

Friends and Overloaded Operators

8.1 Friend Functions 460

Programming Example: An Equality Function 460

Friend Functions 461

Programming Tip: Define Both Accessor Functions and Friend Functions 465

Programming Tip: Use Both Member and Nonmember Functions 469

Programming Example: Money Class (Version 1) 469

Implementation of `digit_to_int` (Optional) 477

Pitfall: Leading Zeros in Number Constants 478

The `const` Parameter Modifier 480

Pitfall: Inconsistent Use of `const` 482

8.2 Overloading Operators 486

Overloading Operators 486

Constructors for Automatic Type Conversion 490

Overloading Unary Operators 492

Overloading `>>` and `<<` 493

Chapter Summary 505

Answers to Self-Test Exercises 506

Programming Projects 514



Friends and Overloaded Operators

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 7.

8.1 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 6, 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 6. In Display 8.1 we repeat this final version of the class `DayOfYear`. In Display 8.1 we 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.

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 8.1 this Boolean expression is used to control an *if-else* statement.

The definition of the function `equal` is straightforward. 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 8.1. 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 8.1. To read the

**Display 8.1 Equality Function (part 1 of 3)**

```
//Program to demonstrate the function equal. The class DayOfYear
//is the same as in Self-Test Exercise 23-24 in Chapter 6.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int the_month, int the_day);
    //Precondition: the_month and the_day form a
    //possible date. Initializes the date according to
    //the arguments.

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

    void input();

    void output();

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

    int get_day();
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};

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

int main()
{
    DayOfYear today, bach_birthday(3, 21);
```

Display 8.1 Equality Function (part 2 of 3)

```

    cout << "Enter today's date:\n";
    today.input();
    cout << "Today's date is ";
    today.output();

    cout << "J. S. Bach's birthday is ";
    bach_birthday.output();

    if ( equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}

bool equal(DayOfYear date1, DayOfYear date2)
{
    return ( date1.get_month() == date2.get_month() &&
            date1.get_day() == date2.get_day() );
}

DayOfYear::DayOfYear(int the_month, int the_day)
    : month(the_month), day(the_day)
{
    check_date();
}

int DayOfYear::get_month()
{
    return month;
}

int DayOfYear::get_day()
{
    return day;
}

```

Omitted function and constructor definitions are as in Chapter 6, Self-Test Exercises 14 and 24, but those details are not needed for what we are doing here.

Display 8.1 Equality Function (*part 3 of 3*)

```
//Uses iostream:
void DayOfYear::input()
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}

//Uses iostream:
void DayOfYear::output()
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

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!
```

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 8.1 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 above definition of `equal` will be legal.

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 8.2 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 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 8.2 does 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.

Friends can access private members.

A friend is not a member.

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

**Display 8.2 Equality Function as a Friend (part 1 of 2)**

```
//Demonstrates the function equal.
//In this version equal is a friend of the class DayOfYear.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    //Precondition: date1 and date2 have values.
    //Returns true if date1 and date2 represent the same date;
    //otherwise, returns false.

    DayOfYear(int the_month, int the_day);
    //Precondition: the_month and the_day form a
    //possible date. Initializes the date according to
    //the arguments.

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

    void input();

    void output();

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

    int get_day();
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

Display 8.2 Equality Function as a Friend (part 2 of 2)

```
int main()
{
    <The main part of the program is the same as in Display 8.1.>
}

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

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 8.1.>

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 8.2. 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 8.2) or not as a friend and use accessor functions (as in Display 8.1). 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 8.1). 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 8.1 (and Display 8.2) 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 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 8.3 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.

long

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 two bytes are used to store the *int* type.¹ The result of the two-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 two bytes usually implement the type *long* in four bytes. Values of type *long* are integers just like the values of the type *int*, except that the four-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 8.3). 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.

input

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 8.3). 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

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

**Display 8.3 Money Class—Version 1 (part 1 of 5)**

```
//Program to demonstrate the class Money.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

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

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

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

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

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

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

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

Display 8.3 Money Class—Version 1 (part 2 of 5)

```

    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 have been sent to the output stream outs.
private:
    long all_cents;
};

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

int main()
{
    Money your_amount, my_amount(10, 9), our_amount;
    cout << "Enter an amount of money: ";
    your_amount.input(cin);
    cout << "Your amount is ";
    your_amount.output(cout);
    cout << endl;
    cout << "My amount is ";
    my_amount.output(cout);
    cout << endl;

    if (equal(your_amount, my_amount))
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";
    our_amount = add(your_amount, my_amount);
    your_amount.output(cout);
    cout << " + ";
    my_amount.output(cout);
    cout << " equals ";
    our_amount.output(cout);
    cout << endl;
    return 0;
}

```

Display 8.3 Money Class—Version 1 (part 3 of 5)

```
Money add(Money amount1, Money amount2)
{
    Money temp;

    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

bool equal(Money amount1, Money amount2)
{
    return (amount1.all_cents == amount2.all_cents);
}

