

6

Introducing Operator Overloading

chapter objectives

- 6.1** The basics of operator overloading
- 6.2** Overloading binary operators
- 6.3** Overloading the relational and logical operators
- 6.4** Overloading a unary operator
- 6.5** Using friend operator functions
- 6.6** A closer look at the assignment operator
- 6.7** Overloading the [] subscript operator

THIS chapter introduces another important C++ feature: operator overloading. This feature allows you to define the meaning of the C++ operators relative to classes that you define. By overloading operators, you can seamlessly add new data types to your program.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Show how to overload the constructor for the following class so that uninitialized objects can also be created. (When creating uninitialized objects, give **x** and **y** the value 0.)

```
class myclass {  
    int x, y;  
public:  
    myclass(int i, int j) { x=i; y=j; }  
    // ...  
};
```

2. Using the class from Question 1, show how you can avoid overloading **myclass()** by using default arguments.
3. What is wrong with the following declaration?

```
int f(int a=0, double balance);
```
4. What is wrong with these two overloaded functions?

```
void f(int a);  
void f(int &a);
```
5. When is it appropriate to use default arguments? When is it probably a bad idea?
6. Given the following class definition, is it possible to dynamically allocate an array of these objects?


```

class test {
    char *p;
    int *q;
    int count;
public:
    test(char *x, int *y, int c) {
        p = x;
        q = y;
        count = c;
    }
    // ...
};

```

7. What is a copy constructor and under what circumstances is it called?

6.1

THE BASICS OF OPERATOR OVERLOADING

Operator overloading resembles function overloading. In fact, operator overloading is really just a type of function overloading. However, some additional rules apply. For example, an operator is always overloaded relative to a user-defined type, such as a class. Other differences will be discussed as needed.

When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.

To overload an operator, you create an *operator function*. Most often an operator function is a member or a friend of the class for which it is defined. However, there is a slight difference between a member operator function and a friend operator function. The first part of this chapter discusses the creation of member operator functions. Then friend operator functions are discussed.

The general form of a member operator function is shown here:

```

return-type class-name::operator#(arg-list)
{
    // operation to be performed
}

```

The return type of an operator function is often the class for which it is defined. (However, an operator function is free to return any type.) The operator being overloaded is substituted for the #. For example, if the + is being overloaded, the function name would be **operator +**. The contents of *arg-list* vary depending upon how the operator function is implemented and the type of operator being overloaded.

There are two important restrictions to remember when you are overloading an operator. First, the precedence of the operator cannot be changed. Second, the number of operands that an operator takes cannot be altered. For example, you cannot overload the / operator so that it takes only one operand.

Most C++ operators can be overloaded. The only operators that you cannot overload are shown here:

. :: .* ?

Also, you cannot overload the preprocessor operators. (The .* operator is highly specialized and is beyond the scope of this book.)

Remember that C++ defines operators very broadly, including such things as the [] subscript operators, the () function call operators, **new** and **delete**, and the . (dot) and -> (arrow) operators. However, this chapter concentrates on overloading the most commonly used operators.

Except for the =, operator functions are inherited by any derived class. However, a derived class is free to overload any operator it chooses (including those overloaded by the base class) relative to itself.

You have been using two overloaded operators: << and >>. These operators have been overloaded to perform console I/O. As mentioned, overloading these operators to perform I/O does not prevent them from performing their traditional jobs of left shift and right shift.

While it is permissible for you to have an operator function perform any activity—whether related to the traditional use of the operator or not—it is best to have an overloaded operator's actions stay within the spirit of the operator's traditional use. When you create overloaded operators that stray from this principle, you run the risk of substantially destructuring your program. For example, overloading the / so that the phrase "I like C++" is written to a disk file 300 times is a fundamentally confusing misuse of operator overloading!

The preceding paragraph notwithstanding, there will be times when you need to use an operator in a way not related to its traditional

usage. The two best examples of this are the << and >> operators, which are overloaded for console I/O. However, even in these cases, the left and right arrows provide a visual "clue" to their meaning. Therefore, if you need to overload an operator in a nonstandard way, make the greatest effort possible to use an appropriate operator.

One final point: operator functions cannot have default arguments.

OVERLOADING BINARY OPERATORS

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by **this**.

It is important to understand that operator functions can be written with many variations. The examples here and elsewhere in this chapter are not exhaustive, but they do illustrate several of the most common techniques.

EXAMPLES

1. The following program overloads the + operator relative to the **coord** class. This class is used to maintain X,Y coordinates.

```
// Overload the + relative to coord class.
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2);
};

// Overload + relative to coord class.
coord coord::operator+(coord ob2)
{
```



```
    coord temp;
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // add two objects - this calls operator+()

    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

This program displays the following:

```
(o1+o2) X: 15, Y: 13
```

Let's look closely at this program. The **operator +()** function returns an object of type **coord** that has the sum of each operand's X coordinates in **x** and the sum of the Y coordinates in **y**. Notice that a temporary object called **temp** is used inside **operator +()** to hold the result, and it is this object that is returned. Notice also that neither operand is modified. The reason for **temp** is easy to understand. In this situation (as in most), the **+** has been overloaded in a manner consistent with its normal arithmetic use. Therefore, it was important that neither operand be changed. For example, when you add $10 + 4$, the result is 14, but neither the 10 nor the 4 is modified. Thus, a temporary object is needed to hold the result.

The reason that the **operator +()** function returns an object of type **coord** is that it allows the result of the addition of **coord** objects to be used in larger expressions. For example, the statement

```
o3 = o1 + o2;
```

is valid only because the result of `o1 + o2` is a `coord` object that can be assigned to `o3`. If a different type had been returned, this statement would have been invalid. Further, by returning a `coord` object, the addition operator allows a string of additions. For example, this is a valid statement:

```
o3 = o1 + o2 + o1 + o3;
```

Although there will be situations in which you want an operator function to return something other than an object for which it is defined, most of the time operator functions that you create will return an object of their class. (The major exception to this rule is when the relational and logical operators are overloaded. This situation is examined in Section 6.3, "Overloading the Relational and Logical Operators," later in this chapter.)

One final point about this example. Because a `coord` object is returned, the following statement is also perfectly valid:

```
(o1+o2).get_xy(x, y);
```

Here the temporary object returned by `operator + ()` is used directly. Of course, after this statement has executed, the temporary object is destroyed.

- The following version of the preceding program overloads the `-` and the `=` operators relative to the `coord` class.

```
// Overload the +, -, and = relative to coord class.
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2);
    coord operator-(coord ob2);
    coord operator=(coord ob2);
};

// Overload + relative to coord class.
```



```
coord coord::operator+(coord ob2)
{
    coord temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// Overload - relative to coord class.
coord coord::operator-(coord ob2)
{
    coord temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;

    return temp;
}

// Overload = relative to coord.
coord coord::operator=(coord ob2)
{
    x = ob2.x;
    y = ob2.y;

    return *this; // return the object that is assigned
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // add two objects - this calls operator+()
    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 - o2; // subtract two objects
    o3.get_xy(x, y);
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // assign an object
```

```

o3.get_xy(x, y);
cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";

return 0;
}

```

The **operator-()** function is implemented similarly to **operator+()**. However, the above example illustrates a crucial point when you are overloading an operator in which the order of the operands is important. When the **operator+()** function was created, it did not matter which order the operands were in. (That is, $A+B$ is the same as $B+A$.) However, the subtraction operation is order dependent. Therefore, to correctly overload the subtraction operator, it is necessary to subtract the operand on the right from the operand on the left. Because it is the left operand that generates the call to **operator-()**, the subtraction must be in this order:

```
x = ob2.x;
```



When a binary operator is overloaded, the left operand is passed implicitly to the function and the right operand is passed as an argument.

Now look at the assignment operator function. The first thing you should notice is that the left operand (that is, the object being assigned a value) is modified by the operation. This is in keeping with the normal meaning of assignment. The second thing to notice is that the function returns ***this**. That is, the **operator=()** function returns the object that is being assigned to. The reason for this is to allow a series of assignments to be made. As you should know, in C++, the following type of statement is syntactically correct (and, indeed, very common):

```
a = b = c = d = 0;
```

By returning ***this**, the overloaded assignment operator allows objects of type **coord** to be used in a similar fashion. For example, this is perfectly valid:

```
o3 = o2 = o1;
```

Keep in mind that there is no rule that requires an overloaded assignment function to return the object that receives the assignment. However, if you want the overloaded = to behave relative to its class the way it does for the built-in types, it must return ***this**.

3. It is possible to overload an operator relative to a class so that the operand on the right side is an object of a built-in type, such as an integer, instead of the class for which the operator function is a member. For example, here the + operator is overloaded to add an integer value to a **coord** object:

```
// Overload + for ob + int as well as ob + ob.  
#include <iostream>  
using namespace std;
```

```
class coord {  
    int x, y; // coordinate values  
public:  
    coord() { x=0; y=0; }  
    coord(int i, int j) { x=i; y=j; }  
    void get_xy(int &i, int &j) { i=x; j=y; }  
    coord operator+(coord ob2); // ob + ob  
    coord operator+(int i); // ob + int  
};
```

```
// Overload + relative to coord class.  
coord coord::operator+(coord ob2)
```

```
{  
    coord temp;
```

```
    temp.x = x + ob2.x;  
    temp.y = y + ob2.y;
```

```
    return temp;  
}
```

```
// Overload + for ob + int  
coord coord::operator+(int i)
```

```
{  
    coord temp;
```

```
    temp.x = x + i;  
    temp.y = y + i;  
    return temp;
```

```

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // add two objects - calls operator+(coord)
    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 + 100; // add object + int - calls operator+(int)
    o3.get_xy(x, y);
    cout << "(o1+100) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

It is important to remember that when you are overloading a member operator function so that an object can be used in an operation involving a built-in type, the built-in type must be on the right side of the operator. The reason for this is easy to understand: It is the object on the left that generates the call to the operator function. For instance, what happens when the compiler sees the following statement?

```
o3 = 19 + o1; // int + ob
```

There is no built-in operation defined to handle the addition of an integer to an object. The overloaded **operator + (int i)** function works only when the object is on the left. Therefore, this statement generates a **compile-time error**. (Soon you will see one way around this restriction.)

