Operator Overloading

CHAPTER

8

IN THIS CHAPTER

- **Overloading Unary Operators 320**
- **Overloading Binary Operators 328**
- **Data Conversion 344**
- **UML Class Diagrams 357**
- **Pitfalls of Operator Overloading and Conversion 358**
- **Keywords `explicit` and `mutable` 360**

Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. For example, statements like

```
d3.addobjects(d1, d2);
```

or the similar but equally obscure

```
d3 = d1.addobjects(d2);
```

can be changed to the much more readable

```
d3 = d1 + d2;
```

The rather forbidding term *operator overloading* refers to giving the normal C++ operators, such as `+`, `*`, `<=`, and `+=`, additional meanings when they are applied to user-defined data types. Normally

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

works only with basic types such as `int` and `float`, and attempting to apply it when `a`, `b`, and `c` are objects of a user-defined class will cause complaints from the compiler. However, using overloading, you can make this statement legal even when `a`, `b`, and `c` are user-defined types.

In effect, operator overloading gives you the opportunity to redefine the C++ language. If you find yourself limited by the way the C++ operators work, you can change them to do whatever you want. By using classes to create new kinds of variables, and operator overloading to create new definitions for operators, you can extend C++ to be, in many ways, a new language of your own design.

Another kind of operation, *data type conversion*, is closely connected with operator overloading. C++ handles the conversion of simple types, such as `int` and `float`, automatically; but conversions involving user-defined types require some work on the programmer's part. We'll look at data conversions in the second part of this chapter.

Overloaded operators are not all beer and skittles. We'll discuss some of the dangers of their use at the end of the chapter.

Overloading Unary Operators

Let's start off by overloading a *unary operator*. As you may recall from Chapter 2, unary operators act on only one operand. (An operand is simply a variable acted on by an operator.) Examples of unary operators are the increment and decrement operators `++` and `--`, and the unary minus, as in `-33`.

In the COUNTER example in Chapter 6, "Objects and Classes," we created a class `Counter` to keep track of a count. Objects of that class were incremented by calling a member function:

```
c1.inc_count();
```

That did the job, but the listing would have been more readable if we could have used the increment operator `++` instead:

```
++c1;
```

All dyed-in-the-wool C++ (and C) programmers would guess immediately that this expression increments `c1`.

Let's rewrite COUNTER to make this possible. Here's the listing for COUNTPP1:

```
// countpp1.cpp
// increment counter variable with ++ operator
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;              //count
   public:
      Counter() : count(0)             //constructor
         {  }
      unsigned int get_count()         //return count
         { return count; }
      void operator ++ ()               //increment (prefix)
         {
         ++count;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                     //define and initialize

   cout << "\nc1=" << c1.get_count();  //display
   cout << "\nc2=" << c2.get_count();

   ++c1;                    //increment c1
   ++c2;                    //increment c2
   ++c2;                    //increment c2
```

```
   cout << "\nc1=" << c1.get_count();  //display again
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

In this program we create two objects of class `Counter`: `c1` and `c2`. The counts in the objects are displayed; they are initially 0. Then, using the overloaded `++` operator, we increment `c1` once and `c2` twice, and display the resulting values. Here's the program's output:

```
c1=0     <------- counts are initially 0
c2=0
c1=1     <------- incremented once
c2=2     <------- incremented twice
```

The statements responsible for these operations are

```
++c1;
++c2;
++c2;
```

The `++` operator is applied once to `c1` and twice to `c2`. We use prefix notation in this example; we'll explore postfix later.

The `operator` Keyword

How do we teach a normal C++ operator to act on a user-defined operand? The keyword `operator` is used to overload the `++` operator in this declarator:

```
void operator ++ ()
```

The return type (`void` in this case) comes first, followed by the keyword `operator`, followed by the operator itself (`++`), and finally the argument list enclosed in parentheses (which are empty here). This declarator syntax tells the compiler to call this member function whenever the `++` operator is encountered, provided the operand (the variable operated on by the `++`) is of type `Counter`.

We saw in Chapter 5, "Functions," that the only way the compiler can distinguish between overloaded functions is by looking at the data types and the number of their arguments. In the same way, the only way it can distinguish between overloaded operators is by looking at the data type of their operands. If the operand is a basic type such as an `int`, as in

```
++intvar;
```

then the compiler will use its built-in routine to increment an `int`. But if the operand is a `Counter` variable, the compiler will know to use our user-written `operator++()` instead.

Operator Arguments

In `main()` the `++` operator is applied to a specific object, as in the expression `++c1`. Yet `operator++()` takes no arguments. What does this operator increment? It increments the `count` data in the object of which it is a member. Since member functions can always access the particular object for which they've been invoked, this operator requires no arguments. This is shown in Figure 8.1.

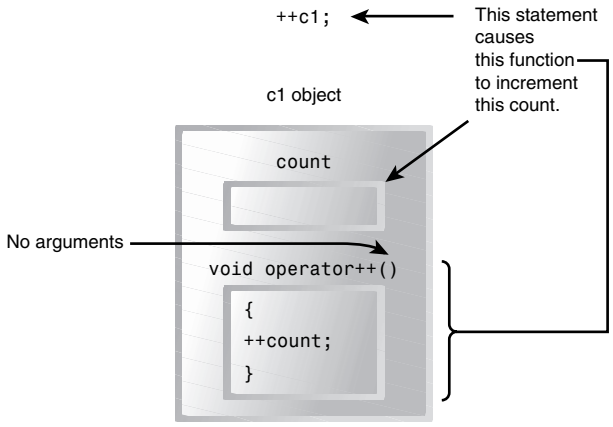

FIGURE 8.1
Overloaded unary operator: no arguments.

Operator Return Values

The `operator++()` function in the COUNTPP1 program has a subtle defect. You will discover it if you use a statement like this in `main()`:

```
c1 = ++c2;
```

The compiler will complain. Why? Because we have defined the `++` operator to have a return type of `void` in the `operator++()` function, while in the assignment statement it is being asked to return a variable of type `Counter`. That is, the compiler is being asked to return whatever value `c2` has after being operated on by the `++` operator, and assign this value to `c1`. So as defined in COUNTPP1, we can't use `++` to increment `Counter` objects in assignments; it must always stand alone with its operand. Of course the normal `++` operator, applied to basic data types such as `int`, would not have this problem.

To make it possible to use our homemade `operator++()` in assignment expressions, we must provide a way for it to return a value. The next program, COUNTPP2, does just that.

```
// countpp2.cpp
// increment counter variable with ++ operator, return value
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;      //count
   public:
      Counter() : count(0)     //constructor
         {  }
      unsigned int get_count() //return count
         { return count; }
      Counter operator ++ ()   //increment count
         {
         ++count;              //increment count
         Counter temp;         //make a temporary Counter
         temp.count = count;   //give it same value as this obj
         return temp;          //return the copy
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                        //c1=0, c2=0

   cout << "\nc1=" << c1.get_count();    //display
   cout << "\nc2=" << c2.get_count();

   ++c1;                                  //c1=1
   c2 = ++c1;                             //c1=2, c2=2

   cout << "\nc1=" << c1.get_count();    //display again
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

Here the `operator++()` function creates a new object of type `Counter`, called `temp`, to use as a return value. It increments the `count` data in its own object as before, then creates the new `temp` object and assigns `count` in the new object the same value as in its own object. Finally, it returns the `temp` object. This has the desired effect. Expressions like

```
++c1
```

now return a value, so they can be used in other expressions, such as

```
c2 = ++c1;
```

as shown in `main()`, where the value returned from `c1++` is assigned to `c2`. The output from this program is

```
c1=0
c2=0
c1=2
c2=2
```

Nameless Temporary Objects

In COUNTPP2 we created a temporary object of type `Counter`, named `temp`, whose sole purpose was to provide a return value for the `++` operator. This required three statements.

```
Counter temp;          // make a temporary Counter object
temp.count = count;    // give it same value as this object
return temp;           // return it
```

There are more convenient ways to return temporary objects from functions and overloaded operators. Let's examine another approach, as shown in the program COUNTPP3:

```
// countpp3.cpp
// increment counter variable with ++ operator
// uses unnamed temporary object
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;        //count
   public:
      Counter() : count(0)       //constructor  no args
         {  }
      Counter(int c) : count(c)  //constructor, one arg
         {  }
      unsigned int get_count()   //return count
         { return count; }
      Counter operator ++ ()     //increment count
         {
         ++count;                  //  increment count, then return
         return Counter(count);    //  an unnamed temporary object
         }                         //  initialized to this count
   };
////////////////////////////////////////////////////////////////
```

```
int main()
   {
   Counter c1, c2;                         //c1=0, c2=0

   cout << "\nc1=" << c1.get_count();     //display
   cout << "\nc2=" << c2.get_count();

   ++c1;                                   //c1=1
   c2 = ++c1;                              //c1=2, c2=2

   cout << "\nc1=" << c1.get_count();     //display again
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

In this program a single statement

```
return Counter(count);
```

does what all three statements did in COUNTPP2. This statement creates an object of type `Counter`. This object has no name; it won't be around long enough to need one. This unnamed object is initialized to the value provided by the argument `count`.

But wait: Doesn't this require a constructor that takes one argument? It does, and to make this statement work we sneakily inserted just such a constructor into the member function list in COUNTPP3.

