4
5  ((cout << "I have ") << amount) << " in my purse.\n";
6
7  and is evaluated as follows:
8
9  First evaluate (cout << "I have "), which returns cout:
10 ((cout << "I have ") << amount) << " in my purse.\n";
11
12 and the string "I have" is output.
13
14 (cout << amount) << " in my purse.\n";
15
16
17 Then evaluate (cout << amount), which returns cout:
18
19 (cout << amount) << " in my purse.\n";
20
21 and the value of amount is output.
22
23 cout << " in my purse.\n";
24
25
26 Then evaluate cout << " in my purse.\n", which returns cout:
27
28 cout << " in my purse.\n";
29
30 and the string "in my purse.\n" is output.
31
32 cout;
33
34 Since there are no more << operators, the process ends.

```

our definition of the class `Money`. The definition of the overloaded operator `<<` is very similar to the member function `output`. In outline form, the definition for the overloaded operator is as follows:

```

ostream& operator <<(ostream& outs, const Money& amount)
{
    <This part is the same as the body of Money::output
    that is given in Display 11.3 (except that all_cents
    is replaced with amount.all_cents).>

    return outs;
}

```

**<< and >> return
a reference**

There is one thing left to explain in the previous function declaration and definition for the overloaded operator <<. What is the meaning of the & in the returned type `ostream&`? The easiest answer is that *whenever an operator (or a function) returns a stream, you must add an & to the end of the name for the returned type*. That simple rule will allow you to overload the operators << and >>. However, although that is a good working rule that will allow you to write your class definitions and programs, it is not very satisfying. You do not need to know what that & really means, but if we explain it, that will remove some of the mystery from the rule that tells you to add an &.

**Returning a
reference**

When you add an & to the name of a returned type, you are saying that the operator (or function) returns a *reference*. All the functions and operators we have seen thus far return values. However, if the returned type is a stream, you cannot simply return the value of the stream. In the case of a stream, the value of the stream is an entire file or the keyboard or the screen, and it may not make sense to return those things. Thus, you want to return only the stream itself rather than the value of the stream. When you add an & to the name of a returned type, you are saying that the operator (or function) returns a **reference**, which means that you are returning the object itself, as opposed to the value of the object.

The extraction operator >> is overloaded in a way that is analogous to what we described for the insertion operator <<. However, with the extraction (input) operator >>, the second argument will be the object that receives the input value, so the second parameter must be an ordinary call-by-reference parameter. In outline form, the definition for the overloaded extraction operator >> is as follows:

```
istream& operator >>(istream& ins, Money& amount)
{
    <This part is the same as the body of
    Money::input given in Display 11.3 (except that
    all_cents is replaced with amount.all_cents).>
    return ins;
}
```

The complete definitions of the overloaded operators << and >> are given in Display 11.8, where we have rewritten the class `Money` yet again. This time we have rewritten the class so that the operators << and >> are overloaded to allow us to use these operators with values of type `Money`.

Overloading >> and <<

The input and output operators >> and << can be overloaded just like any other operators. The value returned must be the stream. The type for the value returned must have the & symbol added to the end of the type name. The function declarations and beginnings of the function definitions are as shown on the next page. See Display 11.8 for an example.

(continued)

FUNCTION DECLARATIONS

```

class Class_Name
{
public:
    . . .

    friend istream& operator >>(istream& Parameter_1,
                               Class_Name& Parameter_2);

    friend ostream& operator <<(ostream& Parameter_3,
                               const Class_Name&
                               Parameter_4);

    . . .

```

Parameter for the stream (points to Parameter_1)

Parameter for the object to receive the input (points to Parameter_2)

DEFINITIONS

```

istream& operator >>(istream& Parameter_1,
                    Class_Name& Parameter_2)
{
    . . .
}

ostream& operator <<(ostream& Parameter_3,
                    const Class_Name& Parameter_4)
{
    . . .
}

```

DISPLAY 11.8 Overloading << and >> (part 1 of 4)

```

1  //Program to demonstrate the class Money
2  #include <iostream>
3  #include <fstream>
4  #include <cstdlib>
5  #include <cctype>
6  using namespace std;
7
8  //Class for amounts of money in U.S. currency.
9  class Money
10 {
11 public:
12     friend Money operator +(const Money& amount1, const Money& amount2);
13     friend Money operator -(const Money& amount1, const Money& amount2);
14     friend Money operator -(const Money& amount);

```

*This is an improved version of the class **Money** that we gave in Display 11.6.*

Although we have omitted some of the comments from Displays 11.5 and 11.6, you should include them.

(continued)

DISPLAY 11.8 Overloading << and >> (part 2 of 4)

```

15     friend bool operator ==(const Money& amount1, const Money& amount2);
16     Money(long dollars, int cents);
17     Money(long dollars);
18     Money( );
19     double get_value( ) const;
20     friend istream& operator >>(istream& ins, Money& amount);
21     //Overloads the >> operator so it can be used to input values of type Money.
22     //Notation for inputting negative amounts is as in -$100.00.
23     //Precondition: If ins is a file input stream, then ins has already been
24     //connected to a file.
25     friend ostream& operator <<(ostream& outs, const Money& amount);
26     //Overloads the << operator so it can be used to output values of type Money.
27     //Precedes each output value of type Money with a dollar sign.
28     //Precondition: If outs is a file output stream,
29     //then outs has already been connected to a file.
30 private:
31     long all_cents;
32 };
33 int digit_to_int(char c);
34 //Used in the definition of the overloaded input operator >>.
35 //Precondition: c is one of the digits '0' through '9'.
36 //Returns the integer for the digit; for example, digit_to_int('3') returns 3.
37
38 int main( )
39 {
40     Money amount;
41     ifstream in_stream;
42     ofstream out_stream;
43
44     in_stream.open("infile.dat");
45     if (in_stream.fail( ))
46     {
47         cout << "Input file opening failed.\n";
48         exit(1);
49     }
50
51     out_stream.open("outfile.dat");
52     if (out_stream.fail( ))
53     {
54         cout << "Output file opening failed.\n";
55         exit(1);
56     }

```

(continued)

DISPLAY 11.8 Overloading << and >> (part 3 of 4)

```

57
58     in_stream >> amount;
59     out_stream << amount
60         << " copied from the file infile.dat.\n";
61     cout << amount
62         << " copied from the file infile.dat.\n";
63
64     in_stream.close( );
65     out_stream.close( );
66
67     return 0;
68 }
69 //Uses iostream, ctype, cstdlib:
70 istream& operator >>(istream& ins, Money& amount)
71 {
72     char one_char, decimal_point,
73         digit1, digit2; //digits for the amount of cents
74     long dollars;
75     int cents;
76     bool negative; //set to true if input is negative.
77
78     ins >> one_char;
79     if (one_char == '-')
80     {
81         negative = true;
82         ins >> one_char; //read '$'
83     }
84     else
85         negative = false;
86     //if input is legal, then one_char == '$'
87
88     ins >> dollars >> decimal_point >> digit1 >> digit2;
89
90     if (one_char != '$' || decimal_point != '.'
91         || !isdigit(digit1) || !isdigit(digit2))
92     {
93         cout << "Error illegal form for money input\n";
94         exit(1);
95     }
96
97     cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);
98
99     amount.all_cents = dollars * 100 + cents;
100    if (negative)
101        amount.all_cents = -amount.all_cents;
102    return ins;
103 }

```

(continued)

DISPLAY 11.8 Overloading << and >> (part 4 of 4)

```

100  int digit_to_int(char c)
101  {
102      return ( static_cast<int>(c) - static_cast<int>('0') );
103  }

104  //Uses cstdlib and iostream:
105  ostream& operator <<(ostream& outs, const Money& amount)
106  {
107      long positive_cents, dollars, cents;
108      positive_cents = labs(amount.all_cents);
109      dollars = positive_cents/100;
110      cents = positive_cents%100;
111
112      if (amount.all_cents < 0)
113          outs << "- $" << dollars << '.';
114      else
115          outs << "$" << dollars << '.';
116
117      if (cents < 10)
118          outs << '0';
119      outs << cents;
120
121      return outs;
122  }
123

```

<The definitions of the member functions and other overloaded operators go here.
See Displays 11.3, 11.4, 11.5, and 11.6 for the definitions.>

infile.dat

(Not changed by program.)

\$1.11 \$2.22 \$3.33

outfile.dat

(After program is run.)

\$1.11 copied from the file infile.dat.

Screen Output

\$1.11 copied from the file infile.dat.

SELF-TEST EXERCISES

17. Here is a definition of a class called `Pairs`. Objects of type `Pairs` can be used in any situation where ordered pairs are needed. Your task is to write implementations of the overloaded operator `>>` and the overloaded operator `<<` so that objects of class `Pairs` are to be input and output in the form `(5,6)(5,-4)(-5,4)` or `(-5,-6)`. You need not implement any constructor or other member, and you need not do any input format checking.

```
#include <iostream>
using namespace std;
class Pairs
{
public:
    Pairs( );
    Pairs(int first, int second);
    //other members and friends
    friend istream& operator >>(istream& ins, Pairs& second);
    friend ostream& operator <<(ostream& outs, const Pairs& second);
private:
    int f;
    int s;
};
```

18. Following is the definition for a class called `Percent`. Objects of type `Percent` represent percentages such as 10% or 99%. Give the definitions of the overloaded operators `>>` and `<<` so that they can be used for input and output with objects of the class `Percent`. Assume that input always consists of an integer followed by the character `'%'`, such as 25%. All percentages are whole numbers and are stored in the `int` member variable named `value`. You do not need to define the other overloaded operators and do not need to define the constructor. You only have to define the overloaded operators `>>` and `<<`.

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

class Percent
{
public:
    friend bool operator ==(const Percent& first,
                           const Percent& second);

    friend bool operator <(const Percent& first,
                           const Percent& second);

    Percent();

    Percent(int percent_value);
```

```

    friend istream& operator >>(istream& ins,
                                Percent& the_object);

    //Overloads the >> operator to input values of type
    //Percent.
    //Precondition: If ins is a file input stream, then ins
    //has already been connected to a file.

    friend ostream& operator <<(ostream& outs,
                                const Percent& a_percent);
    //Overloads the << operator for output values of type
    //Percent.
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
private:
    int value;
};

```

11.3 ARRAYS AND CLASSES

You can combine arrays, structures, and classes to form intricately structured types such as arrays of structures, arrays of classes, and classes with arrays as member variables. In this section we discuss a few simple examples to give you an idea of the possibilities.

