APPENDIX 8

Overloading Operators as Member Operators

In this book we have normally overloaded operators by treating them as friends of the class. For example, in Display 11.5 of Chapter 11 we overloaded the + operator as a friend. We did this by labeling the operator a friend inside the class definition, as follows:

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

We then defined the overloaded operator + outside the class definition (as shown in Display 11.5).

It is also possible to overload the operator + (and other operators) as **member operators.** To overload the + operator as a member operator, the class definition would instead begin as follows:

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

Note that when a binary operator is overloaded as a member operator, there is only one (not two) parameters. The calling object serves as the first parameter. For example, consider the following code:

```
Money cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

When + is overloaded as a member operator, then in the expression `cost + tax`, the variable `cost` is the calling object and `tax` is the one argument to +.

The definition of the member operator + would be as follows:

```
Money Money::operator +(const Money& amount2)
{
    Money temp;
```

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

Notice the following line from this member operator definition:

```
temp.all_cents = all_cents + amount2.all_cents;
```

The first argument to + is an unqualified `all_cents`, and so it is the member variable `all_cents` *of the calling object*.

Overloading an operator as a member variable can seem strange at first, but it is easy to get used to the new details. Many experts advocate always overloading operators as member operators rather than as friends. That is more in the spirit of object-oriented programming. However, there is a big disadvantage to overloading a binary operator as a member operator. When you overload a binary operator as a member operator, the two arguments are no longer symmetric. One is a calling object and only the second "argument" is a true argument. This is unaesthetic, but it also has a very practical shortcoming. Any automatic type conversion will only apply to the second argument. So, for example, the following would be legal:

```
Money base_amount(100, 60), full_amount;
full_amount = base_amount + 25;
```

This is because `Money` has a constructor with one argument of type *`long`*, and so the value 25 will be considered a *`long`* value that is automatically converted to a value of type `Money`.

However, if you overload + as a member operator, then you cannot reverse the two arguments to +. The following is illegal:

```
full_amount = 25 + base_amount;
```

This is because 25 cannot be a calling object. Conversion of *`long`* values to type `Money` works for arguments but not for calling objects.

On the other hand, if you overload + as a friend, then the following is perfectly legal:

```
full_amount = 25 + base_amount;
```