```
Counter(int c) : count(c)      //constructor, one arg
   { }
```

Once the unnamed object is initialized to the value of `count`, it can then be returned. The output of this program is the same as that of COUNTPP2.

The approaches in both COUNTPP2 and COUNTPP3 involve making a copy of the original object (the object of which the function is a member), and returning the copy. (Another approach, as we'll see in Chapter 11, "Virtual Functions," is to return the value of the original object using the `this` pointer.)

Postfix Notation

So far we've shown the increment operator used only in its prefix form.

```
++c1
```

What about postfix, where the variable is incremented after its value is used in the expression?

```
c1++
```

To make both versions of the increment operator work, we define two overloaded ++ operators, as shown in the POSTFIX program:

```
// postfix.cpp
// overloaded ++ operator in both prefix and postfix
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;         //count
   public:
      Counter() : count(0)        //constructor  no args
         {  }
      Counter(int c) : count(c)   //constructor, one arg
         {  }
      unsigned int get_count() const //return count
         { return count; }

      Counter operator ++ ()      //increment count (prefix)
         {                        //increment count, then return
         return Counter(++count); //an unnamed temporary object
         }                        //initialized to this count

      Counter operator ++ (int)   //increment count (postfix)
         {                        //return an unnamed temporary
         return Counter(count++); //object initialized to this
         }                        //count, then increment count
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                      //c1=0, c2=0

   cout << "\nc1=" << c1.get_count();    //display
   cout << "\nc2=" << c2.get_count();

   ++c1;                                //c1=1
   c2 = ++c1;                           //c1=2, c2=2 (prefix)

   cout << "\nc1=" << c1.get_count();    //display
   cout << "\nc2=" << c2.get_count();

   c2 = c1++;                           //c1=3, c2=2 (postfix)
```

```
   cout << "\nc1=" << c1.get_count();    //display again
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

Now there are two different declarators for overloading the ++ operator. The one we've seen before, for prefix notation, is

```
Counter operator ++ ()
```

The new one, for postfix notation, is

```
Counter operator ++ (int)
```

The only difference is the `int` in the parentheses. This `int` isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler to create the postfix version of the operator. The designers of C++ are fond of recycling existing operators and keywords to play multiple roles, and `int` is the one they chose to indicate postfix. (Well, can you think of a better syntax?) Here's the output from the program:

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

We saw the first four of these output lines in COUNTPP2 and COUNTPP3. But in the last two lines we see the results of the statement

```
c2=c1++;
```

Here, `c1` is incremented to 3, but `c2` is assigned the value of `c1` before it is incremented, so `c2` retains the value 2.

Of course, you can use this same approach with the decrement operator (`--`).

Overloading Binary Operators

Binary operators can be overloaded just as easily as unary operators. We'll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators.

Arithmetic Operators

In the ENGLCON program in Chapter 6 we showed how two English `Distance` objects could be added using a member function `add_dist()`:

```
dist3.add_dist(dist1, dist2);
```

By overloading the + operator we can reduce this dense-looking expression to

```
dist3 = dist1 + dist2;
```

Here's the listing for ENGLPLUS, which does just that:

```
// englplus.cpp
// overloaded '+' operator adds two Distances
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                     //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      Distance operator + ( Distance ) const;  //add 2 distances
   };
//-----------------------------------------------------------
                                  //add this distance to d2
Distance Distance::operator + (Distance d2) const  //return sum
   {
   int f = feet + d2.feet;        //add the feet
   float i = inches + d2.inches;  //add the inches
   if(i >= 12.0)                  //if total exceeds 12.0,
      {                           //then decrease inches
      i -= 12.0;                  //by 12.0 and
      f++;                        //increase feet by 1
      }                           //return a temporary Distance
   return Distance(f,i);          //initialized to sum
   }
////////////////////////////////////////////////////////////////
```

```
int main()
   {
   Distance dist1, dist3, dist4;   //define distances
   dist1.getdist();                //get dist1 from user

   Distance dist2(11, 6.25);       //define, initialize dist2

   dist3 = dist1 + dist2;          //single '+' operator

   dist4 = dist1 + dist2 + dist3;  //multiple '+' operators
                                   //display all lengths
   cout << "dist1 = ";  dist1.showdist(); cout << endl;
   cout << "dist2 = ";  dist2.showdist(); cout << endl;
   cout << "dist3 = ";  dist3.showdist(); cout << endl;
   cout << "dist4 = ";  dist4.showdist(); cout << endl;
   return 0;
   }
```

To show that the result of an addition can be used in another addition as well as in an assignment, another addition is performed in `main()`. We add `dist1`, `dist2`, and `dist3` to obtain `dist4` (which should be double the value of `dist3`), in the statement

```
dist4 = dist1 + dist2 + dist3;
```

Here's the output from the program:

```
Enter feet: 10
Enter inches: 6.5

dist1 = 10'-6.5"     <------ from user
dist2 = 11'-6.25"    <------ initialized in program
dist3 = 22'-0.75"    <------ dist1+dist2
dist4 = 44'-1.5"     <------ dist1+dist2+dist3
```

In class `Distance` the declaration for the `operator+()` function looks like this:

```
Distance operator + ( Distance );
```

This function has a return type of `Distance`, and takes one argument of type `Distance`.

In expressions like

```
dist3 = dist1 + dist2;
```

it's important to understand how the return value and arguments of the operator relate to the objects. When the compiler sees this expression it looks at the argument types, and finding only type `Distance`, it realizes it must use the `Distance` member function `operator+()`. But what does this function use as its argument—`dist1` or `dist2`? And doesn't it need two arguments, since there are two numbers to be added?

Here's the key: The argument on the *left side* of the operator (`dist1` in this case) is the object of which the operator is a member. The object on the *right side* of the operator (`dist2`) must be furnished as an argument to the operator. The operator returns a value, which can be assigned or used in other ways; in this case it is assigned to `dist3`. Figure 8.2 shows how this looks.

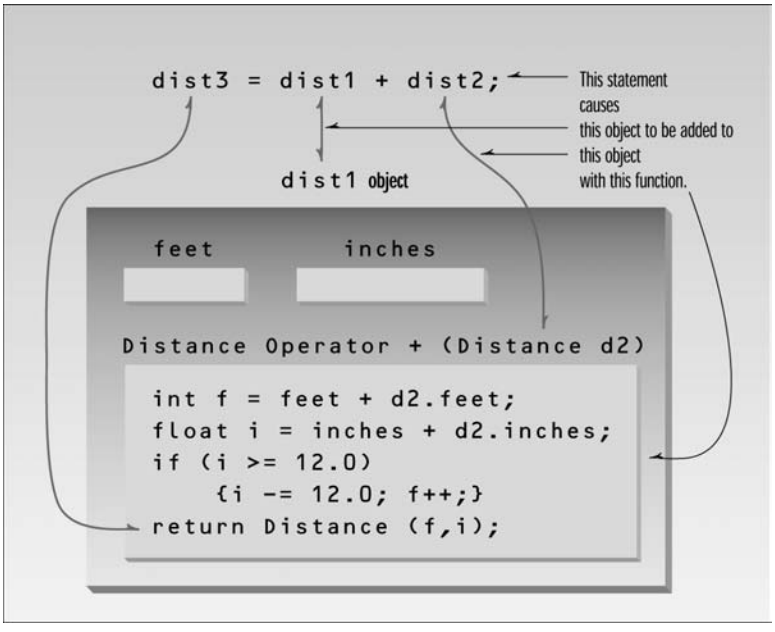

FIGURE 8.2
Overloaded binary operator: one argument.

In the `operator+()` function, the left operand is accessed directly—since this is the object of which the operator is a member—using `feet` and `inches`. The right operand is accessed as the function's argument, as `d2.feet` and `d2.inches`.

We can generalize and say that an overloaded operator always requires one less argument than its number of operands, since one operand is the object of which the operator is a member. That's why unary operators require no arguments. (This rule does not apply to friend functions and operators, C++ features we'll discuss in Chapter 11.)

To calculate the return value of `operator+()` in ENGLPLUS, we first add the `feet` and `inches` from the two operands (adjusting for a carry if necessary). The resulting values, `f` and `i`, are then used to initialize a nameless `Distance` object, which is returned in the statement

```
return Distance(f, i);
```

This is similar to the arrangement used in COUNTPP3, except that the constructor takes two arguments instead of one. The statement

```
dist3 = dist1 + dist2;
```

in `main()` then assigns the value of the nameless `Distance` object to `dist3`. Compare this intuitively obvious statement with the use of a function call to perform the same task, as in the ENGLCON example in Chapter 6.

Similar functions could be created to overload other operators in the `Distance` class, so you could subtract, multiply, and divide objects of this class in natural-looking ways.