Arrays of Classes

The base type of an array may be any type, including types that you define, such as structure and class types. If you want each indexed variable to contain items of different types, make the array an array of structures. For example, suppose you want an array to hold ten weather data points, where each data point is a wind velocity and a wind direction (north, south, east, or west). You might use the following type definition and array declaration:

```

struct WindInfo
{
    double velocity; //in miles per hour
    char direction; // 'N', 'S', 'E', or 'W'
};

WindInfo data_point[10];

```

To fill the array `data_point`, you could use the following *for* loop:

```

int i;
for (i = 0; i < 10; i++)

```

```

{
    cout << "Enter velocity for "
          << i << " numbered data point: ";
    cin >> data_point[i].velocity;
    cout << "Enter direction for that data point"
          << " (N, S, E, or W): ";
    cin >> data_point[i].direction;
}

```

The way to read an expression such as `data_point[i].velocity` is left to right and very carefully. First, `data_point` is an array. So, `data_point[i]` is the *i*th indexed variable of this array. An indexed variable of this array is of type `WindInfo`, which is a structure with two member variables named `velocity` and `direction`. So, `data_point[i].velocity` is the member variable named `velocity` for the *i*th array element. Less formally, `data_point[i].velocity` is the wind velocity for the *i*th data point. Similarly, `data_point[i].direction` is the wind direction for the *i*th data point.

The ten data points in the array `data_point` can be written to the screen with the following *for* loop:

```

for (i = 0; i < 10; i++)
    cout << "Wind data point number " << i << ": \n"
          << data_point[i].velocity
          << " miles per hour\n"
          << "direction " << data_point[i].direction
          << endl;

```

Display 11.9 contains the definition for a class called `Money`. Objects of the class `Money` are used to represent amounts of money in U.S. currency. The definitions of the member functions, member operations, and friend functions for this class can be found in Displays 11.3 through 11.8 and in the answer to Self-Test Exercise 13. You can have arrays whose base type is the type `Money`. A simple example is given in Display 11.9. That program reads in a list of five amounts of money and computes how much each amount differs from the largest of the five amounts. Notice that an array whose base type is a class is treated basically the same as any other array. In fact, the program in Display 11.9 is very similar to the program in Display 7.1, except that in Display 11.9 the base type is a class.

When an array of classes is declared, the default constructor is called to initialize the indexed variables, so it is important to have a default constructor for any class that will be the base type of an array. An array of classes is manipulated just like an array with a simple base type like `int` or `double`. For example, the difference between each amount and the largest amount is stored in an array named `difference`, as follows:

Constructor call

```

Money difference[5];
for (i = 0; i < 5; i++)
    difference[i] = max - amount[i];

```

DISPLAY 11.9 Program Using an Array of Money Objects *(part 1 of 2)*

```

1  //This is the definition for the class Money.
2  //Values of this type are amounts of money in U.S. currency.
3  #include <iostream>
4  using namespace std;

5  class Money
6  {
7  public:
8  friend Money operator +(const Money& amount1, const Money& amount2);
9  //Returns the sum of the values of amount1 and amount2.

10 friend Money operator -(const Money& amount1, const Money& amount2);
11 //Returns amount1 minus amount2.

12 friend Money operator -(const Money& amount);
13 //Returns the negative of the value of amount.

14 friend bool operator ==(const Money& amount1, const Money& amount2);
15 //Returns true if amount1 and amount2 have the same value; false otherwise.

16 friend bool operator <(const Money& amount1, const Money& amount2);
17 //Returns true if amount1 is less than amount2; false otherwise.

18 Money(long dollars, int cents);
19 //Initializes the object so its value represents an amount with
20 //the dollars and cents given by the arguments. If the amount
21 //is negative, then both dollars and cents should be negative.

22 Money(long dollars);
23 //Initializes the object so its value represents $dollars.00.

24 Money( );
25 //Initializes the object so its value represents $0.00.

26 double get_value( ) const;
27 //Returns the amount of money recorded in the data portion of the calling
28 //object.

29 friend istream& operator >>(istream& ins, Money& amount);
30 //Overloads the >> operator so it can be used to input values of type
31 //Money. Notation for inputting negative amounts is as in - $100.00.
32 //Precondition: If ins is a file input stream, then ins has already been
33 //connected to a file.

34
35 friend ostream& operator <<(ostream& outs, const Money& amount);
36 //Overloads the << operator so it can be used to output values of type
37 //Money. Precedes each output value of type Money with a dollar sign.
38 //Precondition: If outs is a file output stream, then outs has already been
39 //connected to a file.

40 private:
41     long all_cents;
42 };
43

```

(continued)

DISPLAY 11.9 Program Using an Array of Money Objects (part 2 of 2)

<The definitions of the member functions and the overloaded operators goes here.>

```

44  //Reads in 5 amounts of money and shows how much each
45  //amount differs from the largest amount.
46  int main( )
47  {
48      Money amount[5], max;
49      int i;
50      cout << "Enter 5 amounts of money:\n";
51      cin >> amount[0];
52      max = amount[0];
53      for (i = 1; i < 5; i++)
54      {
55          cin >> amount[i];
56          if (max < amount[i])
57              max = amount[i];
58          //max is the largest of amount[0], . . . , amount[i].
59      }
60      Money difference[5];
61      for (i = 0; i < 5; i++)
62          difference[i] = max - amount[i];
63      cout << "The highest amount is " << max << endl;
64      cout << "The amounts and their\n"
65           << "differences from the largest are:\n";
66      for (i = 0; i < 5; i++)
67      {
68          cout << amount[i] << " off by "
69               << difference[i] << endl;
70      }
71      return 0;
72  }
```

Sample Dialogue

```

Enter 5 amounts of money:
$5.00 $10.00 $19.99 $20.00 $12.79
The highest amount is $20.00
The amounts and their
differences from the largest are:
$5.00 off by $15.00
$10.00 off by $10.00
$19.99 off by $0.01
$20.00 off by $0.00
$12.79 off by $7.21
```

SELF-TEST EXERCISES

19. Give a type definition for a structure called `Score` that has two member variables called `home_team` and `opponent`. Both member variables are of type `int`. Declare an array called `game` that is an array with ten elements of type `Score`. The array `game` might be used to record the scores of each of ten games for a sports team.
20. Write a program that reads in five amounts of money, doubles each amount, and then writes out the doubled values to the screen. Use one array with `Money` as the base type. (*Hint:* Use Display 11.9 as a guide, but this program will be simpler than the one in Display 11.9.)

Arrays as Class Members

You can have a structure or class that has an array as a member variable. For example, suppose you are a speed swimmer and want a program to keep track of your practice times for various distances. You can use the structure `my_best` (of the type `Data` given next) to record a distance (in meters) and the times (in seconds) for each of ten practice tries swimming that distance:

```
struct Data
{
    double time[10];
    int distance;
};

Data my_best;
```

The structure `my_best`, declared above, has two member variables: One, named `distance`, is a variable of type `int` (to record a distance); the other, named `time`, is an array of ten values of type `double` (to hold times for ten practice tries at the specified distance). To set the distance equal to 20 (meters), you can use the following:

```
my_best.distance = 20;
```

You can set the ten array elements with values from the keyboard as follows:

```
cout << "Enter ten times (in seconds):\n";
for (int i = 0; i < 10; i++)
    cin >> my_best.time[i];
```

The expression `my_best.time[i]` is read left to right: `my_best` is a structure; `my_best.time` is the member variable named `time`. Since `my_best.time` is an array, it makes sense to add an index. So, the expression `my_best.time[i]` is the *i*th indexed variable of the array `my_best.time`. If you use a class rather than a structure type, then you can do all your array manipulations with member functions and avoid such confusing expressions. This is illustrated in the following Programming Example.

PROGRAMMING EXAMPLE**A Class for a Partially Filled Array**

Display 11.10 shows the definition for a class called `TemperatureList`, whose objects are lists of temperatures. You might use an object of type `TemperatureList` in a program that does weather analysis. The list of temperatures is kept in the member variable `list`, which is an array. Since this array will typically be only partially filled, a second member variable, called `size`, is used to keep track of how much of the array is used. The value of `size` is the number of indexed variables of the array `list` that are being used to store values.

An object of type `TemperatureList` is declared like an object of any other type. For example, the following declares `my_data` to be an object of type `TemperatureList`:

```
TemperatureList my_data;
```

This declaration calls the default constructor with the new object `my_data`, and so the object `my_data` is initialized so that the member variable `size` has the value 0, indicating an empty list.

Once you have declared an object such as `my_data`, you can add an item to the list of temperatures (that is, to the member array `list`) with a call to the member function `add_temperature` as follows:

```
my_data.add_temperature(77);
```

In fact, this is the only way you can add a temperature to the list `my_data`, since the array `list` is a private member variable. Notice that when you add an item with a call to the member function `add_temperature`, the function call first tests to see if the array `list` is full and adds the value only if the array is not full.

The class `TemperatureList` is very specialized. The only things you can do with an object of the class `TemperatureList` are to initialize the list so it is empty, add items to the list, check if the list is full, and output the list. To output the temperatures stored in the object `my_data` (declared previously), the call would be as follows:

```
cout << my_data;
```

With the class `TemperatureList` you cannot delete a temperature from the list (array) of temperatures. You can, however, erase the entire list and start over with an empty list by calling the default constructor, as follows:

```
my_data = TemperatureList();
```

The type `TemperatureList` uses almost no properties of temperatures. You could define a similar class for lists of pressures or lists of distances or lists of any other data expressed as values of type *double*. To save yourself the trouble of defining all these different classes, you could define a single class that represents an arbitrary list of values of type *double* without specifying what the values represent.