4. You can use a reference parameter in an operator function. For example, this is a perfectly acceptable way to overload the **+** operator relative to the **coord** class:

```

// Overload + relative to coord class using references.
coord coord::operator+(coord &ob2)
{
    coord temp;

```



```

temp.x = x + ob2.x;
temp.y = y + ob2.y;

return temp;
}

```

One reason for using a reference parameter in an operator function is efficiency. Passing objects as parameters to functions often incurs a large amount of overhead and consumes a significant number of CPU cycles. However, passing the address of an object is always quick and efficient. If the operator is going to be used often, using a reference parameter will generally improve performance significantly.

Another reason for using a reference parameter is to avoid the trouble caused when a copy of an operand is destroyed. As you know from previous chapters, when an argument is passed by value, a copy of that argument is made. If that object has a destructor function, when the function terminates, the copy's destructor is called. In some cases it is possible for the destructor to destroy something needed by the calling object. If this is the case, using a reference parameter instead of a value parameter is an easy (and efficient) way around the problem. Of course, you could also define a copy constructor that would prevent this problem in the general case.

EXERCISES

1. Relative to **coord**, overload the * and / operators. Demonstrate that they work.
2. Why would the following be an inappropriate use of an overloaded operator?

```

coord coord::operator%(coord ob)
{
    double i;

    cout << "Enter a number: ";
    cin >> i;
    cout << "root of " << i << " is ";
}

```



```

    cout << sqr(i);
}

```

3. On your own, experiment by changing the return types of the operator functions to something other than **coord**. See what types of errors result.

OVERLOADING THE RELATIONAL AND LOGICAL OPERATORS

It is possible to overload the relational and logical operators. When you overload the relational and logical operators so that they behave in their traditional manner, you will not want the operator functions to return an object of the class for which they are defined. Instead, they will return an integer that indicates either true or false. This not only allows these operator functions to return a true/false value, it also allows the operators to be integrated into larger relational and logical expressions that involve other types of data.

*If you are using a modern C++ compiler, you can also have an overloaded relational or logical operator function return a value of type **bool**, although there is no advantage to doing so. As explained in Chapter 1, the **bool** type defines only two values: **true** and **false**. These values are automatically converted into nonzero and 0 values. Integer nonzero and 0 values are automatically converted into **true** and **false**.*

EXAMPLE

1. In the following program, the **==** and **&&** operators are overloaded:

```

// Overload the == and && relative to coord class.
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values

```

C++

```
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    int operator==(coord ob2);
    int operator&&(coord ob2);
};

// Overload the == operator for coord.
int coord::operator==(coord ob2)
{
    return x==ob2.x && y==ob2.y;
}

// Overload the && operator for coord.
int coord::operator&&(coord ob2)
{
    return (x && ob2.x) && (y && ob2.y);
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3(10, 10), o4(0, 0);

    if(o1==o2) cout << "o1 same as o2\n";
    else cout << "o1 and o2 differ\n";

    if(o1==o3) cout << "o1 same as o3\n";
    else cout << "o1 and o3 differ\n";

    if(o1&&o2) cout << "o1 && o2 is true\n";
    else cout << "o1 && o2 is false\n";

    if(o1&&o4) cout << "o1 && o4 is true\n";
    else cout << "o1 && o4 is false\n";

    return 0;
}
```

C++



EXERCISE

1. Overload the < and > operators relative to the **coord** class.

OVERLOADING A UNARY OPERATOR

Overloading a unary operator is similar to overloading a binary operator except that there is only one operand to deal with. When you overload a unary operator using a member function, the function has no parameters. Since there is only one operand, it is this operand that generates the call to the operator function. There is no need for another parameter.

EXAMPLES

1. The following program overloads the increment operator (**++**) relative to the **coord** class:

```
// Overload ++ relative to coord class.
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values

public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator++();
};

// Overload ++ for coord class.
coord coord::operator++()
{
    x++;
    y++;
}
```

```

    return *this;
}

int main()
{
    coord o1(10, 10);
    int x, y;

    ++o1; // increment an object
    o1.get_xy(x, y);
    cout << "(++o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

Since the increment operator is designed to increase its operand by 1, the overloaded `++` modifies the object it operates upon. The function also returns the object that it increments. This allows the increment operator to be used as part of a larger statement, such as this:

```
o2 = ++o1;
```

As with the binary operators, there is no rule that says you must overload a unary operator so that it reflects its normal meaning. However, most of the time this is what you will want to do.

- In early versions of C++, when an increment or decrement operator was overloaded, there was no way to determine whether an overloaded `++` or `--` preceded or followed its operand. That is, assuming the preceding program, these two statements would have been identical:

```
o1++;
++o1;
```

However, the modern specification for C++ has defined a way by which the compiler can distinguish between these two statements. To accomplish this, create two versions of the **operator ++()** function. The first is defined as shown in the preceding example. The second is declared like this:

```
coord coord::operator++(int n&notused);
```


If the `++` precedes its operand, the **operator ++()** function is called. However, if the `++` follows its operand, the **operator ++(int notused)** function is used. In this case, **notused** will always be passed the value 0. Therefore, if the difference between prefix and postfix increment or decrement is important to your class objects, you will need to implement both operator functions.

- As you know, the minus sign is both a binary and a unary operator in C++. You might be wondering how you can overload it so that it retains both of these uses relative to a class that you create. The solution is actually quite easy: you simply overload it twice, once as a binary operator and once as a unary operator. This program shows how:

```
// Overload the - relative to coord class.
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator-(coord ob2); // binary minus
    coord operator-(); // unary minus
};

// Overload - relative to coord class.
coord coord::operator-(coord ob2)
{
    coord temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;

    return temp;
}

// Overload unary - for coord class.
coord coord::operator-()
{
    coord temp;
    temp.x = -x;
    temp.y = -y;
    return temp;
}
```


C++

```

{
    x = -x;
    y = -y;
    return *this;
}

int main()
{
    coord o1(10, 10), o2(5, 7);
    int x, y;

    o1 = o1 - o2; // subtraction
    o1.get_xy(x, y);
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o1 = -o1; // negation
    o1.get_xy(x, y);
    cout << "(-o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

As you can see, when the minus is overloaded as a binary operator, it takes one parameter. When it is overloaded as a unary operator, it takes no parameter. This difference in the number of parameters is what makes it possible for the minus to be overloaded for both operations. As the program indicates, when the minus sign is used as a binary operator, the **operator-(coord ob2)** function is called. When it is used as a unary minus, the **operator-()** function is called.

EXERCISES

1. Overload the `--` operator for the **coord** class. Create both its prefix and postfix forms.
2. Overload the `+` operator for the **coord** class so that it is both a binary operator (as shown earlier) and a unary operator. When it is used as a unary operator, have the `+` make any negative coordinate value positive.

**U** **USING FRIEND OPERATOR FUNCTIONS**

As mentioned at the start of this chapter, it is possible to overload an operator relative to a class by using a friend rather than a member function. As you know, a friend function does not have a **this** pointer. In the case of a binary operator, this means that a friend operator function is passed both operands explicitly. For unary operators, the single operand is passed. All other things being equal, there is no reason to use a friend rather than a member operator function, with one important exception, which is discussed in the examples.

You cannot use a friend to overload the assignment operator. The assignment operator can be overloaded only by a member operator function.

EXAMPLES

1. Here **operator +()** is overloaded for the **coord** class by using a friend function:

```
// Overload the + relative to coord class using a friend.
#include <iostream>
using namespace std;

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator+(coord ob1, coord ob2);
};

// Overload + using a friend.
coord operator+(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x + ob2.x;
    temp.y = ob1.y + ob2.y;
}
```

C++

```

return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // add two objects - this calls operator+()
    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

Notice that the left operand is passed to the first parameter and the right operand is passed to the second parameter.

- Overloading an operator by using a friend provides one very important feature that member functions do not. Using a friend operator function, you can allow objects to be used in operations involving built-in types in which the built-in type is on the left side of the operator. As you saw earlier in this chapter, it is possible to overload a binary member operator function such that the left operand is an object and the right operand is a built-in type. But it is not possible to use a member function to allow the built-in type to occur on the left side of the operator. For example, assuming an overloaded member operator function, the first statement shown here is legal; the second is not:

```

ob1 = ob2 + 10; // legal
ob1 = 10 + ob2; // illegal

```

While it is possible to organize such statements like the first, always having to make sure that the object is on the left side of the operand and the built-in type on the right can be a cumbersome restriction. The solution to this problem is to make the overloaded operator functions friends and define both possible situations.

As you know, a friend operator function is explicitly passed *both* operands. Thus, it is possible to define one overloaded friend function so that the left operand is an object and the right

operand is the other type. Then you could overload the operator again with the left operand being the built-in type and the right operand being the object. The following program illustrates this method:

```
// Use friend operator functions to add flexibility.
#include <iostream>
using namespace std;
```

```
class coord {
    int x, y; // coordinate values

public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator+(coord obl, int i);
    friend coord operator+(int i, coord obl);
};
```

```
// Overload for ob + int.
coord operator+(coord obl, int i)
{
    coord temp;

    temp.x = obl.x + i;
    temp.y = obl.y + i;

    return temp;
}
```

```
// Overload for int + ob.
coord operator+(int i, coord obl)
{
    coord temp;

    temp.x = obl.x + i;
    temp.y = obl.y + i;

    return temp;
}
```

```
int main()
{
```



```
coord o1(10, 10);
int x, y;

o1 = o1 + 10; // object + integer
o1.get_xy(x, y);
cout << "(o1+10) X: " << x << ", Y: " << y << "\n";

o1 = 99 + o1; // integer + object
o1.get_xy(x, y);
cout << "(99+o1) X: " << x << ", Y: " << y << "\n";

return 0;
}
```

As a result of overloading friend operator functions for both situations, both of these statements are now valid:

```
o1 = o1 + 10;
o1 = 99 + o1;
```

3. If you want to use a friend operator function to overload either the `++` or `--` unary operator, you must pass the operand to the function as a reference parameter. This is because friend functions do not have **this** pointers. Remember that the increment and decrement operators imply that the operand will be modified. However, if you overload these operators by using a friend that uses a value parameter, any modifications that occur to the parameter inside the friend operator function will not affect the object that generated the call. And since no pointer to the object is passed implicitly (that is, there is no **this** pointer) when a friend is used, there is no way for the increment or decrement to affect the operand.

However, if you pass the operand to the friend as a reference parameter, changes that occur inside the friend function affect the object that generates the call. For example, here is a program that overloads the `++` operator by using a friend function:

```
// Overload the ++ using a friend.
#include <iostream>
using namespace std;
```



```

class coord {
    int x, y; // coordinate values
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator++(coord &ob);
};

// Overload ++ using a friend.
coord operator++(coord &ob) // use reference parameter
{
    ob.x++;
    ob.y++;

    return ob; // return object generating the call
}

int main()
{
    coord o1(10, 10);
    int x, y;

    ++o1; // o1 is passed by reference
    o1.get_xy(x, y);
    cout << "(++o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

If you are using a modern compiler, you can also distinguish between the prefix and postfix forms of the increment or decrement operators when using a friend operator function in much the same way you did when using member functions. You simply add an integer parameter when defining the postfix version. For example, here are the prototypes for both the prefix and postfix versions of the increment operator relative to the **coord** class:

```

coord operator++(coord &ob); // prefix
coord operator++(coord &ob, int notused); // postfix

```



If the `++` precedes its operand, the **operator ++(coord &ob)** function is called. However, if the `++` follows its operand, the **operator ++(coord &ob, int notused)** function is used. In this case, **notused** will be passed the value 0.