Concatenating Strings

The + operator cannot be used to concatenate C-strings. That is, you can't say

```
str3 = str1 + str2;
```

where `str1`, `str2`, and `str3` are C-string variables (arrays of type `char`), as in "cat" plus "bird" equals "catbird." However, if we use our own `String` class, as shown in the STROBJ program in Chapter 6, we can overload the + operator to perform such concatenation. This is what the Standard C++ `string` class does, but it's easier to see how it works in our less ambitious `String` class. Overloading the + operator to do something that isn't strictly addition is another example of redefining the C++ language. Here's the listing for STRPLUS:

```
// strplus.cpp
// overloaded '+' operator concatenates strings
#include <iostream>
using namespace std;
#include <string.h>      //for strcpy(), strcat()
#include <stdlib.h>      //for exit()
////////////////////////////////////////////////////////////
class String             //user-defined string type
   {
   private:
      enum { SZ=80 };               //size of String objects
      char str[SZ];                 //holds a string
   public:
      String()                      //constructor, no args
         { strcpy(str, ""); }
      String( char s[] )            //constructor, one arg
         { strcpy(str, s); }
      void display() const          //display the String
         { cout << str; }
      String operator + (String ss) const  //add Strings
```

```
        {
        String temp;                //make a temporary String
        if( strlen(str) + strlen(ss.str) < SZ )
           {
           strcpy(temp.str, str);   //copy this string to temp
           strcat(temp.str, ss.str);  //add the argument string
           }
        else
           { cout << "\nString overflow"; exit(1); }
        return temp;                //return temp String
        }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   String s1 = "\nMerry Christmas!  ";   //uses constructor 2
   String s2 = "Happy new year!";        //uses constructor 2
   String s3;                            //uses constructor 1

   s1.display();                         //display strings
   s2.display();
   s3.display();

   s3 = s1 + s2;                         //add s2 to s1,
                                         //assign to s3
   s3.display();                         //display s3
   cout << endl;
   return 0;
   }
```


The program first displays three strings separately. (The third is empty at this point, so nothing is printed when it displays itself.) Then the first two strings are concatenated and placed in the third, and the third string is displayed again. Here's the output:

```
Merry Christmas!  Happy new year!     <------- s1, s2, and s3 (empty)
Merry Christmas!  Happy new year!     <------- s3 after concatenation
```

By now the basics of overloading the + operator should be somewhat familiar. The declarator

```
String operator + (String ss)
```

shows that the + operator takes one argument of type `String` and returns an object of the same type. The concatenation process in `operator+()` involves creating a temporary object of type `String`, copying the string from our own `String` object into it, concatenating the argument string using the library function `strcat()`, and returning the resulting temporary string. Note that we can't use the

```
return String(string);
```

approach, where a nameless temporary `String` is created, because we need access to the temporary `String` not only to initialize it, but to concatenate the argument string to it.

We must be careful that we don't overflow the fixed-length strings used in the `String` class. To prevent such accidents in the `operator+()` function, we check that the combined length of the two strings to be concatenated will not exceed the maximum string length. If they do, we print an error message instead of carrying out the concatenation operation. (We could handle errors in other ways, like returning 0 if an error occurred, or better yet, throwing an exception, as discussed in Chapter 14, "Templates and Exceptions.")

Remember that using an `enum` to set the constant value `SZ` is a temporary fix. When all compilers comply with Standard C++ you can change it to

```
static const int SZ = 80;
```

Multiple Overloading

We've seen different uses of the + operator: to add English distances and to concatenate strings. You could put both these classes together in the same program, and C++ would still know how to interpret the + operator: It selects the correct function to carry out the "addition" based on the type of operand.

Comparison Operators

Let's see how to overload a different kind of C++ operator: comparison operators.

Comparing Distances

In our first example we'll overload the *less than* operator (<) in the `Distance` class so that we can compare two distances. Here's the listing for ENGLESS:

```
// engless.cpp
// overloaded '<' operator compares two Distances
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                  //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                      //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                   //constructor (two args)
```

```
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      bool operator < (Distance) const;  //compare distances
   };
//-----------------------------------------------------------
                                 //compare this distance with d2
bool Distance::operator < (Distance d2) const  //return the sum
   {
   float bf1 = feet + inches/12;
   float bf2 = d2.feet + d2.inches/12;
   return (bf1 < bf2) ? true : false;
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1;                //define Distance dist1
   dist1.getdist();               //get dist1 from user

   Distance dist2(6, 2.5);        //define and initialize dist2
                                  //display distances
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();

   if( dist1 < dist2 )            //overloaded '<' operator
      cout << "\ndist1 is less than dist2";
   else
      cout << "\ndist1 is greater than (or equal to) dist2";
   cout << endl;
   return 0;
   }
```

This program compares a distance entered by the user with a distance, 6'–2.5", initialized by the program. Depending on the result, it then prints one of two possible sentences. Here's some typical output:

```
Enter feet: 5
Enter inches: 11.5
dist1 = 5'-11.5"
dist2 = 6'-2.5"
dist1 is less than dist2
```

The approach used in the `operator<()` function in ENGLESS is similar to overloading the `+` operator in the ENGLPLUS program, except that here the `operator<()` function has a return type of `bool`. The return value is `false` or `true`, depending on the comparison of the two distances. The comparison is made by converting both distances to floating-point feet, and comparing them using the normal `<` operator. Remember that the use of the conditional operator

```
return (bf1 < bf2) ? true : false;
```

is the same as

```
if(bf1 < bf2)
   return true;
else
   return false;
```

Comparing Strings

Here's another example of overloading an operator, this time the *equal to* (==) operator. We'll use it to compare two of our homemade `String` objects, returning `true` if they're the same and `false` if they're different. Here's the listing for STREQUAL:

```
//strequal.cpp
//overloaded '==' operator compares strings
#include <iostream>
using namespace std;
#include <string.h>      //for strcmp()
////////////////////////////////////////////////////////////
class String             //user-defined string type
   {
   private:
      enum { SZ = 80 };               //size of String objects
      char str[SZ];                   //holds a string
   public:
      String()                        //constructor, no args
         { strcpy(str, ""); }
      String( char s[] )              //constructor, one arg
         { strcpy(str, s); }
      void display() const            //display a String
         { cout << str; }
      void getstr()                   //read a string
         { cin.get(str, SZ); }
      bool operator == (String ss) const  //check for equality
         {
         return ( strcmp(str, ss.str)==0 ) ? true : false;
         }
   };
////////////////////////////////////////////////////////////
```

```
int main()
   {
   String s1 = "yes";
   String s2 = "no";
   String s3;

   cout << "\nEnter 'yes' or 'no': ";
   s3.getstr();                          //get String from user

   if(s3==s1)                            //compare with "yes"
      cout << "You typed yes\n";
   else if(s3==s2)                       //compare with "no"
      cout << "You typed no\n";
   else
      cout << "You didn't follow instructions\n";
   return 0;
   }
```

The `main()` part of this program uses the == operator twice, once to see if a string input by the user is "yes" and once to see if it's "no." Here's the output when the user types "yes":

```
Enter 'yes' or 'no': yes
You typed yes
```

The `operator==()` function uses the library function `strcmp()` to compare the two C-strings. This function returns 0 if the strings are equal, a negative number if the first is less than the second, and a positive number if the first is greater than the second. Here *less than* and *greater than* are used in their lexicographical sense to indicate whether the first string appears before or after the second in an alphabetized listing.

Other comparison operators, such as < and >, could also be used to compare the lexicographical value of strings. Or, alternatively, these comparison operators could be redefined to compare string lengths. Since you're the one defining how the operators are used, you can use any definition that seems appropriate to your situation.

Arithmetic Assignment Operators

Let's finish up our exploration of overloaded binary operators with an arithmetic assignment operator: the += operator. Recall that this operator combines assignment and addition into one step. We'll use this operator to add one English distance to a second, leaving the result in the first. This is similar to the ENGLPLUS example shown earlier, but there is a subtle difference. Here's the listing for ENGLPLEQ:

```
// englpleq.cpp
// overloaded '+=' assignment operator
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                      //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                          //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                       //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()                //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const         //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      void operator += ( Distance );
   };
//--------------------------------------------------------------
                                    //add distance to this one
void Distance::operator += (Distance d2)
   {
   feet += d2.feet;                 //add the feet
   inches += d2.inches;             //add the inches
   if(inches >= 12.0)               //if total exceeds 12.0,
      {                             //then decrease inches
      inches -= 12.0;               //by 12.0 and
      feet++;                       //increase feet
      }                             //by 1
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1;                  //define dist1
   dist1.getdist();                 //get dist1 from user
   cout << "\ndist1 = ";  dist1.showdist();

   Distance dist2(11, 6.25);        //define, initialize dist2
   cout << "\ndist2 = ";  dist2.showdist();

   dist1 += dist2;                  //dist1 = dist1 + dist2
   cout << "\nAfter addition,";
```

```
   cout << "\ndist1 = ";  dist1.showdist();
   cout << endl;
   return 0;
   }
```

In this program we obtain a distance from the user and add to it a second distance, initialized to 11'–6.25" by the program. Here's a sample of interaction with the program:

```
Enter feet: 3
Enter inches: 5.75
dist1 = 3'-5.75"
dist2 = 11'-6.25"
After addition,
dist1 = 15'-0"
```

In this program the addition is carried out in `main()` with the statement

```
dist1 += dist2;
```

This causes the sum of `dist1` and `dist2` to be placed in `dist1`.

Notice the difference between the function used here, `operator+=()`, and that used in ENGLPLUS, `operator+()`. In the earlier `operator+()` function, a new object of type `Distance` had to be created and returned by the function so it could be assigned to a third `Distance` object, as in

```
dist3 = dist1 + dist2;
```

In the `operator+=()` function in ENGLPLEQ, the object that takes on the value of the sum is the object of which the function is a member. Thus it is `feet` and `inches` that are given values, not temporary variables used only to return an object. The `operator+=()` function has no return value; it returns type `void`. A return value is not necessary with arithmetic assignment operators such as `+=`, because the result of the assignment operator is not assigned to anything. The operator is used alone, in expressions like the one in the program.

```
dist1 += dist2;
```

If you wanted to use this operator in more complex expressions, like

```
dist3 = dist1 += dist2;
```

then you would need to provide a return value. You can do this by ending the `operator+=()` function with a statement like

```
return Distance(feet, inches);
```

in which a nameless object is initialized to the same values as this object and returned.