DISPLAY 11.10 Program for a Class with an Array Member (part 1 of 2)

```

1  //This is a definition for the class
2  //TemperatureList. Values of this type are lists of Fahrenheit temperatures.
3
4  #include <iostream>
5  #include <cstdlib>
6  using namespace std;
7
8  const int MAX_LIST_SIZE = 50;
9
10 class TemperatureList
11 {
12 public:
13     TemperatureList( );
14     //Initializes the object to an empty list.
15
16     void add_temperature(double temperature);
17     //Precondition: The list is not full.
18     //Postcondition: The temperature has been added to the list.
19
20     bool full( ) const;
21     //Returns true if the list is full; false otherwise.
22
23     friend ostream& operator <<(ostream& outs,
24         const TemperatureList& the_object);
25     //Overloads the << operator so it can be used to output values of
26     //type TemperatureList. Temperatures are output one per line.
27     //Precondition: If outs is a file output stream, then outs
28     //has already been connected to a file.
29 private:
30     double list[MAX_LIST_SIZE]; //of temperatures in Fahrenheit
31     int size; //number of array positions filled
32 };
33
34 //This is the implementation for the class TemperatureList.
35
36 TemperatureList::TemperatureList( ) : size(0)
37 {
38     //Body intentionally empty.
39 }
40 void TemperatureList::add_temperature(double temperature)
41 { //Uses iostream and cstdlib:
42     if ( full( ) )
43     {
44         cout << "Error: adding to a full list.\n";
45         exit(1);
46     }

```

(continued)

DISPLAY 11.10 Program for a Class with an Array Member (part 2 of 2)

```

47     else
48     {
49         list[size] = temperature;
50         size = size + 1;
51     }
52 }
53 bool TemperatureList::full( ) const
54 {
55     return (size == MAX_LIST_SIZE);
56 }
57 //Uses iostream:
58 ostream& operator <<(ostream& outs, const TemperatureList& the_object)
59 {
60     for (int i = 0; i < the_object.size; i++)
61         outs << the_object.list[i] << " F\n";
62     return outs;
63 }

```

SELF-TEST EXERCISES

21. Change the class `TemperatureList` given in Display 11.10 by adding a member function called `get_size`, which takes no arguments and returns the number of temperatures on the list.
22. Change the type `TemperatureList` given in Display 11.10 by adding a member function called `get_temperature`, which takes one `int` argument that is an integer greater than or equal to 0 and strictly less than `MAX_LIST_SIZE`. The function returns a value of type `double`, which is the temperature in that position on the list. So, with an argument of 0, `get_temperature` returns the first temperature; with an argument of 1, it returns the second temperature, and so forth. Assume that `get_temperature` will not be called with an argument that specifies a location on the list that does not currently contain a temperature.

11.4 CLASSES AND DYNAMIC ARRAYS

With all appliances and means to boot.

WILLIAM SHAKESPEARE, *King Henry IV, Part III*

A dynamic array can have a base type that is a class. A class can have a member variable that is a dynamic array. You can combine the techniques you

learned about classes and the techniques you learned about dynamic arrays in just about any way. There are a few more things to worry about when using classes and dynamic arrays, but the basic techniques are the ones that you have already used. Let's start with an example.

PROGRAMMING EXAMPLE

A String Variable Class

In Chapter 8 we showed you how to define array variables to hold C strings. In the previous section you learned how to define dynamic arrays so that the size of the array can be determined when your program is run. In this example we will define a class called `StringVar` whose objects are string variables. An object of the class `StringVar` will be implemented using a dynamic array whose size is determined when your program is run. So objects of type `StringVar` will have all the advantages of dynamic arrays, but they will also have some additional features. We will define `StringVar`'s member functions so that if you try to assign a string that is too long to an object of type `StringVar`, you will get an error message. The version we define here provides only a small collection of operations for manipulating string objects. In Programming Project 1 you are asked to enhance the class definition by adding more member functions and overloaded operators.

Since you could use the standard class `string`, as discussed in Chapter 8, you do not really need the class `StringVar`, but it will be a good exercise to design and code it.

Constructors

The definition for the type `StringVar` is given in Display 11.11. One constructor for the class `StringVar` takes a single argument of type `int`. This argument determines the maximum allowable length for a string value stored in the object. A default constructor creates an object with a maximum allowable length of 100. Another constructor takes an array argument that contains a C string of the kind discussed in Chapter 8. Note that this means the argument to this constructor can be a quoted string. This constructor initializes the object so that it can hold any string whose length is less than or equal to the length of its argument, and it initializes the object's string value to a copy of the value of its argument. For the moment, ignore the constructor that is labeled *Copy constructor*. Also ignore the member function named `~StringVar`. Although it may look like one, `~StringVar` is not a constructor. We will discuss these two new kinds of member functions in later subsections. The meanings of the remaining member functions for the class `StringVar` are straight forward.

Size of string value

A simple demonstration program is given in Display 11.11. Two objects, `your_name` and `our_name`, are declared within the definition of the function `conversation`. The object `your_name` can contain any string that is `max_name_size` or fewer characters long. The object `our_name` is initialized to the string value "Borg" and can have its value changed to any other string of length 4 or less.

DISPLAY 11.11 Program Using the StringVar Class (part 1 of 2)

```

1  //This is the definition for the class StringVar
2  //whose values are strings. An object is declared as follows.
3  //Note that you use (max_size), not [max_size]
4  //StringVar the_object(max_size);
5  //where max_size is the longest string length allowed.
6  #include <iostream>
7  using namespace std;
8
9  class StringVar
10 {
11 public:
12     StringVar(int size);
13     //Initializes the object so it can accept string values up to size
14     //in length. Sets the value of the object equal to the empty string.
15
16     StringVar( );
17     //Initializes the object so it can accept string values of length 100
18     //or less. Sets the value of the object equal to the empty string.
19
20     StringVar(const char a[]);
21     //Precondition: The array a contains characters terminated with '\0'.
22     //Initializes the object so its value is the string stored in a and
23     //so that it can later be set to string values up to strlen(a) in length.
24
25     StringVar(const StringVar& string_object);
26     //Copy constructor.
27
28     ~StringVar( );
29     //Returns all the dynamic memory used by the object to the freestore.
30
31     int length( ) const;
32     //Returns the length of the current string value.
33
34     void input_line(istream& ins);
35     //Precondition: If ins is a file input stream, then ins has been
36     //connected to a file.
37     //Action: The next text in the input stream ins, up to '\n', is copied
38     //to the calling object. If there is not sufficient room, then
39     //only as much as will fit is copied.
40
41     friend ostream& operator <<(ostream& outs, const StringVar& the_string);
42     //Overloads the << operator so it can be used to output values
43     //of type StringVar
44     //Precondition: If outs is a file output stream, then outs
45     //has already been connected to a file.

```

(continued)

DISPLAY 11.11 Program Using the StringVar Class (part 2 of 2)

```

45  private:
46      char *value; //pointer to dynamic array that holds the string value.
47      int max_length; //declared max length of any string value.
48  };
49
50
51  <The definitions of the member functions and overloaded operators go here>
52
53  //Program to demonstrate use of the class StringVar.
54
55  void conversation(int max_name_size);
56  //Carries on a conversation with the user.
57
58  int main( )
59  {
60      using namespace std;
61      conversation(30);
62      cout << "End of demonstration.\n";
63      return 0;
64  }
65
66  //This is only a demonstration function:
67  void conversation(int max_name_size)
68  {
69      using namespace std;
70
71      StringVar your_name(max_name_size), our_name("Borg");
72
73      cout << "What is your name?\n";
74      your_name.input_line(cin);
75      cout << "We are " << our_name << endl;
76      cout << "We will meet again " << your_name << endl;
77  }

```

Memory is returned to the freestore when the function call ends.

Determines the size of the dynamic array

Sample Dialogue

```

What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
End of demonstration

```

As we indicated at the beginning of this subsection, the class `StringVar` is implemented using a dynamic array. The implementation is shown in Display 11.12. When an object of type `StringVar` is declared, a constructor is called to initialize the object. The constructor uses the *new* operator to create a new dynamic array of characters for the member variable `value`. The string value is stored in the array `value` as an ordinary string value, with `'\0'` used to mark the end of the string. Notice that the size of this array is not determined until the object is declared, at which point the constructor is called and the argument to the constructor determines the size of the dynamic array. As illustrated in Display 11.11, this argument can be a variable of type *int*. Look at the declaration of the object `your_name` in the definition of the function `conversation`. The argument to the constructor is the call-by-value parameter `max_name_size`. Recall that a call-by-value parameter is a local variable, so `max_name_size` is a variable. Any *int* variable may be used as the argument to the constructor in this way.

The implementation of the member functions `length`, `input_line`, and the overloaded output operator `<<` are all straightforward. In the next few subsections we discuss the function `~StringVar` and the constructor labeled *Copy constructor*.

Implementation

Destructors

There is one problem with dynamic variables. They do not go away unless your program makes a suitable call to *delete*. Even if the dynamic variable was created using a local pointer variable and the local pointer variable goes away at the end of a function call, the dynamic variable will remain unless there is a call to *delete*. If you do not eliminate dynamic variables with calls to *delete*, they will continue to occupy memory space, which may cause your program to abort because it used up all the memory in the freestore. Moreover, if the dynamic variable is embedded in the implementation of a class, the programmer who uses the class does not know about the dynamic variable and cannot be expected to perform the call to *delete*. In fact, since the data members are normally private members, the programmer normally *cannot* access the needed pointer variables and so *cannot* call *delete* with these pointer variables. To handle this problem, C++ has a special kind of member function called a *destructor*.

A **destructor** is a member function that is called automatically when an object of the class passes out of scope. This means that if your program contains a local variable that is an object with a destructor, then when the function call ends, the destructor is called automatically. If the destructor is defined correctly, the destructor calls *delete* to eliminate all the dynamic variables created by the object. This may be done with a single call to *delete* or it may require several calls to *delete*. You might also want your destructor to perform some other cleanup details as well, but returning memory to the freestore is the main job of the destructor.