EXERCISES

1. Overload the `-` and `/` operators for the **coord** class using friend functions.
 2. Overload the **coord** class so it can use **coord** objects in operations in which an integer value can be multiplied by each coordinate. Allow the operations to use either order: `ob * int` or `int * ob`.
 3. Explain why the solution to Exercise 2 requires the use of friend operator functions.
 4. Using a friend, show how to overload the `--` relative to the **coord** class. Define both the prefix and postfix forms.
-

A CLOSER LOOK AT THE ASSIGNMENT OPERATOR

As you have seen, it is possible to overload the assignment operator relative to a class. By default, when the assignment operator is applied to an object, a bitwise copy of the object on the right is put into the object on the left. If this is what you want, there is no reason to provide your own **operator = ()** function. However, there are cases in which a strict bitwise copy is not desirable. You saw some examples of this in Chapter 3, in cases in which an object allocates memory. In these types of situations, you will want to provide a special assignment operation.

EXAMPLE

1. Here is another version of the `strtype` class that you have seen in various forms in the preceding chapters. However, this version overloads the `=` operator so that the pointer `p` is not overwritten by an assignment operation.

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

class strtype {
    char *p;
    int len;
public:
    strtype(char *s);
    ~strtype() {
        cout << "Freeing " << (unsigned) p << '\n';
        delete [] p;
    }
    char *get() { return p; }
    strtype &operator=(strtype &ob);
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;

    p = new char [l];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }

    len = l;
    strcpy(p, s);
}
```

```
// Assign an object.
strtype &strtype::operator=(strtype &ob)
{
    // see if more memory is needed
    if(len < ob.len) { // need to allocate more memory
        delete [] p;
        p = new char [ob.len];
        if(!p) {
            cout << "Allocation error\n";
            exit(1);
        }
    }
    len = ob.len;
    strcpy(p, ob.p);
    return *this;
}

int main()
{
    strtype a("Hello"), b("There");

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    a = b; // now p is not overwritten

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    return 0;
}
```

As you can see, the overloaded assignment operator prevents **p** from being overwritten. It first checks to see if the object on the left has allocated enough memory to hold the string that is being assigned to it. If it hasn't, that memory is freed and another portion is allocated. Then the string is copied to that memory and the length is copied into **len**.

Notice two other important features about the **operator =()** function. First, it takes a reference parameter. This prevents a copy of the object on the right side of the assignment from being made. As you know from previous chapters, when a copy of an object is made when passed to a function, that copy is destroyed

when the function terminates. In this case, destroying the copy would call the destructor function, which would free **p**.

However, this is the same **p** still needed by the object used as an argument. Using a reference parameter prevents this problem.

The second important feature of the **operator = ()** function is that it returns a reference, not an object. The reason for this is the same as the reason it uses a reference parameter. When a function returns an object, a temporary object is created that is destroyed after the return is complete. However, this means that the temporary object's destructor will be called, causing **p** to be freed, but **p** (and the memory it points to) is still needed by the object being assigned a value. Therefore, by returning a reference, you prevent a temporary object from being created.

**Note**

As you learned in Chapter 5, creating a copy constructor is another way to prevent both of the problems described in the preceding two paragraphs. But the copy constructor might not be as efficient a solution as using a reference parameter and a reference return type. This is because using a reference prevents the overhead associated with copying an object in either circumstance. As you can see, there are often several ways to accomplish the same end in C++. Learning to choose between them is part of becoming an excellent C++ programmer.

EXERCISE

1. Given the following class declaration, fill in all the details that will create a dynamic array type. That is, allocate memory for the array, storing a pointer to this memory in **p**. Store the size of the array, in bytes, in **size**. Have **put()** return a reference to the specified element, and have **get()** return the value of a specified element. Don't allow the boundaries of the array to be overrun. Also, overload the assignment operator so that the allocated memory of each array is not accidentally destroyed when one array is assigned to another. (In the next section you will see a way to improve your solution to this exercise.)


```

class dynarray {
    int *p;
    int size;
public:
    dynarray(int s); // pass size of array in s
    int &put(int i); // return reference to element i
    int get(int i); // return value of element i
    // create operator=( ) function
};

```

6.7**O**VERLOADING THE []
SUBSCRIPT OPERATOR

The last operator that we will overload is the [] array subscripting operator. In C++, the [] is considered a binary operator for the purposes of overloading. The [] can be overloaded only by a member function. Therefore, the general form of a member **operator[]()** function is as shown here:

```

type class-name::operator[ ](int index)
{
    // ...
}

```

Technically, the parameter does not have to be of type **int**, but **operator[]()** functions are typically used to provide array subscripting and as such an integer value is generally used.

To understand how the [] operator works, assume that an object called **O** is indexed as shown here:

```
O[9]
```

This index will translate into the following call to the **operator[]()** function:

```
O.operator[ ](9)
```

That is, the value of the expression within the subscripting operator is passed to the **operator[]()** function in its explicit parameter. The **this** pointer will point to **O**, the object that generated the call.

EXAMPLES

1. In the following program, **arraytype** declares an array of five integers. Its constructor function initializes each member of the array. The overloaded **operator[]()** function returns the value of the element specified by its parameter.

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

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    int operator[](int i) { return a[i]; }
};

int main()
{
    arraytype ob;
    int i;

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    return 0;
}
```

This program displays the following output:

```
0 1 2 3 4
```

The initialization of the array **a** by the constructor in this and the following programs is for the sake of illustration only. It is not required.

2. It is possible to design the **operator[]()** function in such a way that the **[]** can be used on both the left and right sides of an assignment statement. To do this, return a reference to the element being indexed. For example, this program makes this change and illustrates its use:

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

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    int &operator[](int i) { return a[i]; }
};

int main()
{
    arraytype ob;
    int i;

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    cout << "\n";

    // add 10 to each element in the array
    for(i=0; i<SIZE; i++)
        ob[i] = ob[i]+10; // [] on left of =

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    return 0;
}
```

This program displays the following output:

```
0 1 2 3 4
10 11 12 13 14
```


Because the `operator[]()` function now returns a reference to the array element indexed by `i`, it can be used on the left side of an assignment to modify an element of the array. (Of course, it can still be used on the right side as well.) As you can see, this makes objects of `arraytype` act like normal arrays.

3. One advantage of being able to overload the `[]` operator is that it allows a better means of implementing safe array indexing. Earlier in this book you saw a simplified way to implement a safe array that relied upon functions such as `get()` and `put()` to access the elements of the array. Here you will see a better way to create a safe array that utilizes an overloaded `[]` operator. Recall that a safe array is an array that is encapsulated within a class that performs bounds checking. This approach prevents the array boundaries from being overrun. By overloading the `[]` operator for such an array, you allow it to be accessed just like a regular array.

To create a safe array, simply add bounds checking to the `operator[]()` function. The `operator[]()` must also return a reference to the element being indexed. For example, this program adds a range check to the previous array program and proves that it works by generating a boundary error:

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    int &operator[](int i);
};

// Provide range checking for arraytype.
int &arraytype::operator[](int i)
{
```

```
    if(i<0 || i> SIZE-1) {
        cout <<  \nIndex \value of ";
        cout << i <<  is out of bounds.\n";
        exit(1);
    }
    return a[i];
}

int main()
{
    arraytype ob;
    int i;

    // this is OK
    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    /* this generates a run-time error because
       SIZE+100 is out of range */
    ob[SIZE+100] = 99; // error!

    return 0;
}
```

In this program, when the statement

```
ob[SIZE+100] = 99;
```

executes, the boundary error is intercepted by **operator[]()** and the program is terminated before any damage can be done.

Because the overloading of the **[]** operator allows you to create safe arrays that look and act just like regular arrays, they can be seamlessly integrated into your programming environment. But be careful. A safe array adds overhead that might not be acceptable in all situations. In fact, the added overhead is why C++ does not perform boundary checking on arrays in the first place. However, in applications in which you want to be sure that a boundary error does not take place, a safe array will be worth the effort.

EXERCISES

1. Modify Example 1 in Section 6.6 so that `strtype` overloads the `[]` operator. Have this operator return the character at the specified index. Also, allow the `[]` to be used on the left side of the assignment statement. Demonstrate its use.
2. Modify your answer to Exercise 1 from Section 6.6 so that it uses `[]` to index the dynamic array. That is, replace the `get()` and `put()` functions with the `[]` operator.

SKILLS CHECK


Mastery

Skills Check

At this point you should be able to perform the following exercises and answer the questions.

1. Overload the `>>` and `<<` shift operators relative to the `coord` class so that the following types of operations are allowed:

```
ob << integer
ob >> integer
```

Make sure your operators shift the `x` and `y` values by the amount specified.

2. Given the class

```
class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
};
```



```

}
three_d() { x=0; y=0; z=0; }
void get(int &i, int &j, int &k)
{
    i = x; j = y; k = z;
}
};

```

overload the +, -, ++, and -- operators for this class. (For the increment and decrement operators, overload only the prefix form.)