The Subscript Operator ([])

The subscript operator, [], which is normally used to access array elements, can be overloaded. This is useful if you want to modify the way arrays work in C++. For example, you might want to make a "safe" array: One that automatically checks the index numbers you use to access the array, to ensure that they are not out of bounds. (You can also use the vector class, described in Chapter 15, "The Standard Template Library.")

To demonstrate the overloaded subscript operator, we must return to another topic, first mentioned in Chapter 5: returning values from functions by reference. To be useful, the overloaded subscript operator must return by reference. To see why this is true, we'll show three example programs that implement a safe array, each one using a different approach to inserting and reading the array elements:

- Separate put() and get() functions
- A single access() function using return by reference
- The overloaded [] operator using return by reference

All three programs create a class called safearay, whose only member data is an array of 100 int values, and all three check to ensure that all array accesses are within bounds. The main() program in each program tests the class by filling the safe array with values (each one equal to 10 times its array index) and then displaying them all to assure the user that everything is working as it should.

Separate get() and put() Functions

The first program provides two functions to access the array elements: putel() to insert a value into the array, and getel() to find the value of an array element. Both functions check the value of the index number supplied to ensure it's not out of bounds; that is, less than 0 or larger than the array size (minus 1). Here's the listing for ARROVER1:

```
// arrover1.cpp
// creates safe array (index values are checked before access)
// uses separate put and get functions
#include <iostream>
using namespace std;
#include <process.h>                  // for exit()
const int LIMIT = 100;
////////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
```

```
      void putel(int n, int elvalue)  //set value of element
         {
         if( n< 0 || n>=LIMIT )
            { cout << "\nIndex out of bounds"; exit(1); }
         arr[n] = elvalue;
         }
      int getel(int n) const          //get value of element
         {
         if( n< 0 || n>=LIMIT )
            { cout << "\nIndex out of bounds"; exit(1); }
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)  // insert elements
      sa1.putel(j, j*10);

   for(j=0; j<LIMIT; j++)      // display elements
      {
      int temp = sa1.getel(j);
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```


The data is inserted into the safe array with the `putel()` member function, and then displayed with `getel()`. This implements a safe array; you'll receive an error message if you attempt to use an out-of-bounds index. However, the format is a bit crude.

Single `access()` Function Returning by Reference

As it turns out, we can use the same member function both to insert data into the safe array and to read it out. The secret is to return the value from the function by reference. This means we can place the function on the left side of the equal sign, and the value on the right side will be assigned to the variable returned by the function, as explained in Chapter 5. Here's the listing for ARROVER2:

```
// arrover2.cpp
// creates safe array (index values are checked before access)
// uses one access() function for both put and get
#include <iostream>
using namespace std;
```

```
#include <process.h>            //for exit()
const int LIMIT = 100;          //array size
////////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      int& access(int n)        //note: return by reference
         {
         if( n< 0 || n>=LIMIT )
            { cout << "\nIndex out of bounds"; exit(1); }
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)   //insert elements
      sa1.access(j) = j*10;     //*left* side of equal sign

   for(j=0; j<LIMIT; j++)       //display elements
      {
      int temp = sa1.access(j); //*right* side of equal sign
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```

The statement

```
sa1.access(j) = j*10;     // *left* side of equal sign
```

causes the value `j*10` to be placed in `arr[j]`, the return value of the function.

It's perhaps slightly more convenient to use the same function for input and output of the safe array than it is to use separate functions; there's one less name to remember. But there's an even better way, with no names to remember at all.

Overloaded `[]` Operator Returning by Reference

To access the safe array using the same subscript (`[]`) operator that's used for normal C++ arrays, we overload the subscript operator in the `safearay` class. However, since this operator is commonly used on the left side of the equal sign, this overloaded function must return by reference, as we showed in the previous program. Here's the listing for ARROVER3:

```
// arrover3.cpp
// creates safe array (index values are checked before access)
// uses overloaded [] operator for both put and get

#include <iostream>
using namespace std;
#include <process.h>           //for exit()
const int LIMIT = 100;         //array size
////////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      int& operator [](int n)  //note: return by reference
         {
         if( n< 0 || n>=LIMIT )
            { cout << "\nIndex out of bounds"; exit(1); }
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)  //insert elements
      sa1[j] = j*10;           //*left* side of equal sign

   for(j=0; j<LIMIT; j++)      //display elements
      {
      int temp = sa1[j];       //*right* side of equal sign
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```

In this program we can use the natural subscript expressions

```
sa1[j] = j*10;
```

and

```
temp = sa1[j];
```

for input and output to the safe array.

Data Conversion

You already know that the = operator will assign a value from one variable to another, in statements like

```
intvar1 = intvar2;
```

where `intvar1` and `intvar2` are integer variables. You may also have noticed that = assigns the value of one user-defined object to another, provided they are of the same type, in statements like

```
dist3 = dist1 + dist2;
```

where the result of the addition, which is type `Distance`, is assigned to another object of type `Distance`, `dist3`. Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler doesn't need any special instructions to use = for the assignment of user-defined objects such as `Distance` objects.

Thus, assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign. But what happens when the variables on different sides of the = are of different types? This is a more thorny question, to which we will devote the balance of this chapter. We'll first review how the compiler handles the conversion of basic types, which it does automatically. Then we'll explore several situations where the compiler doesn't handle things automatically and we need to tell it what to do. These include conversions between basic types and user-defined types, and conversions between different user-defined types.

You might think it represents poor programming practice to convert routinely from one type to another. After all, languages such as Pascal go to considerable trouble to keep you from doing such conversions. However, the philosophy in C++ (and C) is that the flexibility provided by allowing conversions outweighs the dangers. This is a controversial issue; we'll return to it at the end of this chapter.

Conversions Between Basic Types

When we write a statement like

```
intvar = floatvar;
```

where `intvar` is of type `int` and `floatvar` is of type `float`, we are assuming that the compiler will call a special routine to convert the value of `floatvar`, which is expressed in floating-point format, to an integer format so that it can be assigned to `intvar`. There are of course many such conversions: from `float` to `double`, `char` to `float`, and so on. Each such conversion has

its own routine, built into the compiler and called up when the data types on different sides of the equal sign so dictate. We say such conversions are *implicit* because they aren't apparent in the listing.

Sometimes we want to force the compiler to convert one type to another. To do this we use the cast operator. For instance, to convert `float` to `int`, we can say

```
intvar = static_cast<int>(floatvar);
```

Casting provides *explicit* conversion: It's obvious in the listing that `static_cast<int>()` is intended to convert from `float` to `int`. However, such explicit conversions use the same built-in routines as implicit conversions.

Conversions Between Objects and Basic Types

When we want to convert between user-defined data types and basic types, we can't rely on built-in conversion routines, since the compiler doesn't know anything about user-defined types besides what we tell it. Instead, we must write these routines ourselves.

Our next example shows how to convert between a basic type and a user-defined type. In this example the user-defined type is (surprise!) the English `Distance` class from previous examples, and the basic type is `float`, which we use to represent meters, a unit of length in the metric measurement system.

The example shows conversion both from `Distance` to `float`, and from `float` to `Distance`. Here's the listing for ENGLCONV:

```
// englconv.cpp
// conversions: Distance to meters, meters to Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                  //English Distance class
   {
   private:
      const float MTF;          //meters to feet
      int feet;
      float inches;
   public:                      //constructor (no args)
      Distance() : feet(0), inches(0.0), MTF(3.280833F)
         {  }                   //constructor (one arg)
      Distance(float meters) : MTF(3.280833F)
         {                      //convert meters to Distance
         float fltfeet = MTF * meters;  //convert to float feet
         feet = int(fltfeet);           //feet is integer part
         inches = 12*(fltfeet-feet);    //inches is what's left
         }                      //constructor (two args)
```

```
      Distance(int ft, float in) : feet(ft),
                                   inches(in), MTF(3.280833F)
         {  }
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      operator float() const      //conversion operator
         {                        //converts Distance to meters
         float fracfeet = inches/12;     //convert the inches
         fracfeet += static_cast<float>(feet); //add the feet
         return fracfeet/MTF;            //convert to meters
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   float mtrs;
   Distance dist1 = 2.35F;        //uses 1-arg constructor to
                                  //convert meters to Distance
   cout << "\ndist1 = "; dist1.showdist();

   mtrs = static_cast<float>(dist1); //uses conversion operator
                                     //for Distance to meters
   cout << "\ndist1 = " << mtrs << " meters\n";

   Distance dist2(5, 10.25);      //uses 2-arg constructor

   mtrs = dist2;                  //also uses conversion op
   cout << "\ndist2 = " << mtrs << " meters\n";

// dist2 = mtrs;                  //error, = won't convert
   return 0;
   }
```

In `main()` the program first converts a fixed `float` quantity—2.35, representing meters—to feet and inches, using the one-argument constructor:

```
Distance dist1 = 2.35F;
```

Going in the other direction, it converts a `Distance` to meters in the statements

```
mtrs = static_cast<float>(dist2);
```

and

```
mtrs = dist2;
```

Here's the output:

```
dist1 = 7'-8.51949"      <------- this is 2.35 meters
dist1 = 2.35 meters      <------- this is 7'–8.51949"
dist2 = 1.78435 meters   <------- this is 5'–10.25"
```

We've seen how conversions are performed using simple assignment statements in `main()`. Now let's see what goes on behind the scenes, in the `Distance` member functions. Converting a user-defined type to a basic type requires a different approach than converting a basic type to a user-defined type. We'll see how both types of conversions are carried out in ENGLCONV.

From Basic to User-Defined

To go from a basic type—`float` in this case—to a user-defined type such as `Distance`, we use a constructor with one argument. These are sometimes called *conversion constructors*. Here's how this constructor looks in ENGLCONV:

```
Distance(float meters)
   {
   float fltfeet = MTF * meters;
   feet = int(fltfeet);
   inches = 12 * (fltfeet-feet);
   }
```

This function is called when an object of type `Distance` is created with a single argument. The function assumes that this argument represents meters. It converts the argument to feet and inches, and assigns the resulting values to the object. Thus the conversion from meters to `Distance` is carried out along with the creation of an object in the statement

```
Distance dist1 = 2.35;
```

From User-Defined to Basic

What about going the other way, from a user-defined type to a basic type? The trick here is to create something called a *conversion operator*. Here's where we do that in ENGLCONV:

```
operator float()
   {
   float fracfeet = inches/12;
   fracfeet += float(feet);
   return fracfeet/MTF;
   }
```

This operator takes the value of the `Distance` object of which it is a member, converts it to a `float` value representing meters, and returns this value.

This operator can be called with an explicit cast

```
mtrs = static_cast<float>(dist1);
```

or with a simple assignment

```
mtrs = dist2;
```

Both forms convert the `Distance` object to its equivalent `float` value in meters.

Conversion Between C-Strings and `String` Objects

Here's another example that uses a one-argument constructor and a conversion operator. It operates on the `String` class that we saw in the STRPLUS example earlier in this chapter.