VideoNote
Arrays of Classes using
Dynamic Arrays

The member function `~StringVar` is the destructor for the class `StringVar` shown in Display 11.11. Like a constructor, a destructor always has the same name as the class it is a member of, but the destructor has the tilde symbol, `~`, at the beginning of its name (so you can tell that it is a destructor and not a constructor). Like a constructor, a destructor has no type for the value returned, not even the type `void`. A destructor has no parameters. Thus, a class can have only one destructor; you cannot overload the destructor for a class. Otherwise, a destructor is defined just like any other member function.

Notice the definition of the destructor `~StringVar` given in Display 11.12. `~StringVar` calls `delete` to eliminate the dynamic array pointed to by the member pointer variable `value`. Look again at the function conversation in the sample program shown in Display 11.11. The local variables `your_name` and `our_name` both create dynamic arrays. If this class did not have a destructor, then after the call to `conversation` has ended, these dynamic arrays would still be occupying memory, even though they are useless to the program. This would not be a problem here because the sample program ends soon after the call to `conversation` is completed; but if you wrote a program that made repeated calls to functions like `conversation`, and if the class `StringVar` did not have a suitable destructor, then the function calls could consume all the memory in the freestore and your program would then end abnormally.

DISPLAY 11.12 Implementation of `StringVar` (part 1 of 2)

```

1  //This is the implementation of the class StringVar.
2  //The definition for the class StringVar is in Display 11.11.
3  #include <cstdlib>
4  #include <cstdint>
5  #include <cstring>
6
7  //Uses cstdint and cstdlib:
8  StringVar::StringVar(int size) : max_length(size)
9  {
10     value = new char[max_length + 1]; //+1 is for '\0'.
11     value[0] = '\0';
12 }
13
14 //Uses cstdint and cstdlib:
15 StringVar::StringVar( ) : max_length(100)
16 {
17     value = new char[max_length + 1]; //+1 is for '\0'.
18     value[0] = '\0';
19 }
20
21 //Uses cstring, cstdint, and cstdlib:
22 StringVar::StringVar(const char a[]) : max_length(strlen(a))
23 {
24     value = new char[max_length + 1]; //+1 is for '\0'.

```

(continued)

DISPLAY 11.12 Implementation of StringVar (part 2 of 2)

```

25     strcpy(value, a);
26 }
27 //Uses cstring, cstdint, and cstdlib:
28 StringVar::StringVar(const StringVar& string_object)
29     : max_length(string_object.length( ))
30 {
31     value = new char[max_length + 1]; //+1 is for '\0'.
32     strcpy(value, string_object.value);
33 }
34 StringVar::~StringVar( )
35 {
36     delete [] value;
37 }
38
39 //Uses cstring:
40 int StringVar::length( ) const
41 {
42     return strlen(value);
43 }
44
45 //Uses iostream:
46 void StringVar::input_line(istream& ins)
47 {
48     ins.getline(value, max_length + 1);
49 }
50
51 //Uses iostream:
52 ostream& operator <<(ostream& outs, const StringVar& the_string)
53 {
54     outs << the_string.value;
55     return outs;
56 }

```

Copy constructor (discussed later in this chapter)

Destructor

Destructor

A **destructor** is a member function of a class that is called automatically when an object of the class goes out of scope. Among other things, this means that if an object of the class type is a local variable for a function, then the destructor is automatically called as the last action before the function call ends. Destructors are used to eliminate any dynamic variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the freestore. Destructors may perform other cleanup tasks as well. The name of a destructor must consist of the tilde symbol, ~, followed by the name of the class.

PITFALL Pointers as Call-by-Value Parameters

When a call-by-value parameter is of a pointer type, its behavior can be subtle and troublesome. Consider the function call shown in Display 11.13. The parameter `temp` in the function `sneaky` is a call-by-value parameter, and hence it is a local variable. When the function is called, the value of `temp` is set to the value of the argument `p` and the function body is executed. Since `temp` is a local variable, no changes to `temp` should go outside of the function `sneaky`. In particular, the value of the pointer variable `p` should not be changed. Yet the sample dialogue makes it look as if the value of the pointer variable `p` had changed. Before the call to the function `sneaky`, the value of `*p` was 77, and after the call to `sneaky` the value of `*p` is 99. What has happened?

DISPLAY 11.13 A Call-by-Value Pointer Parameter (part 1 of 2)

```

1  //Program to demonstrate the way call-by-value parameters
2  //behave with pointer arguments.
3  #include <iostream>
4  using namespace std;
5
6  typedef int* IntPtr;
7
8  void sneaky(IntPtr temp);
9
10 int main( )
11 {
12     IntPtr p;
13
14     p = new int;
15     *p = 77;
16     cout << "Before call to function *p == "
17          << *p << endl;
18
19     sneaky(p);
20
21     cout << "After call to function *p == "
22          << *p << endl;
23
24     return 0;
25 }
26
27 void sneaky(IntPtr temp)
28 {
29     *temp = 99;
30     cout << "Inside function call *temp == "
31          << *temp << endl;

```

(continued)

DISPLAY 11.13 A Call-by-Value Pointer Parameter (part 2 of 2)**Sample Dialogue**

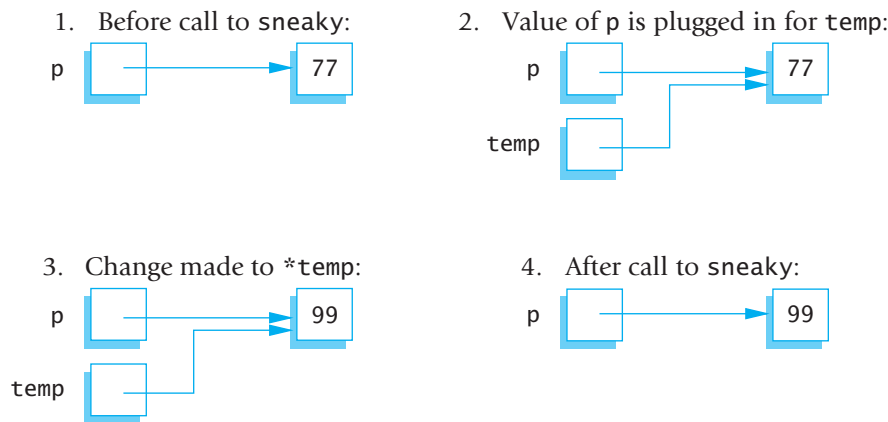
```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

The situation is diagrammed in Display 11.14. Although the sample dialogue may make it look as if `p` were changed, the value of `p` was not changed by the function call to `sneaky`. Pointer `p` has two things associated with it: `p`'s pointer value and the value stored where `p` points. Now, the value of `p` is a pointer (that is, a memory address). After the call to `sneaky`, the variable `p` contains the same pointer value (that is, the same memory address). The call to `sneaky` has changed the value of the variable pointed to by `p`, but it has not changed the value of `p` itself.

If the parameter type is a class or structure type that has member variables of a pointer type, the same kind of surprising changes can occur with call-by-value arguments of the class type. However, for class types, you can avoid (and control) these surprise changes by defining a *copy constructor*, as described in the next subsection. ■

Copy Constructors

A **copy constructor** is a constructor that has one parameter that is of the same type as the class. The one parameter must be a call-by-reference parameter, and normally the parameter is preceded by the *const* parameter modifier, so it is a constant parameter. In all other respects, a copy constructor is defined in the same way as any other constructor and can be used just like other constructors.

DISPLAY 11.14 The Function Call `sneaky(p);`

Called when an
object is declared

For example, a program that uses the class `StringVar` defined in Display 11.11 might contain the following:

```
StringVar line(20), motto("Constructors can help.");
cout << "Enter a string of length 20 or less:\n";
line.input_line(cin);
StringVar temp(line); //Initialized by the copy constructor.
```

The constructor used to initialize each of the three objects of type `StringVar` is determined by the type of the argument given in parentheses after the object's name. The object `line` is initialized with the constructor that has a parameter of type `int`; the object `motto` is initialized by the constructor that has a parameter of type `const char a[]`. Similarly, the object `temp` is initialized by the constructor that has one argument of type `const StringVar&`. When used in this way, a copy constructor is being used just like any other constructor.

A copy constructor should be defined so that the object being initialized becomes a complete, independent copy of its argument. So, in the declaration

```
StringVar temp(line);
```

the member variable `temp.value` is not simply set to the same value as `line.value`; that would produce two pointers pointing to the same dynamic array. The definition of the copy constructor is shown in Display 11.12. Note that in the definition of the copy constructor, a new dynamic array is created and the contents of one dynamic array are copied to the other dynamic array. Thus, in the previous declaration, `temp` is initialized so that its string value is equal to the string value of `line`, but `temp` has a separate dynamic array. Thus, any change that is made to `temp` has no effect on `line`.

As you have seen, a copy constructor can be used just like any other constructor. A copy constructor is also called automatically in certain other situations. Roughly speaking, whenever C++ needs to make a copy of an object, it automatically calls the copy constructor. In particular, the copy constructor is called automatically in three circumstances: (1) when a class object is declared and is initialized by another object of the same type, (2) when a function returns a value of the class type, and (3) whenever an argument of the class type is "plugged in" for a call-by-value parameter. In this case, the copy constructor defines what is meant by "plugging in."

To see why you need a copy constructor, let's see what would happen if we did not define a copy constructor for the class `StringVar`. Suppose we did not include the copy constructor in the definition of the class `StringVar` and suppose we used a call-by-value parameter in a function definition, for example:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
        << the_string << endl;
}
```

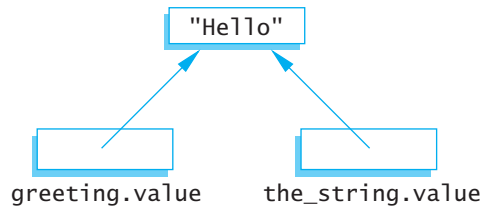
Call-by-value
parameters

Why a copy
constructor is
needed

Consider the following code, which includes a function call:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