3. Rewrite your answer to Question 2 so that it uses reference parameters instead of value parameters to the operator functions. (Hint: You will need to use friend functions for the increment and decrement operators.)
4. How do friend operator functions differ from member operator functions?
5. Explain why you might need to overload the assignment operator.
6. Can **operator =()** be a friend function?
7. Overload the + for the **three_d** class in Question 2 so that it accepts the following types of operations:


```

ob + int;
int + ob;

```
8. Overload the ==, !=, and || operators relative to the **three_d** class from Question 2.
9. Explain the main reason for overloading the [] operator.



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

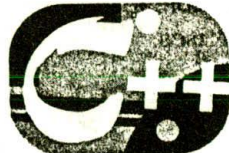
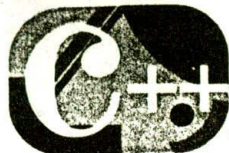
1. Create a **strtype** class that allows the following types of operators:

- ▼ string concatenation using the + operator
- ▼ string assignment using the = operator
- ▼ string comparisons using <, >, and ==

Feel free to use fixed-length strings. This is a challenging assignment, but with some thought (and experimentation), you should be able to accomplish it.







7

Inheritance

chapter objectives

- 7.1** Base class access control
- 7.2** Using protected members
- 7.3** Constructors, destructors, and inheritance
- 7.4** Multiple inheritance
- 7.5** Virtual base classes

YOU were introduced to the concept of inheritance earlier in this book. Now it is time to explore it more thoroughly. Inheritance is one of the three principles of OOP and, as such, it is an important feature of C++. Inheritance does more than just support the concept of hierarchical classification; in Chapter 10 you will learn how inheritance provides support for polymorphism, another principal feature of OOP.

The topics covered in this chapter include base class access control and the **protected** access specifier, inheriting multiple base classes, passing arguments to base class constructors, and virtual base classes.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. When an operator is overloaded, does it lose any of its original functionality?
2. Must an operator be overloaded relative to a user-defined type, such as a class?
3. Can the precedence of an overloaded operator be changed? Can the number of operands be altered?
4. Given the following partially completed program, fill in the needed operator functions:

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

class array {
    int nums[10];
public:
    array();
    void set(int n[10]);
    void show();
    array operator+(array ob2);
    array operator-(array ob2);
    int operator==(array ob2);
```

```
};

array::array()
{
    int i;
    for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i=0; i<10; i++)
        cout << nums[i] << ' ';

    cout << "\n";
}

// Fill in operator functions.

int main()
{
    array o1, o2, o3;

    int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    o1.set(i);
    o2.set(i);

    o3 = o1 + o2;
    o3.show();

    o3 = o1 - o2;
    o3.show();

    if(o1==o2) cout << "o1 equals o2\n";
    else cout << "o1 does not equal o2\n";
}
```

C++

```

if (o1==o3) cout << "o1 equals o3\n";
else cout << "o1 does not equal o3\n";

return 0;
}

```

Have the overloaded + add each element of each operand. Have the overloaded - subtract each element of the right operand from the left. Have the overloaded == return true if each element of each operand is the same and return false otherwise.

- Convert the solution to Exercise 4 so it overloads the operators by using friend functions.
- Using the class and support functions from Exercise 4, overload the ++ operator by using a member function and overload the -- operator by using a friend. (Overload only the prefix forms of ++ and --.)
- Can the assignment operator be overloaded by using a friend function?

7.1***B*ASE CLASS ACCESS CONTROL**

When one class inherits another, it uses this general form:

```

class derived-class-name : access base-class-name {
    // ...
}

```

Here *access* is one of three keywords: **public**, **private**, or **protected**. A discussion of the **protected** access specifier is deferred until the next section of this chapter. The other two are discussed here.

The access specifier determines how elements of the base class are inherited by the derived class. When the access specifier for the inherited base class is **public**, all public members of the base become public members of the derived class. If the access specifier is **private**, all public members of the base class become private members of the derived class. In either case, any private members of the base remain private to it and are inaccessible by the derived class.

It is important to understand that if the access specifier is **private**, public members of the base become private members of the der

class, but these members are still accessible by member functions of the derived class.

Technically, *access* is optional. If the specifier is not present, it is **private** by default if the derived class is a **class**. If the derived class is a **struct**, **public** is the default in the absence of an explicit access specifier. Frankly, most programmers explicitly specify *access* for the sake of clarity.

EXAMPLES

1. Here is a short base class and a derived class that inherits it (as public):

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

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Inherit as public.
class derived : public base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() { cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setx(10); // access member of base class
    ob.sety(20); // access member of derived class

    ob.showx(); // access member of base class
    ob.showy(); // access member of derived class

    return 0;
}
```

As this program illustrates, because **base** is inherited as public, the public members of **base-setx()** and **showx()** - become public members of **derived** and are, therefore, accessible by any other part of the program. Specifically, they are legally called within **main()**.

2. It is important to understand that just because a derived class inherits a base as public, it does not mean that the derived class has access to the base's private members. For example, this addition to **derived** from the preceding example is incorrect:

```
class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Inherit as public - this has an error!
class derived : public base {
    int y;
public:
    void sety(int n) { y = n; }

    /* Cannot access private member of base class.
       x is a private member of base and not available
       within derived. */
    void show_sum() { cout << x+y << '\n'; } // Error!

    void showy() { cout << y << '\n'; }
};
```

In this example, the **derived** class attempts to access **x**, which is a private member of **base**. This is an error because the private parts of a base class remain private to it *no matter how it is inherited*.

3. Here is a variation of the program shown in Example 1; this time **derived** inherits **base** as private. This change causes the program to be in error, as indicated in the comments.

```
// This program contains an error.
#include <iostream>
using namespace std;
```



```

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Inherit base as private.
class derived : private base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() { cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setx(10); // ERROR - now private to derived class
    ob.sety(20); // access member of derived class - OK

    ob.showx(); // ERROR - now private to derived class
    ob.showy(); // access member of derived class - OK

    return 0;
}

```

As the comments in this (incorrect) program illustrate, both **showx()** and **setx()** become private to **derived** and are not accessible outside of it.

Keep in mind that **showx()** and **setx()** are still public within **base** no matter how they are inherited by some derived class. This means that an object of type **base** could access these functions anywhere. However, relative to objects of type **derived**, they become private. For example, given this fragment:

```

base base_ob;

base_ob.setx(1); // is legal because base_ob is of type base

```

the call to **setx()** is legal because **setx()** is public within **base**.

4. As stated, even though public members of a base class become private members of a derived class when inherited using the **private** specifier, they are still accessible *within* the derived class. For example, here is a "fixed" version of the preceding program:

```
// This program is fixed.
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Inherit base as private.
class derived : private base {
    int y;
public:
    // setx is accessible from within derived
    void setxy(int n, int m) { setx(n); y = m; }
    // showx is accessible from within derived
    void showxy() { showx(); cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setxy(10, 20);

    ob.showxy();

    return 0;
}
```

In this case, the functions **setx()** and **showx()** are accessed inside the derived class, which is perfectly legal because they are private members of that class.

EXERCISES

1. Examine this skeleton:

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

class mybase {
    int a, b;
public:
    int c;
    void setab(int i, int j) { a = i; b = j; }
    void getab(int &i, int &j) { i = a; j = b; }
};

class derived1 : public mybase {
    // ...
};

class derived2 : private mybase {
    // ...
};

int main()
{
    derived1 o1;
    derived2 o2;
    int i, j;

    // ...
}
```

Within **main()**, which of the following statements are legal?

- A. o1.getab(i, j);
 - B. o2.getab(i, j);
 - C. o1.c = 10;
 - D. o2.c = 10;
2. What happens when a public member is inherited as public?
What happens when it is inherited as private?
3. If you have not done so, try all the examples presented in this section. On your own, try various changes relative to the access specifiers and observe the results.
-

U **USING PROTECTED MEMBERS**

As you know from the preceding section, a derived class does not have access to the private members of the base class. This means that if the derived class needs access to some member of the base, that member must be public. However, there will be times when you want to keep a member of a base class private but still allow a derived class access to it. To accomplish this goal, C++ includes the **protected** access specifier.

The **protected** access specifier is equivalent to the **private** specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible.

The **protected** access specifier can occur anywhere in the class declaration, although typically it occurs after the (default) private members are declared and before the public members. The full general form of a class declaration is shown here:

```
class class-name {  
    // private members  
protected: // optional  
    // protected members  
public:  
    // public members  
};
```

[When a protected member of a base class is inherited as public by the derived class, it becomes a protected member of the derived class. If the base is inherited as private, a protected member of the base becomes a private member of the derived class.]

A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class. (Of course, private members of the base class remain private to it and are not accessible by the derived class.)

The **protected** access specifier can also be used with structures.

EXAMPLES

1. This program illustrates how public, private, and protected members of a class can be accessed:

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

class samp {
    // private by default
    int a;
protected: // still private relative to samp
    int b;
public:
    int c;

    samp(int n, int m) { a = n; b = m; }
    int geta() { return a; }
    int getb() { return b; }
};

int main()
{
    samp ob(10, 20);

    // ob.b = 99; Error! b is protected and thus private
    ob.c = 30; // OK, c is public

    cout << ob.geta() << ' ';
    cout << ob.getb() << ' ' << ob.c << '\n';

    return 0;
}
```

As you can see, the commented-out line is not permissible in **main()** because **b** is protected and is thus still private to **samp**.

2. The following program illustrates what occurs when protected members are inherited as public:

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

```
class base {
protected: // private to base
    int a, b; // but still accessible by derived
public:
    void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
    int c;
public:
    void setc(int n) { c = n; }

    // this function has access to a and b from base
    void showabc() {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};

int main()
{
    derived ob;

    /* a and b are not accessible here because they are
       private to both base and derived. */

    ob.setab(1, 2);
    ob.setc(3);

    ob.showabc();

    return 0;
}
```

Because **a** and **b** are protected in **base** and inherited as public by **derived**, they are available for use by member functions of **derived**. However, outside of these two classes, **a** and **b** are effectively private and unaccessible.