```
// strconv.cpp
// convert between ordinary strings and class String
#include <iostream>
using namespace std;
#include <string.h>              //for strcpy(), etc.
////////////////////////////////////////////////////////////////
class String                     //user-defined string type
   {
   private:
      enum { SZ = 80 };          //size of all String objects
      char str[SZ];              //holds a C-string
   public:
      String()                   //no-arg constructor
         { str[0] = '\0'; }
      String( char s[] )         //1-arg constructor
         { strcpy(str, s); }     //   convert C-string to String
      void display() const       //display the String
         { cout << str; }
      operator char*()           //conversion operator
         { return str; }         //convert String to C-string
   };
////////////////////////////////////////////////////////////////
int main()
   {
   String s1;                    //use no-arg constructor
                                 //create and initialize C-string
   char xstr[] = "Joyeux Noel! ";

   s1 = xstr;                    //use 1-arg constructor
                                 //   to convert C-string to String
   s1.display();                 //display String
```

```
String s2 = "Bonne Annee!";  //uses 1-arg constructor
                             //to initialize String
cout << static_cast<char*>(s2); //use conversion operator
cout << endl;                  //to convert String to C-string
return 0;                       //before sending to << op
}
```

The one-argument constructor converts a normal string (an array of `char`) to an object of class `String`:

```
String(char s[])
   { strcpy(str, s); }
```

The C-string `s` is passed as an argument, and copied into the `str` data member in a newly created `String` object, using the `strcpy()` library function.

This conversion will be applied when a `String` is created, as in

```
String s2 = "Bonne Annee!";
```

or it will be applied in assignment statements, as in

```
s1 = xstr;
```

where `s1` is type `String` and `xstr` is a C-string.

A conversion operator is used to convert from a `String` type to a C-string:

```
operator char*()
   { return str; }
```

The asterisk in this expression means *pointer to*. We won't explore pointers until Chapter 10, but its use here is not hard to figure out. It means *pointer to* `char`, which is very similar to *array of type* `char`. Thus `char*` is similar to `char[]`. It's another way of specifying a C-string data type.

The conversion operator is used by the compiler in the statement

```
cout << static_cast<char*>(s2);
```

Here the `s2` variable is an argument supplied to the overloaded operator <<. Since the << operator doesn't know anything about our user-defined `String` type, the compiler looks for a way to convert `s2` to a type that << does know about. We specify the type we want to convert it to with the `char*` cast, so it looks for a conversion from `String` to C-string, finds our operator `char*()` function, and uses it to generate a C-string, which is then sent on to << to be displayed. (The effect is similar to calling the `String::display()` function, but given the ease and intuitive clarity of displaying with <<, the `display()` function is redundant and could be removed.)

Here's the output from STRCONV:

```
Joyeux Noel! Bonne Annee!
```

The STRCONV example demonstrates that conversions take place automatically not only in assignment statements but in other appropriate places, such as in arguments sent to operators (such as <<) or functions. If you supply an operator or a function with arguments of the wrong type, they will be converted to arguments of an acceptable type, provided you have defined such a conversion.

Note that you can't use an explicit assignment statement to convert a `String` to a C-string:

```
xstr = s2;
```

The C-string `xstr` is an array, and you can't normally assign to arrays (although as we'll see in Chapter 11, when you overload the assignment operator, all sorts of things are possible).

Conversions Between Objects of Different Classes

What about converting between objects of different user-defined classes? The same two methods just shown for conversions between basic types and user-defined types also apply to conversions between two user-defined types. That is, you can use a one-argument constructor or you can use a conversion operator. The choice depends on whether you want to put the conversion routine in the class declaration of the source object or of the destination object. For example, suppose you say

```
objecta = objectb;
```

where `objecta` is a member of class `A` and `objectb` is a member of class `B`. Is the conversion routine located in class `A` (the destination class, since `objecta` receives the value) or class `B` (the source class)? We'll look at both cases.

Two Kinds of Time

Our example programs will convert between two ways of measuring time: 12-hour time and 24-hour time. These methods of telling time are sometimes called *civilian time* and *military time*. Our `time12` class will represent civilian time, as used in digital clocks and airport flight departure displays. We'll assume that in this context there is no need for seconds, so `time12` uses only hours (from 1 to 12), minutes, and an "a.m." or "p.m." designation. Our `time24` class, which is for more exacting applications such as air navigation, uses hours (from 00 to 23), minutes, and seconds. Table 8.1 shows the differences.

TABLE 8.1 12-Hour and 24-Hour Time

12-Hour Time	*24-Hour Time*
12:00 a.m. (midnight)	00:00
12:01 a.m.	00:01
1:00 a.m.	01:00
6:00 a.m.	06:00
11:59 a.m	11:59
12:00 p.m. (noon)	12:00
12:01 p.m.	12:01
6:00 p.m.	18:00
11:59 p.m.	23:59

Note that 12 a.m. (midnight) in civilian time is 00 hours in military time. There is no 0 hour in civilian time.

Routine in Source Object

The first example program shows a conversion routine located in the source class. When the conversion routine is in the source class, it is commonly implemented as a conversion operator. Here's the listing for TIMES1:


```
//times1.cpp
//converts from time24 to time12 using operator in time24
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class time12
   {
   private:
      bool pm;                        //true = pm, false = am
      int hrs;                        //1 to 12
      int mins;                       //0 to 59
   public:                            //no-arg constructor
      time12() : pm(true), hrs(0), mins(0)
         {  }
                                      //3-arg constructor
      time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
         {  }
      void display() const            //format: 11:59 p.m.
         {
         cout << hrs << ':';
```

```
         if(mins < 10)
            cout << '0';             //extra zero for "01"
         cout << mins << ' ';
         string am_pm = pm ? "p.m." : "a.m.";
         cout << am_pm;
         }
   };
////////////////////////////////////////////////////////////////
class time24
   {
   private:
      int hours;                     //0 to 23
      int minutes;                   //0 to 59
      int seconds;                   //0 to 59
   public:                           //no-arg constructor
      time24() : hours(0), minutes(0), seconds(0)
         {  }
      time24(int h, int m, int s) :  //3-arg constructor
            hours(h), minutes(m), seconds(s)
         {  }
      void display() const           //format: 23:15:01
         {
         if(hours < 10)    cout << '0';
         cout << hours << ':';
         if(minutes < 10)  cout << '0';
         cout << minutes << ':';
         if(seconds < 10)  cout << '0';
         cout << seconds;
         }
      operator time12() const;       //conversion operator
   };
//-------------------------------------------------------------
time24::operator time12() const           //conversion operator
   {
   int hrs24 = hours;
   bool pm = hours < 12 ? false : true;   //find am/pm
                                          //round secs
   int roundMins = seconds < 30 ? minutes : minutes+1;
   if(roundMins == 60)                    //carry mins?
      {
      roundMins=0;
      ++hrs24;
      if(hrs24 == 12 || hrs24 == 24)      //carry hrs?
         pm = (pm==true) ? false : true;  //toggle am/pm
      }
   int hrs12 = (hrs24 < 13) ? hrs24 : hrs24-12;
```

```
   if(hrs12==0)                          //00 is 12 a.m.
      { hrs12=12; pm=false; }
   return time12(pm, hrs12, roundMins);
   }
//////////////////////////////////////////////////////////////
int main()
   {
   int h, m, s;

   while(true)
      {                                  //get 24-hr time from user
      cout << "Enter 24-hour time: \n";
      cout << "   Hours (0 to 23): "; cin >> h;
      if(h > 23)                         //quit if hours > 23
         return(1);
      cout << "   Minutes: ";  cin >> m;
      cout << "   Seconds: ";  cin >> s;

      time24 t24(h, m, s);               //make a time24
      cout << "You entered: ";           //display the time24
      t24.display();

      time12 t12 = t24;                  //convert time24 to time12

      cout << "\n12-hour time: ";        //display equivalent time12
      t12.display();
      cout << "\n\n";
      }
   return 0;
   }
```

In the `main()` part of TIMES1 we define an object of type `time24`, called `t24`, and give it values for hours, minutes, and seconds obtained from the user. We also define an object of type `time12`, called `t12`, and initialize it to `t24` in the statement