Assuming there is no copy constructor, things proceed as follows: When the function call is executed, the value of `greeting` is copied to the local variable `the_string`, so `the_string.value` is set equal to `greeting.value`. But these are pointer variables, so during the function call, `the_string.value` and `greeting.value` point to the same dynamic array, as follows:



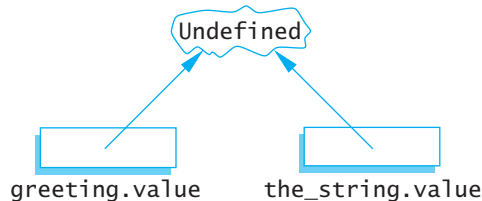
When the function call ends, the destructor for `StringVar` is called to return the memory used by `the_string` to the freestore. The definition of the destructor contains the following statement:

```
delete [] value;
```

Since the destructor is called with the object `the_string`, this statement is equivalent to:

```
delete [] the_string.value;
```

which changes the picture to the following:



Since `greeting.value` and `the_string.value` point to the same dynamic array, deleting `the_string.value` is the same as deleting `greeting.value`. Thus, `greeting.value` is undefined when the program reaches the statement

```
cout << "After call: " << greeting << endl;
```

This `cout` statement is therefore undefined. The `cout` statement may by chance give you the output you want, but sooner or later the fact that `greeting.value` is undefined will produce problems. One major problem occurs when the object `greeting` is a local variable in some function. In this case the destructor will be called with `greeting` when the function call ends. That destructor call will be equivalent to

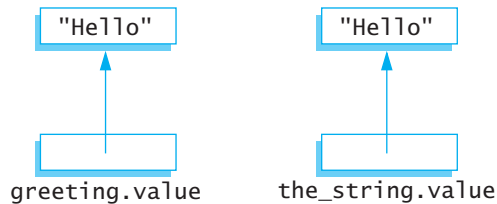
```
delete [] greeting.value;
```

But, as we just saw, the dynamic array pointed to by `greeting.value` has already been deleted once, and now the system is trying to delete it a second time. Calling *delete* twice to delete the same dynamic array (or other variable created with *new*) can produce a serious system error that can cause your program to crash.

That was what would happen if there were no copy constructor. Fortunately, we included a copy constructor in our definition of the class `StringVar`, so the copy constructor is called automatically when the following function call is executed:

```
StringVar greeting("Hello");
show_string(greeting);
```

The copy constructor defines what it means to “plug in” the argument `greeting` for the call-by-value parameter `the_string`, so that now the picture is as follows:



Thus, any change that is made to `the_string.value` has no effect on the argument `greeting`, and there are no problems with the destructor. If the destructor is called for `the_string` and then called for `greeting`, each call to the destructor deletes a different dynamic array.

Returned value

When a function returns a value of a class type, the copy constructor is called automatically to copy the value specified by the return statement. If there is no copy constructor, then problems similar to what we described for value parameters will occur.

When you need a copy constructor

If a class definition involves pointers and dynamically allocated memory using the *new* operator, then you need to include a copy constructor. Classes that do not involve pointers or dynamically allocated memory do not need a copy constructor.

Assignment statements

Contrary to what you might expect, the copy constructor is *not* called when you set one object equal to another using the assignment operator.² However, if you do not like what the default assignment operator does, you can redefine the assignment operator in the way described in the subsection entitled “Overloading the Assignment Operator.”

² C++ makes a careful distinction between initialization (the three cases where the copy constructor is called) and assignment. Initialization uses the copy constructor to create a new object; the assignment operator takes an existing object and modifies it so that it is an identical copy (in all but location) of the right-hand side of the assignment.

Copy Constructor

A **copy constructor** is a constructor that has one call-by-reference parameter that is of the same type as the class. The one parameter must be a call-by-reference parameter. Normally, the parameter is also a constant parameter, that is, preceded by the *const* parameter modifier. The copy constructor for a class is called automatically whenever a function returns a value of the class type. The copy constructor is also called automatically whenever an argument is “plugged in” for a call-by-value parameter of the class type. A copy constructor can also be used in the same ways as other constructors.

Any class that uses pointers and the *new* operator should have a copy constructor.

The Big Three

The **copy constructor**, the **=operator**, and the **destructor** are called the **big three** because experts say that if you need to define any of them, then you need to define all three. If any of these is missing, the compiler will create it, but it may not behave as you want. So it pays to define them yourself. The copy constructor and overloaded =operator that the compiler generates for you will work fine if all member variables are of predefined types such as *int* and *double*, but they may misbehave on classes that have class or pointer member variables. For any class that uses pointers and the *new* operator, it is safest to define your own copy constructor, overloaded =, and destructor.

= must be a
member

SELF-TEST EXERCISES

23. If a class is named `MyClass` and it has a constructor, what is the constructor named? If `MyClass` has a destructor, what is the destructor named?
24. Suppose you change the definition of the destructor in Display 11.12 to the following. How would the sample dialogue in Display 11.11 change?

```
StringVar::~~StringVar()
{
    cout << endl
        << "Good-bye cruel world! The short life of\n"
        << "this dynamic array is about to end.\n";
    delete [] value;
}
```

25. The following is the first line of the copy constructor definition for the class `StringVar`. The identifier `StringVar` occurs three times and means something slightly different each time. What does it mean in each of the three cases?

```
StringVar::StringVar(const StringVar& string_object)
```

26. Answer these questions about destructors.
- What is a destructor and what must the name of a destructor be?
 - When is a destructor called?
 - What does a destructor actually do?
 - What should a destructor do?

Overloading the Assignment Operator

Suppose `string1` and `string2` are declared as follows:

```
StringVar string1(10), string2(20);
```

The class `StringVar` was defined in Displays 11.11 and 11.12. If `string2` has somehow been given a value, then the following assignment statement is defined, but its meaning may not be what you would like it to be:

```
string1 = string2;
```

As usual, this predefined version of the assignment statement copies the value of each of the member variables of `string2` to the corresponding member variables of `string1`, so the value of `string1.max_length` is changed to be the same as `string2.max_length` and the value of `string1.value` is changed to be the same as `string2.value`. But this can cause problems with `string1` and probably even cause problems for `string2`.

The member variable `string1.value` contains a pointer, and the assignment statement sets this pointer equal to the same value as `string2.value`. Thus, both `string1.value` and `string2.value` point to the same place in memory. If you change the string value in `string1`, you will therefore also change the string value in `string2`. If you change the string value in `string2`, you will change the string value in `string1`.

In short, the predefined assignment statement does not do what we would like an assignment statement to do with objects of type `StringVar`. Using the predefined version of the assignment operator with the class `StringVar` can only cause problems. The way to fix this is to overload the assignment operator = so that it does what we want it to do with objects of the class `StringVar`.

The assignment operator cannot be overloaded in the way we have overloaded other operators, such as `<<` and `+`. When you overload the assignment operator, it must be a member of the class; it cannot be a friend of the class. To add an overloaded version of the assignment operator to the class `StringVar`, the definition of `StringVar` should be changed to the following:



VideoNote
Overloading = and == for
a Class

```

class StringVar
{
public:
    void operator =(const StringVar& right_side);
    //Overloads the assignment operator = to copy a string
    //from one object to another.
    <The rest of the definition of the class can be the same as in
    Display 11.11.>

```

The assignment operator is then used just as you always use the assignment operator. For example, consider the following:

```
string1 = string2;
```

In this call, string1 is the calling object and string2 is the argument to the member operator =.

Calling object
for =

The definition of the assignment operator can be as follows:

```

//The following is acceptable, but
//we will give a better definition:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if ((new_length) > max_length)
        new_length = max_length;

    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}

```

Notice that the length of the string in the object on the right side of the assignment operator is checked. If it is too long to fit in the object on the left side of the assignment operator (which is the calling object), then only as many characters as will fit are copied to the object receiving the string. But suppose you do not want to lose any characters in the copying process. To fit in all the characters, you can create a new, larger dynamic array for the object on the left-hand side of the assignment operator. You might try to redefine the assignment operator as follows:

```

//This version has a bug:
void StringVar::operator =(const StringVar& right_side)
{
    delete [] value;
    int new_length = strlen(right_side.value);
    max_length = new_length;
    value = new char[max_length + 1];
    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}

```

This version has a problem when used in an assignment with the same object on both sides of the assignment operator, like the following:

```
my_string = my_string;
```

When this assignment is executed, the first statement executed is

```
delete [] value;
```

But the calling object is `my_string`, so this means

```
delete [] my_string.value;
```

So, the string value in `my_string` is deleted and the pointer `my_string.value` is undefined. The assignment operator has corrupted the object `my_string`, and this run of the program is probably ruined.

One way to fix this bug is to first check whether there is sufficient room in the dynamic array member of the object on the left-hand side of the assignment operator and to delete the array only if extra space is needed. Our final definition of the overloaded assignment operator does just such a check:

```
//This is our final version:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if (new_length > max_length)
    {
        delete [] value;
        max_length = new_length;
        value = new char[max_length + 1];
    }
    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

For many classes, the obvious definition for overloading the assignment operator does not work correctly when the same object is on both sides of the assignment operator. You should always check this case and be careful to write your definition of the overloaded assignment operator so that it also works in this case.

SELF-TEST EXERCISE

27. a. Explain carefully why no overloaded assignment operator is needed when the only data consists of built-in types.
- b. Same as part (a) for a copy constructor.
- c. Same as part (a) for a destructor.

CHAPTER SUMMARY

- A **friend** function of a class is an ordinary function except that it has access to the private members of the class, just like the member functions do.
- If your classes each have a full set of accessor and mutator functions, then the only reason to make a function a friend is to make the definition of the friend function simpler and more efficient, but that is often reason enough.
- A parameter of a class type that is not changed by the function should normally be a constant parameter.
- Operators, such as + and ==, can be overloaded so they can be used with objects of a class type that you define.
- When overloading the >> or << operators, the type returned should be a stream type and must be a reference, which is indicated by appending an & to the name of the returned type.
- The base type of an array can be a structure or class type. A structure or class can have an array as a member variable.
- A **destructor** is a special kind of member function for a class. A destructor is called automatically when an object of the class passes out of scope. The main reason for destructors is to return memory to the freestore so the memory can be reused.
- A **copy constructor** is a constructor that has a single argument that is of the same type as the class. If you define a copy constructor, it will be called automatically whenever a function returns a value of the class type and whenever an argument is “plugged in” for a call-by-value parameter of the class type. Any class that uses pointers and the operator *new* should have a copy constructor.
- The assignment operator = can be overloaded for a class so that it behaves as you wish for that class. However, it must be overloaded as a member of the class; it cannot be overloaded as a friend. Any class that uses pointers and the operator *new* should overload the assignment operator for use with that class.

Answers to Self-Test Exercises

1.