Money::Money(long dollars, int cents)
{
    if(dollars*cents < 0) //If one is negative and one is positive
    {
        cout << "Illegal values for dollars and cents.\n";
        exit(1);
    }
    all_cents = dollars*100 + cents;
}

Money::Money(long dollars) : all_cents(dollars*100)
{
    //Body intentionally blank.
}

Money::Money() : all_cents(0)
{
    //Body intentionally blank.
}

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

Display 8.3 Money Class—Version 1 (part 4 of 5)

```

//Uses iostream, ctype, cstdlib:
void Money::input(istream& ins)
{
    char one_char, decimal_point,
        digit1, digit2; //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if ( one_char != '$' || decimal_point != '.'
        || !isdigit(digit1) || !isdigit(digit2) )
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }
    cents = digit_to_int(digit1)*10 + digit_to_int(digit2);

    all_cents = dollars*100 + cents;
    if (negative)
        all_cents = -all_cents;
}

```

Display 8.3 Money Class—Version 1 (part 5 of 5)

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

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

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

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

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

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*.)

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.

digit_to_int

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 8.3. 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;
```

output

The member function *output* (Display 8.3) 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 8.3 is reproduced below:

```
int digit_to_int(char c)
{
    return ( int(c) - 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 5, 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 `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, `int('3')` is not 3, but is some other number. We need to convert `int(c)` to the number corresponding to `c` (for example, '3' to 3). So let's see how we must adjust `int(c)` to get the number we want.

We know that the digits are in order. So `int('0') + 1` is equal to `int('1')`; `int('1') + 1` is equal to `int('2')`; `int('2') + 1` is equal to `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

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

which is

$$\text{int}('0') - \text{int}('0')$$

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

Now let's consider what happens when `c` has the value `'1'`. The value returned is then `int(c) - int('0')`, which is `int('1') - int('0')`. That equals `(int('0') + 1) - int('0')`, and that, in turn, equals `int('0') - int('0') + 1`. Since `int('0') - 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 8.3:

```
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 8.2. 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 8.2?

- 4 Suppose you wish to add a friend function for subtraction to the class `Money` defined in Display 8.3. What do you need to add to the description of the class `Money` that we gave in Display 8.3? 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 8.3. 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 8.3. 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 8.3; 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 8.3. 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-eight 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 8.3. The `Money` parameters for the friend function `add` can be made into constant parameters as follows:

constant parameter

```
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 above change in the class definition, 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 8.3.

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 8.3:

*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 8.3.

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*.

***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.

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.

In Display 8.4 we have rewritten the definition of the class `Money` given in Display 8.3, 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 8.3, except that the modifier *const* must be used in function headings so that the headings match the function declarations shown in Display 8.4.



Display 8.4 The Class Money with Constant Parameters

```

//Class for amounts of money in U.S. currency.
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.

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

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

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

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

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

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

    void output(ostream& outs) const;
    //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.
private:
    long all_cents;
};

```

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 8.4.
- 9 Why would it be incorrect to add the modifier `const`, as shown below, to the function declaration for the member function `input` of the class `Money` given in Display 8.4?

```
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 follow:

```
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?

8.2 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 8.3). 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 below:

```
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 8.3) 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 8.5 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 8.5, 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 8.5, 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 8.5.

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 8.5. What do you need to add to the description of `Money` given in Display 8.5?



Display 8.5 Overloading Operators (part 1 of 2)

```
//Program to demonstrate the class Money. (This is an improved version of
//the class Money that we gave in Display 8.3 and rewrote in Display 8.4.)
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool operator ==(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if amount1 and amount2 have the same value;
    //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;
};

<Any extra function declarations from Display 8.3 go here. >

int main()
{
    Money cost(1, 50), tax(0, 15), total;
    total = cost + tax;

    cout << "cost = ";
    cost.output(cout);
    cout << endl;
```

Some comments from Display 8.4 have been omitted to save space in this book, but they should be included in a real program.

Display 8.5 Overloading Operators (part 2 of 2)

```

    cout << "tax = ";
    tax.output(cout);
    cout << endl;
    cout << "total bill = ";
    total.output(cout);
    cout << endl;
    if (cost == tax)
        cout << "Move to another state.\n";
    else
        cout << "Things seem normal.\n";
    return 0;
}

Money operator +(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

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

```

<The definitions of the member functions are the same as in Display 8.3 except that *const* is added to the function headings in various places so that the function headings match the function declarations in the above 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 8.3.>

Output

```

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

```

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 9.)
- 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 Chapter 12. 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.

- 14 Suppose you wish to overload the operator `<=` so that it applies to the type `Money` defined in Display 8.5. What do you need to add to the description of `Money` given in Display 8.5?
- 15 Is it possible using operator overloading to change the behavior of `+` on integers? Why or why not?

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 8.5, 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 8.5 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
```

the system 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 8.3.)