```
time12 t12 = t24;
```

Since these objects are from different classes, the assignment involves a conversion, and—as we specified—in this program the conversion operator is a member of the `time24` class. Here's its declarator:

```
time24::operator time12() const            //conversion operator
}
```

This function transforms the object of which it is a member to a `time12` object, and returns this object, which `main()` then assigns to `t12`. Here's some interaction with TIMES1:

```
Enter 24-hour time:
   Hours (0 to 23): 17
   Minutes: 59
   Seconds: 45
You entered: 17:59:45
12-hour time: 6:00 p.m.
```

The seconds value is rounded up, pushing the 12-hour time from 5:59 p.m. to 6:00 p.m. Entering an hours value greater than 23 causes the program to exit.

Routine in Destination Object

Let's see how the same conversion is carried out when the conversion routine is in the destination class. In this situation it's common to use a one-argument constructor. However, things are complicated by the fact that the constructor in the destination class must be able to access the data in the source class to perform the conversion. The data in `time24`—`hours`, `minutes` and `seconds`—is private, so we must provide special member functions in `time24` to allow direct access to it. These are called `getHrs()`, `getMins()`, and `getSecs()`.

Here's the listing for TIMES2:

```
//times2.cpp
//converts from time24 to time12 using constructor in time12
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class time24
   {
   private:
      int hours;                    //0 to 23
      int minutes;                  //0 to 59
      int seconds;                  //0 to 59
   public:                          //no-arg constructor
      time24() : hours(0), minutes(0), seconds(0)
         {  }
      time24(int h, int m, int s) :  //3-arg constructor
             hours(h), minutes(m), seconds(s)
         {  }
      void display() const          //format 23:15:01
         {
         if(hours < 10)    cout << '0';
         cout << hours << ':';
         if(minutes < 10)  cout << '0';
         cout << minutes << ':';
         if(seconds < 10)  cout << '0';
         cout << seconds;
         }
```

```
      int getHrs() const   { return hours; }
      int getMins() const  { return minutes; }
      int getSecs() const  { return seconds; }
   };
////////////////////////////////////////////////////////////////
class time12
   {
   private:
      bool pm;                        //true = pm, false = am
      int hrs;                        //1 to 12
      int mins;                       //0 to 59
   public:                            //no-arg constructor
      time12() : pm(true), hrs(0), mins(0)
         {  }
      time12(time24);                 //1-arg constructor
                                      //3-arg constructor
      time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
         {  }
      void display() const
         {
         cout << hrs << ':';
         if(mins < 10)  cout << '0';  //extra zero for "01"
         cout << mins << ' ';
         string am_pm = pm ? "p.m." : "a.m.";
         cout << am_pm;
         }
   };
//-------------------------------------------------------------
time12::time12( time24 t24 )          //1-arg constructor
   {                                  //converts time24 to time12
   int hrs24 = t24.getHrs();          //get hours
                                      //find am/pm
   pm = t24.getHrs() < 12 ? false : true;

   mins = (t24.getSecs() < 30) ?      //round secs
                         t24.getMins() : t24.getMins()+1;
   if(mins == 60)                     //carry mins?
      {
      mins=0;
      ++hrs24;
      if(hrs24 == 12 || hrs24 == 24)      //carry hrs?
         pm = (pm==true) ? false : true;  //toggle am/pm
      }
   hrs = (hrs24 < 13) ? hrs24 : hrs24-12; //convert hrs
   if(hrs==0)                         //00 is 12 a.m.
```

```
      { hrs=12; pm=false; }
   }
//////////////////////////////////////////////////////////////
int main()
   {
   int h, m, s;

   while(true)
      {                              //get 24-hour time from user
      cout << "Enter 24-hour time: \n";
      cout << "   Hours (0 to 23): "; cin >> h;
      if(h > 23)                     //quit if hours > 23
         return(1);
      cout << "   Minutes: ";  cin >> m;
      cout << "   Seconds: ";  cin >> s;

      time24 t24(h, m, s);           //make a time24
      cout << "You entered: ";       //display the time24
      t24.display();

      time12 t12 = t24;              //convert time24 to time12

      cout << "\n12-hour time: ";    //display equivalent time12
      t12.display();
      cout << "\n\n";
      }
   return 0;
   }
```

The conversion routine is the one-argument constructor from the `time12` class. This function sets the object of which it is a member to values that correspond to the `time24` values of the object received as an argument. It works in much the same way as the conversion operator in TIMES1, except that it must work a little harder to access the data in the `time24` object, using `getHrs()` and similar functions.

The `main()` part of TIMES2 is the same as that in TIMES1. The one-argument constructor again allows the `time24`-to-`time12` conversion to take place in the statement

```
time12 t12 = t24;
```

The output is similar as well. The difference is behind the scenes, where the conversion is handled by a constructor in the destination object rather than a conversion operator in the source object.

Conversions: When to Use What

When should you use the one-argument constructor in the destination class, as opposed to the conversion operator in the source class? Mostly you can take your pick. However, sometimes the choice is made for you. If you have purchased a library of classes, you may not have access to their source code. If you use an object of such a class as the source in a conversion, you'll have access only to the destination class, and you'll need to use a one-argument constructor. If the library class object is the destination, you must use a conversion operator in the source.

UML Class Diagrams

We introduced the UML in Chapter 1, "The Big Picture." Now that you know something about classes, let's take a look at our first UML feature: the *class diagram*. This diagram offers a new way of looking at object-oriented programs, and may throw some additional light on the workings of the TIMES1 and TIMES2 programs.

Looking at the listing for TIMES1 we can see that there are two classes: `time12` and `time24`. In a UML class diagram, classes are represented by rectangles, as shown in Figure 8.3.

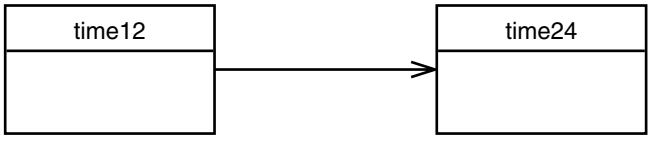

FIGURE 8.3
UML class diagram of the TIMES1 program.

Each class rectangle is divided into sections by horizontal lines. The class name goes in the top section. We don't show them here, but you can include sections for member data (called *attributes* in the UML) and member functions (called *operations*).

Associations

Classes may have various kinds of relationships with each other. The classes in TIMES1 are related by *association*. We indicate this with a line connecting their rectangles. (We'll see what another kind of class relationship, generalization, looks like in Chapter 9, "Inheritance.")

What constitutes an association? Conceptually, the real-world entities that are represented by classes in the program have some kind of obvious relationship. Drivers are related to cars, books are related to libraries, race horses are related to race tracks. If such entities were classes in a program, they would be related by association.

In the TIMES2.CPP program, we can see that class `time12` is associatd with class `time24` because we are converting objects of one class into objects of the other.

A class association actually implies that objects of the classes, rather than the classes themselves, have some kind of relationship. Typically, two classes are associated if an object of one class calls a member function (an *operation*) of an object of the other class. An association might also exist if an attribute of one class is an object of the other class.

In the TIMES1 program, an object of the `time12` class, called `t12`, calls the conversion routine `operator time12()` in the object `t24` of the `time24` class. This happens in `main()` in the statement

```
time12 t12 = t24;               //convert time24 to time12
```

Such a call is represented by an association line between the two classes.

Navigability

We can add an open arrowhead to indicate the direction or *navigability* of the association. (As we'll see later, closed arrowheads have a different meaning.) Because `time12` calls `time24`, the arrow points from `time12` to `time24`. It's called a *unidirectional* association because it only goes one way. If each of two classes called an operation in the other, there would be arrowheads on both ends of the line and it would be called a *bidirectional* association. As are many things in the UML, navigability arrows are optional.

Pitfalls of Operator Overloading and Conversion

Operator overloading and type conversions give you the opportunity to create what amounts to an entirely new language. When `a`, `b`, and `c` are objects from user-defined classes, and `+` is overloaded, the statement

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

can mean something quite different than it does when `a`, `b`, and `c` are variables of basic data types. The ability to redefine the building blocks of the language can be a blessing in that it can make your listing more intuitive and readable. It can also have the opposite effect, making your listing more obscure and hard to understand. Here are some guidelines.

Use Similar Meanings

Use overloaded operators to perform operations that are as similar as possible to those performed on basic data types. You could overload the `+` sign to perform subtraction, for example, but that would hardly make your listings more comprehensible.

Overloading an operator assumes that it makes sense to perform a particular operation on objects of a certain class. If we're going to overload the + operator in class `X`, the result of adding two objects of class `X` should have a meaning at least somewhat similar to addition. For example, in this chapter we showed how to overload the + operator for the English `Distance` class. Adding two distances is clearly meaningful. We also overloaded + for the `String` class. Here we interpret the addition of two strings to mean placing one string after another to form a third. This also has an intuitively satisfying interpretation. But for many classes it may not be reasonable to talk about "adding" their objects. You probably wouldn't want to add two objects of a class called `employee` that held personal data, for example.

Use Similar Syntax

Use overloaded operators in the same way you use basic types. For example, if `alpha` and `beta` are basic types, the assignment operator in the statement

```
alpha += beta;
```

sets `alpha` to the sum of `alpha` and `beta`. Any overloaded version of this operator should do something analogous. It should probably do the same thing as