3. As mentioned earlier, when a base class is inherited as protected, public and protected members of the base class become protected members of the derived class. For example, here the preceding program is changed slightly, inheriting **base** as protected instead of public:

```
// This program will not compile.
#include <iostream>
using namespace std;

class base {
protected: // private to base
    int a, b; // but still accessible by derived
public:
    void setab(int n, int m) { a = n; b = m; }
};

class derived : protected base { // inherit as protected
    int c;
public:
    void setc(int n) { c = n; }

    // this function has access to a and b from base
    void showabc() {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};

int main()
{
    derived ob;

    // ERROR: setab() is now a protected member of base.
    ob.setab(1, 2); // setab() is not accessible here.

    ob.setc(3);

    ob.showabc();

    return 0;
}
```

As the comments now describe, because **base** is inherited as protected, its public and protected elements become protected members of **derived** and are therefore inaccessible within **main()**.

**EXERCISES**

1. What happens when a protected member is inherited as public? What happens when it is inherited as private? What happens when it is inherited as protected?
2. Explain why the protected category is needed.
3. In Exercise 1 from Section 7.1, if the **a** and **b** inside **myclass** were made into protected instead of private (by default) members, would any of your answers to that exercise change? If so, how?

7.3**C**ONSTRUCTORS, DESTRUCTORS,
AND INHERITANCE

It is possible for the base class, the derived class, or both to have constructor and/or destructor functions. Several issues that relate to these situations are examined in this section.

When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order. That is, the base class constructor is executed before the constructor in the derived class. The reverse is true for destructor functions: the derived class's destructor is executed before the base class's destructor.

If you think about it, it makes sense that constructor functions are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it performs is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

On the other hand, a derived class's destructor must be executed before the destructor of the base class because the base class underlies the derived class. If the base class's destructor were executed first, it would imply the destruction of the derived class. Thus, the derived class's destructor must be called before the object goes out of existence.

So far, none of the preceding examples have passed arguments to either a derived or base class constructor. However, it is possible to do this. When only the derived class takes an initialization, arguments are

passed to the derived class's constructor in the normal fashion. However, if you need to pass an argument to the constructor of the base class, a little more effort is needed. To accomplish this, a chain of argument passing is established. First, all necessary arguments to both the base class and the derived class are passed to the derived class's constructor. Using an expanded form of the derived class's constructor declaration, you then pass the appropriate arguments along to the base class. The syntax for passing along an argument from the derived class to the base class is shown here:

```
derived-constructor (arg-list) : base(arg-list) {
    // body of derived class constructor
}
```

Here *base* is the name of the base class. It is permissible for both the derived class and the base class to use the same argument. It is also possible for the derived class to ignore all arguments and just pass them along to the base.

EXAMPLES

- Here is a very short program that illustrates when base class and derived class constructor and destructor functions are executed:

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

class base {
public:
    base() { cout << "Constructing base class\n"; }
    ~base() { cout << "Destructing base class\n"; }
};

class derived : public base {
public:
    derived() { cout << "Constructing derived class\n"; }
    ~derived() { cout << "Destructing derived class\n"; }
};

int main()
{
```

C++

```

    derived o;

    return 0;
}

```

This program displays the following output:

```

Constructing base class
Constructing derived class
Destructing derived class
Destructing base class

```

As you can see, the constructors are executed in order of derivation and the destructors are executed in reverse order.

2. This program shows how to pass an argument to a derived class's constructor:

```

#include <iostream>
using namespace std;

class base {
public:
    base() { cout << "Constructing base class\n"; }
    ~base() { cout << "Destructing base class\n"; }
};

class derived : public base {
    int j;
public:
    derived(int n) {
        cout << "Constructing derived class\n";
        j = n;
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showj();

    return 0;
}

```


Notice that the argument is passed to the derived class's constructor in the normal fashion.

3. In the following example, both the derived class and the base class constructors take arguments. In this specific case, both use the same argument, and the derived class simply passes along the argument to the base.

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

class base {
    int i;
public:
    base(int n) {
        cout << "Constructing base class\n";
        i = n;
    }
    ~base() { cout << "Destructing base class\n"; }
    void showi() { cout << i << '\n'; }
};

class derived : public base {
    int j;
public:
    derived(int n) : base(n) { // pass arg to base class
        cout << "Constructing derived class\n";
        j = n;
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showi();
    o.showj();

    return 0;
}
```

Pay special attention to the declaration of **derived**'s constructor. Notice how the parameter **n** (which receives the initialization argument) is both used by **derived()** and passed to **base()**.

4. In most cases, the constructor functions for the base and derived classes will *not* use the same argument. When this is the case and you need to pass one or more arguments to each, you must pass to the derived class's constructor *all* arguments needed by *both* the derived class and the base class. Then the derived class simply passes along to the base those arguments required by it. For example, this program shows how to pass an argument to the derived class's constructor and another one to the base class:

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

class base {
    int i;
public:
    base(int n) {
        cout << "Constructing base class\n";
        i = n;
    }
    ~base() { cout << "Destructing base class\n"; }
    void showi() { cout << i << '\n'; }
};

class derived : public base {
    int j;
public:
    derived(int n, int m) : base(m) { // pass arg to base class
        cout << "Constructing derived class\n";
        j = n;
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10, 20);
```

```

o.showi();
o.showj();

return 0;
}

```

5. It is not necessary for the derived class' constructor to actually use an argument in order to pass one to the base class. If the derived class does not need an argument, it ignores the argument and simply passes it along. For example, in this fragment, parameter **n** is not used by **derived()**. Instead, it is simply passed to **base()**:

```

class base {
    int i;
public:
    base(int n) {
        cout << "Constructing base class\n";
        i = n;
    }
    ~base() { cout << "Destructing base class\n"; }
    void showi() { cout << i << '\n'; }
};

```

```

class derived : public base {
    int j;
public:
    derived(int n) : base(n) { // pass arg to base class
        cout << "Constructing derived class\n";
        j = 0; // n not used here
    }
    ~derived() { cout << "Destructing derived class\n"; }
    void showj() { cout << j << '\n'; }
};

```

EXERCISES

- Given the following skeleton, fill in the constructor function for **myderived**. Have it pass along a pointer to an initialization string to **mybase**. Also, have **myderived()** initialize **len** to the length of the string.

C++

```

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

class mybase {
    char str[80];
public:
    mybase(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

class myderived : public mybase {
    int len;
public:
    // add myderived() here
    int getlen() { return len; }
    void show() { cout << get() << '\n'; }
};

int main()
{
    myderived ob("hello");

    ob.show();
    cout << ob.getlen() << '\n';

    return 0;
}

```

2. Using the following skeleton, create appropriate **car()** and **truck()** constructor functions. Have each pass along appropriate arguments to **vehicle**. In addition, have **car()** initialize **passengers** as specified when an object is created. Have **truck()** initialize **loadlimit** as specified when an object is created.

```

#include <iostream>
using namespace std;

// A base class for various types of vehicles.
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)

```

```
{
    num_wheels = w; range = r;
}
void showv()
{
    cout << "Wheels: " << num_wheels << '\n';
    cout << "Range: " << range << '\n';
}
};

class car : public vehicle {
    int passengers;
public:
    // insert car() constructor here
    void show()
    {
        showv();
        cout << "Passengers: " << passengers << '\n';
    }
};

class truck : public vehicle {
    int loadlimit;
public:
    // insert truck() constructor here
    void show()
    {
        showv();
        cout << "loadlimit " << loadlimit << '\n';
    }
};

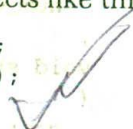
int main()
{
    car c(5, 4, 500);
    truck t(30000, 12, 1200);

    cout << "Car: \n";
    c.show();
    cout << "\nTruck:\n";
    t.show();

    return 0;
}
```

Have `car()` and `truck()` declare objects like this:

```
car ob(passengers, wheels, range);  
truck ob(loadlimit, wheels, range);
```

**7.4****M**ULTIPLE INHERITANCE

There are two ways that a derived class can inherit more than one base class. First, a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy. In this case, the original base class is said to be an *indirect* base class of the second derived class. (Keep in mind that any class—no matter how it is created—can be used as a base class.) Second, a derived class can directly inherit more than one base class. In this situation, two or more base classes are combined to help create the derived class. There are several issues that arise when multiple base classes are involved, and these issues are examined in this section.

When a base class is used to derive a class that is used as a base class for another derived class, the constructor functions of all three classes are called in order of derivation. (This is a generalization of the principle you learned earlier in this chapter.) Also, destructor functions are called in reverse order. Thus, if class *B1* is inherited by *D1*, and *D1* is inherited by *D2*, *B1*'s constructor is called first, followed by *D1*'s, followed by *D2*'s. The destructors are called in reverse order.

When a derived class directly inherits multiple base classes, it uses this expanded declaration:

```
class derived-class-name : access base1, access base2, . . . , access baseN  
{  
    // ... body of class  
}
```

Here *base1* through *baseN* are the base class names and *access* is the access specifier, which can be different for each base class. When multiple base classes are inherited, constructors are executed in the order, left to right, that the base classes are specified. Destructors are executed in the opposite order.

When a class inherits multiple base classes that have constructors that require arguments, the derived class passes the necessary arguments to them by using this expanded form of the derived class' constructor function:

```
derived-constructor(arg-list) : base1(arg-list), base2(arg-list), ...,
    baseN(arg-list)
{
    // body of derived class constructor
}
```

Here *base1* through *baseN* are the names of the base classes.

When a derived class inherits a hierarchy of classes, each derived class in the chain must pass back to its preceding base any arguments it needs.

EXAMPLES

1. Here is an example of a derived class that inherits a class derived from another class. Notice how arguments are passed along the chain from **D2** to **B1**.