```
bool before(DayOfYear date1, DayOfYear date2)
{
    return ((date1.get_month() < date2.get_month())
        || (date1.get_month() == date2.get_month()
        && date1.get_day ( ) < date2.get_day ( )));
}
```

The previous Boolean expression says that `date1` is before `date2`, provided the month of `date1` is before the month of `date2` or that the months are the same and the day of `date1` is before the day of `date2`.

2. A friend function and a member function are alike in that they both can use any member of the class (either public or private) in their function definition. However, a friend function is defined and used just like an ordinary function; the dot operator is not used when you call a friend function, and no type qualifier is used when you define a friend function. A member function, on the other hand, is called using an object name and the dot operator. Also, a member function definition includes a type qualifier consisting of the class name and the scope resolution operator `::`.
3. The modified definition of the class `DayOfYear` is shown below. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 11.2 should be included in this definition.

```
class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    friend bool after(DayOfYear date1, DayOfYear date2);
    //Precondition: date1 and date2 have values.
    //Returns true if date1 follows date2 on the calendar;
    //otherwise, returns false.

    DayOfYear(int the_month, int the_day);
    DayOfYear();
    void input();
    void output();
    int get_month();
    int get_day();
private:
    void check_date( );
    int month;
    int day;
};
```

You also must add the following definition of the function `after`:

```
bool after(DayOfYear date1, DayOfYear date2)
{
    return ((date1.month > date2.month) ||
            ((date1.month == date2.month) && (date1.day > date2.day)))
}
```

4. The modified definition of the class `Money` is shown here. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 11.3 should be included in this definition.

```

class Money
{
public:
    friend Money subtract(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have values.
    //Returns amount1 minus amount2.

    friend bool equal(Money amount1, Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value();
    void input(istream& ins);
    void output(ostream& outs);
private:
    long all_cents;
};

```

You also must add the following definition of the function subtract:

```

Money subtract(Money amount1, Money amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents
                    - amount2.all_cents;
    return temp;
}

```

5. The modified definition of the class Money is shown here. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 11.3 should be included in this definition.

```

class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    friend bool equal(Money amount1, Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value();
    void input(istream& ins);

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: A dollar sign and the amount of money
    //recorded in the calling object has been sent to the
    //output stream outs.

```

```

void output();
//Postcondition: A dollar sign and the amount of money
//recorded in the calling object has been output to the
//screen.
private:
    long all_cents;
};

```

You also must add the following definition of the function name `output`. (The old definition of `output` stays, so that there are two definitions of `output`.)

```

void Money::output()
{
    output(cout);
}

```

The following longer version of the function definition also works:

```

//Uses cstdlib and iostream
void Money::output()
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        cout << "-$" << dollars << '.';
    else
        cout << "$" << dollars << '.';

    if (cents < 10)
        cout << '0';
    cout << cents;
}

```

You can also overload the member function `input` so that a call like

```
purse.input();
```

means the same as

```
purse.input(cin);
```

And, of course, you can combine this enhancement with the enhancements from previous Self-Test Exercises to produce one highly improved class `Money`.

6. If the user enters `$-9.95` (instead of `-$9.95`), the function `input` will read the `'$'` as the value of `one_char`, the `-9` as the value of `dollars`, the `'.'` as the value of `decimal_point`, and the `'9'` and `'5'` as the values of `digit1` and `digit2`. That means it will set `dollars` equal to `-9` and `cents` equal

to 95 and so set the amount equal to a value that represents $-\$9.00$ plus 0.95, which is $-\$8.05$. One way to catch this problem is to test if the value of `dollars` is negative (since the value of `dollars` should be an absolute value). To do this, rewrite the error message portion as follows:

```
if (one_char != '$' || decimal_point != '.'
    || !isdigit(digit1) || !isdigit(digit2)
    || dollars < 0) ← New
{
    cout << "Error illegal form for money input\n";
    exit(1);
}
```

This code still will not give an error message for incorrect input with zero dollars as in $\$-0.95$. However, with the material we have learned thus far, a test for this case, while certainly possible, would significantly complicate the code and make it harder to read.

```
7. #include <iostream>
   using namespace std;
   int main( )
   {
       int x;
       cin >> x;
       cout << x << endl;
       return 0;
   }
```

If the compiler interprets input with a leading 0 as a base-8 numeral, then with input data 077, the output should be 63. The output should be 77 if the compiler does not interpret data with a leading 0 as indicating base 8.

8. The only change from the version given in Display 11.3 is that the modifier *const* is added to the function heading, so the definition is

```
double Money::get_value() const
{
    return (all_cents * 0.01);
}
```

9. The member function `input` changes the value of its calling object, and so the compiler will issue an error message if you add the *const* modifier.
10. *Similarities*: Each parameter call protects the caller's argument from change. *Differences*: The call-by-value makes a copy of the caller's argument, so it uses more memory than a call-by-constant-reference.
11. In the *const* `int x = 17;` declaration, the *const* keyword promises the compiler that code written by the author will not change the value of `x`.

In the `int f() const;` declaration, the `const` keyword is a promise to the compiler that code written by the author to implement function `f` will not change anything in the calling object.

In the `int g(const A& x);` declaration, the `const` keyword is a promise to the compiler that code written by the class author will not change the argument plugged in for `x`.

12. The difference between a (binary) operator (such as `+`, `*`, `/`, and so forth) and a function involves the syntax of how they are called. In a function call, the arguments are given in parentheses after the function name. With an operator, the arguments are given before and after the operator. Also, you must use the reserved word *operator* in the declaration and in the definition of an overloaded operator.
13. The modified definition of the class `Money` is shown here. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 11.5 should be included in this definition.

```
class Money
{
public:
    friend Money operator +(const Money& amount1,
                           const Money& amount2);
    friend bool operator ==(const Money& amount1,
                           const Money& amount2);
    friend bool operator <(const Money& amount1,
                           const Money& amount2);
    //Precondition: amount1 and amount2 have been given
    //values.
    //Returns true if amount1 is less than amount2;
    //otherwise, returns false.
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value() const;
    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};
```

You also must add the following definition of the overloaded operator `<`:

```
bool operator <(const Money& amount1,
               const Money& amount2)
{
    return (amount1.all_cents < amount2.all_cents);
}
```

14. The modified definition of the class `Money` is shown here. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 11.5 should be included in this definition. We have included the changes from the previous exercises in this answer, since it is natural to use the overloaded `<` operator in the definition of the overloaded `<=` operator.

```
class Money
{
public:
    friend Money operator +(const Money& amount1,
                           const Money& amount2);
    friend bool operator ==(const Money& amount1,
                           const Money& amount2);
    friend bool operator <(const Money& amount1,
                           const Money& amount2);
    //Precondition: amount1 and amount2 have been given
    //values.
    //Returns true if amount1 is less than amount2;
    //otherwise, returns false.
    friend bool operator <=(const Money& amount1,
                           const Money& amount2);
    //Precondition: amount1 and amount2 have been given
    //values.
    //Returns true if amount1 is less than or equal to
    //amount2; otherwise, returns false.
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value() const;
    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};
```

You also must add the following definition of the overloaded operator `<=` (as well as the definition of the overloaded operator `<` given in the previous exercise):

```
bool operator <=(const Money& amount1,
                 const Money& amount2)
{
    return ((amount1.all_cents < amount2.all_cents)
           || (amount1.all_cents == amount2.all_cents));
}
```

15. When overloading an operator, at least one of the arguments to the operator must be of a class type. This prevents changing the behavior of `+` for integers. Actually, this requirement prevents changing the effect of any operator on any built-in type.

```

16. //Uses cmath (for floor):
    Money::Money(double amount)
    {
        all_cents = floor(amount * 100);
    }

```

This definition simply discards any amount that is less than one cent. For example, it converts 12.34999 to the integer 1234, which represents the amount \$12.34. It is possible to define the constructor to instead do other things with any fraction of a cent.

```

17. istream& operator >>(istream& ins, Pairs& second)
    {
        char ch;
        ins >> ch;          //discard initial '('
        ins >> second.f;
        ins >> ch;          //discard comma ','
        ins >> second.s;
        ins >> ch;          //discard final ')'
        return ins;
    }

    ostream& operator <<(ostream& outs, const Pairs& second)
    {
        outs << '(';
        outs << second.f;
        outs << ','; //You might prefer ", "
                   //to get an extra space
        outs << second.s;
        outs << ')';
        return outs;
    }

18. //Uses iostream:
    istream& operator >>(istream& ins, Percent& the_object)
    {
        char percent_sign;
        ins >> the_object.value;
        ins >> percent_sign; //Discards the % sign.
        return ins;
    }

    //Uses iostream:
    ostream& operator <<(ostream& outs,
        const Percent& a_percent)
    {
        outs << a_percent.value << '%';
        return outs;
    }

```


19. `struct` Score
- ```
{
 int home_team;
 int opponent;
};
Score game[10];
```
20. *//Reads in 5 amounts of money, doubles each amount,  
//and outputs the results.*  
`#include <iostream>`

<The definitions for the Money class go here>

```
int main()
{
 using namespace std;
 Money amount[5];
 int i;
 cout << "Enter 5 amounts of money:\n";
 for (i = 0; i < 5; i++)
 cin >> amount[i];
 for (i = 0; i < 5; i++)
 amount[i] = amount[i] + amount[i];
 cout << "After doubling, the amounts are:\n";
 for (i = 0; i < 5; i++)
 cout << amount[i] << " ";
 cout << endl;
 return 0;
}
```

(You cannot use `2 * amount[i]`, since `*` has not been overloaded for operands of type `Money`.)

21. See answer 22.
22. This answer combines the answers to this and the previous Self-Test Exercise. The class definition would change to the following. (We have deleted some comments from Display 11.10 to save space, but you should include them in your answer.)