```
alpha = alpha + beta;
```

where the + is overloaded.

If you overload one arithmetic operator, you may for consistency want to overload all of them. This will prevent confusion.

Some syntactical characteristics of operators can't be changed. As you may have discovered, you can't overload a binary operator to be a unary operator, or vice versa.

Show Restraint

Remember that if you have overloaded the + operator, anyone unfamiliar with your listing will need to do considerable research to find out what a statement like

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

really means. If the number of overloaded operators grows too large, and if the operators are used in nonintuitive ways, the whole point of using them is lost, and reading the listing becomes harder instead of easier. Use overloaded operators sparingly, and only when the usage is obvious. When in doubt, use a function instead of an overloaded operator, since a function name can state its own purpose. If you write a function to find the left side of a string, for example, you're better off calling it `getleft()` than trying to overload some operator such as `&&` to do the same thing.

Avoid Ambiguity

Suppose you use both a one-argument constructor and a conversion operator to perform the same conversion (`time24` to `time12`, for example). How will the compiler know which conversion to use? It won't. The compiler does not like to be placed in a situation where it doesn't know what to do, and it will signal an error. So avoid doing the same conversion in more than one way.

Not All Operators Can Be Overloaded

The following operators cannot be overloaded: the member access or dot operator (`.`), the scope resolution operator (`::`), and the conditional operator (`?:`). Also, the pointer-to-member operator (`->`), which we have not yet encountered, cannot be overloaded. In case you wondered, no, you can't create new operators (like `*&`) and try to overload them; only existing operators can be overloaded.

Keywords `explicit` and `mutable`

Let's look at two unusual keywords: `explicit` and `mutable`. They have quite different effects, but are grouped together here because they both modify class members. The `explicit` keyword relates to data conversion, but `mutable` has a more subtle purpose.

Preventing Conversions with `explicit`

There may be some specific conversions you have decided are a good thing, and you've taken steps to make them possible by installing appropriate conversion operators and one-argument constructors, as shown in the TIME1 and TIME2 examples. However, there may be other conversions that you don't want to happen. You should actively discourage any conversion that you don't want. This prevents unpleasant surprises.

It's easy to prevent a conversion performed by a conversion operator: just don't define the operator. However, things aren't so easy with constructors. You may want to construct objects using a single value of another type, but you may not want the implicit conversions a one-argument constructor makes possible in other situations. What to do?

Standard C++ includes a keyword, `explicit`, to solve this problem. It's placed just before the declaration of a one-argument constructor. The EXPLICIT example program (based on the ENGLCON program) shows how this looks.

```
//explicit.cpp
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                  //English Distance class
```

```
   {
   private:
      const float MTF;          //meters to feet
      int feet;
      float inches;
   public:                      //no-args constructor
      Distance() : feet(0), inches(0.0), MTF(3.280833F)
         {  }
                                //EXPLICIT one-arg constructor
      explicit Distance(float meters) : MTF(3.280833F)
         {
         float fltfeet = MTF * meters;
         feet = int(fltfeet);
         inches = 12*(fltfeet-feet);
         }
      void showdist()           //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   void fancyDist(Distance);    //declaration
   Distance dist1(2.35F);       //uses 1-arg constructor to
                                //convert meters to Distance

// Distance dist1 = 2.35F;      //ERROR if ctor is explicit
   cout << "\ndist1 = "; dist1.showdist();

   float mtrs = 3.0F;
   cout << "\ndist1 ";
// fancyDist(mtrs);             //ERROR if ctor is explicit

   return 0;
   }
//--------------------------------------------------------------
void fancyDist(Distance d)
   {
   cout << "(in feet and inches) = ";
   d.showdist();
   cout << endl;
   }
```

This program includes a function (`fancyDist()`) that embellishes the output of a `Distance` object by printing the phrase "(in feet and inches)" before the feet and inches figures. The argument to this function is a `Distance` variable, and you can call `fancyDist()` with such a variable with no problem.

The tricky part is that, unless you take some action to prevent it, you can also call `fancyDist()` with a variable of type `float` as the argument:

```
fancyDist(mtrs);
```

The compiler will realize it's the wrong type and look for a conversion operator. Finding a `Distance` constructor that takes type `float` as an argument, it will arrange for this constructor to convert `float` to `Distance` and pass the `Distance` value to the function. This is an *implicit* conversion, one which you may not have intended to make possible.

However, if we make the constructor *explicit*, we prevent implicit conversions. You can check this by removing the comment symbol from the call to `fancyDist()` in the program: the compiler will tell you it can't perform the conversion. Without the explicit keyword, this call is perfectly legal.

As a side effect of the explicit constructor, note that you can't use the form of object initialization that uses an equal sign

```
Distance dist1 = 2.35F;
```

whereas the form with parentheses

```
Distance dist1(2.35F);
```

works as it always has.

Changing `const` Object Data Using `mutable`

Ordinarily, when you create a `const` object (as described in Chapter 6), you want a guarantee that none of its member data can be changed. However, a situation occasionally arises where you want to create `const` objects that have some specific member data item that needs to be modified despite the object's `const`ness.

As an example, let's imagine a window (the kind that Windows programs commonly draw on the screen). It may be that some of the features of the window, such as its scrollbars and menus, are *owned* by the window. Ownership is common in various programming situations, and indicates a greater degree of independence than when one object is an attribute of another. In such a situation an object may remain unchanged, except that its owner may change. A scrollbar retains the same size, color, and orientation, but its ownership may be transferred from one window to another. It's like what happens when your bank sells your mortgage to another bank; all the terms of the mortgage are the same, but the owner is different.

Let's say we want to be able to create `const` scrollbars in which attributes remain unchanged, except for their ownership. That's where the `mutable` keyword comes in. The MUTABLE program shows how this looks.

```
//mutable.cpp
#include <iostream>
#include <string>
using namespace std;
//////////////////////////////////////////////////////////////
class scrollbar
   {
   private:
      int size;                     //related to constness
      mutable string owner;         //not relevant to constness
   public:
      scrollbar(int sz, string own) : size(sz), owner(own)
         {  }
      void setSize(int sz)             //changes size
         { size = sz; }
      void setOwner(string own) const  //changes owner
         { owner = own; }
      int getSize() const              //returns size
         { return size; }
      string getOwner() const          //returns owner
         { return owner; }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   const scrollbar sbar(60, "Window1");

// sbar.setSize(100);             //can't do this to const obj
   sbar.setOwner("Window2");      //this is OK
                                  //these are OK too
   cout << sbar.getSize() << ", " << sbar.getOwner() << endl;
   return 0;
   }
```

The `size` attribute represents the scrollbar data that cannot be modified in `const` objects. The `owner` attribute, however, can change, even if the object is `const`. To permit this, it's made `mutable`. In `main()` we create a `const` object `sbar`. Its size cannot be modified, but its owner can, using the `setOwner()` function. (In a non-`const` object, of course, both attributes could be modified.) In this situation, `sbar` is said to have *logical* `const`*ness*. That means that in theory it can't be modified, but in practice it can, in a limited way.

Summary

In this chapter we've seen how the normal C++ operators can be given new meanings when applied to user-defined data types. The keyword `operator` is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.

Closely related to operator overloading is the issue of *type conversion*. Some conversions take place between user-defined types and basic types. Two approaches are used in such conversions: A one-argument constructor changes a basic type to a user-defined type, and a conversion operator converts a user-defined type to a basic type. When one user-defined type is converted to another, either approach can be used.

Table 8.2 summarizes these conversions.

TABLE 8.2 Type Conversions

	Routine in Destination	*Routine in Source*
Basic to basic	**(Built-In Conversion Operators)**	
Basic to class	Constructor	N/A
Class to basic	N/A	Conversion operator
Class to class	Constructor	Conversion operator

A constructor given the keyword `explicit` cannot be used in implicit data conversion situations. A data member given the keyword `mutable` can be changed, even if its object is `const`.

UML class diagrams show classes and relationships between classes. An association represents a conceptual relationship between the real-world objects that the program's classes represent. Associations can have a direction from one class to another; this is called navigability.

Questions

Answers to these questions can be found in Appendix G.

1. Operator overloading is
   a. making C++ operators work with objects.
   b. giving C++ operators more than they can handle.
   c. giving new meanings to existing C++ operators.
   d. making new C++ operators.
2. Assuming that class `X` does not use any overloaded operators, write a statement that subtracts an object of class `X`, `x1`, from another such object, `x2`, and places the result in `x3`.

3. Assuming that class `X` includes a routine to overload the `-` operator, write a statement that would perform the same task as that specified in Question 2.
4. True or false: The `>=` operator can be overloaded.
5. Write a complete definition for an overloaded operator for the `Counter` class of the COUNTPP1 example that, instead of incrementing the count, decrements it.
6. How many arguments are required in the definition of an overloaded unary operator?
7. Assume a class `C` with objects `obj1`, `obj2`, and `obj3`. For the statement `obj3 = obj1 - obj2` to work correctly, the overloaded `-` operator must

   a. take two arguments.

   b. return a value.

   c. create a named temporary object.

   d. use the object of which it is a member as an operand.
8. Write a complete definition for an overloaded `++` operator for the `Distance` class from the ENGLPLUS example. It should add `1` to the `feet` member data, and make possible statements like

   ```
   dist1++;
   ```
9. Repeat Question 8, but allow statements like the following:

   ```
   dist2 = dist1++;
   ```
10. When used in prefix form, what does the overloaded `++` operator do differently from what it does in postfix form?
11. Here are two declarators that describe ways to add two string objects:

    ```
    void add(String s1, String s2)
    String operator + (String s)
    ```

    Match the following from the first declarator with the appropriate selection from the second:

    function name (`add`) matches _____.

    return value (type `void`) matches _____.

    first argument (`s1`) matches _____.

    second argument (`s2`) matches _____.

    object of which function is a member matches _____.

    a. argument (`s`)

    b. object of which operator is a member

    c. operator (`+`)

    d. return value (type `String`)

    e. no match for this item

12. True or false: An overloaded operator always requires one less argument than its number of operands.
13. When you overload an arithmetic assignment operator, the result
    a. goes in the object to the right of the operator.
    b. goes in the object to the left of the operator.
    c. goes in the object of which the operator is a member.
    d. must be returned.
14. Write the complete definition of an overloaded `++` operator that works with the `String` class from the STRPLUS example and has the effect of changing its operand to uppercase. You can use the library function `toupper()` (header file CCTYPE), which takes as its only argument the character to be changed and returns the changed character (or the same character if no change is necessary).
15. To convert from a user-defined class to a basic type, you would most likely use
    a. a built-in conversion operator.
    b. a one-argument constructor.
    c. an overloaded = operator.
    d. a conversion operator that's a member of the class.
16. True or false: The statement `objA=objB;` will cause a compiler error if the objects are of different classes.
17. To convert from a basic type to a user-defined class, you would most likely use
    a. a built-in conversion operator.
    b. a one-argument constructor.
    c. an overloaded = operator.
    d. a conversion operator that's a member of the class.
18. True or false: If you've defined a constructor to handle definitions like `aclass obj = intvar;` you can also make statements like `obj = intvar;`.
19. If `objA` is in class `A`, and `objB` is in class `B`, and you want to say `objA = objB;`, and you want the conversion routine to go in class `A`, what type of conversion routine might you use?
20. True or false: The compiler won't object if you overload the `*` operator to perform division.
21. In a UML class diagram, an association arises whenever
    a. two classes are in the same program.
    b. one class is descended from another.
    c. two classes use the same global variable.
    d. one class calls a member function in the other class.

22. In the UML, member data items are called _____ and member functions are called _____.
23. True or false: rectangles that symbolize classes have rounded corners.
24. Navigability from class A to class B means that
    a. an object of class A can call an operation in an object of class B.
    b. there is a relationship between class A and class B.
    c. objects can go from class A to class B.
    d. messages from class B are received by class A.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. To the `Distance` class in the ENGLPLUS program in this chapter, add an overloaded `-` operator that subtracts two distances. It should allow statements like `dist3= dist1-dist2;`. Assume that the operator will never be used to subtract a larger number from a smaller one (that is, negative distances are not allowed).

*2. Write a program that substitutes an overloaded `+=` operator for the overloaded `+` operator in the STRPLUS program in this chapter. This operator should allow statements like