```
// Multiple Inheritance
#include <iostream>
using namespace std;

class B1 {
    int a;
public:
    B1(int x) { a = x; }
    int geta() { return a; }
};

// Inherit direct base class.
class D1 : public B1 {
    int b;
public:
    D1(int x, int y) : B1(y) // pass y to B1
    {
        b = x;
    }
    int getb() { return b; }
};
```

```
// Inherit a derived class and an indirect base.
class D2 : public D1 {
    int c;
public:
    D2(int x, int y, int z) : D1(y, z) // pass args to D1
    {
        c = x;
    }

    /* Because bases inherited as public, D2 has access
       to public elements of both B1 and D1. */
    void show() {
        cout << geta() << ' ' << getb() << ' ';
        cout << c << '\n';
    }
};

int main()
{
    D2 ob(1, 2, 3);
    ob.show();
    // geta() and getb() are still public here.
    cout << ob.geta() << ' ' << ob.getb() << '\n';

    return 0;
}
```

The call to **ob.show()** displays 3 2 1. In this example, **B1** is an indirect base class of **D2**. Notice that **D2** has access to the public members of both **D1** and **B1**. As you should remember, when public members of a base class are inherited as public, they become public members of the derived class. Therefore, when **D1** inherits **B1**, **geta()** becomes a public member of **D1**, which becomes a public member of **D2**.

As the program illustrates, each class in a class hierarchy must pass all arguments required by each preceding base class. Failure to do so will generate a compile-time error.

The class hierarchy created in this program is illustrated here:



Before we move on, a short discussion about how to draw C++-style inheritance graphs is in order. In the preceding graph, notice that the arrows point up instead of down. Traditionally, C++ programmers usually draw inheritance charts as directed graphs in which the arrow points from the derived class to the base class. While newcomers sometimes find this approach counter-intuitive, it is nevertheless the way inheritance charts are usually depicted in C++.

2. Here is a reworked version of the preceding program, in which a derived class directly inherits two base classes:

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

// Create first base class.
class B1 {
    int a;
public:
    B1(int x) { a = x; }
    int geta() { return a; }
};

// Create second base class.
class B2 {
    int b;
public:
    B2(int x)
    {
        b = x;
    }
};
```



```

C++
int getb() { return b; }
};

// Directly inherit two base classes.
class D : public B1, public B2 {
    int c;
public:
    // here z and y are passed directly to B1 and B2
    D(int x, int y, int z) : B1(z), B2(y)
    {
        c = x;
    }

    /* Because bases inherited as public, D has access
       to public elements of both B1 and B2. */
    void show() {
        cout << geta() << ' ' << getb() << ' ';
        cout << c << '\n';
    }
};

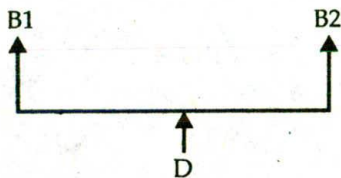
int main()
{
    D ob(1, 2, 3);

    ob.show();

    return 0;
}

```

In this version, the arguments to **B1** and **B2** are passed individually to these classes by **D**. This program creates a class that looks like this:



3. The following program illustrates the order in which constructor and destructor functions are called when a derived class directly inherits multiple base classes:

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

class B1 {
public:
    B1() { cout << "Constructing B1\n"; }
    ~B1() { cout << "Destructing B1\n"; }
};

class B2 {
    int b;
public:
    B2() { cout << "Constructing B2\n"; }
    ~B2() { cout << "Destructing B2\n"; }
};

// Inherit two base classes.
class D : public B1, public B2 {
public:
    D() { cout << "Constructing D\n"; }
    ~D() { cout << "Destructing D\n"; }
};

int main()
{
    D ob;

    return 0;
}
```

This program displays the following:

```
Constructing B1
Constructing B2
Constructing D
Destructing D
Destructing B2
Destructing B1
```

As you have learned, when multiple direct base classes are inherited, constructors are called in order, left to right, as specified in the inheritance list. Destructors are called in reverse order.

EXERCISES

1. What does the following program display? (Try to determine this without actually running the program.)

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

class A {
public:
    A() { cout << "Constructing A\n"; }
    ~A() { cout << "Destructing A\n"; }
};

class B {
public:
    B() { cout << "Constructing B\n"; }
    ~B() { cout << "Destructing B\n"; }
};

class C : public A, public B {
public:
    C() { cout << "Constructing C\n"; }
    ~C() { cout << "Destructing C\n"; }
};

int main()
{
    C ob;

    return 0;
}
```

2. Using the following class hierarchy, create **C**'s constructor so that it initializes **k** and passes on arguments to **A()** and **B()**.

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

class A {
    int i;
public:
    A(int a) { i = a; }
};
```



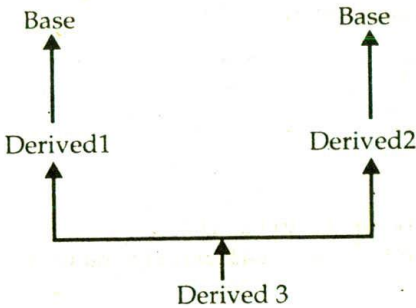
```
class B {  
    int j;  
public:  
    B(int a) { j = a; }  
};
```

```
class C : public A, public B {  
    int k;  
public:  
    /* Create C() so that it initializes k  
       and passes arguments to both A() and B() */  
};
```

7.5

VIRTUAL BASE CLASSES

A potential problem exists when multiple base classes are directly inherited by a derived class. To understand what this problem is, consider the following class hierarchy:



Here the base class *Base* is inherited by both *Derived1* and *Derived2*. *Derived3* directly inherits both *Derived1* and *Derived2*. However, this implies that *Base* is actually inherited twice by *Derived3*—first it is inherited through *Derived1*, and then again through *Derived2*. This causes ambiguity when a member of *Base* is used by *Derived3*. Since two copies of *Base* are included in *Derived3*, is a reference to a member of *Base* referring to the *Base* inherited indirectly through *Derived1* or to the *Base* inherited indirectly through *Derived2*? To resolve this

ambiguity, C++ includes a mechanism by which only one copy of *Base* will be included in *Derived3*. This feature is called a *virtual base class*.

In situations like the one just described, in which a derived class indirectly inherits the same base class more than once, it is possible to prevent two copies of the base from being present in the derived object by having that base class inherited as virtual by any derived classes. Doing this prevents two (or more) copies of the base from being present in any subsequent derived class that inherits the base class indirectly. The **virtual** keyword precedes the base class access specifier when it is inherited by a derived class.

EXAMPLES

1. Here is an example that uses a virtual base class to prevent two copies of **base** from being present in **derived3**.

```
// This program uses a virtual base class.
#include <iostream>
using namespace std;

class base {
public:
    int i;
};

// Inherit base as virtual.
class derived1 : virtual public base {
public:
    int j;
};

// Inherit base as virtual here, too.
class derived2 : virtual public base {
public:
    int k;
};

/* Here, derived3 inherits both derived1 and derived2.
   However, only one copy of base is present.
*/
class derived3 : public derived1, public derived2 {
public:
    int product() { return i * j * k; }
```

```

int main()
{
    derived3 ob;

    ob.i = 10; // unambiguous because only one copy present
    ob.j = 3;
    ob.k = 5;

    cout << "Product is " << ob.product() << '\n';

    return 0;
}

```

If **derived1** and **derived2** had not inherited **base** as virtual, the statement

```
ob.i = 10;
```

would have been ambiguous and a compile-time error would have resulted. (See Exercise 1, below.)

- It is important to understand that when a base class is inherited as virtual by a derived class, that base class still exists within that derived class. For example, assuming the preceding program, this fragment is perfectly valid:

```

derived1 ob;
ob.i = 100;

```

The only difference between a normal base class and a virtual one occurs when an object inherits the base more than once. If virtual base classes are used, only one base class is present in the object. Otherwise, multiple copies will be found.

EXERCISES

- Using the program in Example 1, remove the **virtual** keyword and try to compile the program. See what types of errors result.
- Explain why a virtual base class might be necessary.

SKILLS CHECK



Mastery
Skills Check

At this point you should be able to perform the following exercises and answer the questions.

1. Create a generic base class called **building** that stores the number of floors a building has, the number of rooms, and its total square footage. Create a derived class called **house** that inherits **building** and also stores the number of bedrooms and the number of bathrooms. Next, create a derived class called **office** that inherits **building** and also stores the number of fire extinguishers and the number of telephones. Note: Your solution may differ from the answer given in the back of this book. However, if it is functionally the same, count it as correct.
2. When a base class is inherited as public by the derived class, what happens to its public members? What happens to its private members? If the base is inherited as private by the derived class, what happens to its public and private members?
3. Explain what **protected** means. (Be sure to explain what it means both when referring to members of a class and when it is used as an inheritance access specifier.)
4. When one class inherits another, when are the classes' constructors called? When are their destructors called?
5. Given this skeleton, fill in the details as indicated in the comments:

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

class planet {
protected:
    double distance; // miles from the sun
    int revolve; // in days
public:
```



```
planet(double d, int r) { distance = d; revolve = r; }
};
```

```
class earth : public planet {
    double circumference; // circumference of orbit
public:
    /* Create earth(double d, int r). Have it pass the
       distance and days of revolution back to planet.
       Have it compute the circumference of the orbit.
       (Hint: circumference = 2r*3.1416.)
    */
    /* Create a function called show() that displays the
       information. */
};
```

```
int main()
{
    earth ob(93000000, 365);

    ob.show();

    return 0;
}
```

6. Fix the following program:

```
/* A variation on the vehicle hierarchy. But
   this program contains an error. Fix it. Hint:
   try compiling it as is and observe the error
   messages.
*/
#include <iostream>
using namespace std;

// A base class for various types of vehicles.
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
    }
    void showv()
    {
```

C++

```
    cout << "Wheels: " << num_wheels << '\n';
    cout << "Range: " << range << '\n';
}
};

enum motor {gas, electric, diesel};

class motorized : public vehicle {
    enum motor mtr;
public:
    motorized(enum motor m, int w, int r) : vehicle(w, r)
    {
        mtr = m;
    }
    void showm() {
        cout << "Motor: ";
        switch(mtr) {
            case gas : cout << "Gas\n";
                break;
            case electric : cout << "Electric\n";
                break;
            case diesel : cout << "Diesel\n";
                break;
        }
    }
};

class road_use : public vehicle {
    int passengers;
public:
    road_use(int p, int w, int r) : vehicle(w, r)
    {
        passengers = p;
    }
    void showr()
    {
        cout << "Passengers: " << passengers << '\n';
    }
};

enum steering { power, rack_pinion, manual };

class car : public motorized, public road_use {
    enum steering strng;
public:
```

```
car(enum steering s, enum motor m, int w, int r, int p) :
    road_use(p, w, r), motorized(m, w, r), vehicle(w, r)
{
    strng = s;
}

void show() {
    showr(); showr(); showm();
    cout << "Steering: ";
    switch(strng) {
        case power : cout << "Power\n";
            break;
        case rack_pinion : cout << "Rack and Pinion\n";
            break;
        case manual : cout << "Manual\n";
            break;
    }
}

};

int main()
{
    car c(power, gas, 4, 500, 5);

    c.show();

    return 0;
}
```



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. In Exercise 6 from the preceding Mastery Skills Check section, you might have seen a warning message (or perhaps an error message) concerning the use of the **switch** statement within **car** and **motorized**. Why?
2. As you know from the preceding chapter, most operators overloaded in a base class are available for use in a derived

class. Which one or ones are not? Can you offer a reason why this is the case?