```
class TemperatureList
{
public:
 TemperatureList();

 int get_size() const;
 //Returns the number of temperatures on the list.

 void add_temperature(double temperature);

 double get_temperature(int position) const;
```

```

 //Precondition: 0 <= position < get_size().
 //Returns the temperature that was added in position
 //specified. The first temperature that was added is
 //in position 0.

 bool full() const;

 friend ostream& operator <<(ostream& outs,
 const TemperatureList& the_object);
 private:
 double list[MAX_LIST_SIZE]; //of temperatures in
 //Fahrenheit
 int size; //number of array positions filled
};

```

You also need to add the following member function definitions:

```

int TemperatureList::get_size() const
{
 return size;
}

//Uses iostream and cstdlib:
double TemperatureList::get_temperature (int position) const
{
 if ((position >= size) || (position < 0))
 {
 cout << "Error:"
 << " reading an empty list position.\n";
 exit(1);
 }
 else
 return (list[position]);
}

```

23. The *constructor* is named `MyClass`, the same name as the name of the class. The *destructor* is named `~MyClass`.
24. The dialogue would change to the following:

```

What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
Good-bye cruel world! The short life of
this dynamic array is about to end.
Good-bye cruel world! The short life of
this dynamic array is about to end.
End of demonstration

```

25. The `StringVar` before the `::` is the name of the class. The `StringVar` right after the `::` is the name of the member function. (Remember, a constructor is a member function that has the same name as the class.) The `StringVar` inside the parentheses is the type for the parameter `string_object`.
26.
  - a. A destructor is a member function of a class. A destructor's name always begins with a tilde, `~`, followed by the class name.
  - b. A destructor is called when a class object goes out of scope.
  - c. A destructor actually does whatever the class author programs it to do!
  - d. A destructor is supposed to delete dynamic variables that have been allocated by constructors for the class. Destructors may also do other cleanup tasks.
27. In the case of the assignment operator `=` and the copy constructor, if there are only built-in types for data, the default copy mechanism is exactly what you want, so the default works fine. In the case of the destructor, no dynamic memory allocation is done (no pointers), so the default do-nothing action is again what you want.

## PRACTICE PROGRAMS

*Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.*

1. Modify the definition of the class `Money` shown in Display 11.8 so that all of the following are added:
  - a. The operators `<`, `<=`, `>`, and `>=` have each been overloaded to apply to the type `Money`. (*Hint: See Self-Test Exercise 13.*)
  - b. The following member function has been added to the class definition. (We show the function declaration as it should appear in the class definition. The definition of the function itself will include the qualifier `Money::`.)

```
Money percent(int percent_figure) const;
//Returns a percentage of the money amount in the
//calling object. For example, if percent_figure is 10,
//then the value returned is 10% of the amount of
//money represented by the calling object.
```

For example, if `purse` is an object of type `Money` whose value represents the amount \$100.10, then the call

```
purse.percent(10);
```

returns 10% of \$100.10; that is, it returns a value of type `Money` that represents the amount \$10.01.

2. Self-Test Exercise 17 asked you to overload the operator `>>` and the operator `<<` for a class `Pairs`. Complete and test this exercise. Implement the default constructor and the constructors with one and two `int` parameters. The one-parameter constructor should initialize the first member of the pair; the second member of the pair is to be 0.

Overload binary operator `+` to add pairs according to the rule

$$(a, b) + (c, d) = (a + c, b + d)$$

Overload operator `-` analogously.

Overload operator `*` on `Pairs` and `int` according to the rule

$$(a, b) * c = (a * c, b * c)$$

Write a program to test all the member functions and overloaded operators in your class definition.

3. Self-Test Exercise 18 asked you to overload the operator `>>` and the operator `<<` for a class `Percent`. Complete and test this exercise. Implement the default constructor and the constructor with one `int` parameter. Overload the `+` and `-` operators to add and subtract percents. Also, overload the `*` operator to allow multiplication of a percent by an integer.

Write a program to test all the member functions and overloaded operators in your class definition.

## PROGRAMMING PROJECTS

*Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit [www.myprogramminglab.com](http://www.myprogramminglab.com) to complete many of these Programming Projects online and get instant feedback.*

1. In Chapter 8 we discussed vectors, which are like arrays that can grow in size. Suppose that vectors were not defined in C++. Define a class called `VectorDouble` that is like a class for a vector with base type `double`. Your class `VectorDouble` will have a private member variable for a dynamic array of `doubles`. It will also have two member variables of type `int`; one called `max_count` for the size of the dynamic array of `doubles`; and one called `count` for the number of array positions currently holding values. (`max_count` is the same as the capacity of a vector; `count` is the same as the size of a vector.)

If you attempt to add an element (a value of type `double`) to the vector object of the class `VectorDouble` and there is no more room, then a new dynamic array with twice the capacity of the old dynamic array is created and the values of the old dynamic array are copied to the new dynamic array.

Your class should have all of the following:

- Three constructors: a default constructor that creates a dynamic array for 50 elements, a constructor with one *int* argument for the number of elements in the initial dynamic array, and a copy constructor.
- A destructor.
- A suitable overloading of the assignment operator `=`.
- A suitable overloading of the equality operator `==`. To be equal, the values of `count` and the count array elements must be equal, but the values of `max_count` need not be equal.
- Member functions `push_back`, `capacity`, `size`, `reserve`, and `resize` that behave the same as the member functions of the same names for vectors.
- Two member functions to give your class the same utility as the square brackets: `value_at(i)`, which returns the value of the *i*th element in the dynamic array; and `change_value_at(d, i)`, which changes the *double* value at the *i*th element of the dynamic array to *d*. Enforce suitable restrictions on the arguments to `value_at` and `change_value_at`. (Your class will not work with the square brackets. It can be made to work with square brackets, but we have not covered the material which tells you how to do that.)

2. Define a class for rational numbers. A rational number is a number that can be represented as the quotient of two integers. For example,  $1/2$ ,  $3/4$ ,  $64/2$ , and so forth are all rational numbers. (By  $1/2$ , etc., we mean the everyday meaning of the fraction, not the integer division this expression would produce in a C++ program.) Represent rational numbers as two values of type *int*, one for the numerator and one for the denominator. Call the class `Rational`.

Include a constructor with two arguments that can be used to set the member variables of an object to any legitimate values. Also include a constructor that has only a single parameter of type *int*; call this single parameter `whole_number` and define the constructor so that the object will be initialized to the rational number `whole_number/1`. Also include a default constructor that initializes an object to 0 (that is, to  $0/1$ ).

Overload the input and output operators `>>` and `<<`. Numbers are to be input and output in the form  $1/2$ ,  $15/32$ ,  $300/401$ , and so forth. Note that the numerator, the denominator, or both may contain a minus sign, so  $-1/2$ ,  $15/32$ , and  $-300/-401$  are also possible inputs. Overload all of the following operators so that they correctly apply to the type `Rational`: `==`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, and `/`. Also write a test program to test your class.

(Hints: Two rational numbers  $a/b$  and  $c/d$  are equal if  $a*d$  equals  $c*b$ . If  $b$  and  $d$  are *positive* rational numbers,  $a/b$  is less than  $c/d$  provided  $a*d$  is less than  $c*b$ . You should include a function to normalize the values stored so that, after normalization, the denominator is positive and the numerator and denominator are as small as possible. For example, after normalization  $4/-8$  would be represented the same as  $-1/2$ . You should also write a test program to test your class.)

3. Define a class for complex numbers. A complex number is a number of the form

$$a + b * i$$

where, for our purposes,  $a$  and  $b$  are numbers of type *double*, and  $i$  is a number that represents the quantity  $\sqrt{-1}$ . Represent a complex number as two values of type *double*. Name the member variables *real* and *imaginary*. (The variable for the number that is multiplied by  $i$  is the one called *imaginary*.) Call the class *Complex*.

Include a constructor with two parameters of type *double* that can be used to set the member variables of an object to any values. Also include a constructor that has only a single parameter of type *double*; call this parameter *real\_part* and define the constructor so that the object will be initialized to *real\_part*+0\* $i$ . Also include a default constructor that initializes an object to 0 (that is, to 0+0\* $i$ ). Overload all of the following operators so that they correctly apply to the type *Complex*: `==`, `+`, `-`, `*`, `>>`, and `<<`. You should write a test program to test your class.

(Hints: To add or subtract two complex numbers, you add or subtract the two member variables of type *double*. The product of two complex numbers is given by the following formula:

$$(a + b*i)*(c + d*i) == (a*c - b*d) + (a*d + b*c)*i$$

In the interface file, you should define a constant  $i$  as follows:

```
const Complex i(0, 1);
```

This defined constant  $i$  will be the same as the  $i$  discussed earlier.