```
s1 += s2;
```

where `s2` is added (concatenated) to `s1` and the result is left in `s1`. The operator should also permit the results of the operation to be used in other calculations, as in

```
s3 = s1 += s2;
```

*3. Modify the `time` class from Exercise 3 in Chapter 6 so that instead of a function `add_time()` it uses the overloaded `+` operator to add two times. Write a program to test this class.

*4. Create a class `Int` based on Exercise 1 in Chapter 6. Overload four integer arithmetic operators (`+`, `-`, `*`, and `/`) so that they operate on objects of type `Int`. If the result of any such arithmetic operation exceeds the normal range of `int`s (in a 32-bit environment)—from 2,147,483,648 to –2,147,483,647—have the operator print a warning and terminate the program. Such a data type might be useful where mistakes caused by arithmetic overflow are unacceptable. Hint: To facilitate checking for overflow, perform the calculations using type `long double`. Write a program to test this class.

5. Augment the `time` class referred to in Exercise 3 to include overloaded increment (`++`) and decrement (`--`) operators that operate in both prefix and postfix notation and return values. Add statements to `main()` to test these operators.

6. Add to the `time` class of Exercise 5 the ability to subtract two time values using the overloaded (`-`) operator, and to multiply a time value by a number of type `float`, using the overloaded (`*`) operator.
7. Modify the `fraction` class in the four-function fraction calculator from Exercise 11 in Chapter 6 so that it uses overloaded operators for addition, subtraction, multiplication, and division. (Remember the rules for fraction arithmetic in Exercise 12 in Chapter 3, "Loops and Decisions.") Also overload the == and `!=` comparison operators, and use them to exit from the loop if the user enters 0/1, 0/1 for the values of the two input fractions. You may want to modify the `lowterms()` function so that it returns the value of its argument reduced to lowest terms. This makes it more useful in the arithmetic functions, where it can be applied just before the answer is returned.
8. Modify the `bMoney` class from Exercise 12 in Chapter 7, "Arrays and Strings," to include the following arithmetic operations, performed with overloaded operators:

```
bMoney = bMoney + bMoney
bMoney = bMoney - bMoney
bMoney = bMoney * long double  (price per widget times number of widgets)
long double = bMoney / bMoney  (total price divided by price per widget)
bMoney = bMoney / long double  (total price divided by number of widgets)
```

   Notice that the `/` operator is overloaded twice. The compiler can distinguish between the two usages because the arguments are different. Remember that it's easy to perform arithmetic operations on `bMoney` objects by performing the same operation on their `long double` data.

   Make sure the `main()` program asks the user to enter two money strings and a floating-point number. It should then carry out all five operations and display the results. This should happen in a loop, so the user can enter more numbers if desired.

   Some money operations don't make sense: `bMoney * bMoney` doesn't represent anything real, since there is no such thing as square money; and you can't add `bMoney` to `long double` (what's dollars plus widgets?). To make it impossible to compile such illegal operations, don't include conversion operators for `bMoney` to `long double` or `long double` to `bMoney`. If you do, and you write an expression like

```
bmon2 = bmon1 + widgets;  // doesn't make sense
```

   then the compiler will automatically convert widgets to `bMoney` and carry out the addition. Without them, the compiler will flag such conversions as errors, making it easier to catch conceptual mistakes. Also, make any conversion constructors explicit.

   There are some other plausible money operations that we don't yet know how to perform with overloaded operators, since they require an object on the right side of the operator but not the left:

```
long double * bMoney  // can't do this yet: bMoney only on right
long double / bMoney  // can't do this yet: bMoney only on right
```

   We'll learn how to handle this situation when we discuss friend functions in Chapter 11.

9. Augment the `safearay` class in the ARROVER3 program in this chapter so that the user can specify both the upper and lower bound of the array (indexes running from 100 to 200, for example). Have the overloaded subscript operator check the index each time the array is accessed to ensure that it is not out of bounds. You'll need to add a two-argument constructor that specifies the upper and lower bounds. Since we have not yet learned how to allocate memory dynamically, the member data will still be an array that starts at 0 and runs up to 99, but perhaps you can map the indexes for the `safearay` into different indexes in the real `int` array. For example, if the client selects a range from 100 to 175, you could map this into the range from `arr[0]` to `arr[75]`.
10. For math buffs only: Create a class `Polar` that represents the points on the plain as polar coordinates (radius and angle). Create an overloaded `+operator` for addition of two `Polar` quantities. "Adding" two points on the plain can be accomplished by adding their X coordinates and then adding their Y coordinates. This gives the X and Y coordinates of the "answer." Thus you'll need to convert two sets of polar coordinates to rectangular coordinates, add them, then convert the resulting rectangular representation back to polar.
11. Remember the `sterling` structure? We saw it in Exercise 10 in Chapter 2, "C++ Programming Basics," and in Exercise 11 in Chapter 5, among other places. Turn it into a class, with pounds (type `long`), shillings (type `int`), and pence (type `int`) data items. Create the following member functions:
    - no-argument constructor
    - one-argument constructor, taking type `double` (for converting from decimal pounds)
    - three-argument constructor, taking pounds, shillings, and pence
    - `getSterling()` to get an amount in pounds, shillings, and pence from the user, format £9.19.11
    - `putSterling()` to display an amount in pounds, shillings, and pence, format £9.19.11
    - addition (`sterling + sterling`) using overloaded `+` operator
    - subtraction (`sterling - sterling`) using overloaded `-` operator
    - multiplication (`sterling * double`) using overloaded `*` operator
    - division (`sterling / sterling`) using overloaded `/` operator
    - division (`sterling / double`) using overloaded `/` operator
    - operator `double` (to convert to `double`)

    To perform arithmetic, you could (for example) add each object's data separately: Add the pence, carry, add the shillings, carry, and so on. However, it's easier to use the conversion operator to convert both `sterling` objects to type `double`, perform the arithmetic on the doubles, and convert back to `sterling`. Thus the overloaded `+` operator looks like this:

```
sterling sterling::operator + (sterling s2)
   {
   return sterling( double(sterling(pounds, shillings, pence))
                   + double(s2) );
   }
```

This creates two temporary `double` variables, one derived from the object of which the function is a member, and one derived from the argument `s2`. These `double` variables are then added, and the result is converted back to `sterling` and returned.

Notice that we use a different philosophy with the `sterling` class than with the `bMoney` class. With sterling we use conversion operators, thus giving up the ability to catch illegal math operations but gaining simplicity in writing the overloaded math operators.

12. Write a program that incorporates both the `bMoney` class from Exercise 8 and the `sterling` class from Exercise 11. Write conversion operators to convert between `bMoney` and `sterling`, assuming that one pound (£1.0.0) equals fifty dollars ($50.00). This was the approximate exchange rate in the 19th century when the British Empire was at its height and the pounds-shillings-pence format was in use. Write a `main()` program that allows the user to enter an amount in either currency and then converts it to the other currency and displays the result. Minimize any modifications to the existing `bMoney` and `sterling` classes.