Note that this type conversion will not work unless there is a suitable constructor. For example, the type `Money` (Display 8.5) 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 the above 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 a definition of 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 8.5. 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 statement below, 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 8.5 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 8.6. 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 how we overloaded the negation operator in Display 8.6. 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!)

`++` and `--`

Overloading `>>` and `<<`

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";
```

`<<` is an operator

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 `<<`.



Display 8.6 Overloading a Unary Operator

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

    friend Money operator -(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns amount 1 minus amount2.

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

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

    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;
};

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

Money operator -(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents - amount2.all_cents;
    return temp;
}

Money operator -(const Money& amount)
{
    Money temp;
    temp.all_cents = -amount.all_cents;
    return temp;
}
```

<The other function definitions are the same as in Display 8.5.>

*This is an improved version
of the class Money given in
Display 8.5.*

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

overloading <<

In our previous definitions of the class `Money`, we used the member function `output` to output values of type `Money` (Display 8.3 through Display 8.6). 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 below:

```
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
```

The two operands in the above 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 above expression 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 `<<`.

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

chains of <<

```
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 the above expression? The first thing evaluated is the subexpression:

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

If things are to work out, then the above 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 8.7. 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 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 8.3 (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 above 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*

Display 8.7 << as an Operator

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


means the same as

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

and is evaluated as follows:

First evaluate `(cout << "I have ")`, which returns `cout`:

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




and the string "I have" is output.

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

Then evaluate `(cout << amount)`, which returns `cout`:

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




and the value of amount is output.

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

Then evaluate `cout << " in my purse.\n"`, which returns `cout`:

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



and the string " in my purse.\n" is output.

```
cout;
```

Since there are no more << operators, the process ends.

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.

reference

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 8.3 (except that
    all_cents is replaced with amount.all_cents)>

    return ins;
}
```

The complete definitions of the overloaded operators << and >> are given in Display 8.8, 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 below. See Display 8.8 for an example.

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 → *Parameter_1*

Parameter for the object to receive the input → *Parameter_2*

Definitions

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

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


Display 8.8 Overloading << and >> (part 1 of 4)

```
//Program to demonstrate the class Money.
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    friend Money operator -(const Money& amount1, const Money& amount2);
    friend Money operator -(const Money& amount);
    friend bool operator ==(const Money& amount1, const Money& amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value() const;

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

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

*This is an improved version
of the class Money that we
gave in Display 8.6.*

*Although we have omitted
some of the comments from
Displays 8.5 and 8.6, you
should include them.*

Display 8.8 Overloading << and >> (part 2 of 4)

```
int digit_to_int(char c);  
//Used in the definition of the overloaded input operator >>.  
//Precondition: c is one of the digits '0' through '9'.  
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.  
  
int main()  
{  
    Money amount;  
    ifstream in_stream;  
    ofstream out_stream;  
  
    in_stream.open("infile.dat");  
    if (in_stream.fail())  
    {  
        cout << "Input file opening failed.\n";  
        exit(1);  
    }  
  
    out_stream.open("outfile.dat");  
    if (out_stream.fail())  
    {  
        cout << "Output file opening failed.\n";  
        exit(1);  
    }  
  
    in_stream >> amount;  
    out_stream << amount  
        << " copied from the file infile.dat.\n";  
    cout << amount  
        << " copied from the file infile.dat.\n";  
  
    in_stream.close();  
    out_stream.close();  
  
    return 0;  
}
```

Display 8.8 Overloading << and >> (part 3 of 4)

```

//Uses iostream, ctype, cstdlib:
istream& operator >>(istream& ins, Money& amount)
{
    char one_char, decimal_point,
        digit1, digit2; //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

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

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

    amount.all_cents = dollars*100 + cents;
    if (negative)
        amount.all_cents = -amount.all_cents;

    return ins;
}

int digit_to_int(char c)
{
    return ( int(c) - int('0') );
}

```

Display 8.8 Overloading << and >> (part 4 of 4)

```
//Uses cstdlib and iostream:
ostream& operator <<(ostream& outs, const Money& amount)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(amount.all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

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

    if (cents < 10)
        outs << '0';
    outs << cents;

    return outs;
}
```

<The definitions of the member functions and other overloaded operators go here. See Displays 8.3, 8.4, 8.5, and 8.6 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.

```

    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;
};

```

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 the type returned must be a reference, which is indicated by appending an & to the name of the returned type.

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 above 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 8.2 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 below. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 8.3 should be included in this definition.

```
class Money
{
public:
    friend Money add(Money amount1, Money amount2);

    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 below. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 8.3 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 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-eight 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 eight.

- 8 The only change from the version given in Display 8.3 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 method 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 below. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 8.5 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 below. The part in color is new. We have omitted some comments to save space, but all the comments shown in Display 8.5 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;
}

```

Programming Projects



- 1 Modify the definition of the class Money shown in Display 8.8 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.

- 4 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.

- 5 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 above.