```
delete p;
```

4. Enhance the definition of the class *StringVar* given in Displays 11.11 and 11.12 by adding all of the following:

- Member function *copy\_piece*, which returns a specified substring;
- member function *one\_char*, which returns a specified single character; and
- member function *set\_char*, which changes a specified character

- An overloaded version of the `==` operator (note that only the string values have to be equal; the values of `max_length` need not be the same)
- An overloaded version of `+` that performs concatenation of strings of type `StringVar`
- An overloaded version of the extraction operator `>>` that reads one word (as opposed to `input_line`, which reads a whole line)

If you did the section on overloading the assignment operator, then add it as well. Also write a suitable test program and thoroughly test your class definition.

5. Define a class called `Text` whose objects store lists of words. The class `Text` will be just like the class `StringVar` except that the class `Text` will use a dynamic array with base type `StringVar` rather than base type `char` and will mark the end of the array with a `StringVar` object consisting of a single blank, rather than using `'\0'` as the end marker. Intuitively, an object of the class `Text` represents some text consisting of words separated by blanks. Enforce the restriction that the array elements of type `StringVar` contain no blanks (except for the end marker elements of type `StringVar`).

Your class `Text` will have member functions corresponding to all the member functions of `StringVar`. The constructor with an argument of type `const char a[]` will initialize the `Text` object in the same way as described below for `input_line`. If the C-string argument contains the new-line symbol `'\n'`, that is considered an error and ends the program with an error message.

The member function `input_line` will read blank separated strings and store each string in one element of the dynamic array with base type `StringVar`. Multiple blank spaces are treated the same as a single blank space. When outputting an object of the class `Text`, insert one blank between each value of type `StringVar`. You may either assume that no tab symbols are used or you can treat the tab symbols the same as a blank; if this is a class assignment, ask your instructor how you should treat the tab symbol.

Add the enhancements described in Programming Project 6. The overloaded version of the extraction operator `>>` will fill only one element of the dynamic array.

6. Using dynamic arrays, implement a polynomial class with polynomial addition, subtraction, and multiplication.

*Discussion:* A variable in a polynomial does very little other than act as a placeholder for the coefficients. Hence, the only interesting thing

about polynomials is the array of coefficients and the corresponding exponent. Think about the polynomial

$$x^3 + x + 1$$

One simple way to implement the polynomial class is to use an array of *doubles* to store the coefficients. The index of the array is the exponent of the corresponding term. Where is the term in  $x^3$  in the previous example? If a term is missing, then it simply has a zero coefficient.

There are techniques for representing polynomials of high degree with many missing terms. These use so-called sparse polynomial techniques. Unless you already know these techniques, or learn very quickly, don't use them.

Provide a default constructor, a copy constructor, and a parameterized constructor that enable an arbitrary polynomial to be constructed. Also supply an overloaded operator `=` and a destructor.

Provide these operations:

- polynomial + polynomial
- constant + polynomial
- polynomial + constant
- polynomial - polynomial
- constant - polynomial
- polynomial - constant
- polynomial \* polynomial
- constant \* polynomial
- polynomial \* constant

Supply functions to assign and extract coefficients, indexed by exponent.

Supply a function to evaluate the polynomial at a value of type *double*.

You should decide whether to implement these functions as members, friends, or stand-alone functions.

7. Write a checkbook balancing program. The program will read in the following for all checks that were not cashed as of the last time you balanced your checkbook: the number of each check, the amount of the check, and whether or not it has been cashed. Use an array with a class base type. The class should be a class for a check. There should be three member variables to record the check number, the check amount, and whether or not the check was cashed. The class for a check will have a member variable of type *Money* (as defined in Display 19) to record the check amount. So, you will have a class used within



a class. The class for a check should have accessor and mutator functions as well as constructors and functions for both input and output of a check.

In addition to the checks, the program also reads all the deposits, as well as the old and the new account balance. You may want another array to hold the deposits. The new account balance should be the old balance plus all deposits, minus all checks that have been cashed.

The program outputs the total of the checks cashed, the total of the deposits, what the new balance should be, and how much this figure differs from what the bank says the new balance is. It also outputs two lists of checks: the checks cashed since the last time you balanced your checkbook and the checks still not cashed. Display both lists of checks in sorted order from lowest to highest check number.

If this is a class assignment, ask your instructor if input/output should be done with the keyboard and screen or if it should be done with files. If it is to be done with files, ask your instructor for instructions on file names.

8. Define a class called `List` that can hold a list of values of type *double*. Model your class definition after the class `TemperatureList` given in Display 11.10, but your class `List` will make no reference to temperatures when it outputs values. The values may represent any sort of data items as long as they are of type *double*. Include the additional features specified in Self-Test Exercises 21 and 22. Change the member function names so that they do not refer to temperature.

Add a member function called `get_last` that takes no arguments and returns the last item on the list. The member function `get_last` does not change the list, and it should not be called if the list is empty. Add another member function called `delete_last` that deletes the last element on the list. The member function `delete_last` is a *void* function. Note that when the last element is deleted, the member variable `size` must be adjusted. If `delete_last` is called with an empty list as the calling object, the function call has no effect. Design a program to thoroughly test your definition for the class `List`.

9. Define a class called `StringSet` that will be used to store a set of STL strings. Use an array or a vector to store the strings. Create a constructor that takes as an input parameter an array of strings for the initial values in the set. Then write member functions to add a string to the set, remove a string from the set, clear the entire set, return the number of strings in the set, and output all strings in the set. Overload the `+` operator so that it returns the union of two `StringSet` objects. Also overload the `*` operator so that it returns the intersection of two `StringSet` objects. Write a program to test all member functions and overloaded operators in your class.

10. This programming project requires you to complete Programming Project 9 first.

The field of information retrieval is concerned with finding relevant electronic documents based upon a query. For example, given a group of keywords (the query), a search engine retrieves Web pages (documents) and displays them sorted by relevance to the query. This technology requires a way to compare a document with the query to see which is most relevant to the query.

A simple way to make this comparison is to compute the binary cosine coefficient. The coefficient is a value between 0 and 1, where 1 indicates that the query is very similar to the document and 0 indicates that the query has no keywords in common with the document. This approach treats each document as a set of words. For example, given the following sample document:

"Chocolate ice cream, chocolate milk, and chocolate bars are delicious."

This document would be parsed into keywords where case is ignored and punctuation discarded and turned into the set containing the words {chocolate, ice, cream, milk, and, bars, are, delicious}. An identical process is performed on the query to turn it into a set of strings. Once we have a query  $Q$  represented as a set of words and a document  $D$  represented as a set of words, the similarity between  $Q$  and  $D$  is computed by:

$$Sim = \frac{|Q \cap D|}{\sqrt{|Q|} \sqrt{|D|}}$$

Modify the `StringSet` from Programming Project 12 by adding an additional member function that computes the similarity between the current `StringSet` and an input parameter of type `StringSet`. The `sqr` function is in the `cmath` library.

Create two text files on your disk named `Document1.txt` and `Document2.txt`. Write some text content of your choice in each file, but make sure that each file contains different content. Next, write a program that allows the user to input from the keyboard a set of strings that represents a query. The program should then compare the query to both text files on the disk and output the similarity to each one using the binary cosine coefficient. Test your program with different queries to see if the similarity metric is working correctly.

11. Redo Programming Project 6 from Chapter 9 (or do it for the first time), but this time encapsulate the dynamic array and array size within a class. The class should have public member functions `addEntry` and `deleteEntry`. Make the array and size variables private. This will require

adding functions for getting and setting specific items in the array as well as returning the current size of the array. Add a destructor that frees up the memory allocated to the dynamic array. Also, add a copy constructor and overload the assignment operator so that the dynamic array is properly copied from the object on the right-hand side of the assignment to the object on the left-hand side. Embed your class in a suitable test program.

12. To combat election fraud, your city is instituting a new voting procedure. The ballot has a letter associated with every selection a voter may make. A sample ballot is shown.



VideoNote  
Solution to Programming  
Project 11.12

1. VOTE FOR MAYOR
  - A. Pincher, Penny ☐
  - B. Dover, Skip ☐
  - C. Perman, Sue ☐
2. PROPOSITION 17
  - D. YES ☐
  - E. NO ☐
3. MEASURE 1
  - F. YES ☐
  - G. NO ☐
4. MEASURE 2
  - H. YES ☐
  - I. NO ☐

After submitting the ballot, every voter receives a receipt that has a unique ID number and a record of the voting selections. For example, a voter who submits a ballot for Sue Perman, Yes on Proposition 17, No on Measure 1, and Yes on Measure 2 might receive a receipt with

ID 4925 : CDGH

The next day the city posts all votes on its Web page sorted by ID number. This allows a voter to confirm their submission and allows anyone to count the vote totals for themselves. A sample list for the sample ballot is shown.

| ID   | VOTES |
|------|-------|
| 4925 | CDGH  |
| 4926 | AEGH  |
| 4927 | CDGI  |
| 4928 | BEGI  |
| 4929 | ADFH  |

Write a program that reads the posted voting list from a file and outputs the percent of votes cast for each ballot item. You may assume that the file does not have any header lines. The first line will contain a voter ID and a string representing votes. Define a class named `Voter` that stores an individual's voting record. The class should have a constructor that takes as input a string of votes (for example, "CDGH"), a voter ID, and accessor function(s) that return the person's ID and vote for a specific question. Store each `Voter` instance in an array or vector. Your program should iterate over the array to compute and output the percent of votes cast for each candidate, proposition, and measure. It should then prompt the user to enter a voter ID, iterate over the list again to find the object with that ID, and print his or her votes.

13. Repeat Programming Project 11 from Chapter 10 but use an array to store the movie ratings instead of separate variables. All changes should be internal to the class so the main function to test the class should run identically with either the old `Movie` class or the new `Movie` class that uses an array member variable.

Next, modify the main function so that instead of creating separate variables for each `Movie` object, an array of at least four `Movie` objects is created with sample data. Loop through the array and output the name, MPAA rating, and average rating for each of the four movies.

14. Do Programming Project 16 from Chapter 8 except use a `Racer` class to store information about each race participant. The class should store the racer's name, bib number, finishing position, and all of his or her split times as recorded by the RFID sensors. You can choose appropriate structures to store this information. Include appropriate functions to access or change the racer's information, along with a constructor. Make an array or vector of `Racer` objects to store the entire race results.

The racer's name should come from a separate text file. The information for this file is collected before the race when the participant registers for the event. Listed below is a sample file:

```
100,Bill Rodgers
132,Frank Shorter
182,Joan Benoit
```