3. Following is a reworked version of the **coord** class from the previous chapter. This time it is used as a base for another class called **quad**, which also maintains the quadrant the specific point is in. On your own, run this program and try to understand its output.

```
/* Overload the +, -, and = relative to coord class. Then
   use coord as a base for quad. */
#include <iostream>
using namespace std;
```

```
class coord {
public:
    int x, y; // coordinate values
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2);
    coord operator-(coord ob2);
    coord operator=(coord ob2);
};
```

```
// Overload + relative to coord class.
coord coord::operator+(coord ob2)
```

```
{
    coord temp;

    cout << "Using coord operator+()\n";
```

```
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
```

```
    return temp;
}
```

```
// Overload - relative to coord class.
coord coord::operator-(coord ob2)
```

```
{
    coord temp;

    cout << "Using coord operator-()\n";
```



```
temp.x = x - ob2.x;
temp.y = y - ob2.y;

return temp;
}

// Overload = relative to coord.
coord coord::operator=(coord ob2)
{
    cout << "Using coord operator=()\n";

    x = ob2.x;
    y = ob2.y;

    return *this; // return the object that is assigned to
}

class quad : public coord {
    int quadrant;
public:
    quad() { x = 0; y = 0; quadrant = 0; }
    quad(int x, int y) : coord(x, y)
    {
        if(x>=0 && y>=0) quadrant = 1;
        else if(x<0 && y>=0) quadrant = 2;
        else if(x<0 && y<0) quadrant = 3;
        else quadrant = 4;
    }
    void showq()
    {
        cout << "Point in Quadrant: " << quadrant << '\n';
    }
    quad operator=(coord ob2);
};

quad quad::operator=(coord ob2)
{
    cout << "Using quad operator=()\n";

    x = ob2.x;
    y = ob2.y;
    if(x>=0 && y>=0) quadrant = 1;
    else if(x<0 && y>=0) quadrant = 2;
    else if(x<0 && y<0) quadrant = 3;
    else quadrant = 4;
```

```
    return *this;
}

int main()
{
    quad o1(10, 10), o2(15, 3), o3;
    int x, y;

    o3 = o1 + o2; // add two objects - this calls operator+()
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 - o2; // subtract two objects
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // assign an object
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

4. Again on your own, convert the program shown in Exercise 3 so that it uses friend operator functions.

8

Introducing the C++ I/O System

chapter objectives

8.1 Some C++ I/O basics

8.2 Formatted I/O

8.3 Using **width()**, **precision()**, and **fill()**

8.4 Using I/O manipulators

8.5 Creating your own inserters

8.6 Creating extractors

ALTHOUGH you have been using C++-style I/O since the first chapter of this book, it is time to explore it more fully. Like its predecessor, C, the C++ language includes an I/O system that is both flexible and powerful. It is important to understand that C++ still supports the entire C I/O system. However, C++ supplies a complete set of object-oriented I/O routines. The major advantage of the C++ I/O system is that it can be overloaded relative to classes that you create. Put differently, the C++ I/O system allows you to seamlessly integrate new types that you create.

Like the C I/O system, the C++ object-oriented I/O system makes little distinction between console and file I/O. File and console I/O are really just different perspectives on the same mechanism. The examples in this chapter use console I/O, but the information presented is applicable to file I/O as well. (File I/O is examined in detail in Chapter 9.)

At the time of this writing, there are two versions of the I/O library in use: the older one that is based on the original specifications for C++ and the newer one defined by Standard C++. For the most part the two libraries appear the same to the programmer. This is because the new I/O library is, in essence, simply an updated and improved version of the old one. In fact, the vast majority of the differences between the two occur beneath the surface, in the way that the libraries are implemented—not in the way that they are used. From the programmer's perspective, the main difference is that the new I/O library contains a few additional features and defines some new data types. Thus, the new I/O library is essentially a superset of the old one. Nearly all programs originally written for the old library will compile without substantive changes when the new library is used. Since the old-style I/O library is now obsolete, this book describes only the new I/O library as defined by Standard C++. But most of the information is applicable to the old I/O library as well.

This chapter covers several aspects of C++'s I/O system, including formatted I/O, I/O manipulators, and creating your own I/O inserters and extractors. As you will see, the C++ I/O system shares many features with the C I/O system.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Create a class hierarchy that stores information about airships. Start with a general base class called **airship** that stores the number of passengers and the amount of cargo (in pounds) that can be carried. Then create two derived classes called **airplane** and **balloon** from **airship**. Have **airplane** store the type of engine used (propeller or jet) and range, in miles. Have **balloon** store information about the type of gas used to lift the balloon (hydrogen or helium) and its maximum altitude (in feet). Create a short program that demonstrates this class hierarchy. (Your solution will, no doubt, differ from the answer shown in the back of this book. If it is functionally similar, count it as correct.)
2. What is **protected** used for?
3. Given the following class hierarchy, in what order are the constructor functions called? In what order are the destructor functions called?

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

```
class A {
public:
    A() { cout << "Constructing A\n"; }
    ~A() { cout << "Destructing A\n"; }
};
```

```
class B : public A {
public:
    B() { cout << "Constructing B\n"; }
    ~B() { cout << "Destructing B\n"; }
};
```

```

C++
class C : public B {
public:
    C() { cout << "Constructing C\n"; }
    ~C() { cout << "Destructing C\n"; }
};

int main()
{
    C ob;

    return 0;
}

```

4. Given the following fragment, in what order are the constructor functions called?

```
class myclass : public A, public B, public C { ...
```

5. Fill in the missing constructor functions in this program:

```

#include <iostream>
using namespace std;

class base {
    int i, j;
public:
    // need constructor
    void showij() { cout << i << ' ' << j << '\n'; }
};

class derived : public base {
    int k;
public:
    // need constructor
    void show() { cout << k << ' '; showij(); }
};

int main()
{
    derived ob(1, 2, 3);

    ob.show();
}

```

```
return 0;
}
```

6. In general, when you define a class hierarchy, you begin with the most _____ class and move to the most _____ class. (Fill in the missing words.)

8.1

SOME C++ I/O BASICS

Before we begin our examination of C++ I/O, a few general comments are in order. The C++ I/O system, like the C I/O system, operates through *streams*. Because of your C programming experience, you should already know what a stream is, but here is a summary. A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the I/O system presents the programmer with a consistent interface, even though it operates on devices with differing capabilities. For example, the same function that you use to write to the screen can be used to write to a disk file or to the printer.

As you know, when a C program begins execution, three predefined streams are automatically opened: **stdin**, **stdout**, and **stderr**. A similar thing happens when a C++ program starts running. When a C++ program begins, these four streams are automatically opened:

<u>Stream</u>	<u>Meaning</u>	<u>Default Device</u>
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error	Screen
clog	Buffered version of cerr	Screen

As you have probably guessed, the streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**. You have already been using **cin** and **cout**. The stream **clog** is simply a buffered version of **cerr**. Standard C++ also opens wide (16-bit) character versions of these streams called **wcin**, **wcout**, **wcerr**, and **wclog**, but we won't be

using them in this book. The wide character streams exist to support languages, such as Chinese, that require large character sets.

By default, the standard streams are used to communicate with the console. However, in environments that support I/O redirection, these streams can be redirected to other devices.

As you learned in Chapter 1, C++ provides support for its I/O system in the header file `<iostream>`. In this file, a rather complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of *template classes*. Template classes, also called generic classes, will be discussed more fully in Chapter 11; briefly, a template class defines the form of a class without fully specifying the data upon which it will operate. Once a template class has been defined, specific instances of it can be created. As it relates to the I/O library, Standard C++ creates two specific versions of the I/O template classes: one for 8-bit characters and another for wide characters. This book will discuss only the 8-bit character classes, since they are by far the most frequently used.

The C++ I/O system is built upon two related, but different, template class hierarchies. The first is derived from the low-level I/O class called **basic_streambuf**. This class supplies the basic, low-level input and output operations and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly. The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error-checking, and status information related to stream I/O. **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively.

As explained earlier, the I/O library creates two specific versions of the class hierarchies just described: one for 8-bit characters and one for wide characters. The following table shows the mapping of the template class names to their 8-bit character-based versions (including some that will be used in Chapter 9):

<u>Template Class</u>	<u>8-Bit Character-Based Class</u>
<code>basic_streambuf</code>	<code>streambuf</code>
<code>basic_ios</code>	<code>ios</code>

Template Class-Bit Character-Based Class

basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

The character-based names will be used throughout the remainder of this book, since they are the names that you will use in your programs. They are also the same names that were used by the old I/O library. This is why the old and the new I/O libraries are compatible at the source code level.

One last point: The **ios** class contains many member functions and variables that control or monitor the fundamental operation of a stream. It will be referred to frequently. Just remember that if you include `<iostream>` in your program, you will have access to this important class.

8.2***FORMATTED I/O***

Until now, all examples in this book displayed information to the screen using C++'s default formats. However, it is possible to output information in a wide variety of forms. In fact, you can format data using C++'s I/O system in much the same way that you do using C's **printf()** function. Also, you can alter certain aspects of the way information is input.

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios** class declares a bitmask enumeration called **fmtflags**, in which the following values are defined:

adjustfield	floatfield	right	skipws
basefield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	
fixed	oct	showpos	

C++

These values are used to set or clear the format flags and are defined within **ios**. If you are using an older, nonstandard compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer.

When the **skipws** flag is set, leading whitespace characters (spaces, tabs, and newlines) are discarded when input is being performed on a stream. When **skipws** is cleared, whitespace characters are not discarded.

When the **left** flag is set, output is left justified. When **right** is set, output is right justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags is set, output is right justified by default.

By default, numeric values are output in decimal. However, it is possible to change the number base. Setting the **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.

Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.

By default, when scientific notation is displayed, the e is lowercase. Also, when a hexadecimal value is displayed, the x is lowercase. When **uppercase** is set, these characters are displayed uppercase.

Setting **showpos** causes a leading plus sign to be displayed before positive values.

Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

If the **scientific** flag is set, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.

When **unitbuf** is set, the buffer is flushed after each insertion operation.

When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.

Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

To set a format flag, use the `setf()` function. This function is a member of `ios`. Its most common form is shown here:

```
fmtflags setf(fmtflags flags);
```

This function returns the previous settings of the format flags and turns on those flags specified by *flags*. (All other flags are unaffected.) For example, to turn on the **showpos** flag, you can use this statement:

```
stream.setf(ios::showpos);
```

Here *stream* is the stream you wish to affect. Notice the use of the scope resolution operator. Remember, **showpos** is an enumerated constant within the `ios` class. Therefore, it is necessary to tell the compiler this fact by preceding **showpos** with the class name and the scope resolution operator. If you don't, the constant **showpos** will simply not be recognized.

It is important to understand that `setf()` is a member function of the `ios` class and affects streams created by that class. Therefore, any call to `setf()` is done relative to a specific stream. There is no concept of calling `setf()` by itself. Put differently, there is no concept in C++ of global format status. Each stream maintains its own format status information individually.

It is possible to set more than one flag in a single call to `setf()`, rather than making multiple calls. To do this, OR together the values of the flags you want to set. For example, this call sets the **showbase** and **hex** flags for `cout`:

```
cout.setf(ios::showbase | ios::hex);
```

*Because the format flags are defined within the `ios` class, you must access their values by using `ios` and the scope resolution operator. For example, **showbase** by itself will not be recognized; you must specify `ios::showbase`.*

The complement of `setf()` is `unsetf()`. This member function of `ios` clears one or more format flags. Its most common prototype form is shown here:

```
void unsetf(fmtflags flags);
```

The flags specified by *flags* are cleared. (All other flags are unaffected.)

There will be times when you want to know, but not alter, the current format settings. Since both `setf()` and `unsetf()` alter the setting of one or more flags, `ios` also includes the member function `flags()`, which simply returns the current setting of each format flag. Its prototype is shown here:

```
fmtflags flags();
```

The `flags()` function has a second form that allows you to set *all* format flags associated with a stream to those specified in the argument to `flags()`. The prototype for this version of `flags()` is shown here:

```
fmtflags flags(fmtflags f);
```

When you use this version, the bit pattern found in `f` is copied to the variable used to hold the format flags associated with the stream, thus overwriting all previous flag settings. The function returns the previous settings.

EXAMPLES

1. Here is an example that shows how to set several of the format flags:

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

int main()
{
    // display using default settings
    cout << 123.23 << " hello " << 100 << '\n';
    cout << 10 << ' ' << -10 << '\n';
    cout << 100.0 << "\n\n";

    // now, change formats
    cout.unsetf(ios::dec); // not required by all compilers
    cout.setf(ios::hex | ios::scientific);
    cout << 123.23 << " hello " << 100 << '\n';

    cout.setf(ios::showpos);
    cout << 10 << ' ' << -10 << '\n';
```

```

cout.setf(ios::showpoint | ios::fixed);
cout << 100.0;

return 0;
}

```

This program displays the following output:

```

123.23 hello 100
10 -10
100

1.232300e+02 hello 64
a ffffffff6
+100.000000

```

Notice that the **showpos** flag affects only decimal output. It does not affect the value 10 when output in hexadecimal. Also notice the **unsetf()** call that turns off the **dec** flag (which is on by default). This call is not needed by all compilers. But for some compilers, the **dec** flag overrides the other flags, so it is necessary to turn it off when turning on either **hex** or **oct**. In general, for maximum portability, it is better to set only the number base that you want to use and clear the others.

- The following program illustrates the effect of the **uppercase** flag. It first sets the **uppercase**, **showbase**, and **hex** flags. It then outputs 88 in hexadecimal. In this case, the X used in the hexadecimal notation is uppercase. Next, it clears the **uppercase** flag by using **unsetf()** and again outputs 88 in hexadecimal. This time, the x is lowercase.

```

#include <iostream>
using namespace std;

int main()
{
    cout.unsetf(ios::dec);
    cout.setf(ios::uppercase | ios::showbase | ios::hex);

    cout << 88 << '\n';

    cout.unsetf(ios::uppercase);

    cout << 88 << '\n';
}

```



```
    return 0;
}
```

3. The following program uses **flags()** to display the settings of the format flags relative to **cout**. Pay special attention to the **showflags()** function. You might find it useful in programs you write.

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

void showflags();

int main()
{
    // show default condition of format flags
    showflags();

    cout.setf(ios::oct | ios::showbase | ios::fixed);

    showflags();

    return 0;
}

// This function displays the status of the format flags.
void showflags()
{
    ios::fmtflags f;

    f = cout.flags(); // get flag settings

    if(f & ios::skipws) cout << "skipws on\n";
    else cout << "skipws off\n";

    if(f & ios::left) cout << "left on\n";
    else cout << "left off\n";

    if(f & ios::right) cout << "right on\n";
    else cout << "right off\n";

    if(f & ios::internal) cout << "internal on\n";
    else cout << "internal off\n";
}
```

```
if(f & ios::dec) cout << "dec on\n";
else cout << "dec off\n";

if(f & ios::oct) cout << "oct on\n";
else cout << "oct off\n";

if(f & ios::hex) cout << "hex on\n";
else cout << "hex off\n";

if(f & ios::showbase) cout << "showbase on\n";
else cout << "showbase off\n";

if(f & ios::showpoint) cout << "showpoint on\n";
else cout << "showpoint off\n";

if(f & ios::showpos) cout << "showpos on\n";
else cout << "showpos off\n";

if(f & ios::uppercase) cout << "uppercase on\n";
else cout << "uppercase off\n";

if(f & ios::scientific) cout << "scientific on\n";
else cout << "scientific off\n";

if(f & ios::fixed) cout << "fixed on\n";
else cout << "fixed off\n";

if(f & ios::unitbuf) cout << "unitbuf on\n";
else cout << "unitbuf off\n";

if(f & ios::boolalpha) cout << "boolalpha on\n";
else cout << "boolalpha off\n";

cout << "\n";
}
```

Inside `showflags()`, the local variable `f` is declared to be of type `fmtflags`. If your compiler does not define `fmtflags`, declare this variable as `long` instead. The output from the program is shown here:

```
skipws on
left off
right off
internal off
```

C++

```

dec on
oct off
hex off
showbase off
showpoint off
showpos off
uppercase off
scientific off
fixed off
unitbuf off
boolalpha off

```

```

skipws on
left off
right off
internal off
dec on
oct on
hex off
showbase on
showpoint off
showpos off
uppercase off
scientific off
fixed on
unitbuf off
boolalpha off

```

4. The next program illustrates the second version of `flags()`. It first constructs a flag mask that turns on **showpos**, **showbase**, **oct**, and **right**. It then uses `flags()` to set the flag variable associated with `cout` to these settings. The function `showflags()` verifies that the flags are set as indicated. (This is the same function used in the previous program.)

```

#include <iostream>
using namespace std;

void showflags();

int main()
{
    // show default condition of format flags
    showflags();
}

```



```
// showpos, showbase, oct, right are on, others off
ios::fmtflags f = ios::showpos | ios::showbase |
ios::oct | ios::right;
```

```
cout.flags(f); // set flags
```

```
showflags();
```

```
return 0;
```

```
}
```

EXERCISES

1. Write a program that sets **cout**'s flags so that integers display a + sign when positive values are displayed. Demonstrate that you have set the format flags correctly.
2. Write a program that sets **cout**'s flags so that the decimal point is always shown when floating-point values are displayed. Also, display all floating-point values in scientific notation with an uppercase E.
3. Write a program that saves the current state of the format flags, sets **showbase** and **hex**, and displays the value 100. Then reset the flags to their previous values.

8.3

USING `width()`, `precision()`, AND `fill()`

In addition to the formatting flags, there are three member functions defined by **ios** that set these format parameters: the field width, the precision, and the fill character. These are **width()**, **precision()**, and **fill()**, respectively.

By default, when a value is output, it occupies only as much space as the number of characters it takes to display. However, you can

specify a minimum field width by using the **width()** function. Its prototype is shown here:

```
streamsize width(streamsize w);
```

Here *w* becomes the field width, and the previous field width is returned. The **streamsize** type is defined by **<iostream>** as some form of integer. In some implementations, each time an output operation is performed, the field width returns to its default setting, so it might be necessary to set the minimum field width before each output statement.

After you set a minimum field width, when a value uses less than the specified width, the field is padded with the current fill character (the space, by default) so that the field width is reached. However, keep in mind that if the size of the output value exceeds the minimum field width, the field will be overrun. No values are truncated.

By default, six digits of precision are used. You can set this number by using the **precision()** function. Its prototype is shown here:

```
streamsize precision(streamsize p);
```

Here the precision is set to *p* and the old value is returned.

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill()** function. Its prototype is shown here:

```
char fill(char ch);
```

After a call to **fill()**, *ch* becomes the new fill character, and the old one is returned.

EXAMPLES

1. Here is a program that illustrates the format functions:

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

int main()
{
    cout.width(10);           // set minimum field width
```

```

cout << "hello" << '\n'; // right-justify by default
cout.fill('%');          // set fill character
cout.width(10);          // set width
cout << "hello" << '\n'; // right-justify by default
cout.setf(ios::left);    // left-justify
cout.width(10);          // set width
cout << "hello" << '\n'; // output left justified

cout.width(10);          // set width
cout.precision(10);      // set 10 digits of precision
cout << 123.234567 << '\n';
cout.width(10);          // set width
cout.precision(6);       // set 6 digits of precision
cout << 123.234567 << '\n';

return 0;
}

```

This program displays the following output:

```

hello
%%%hello
hello%%%
123.234567
123.235%%

```

Notice that the field width is set before each output statement.

- The following program shows how to use the C++ I/O format functions to create an aligned table of numbers:

```

// Create a table of square roots and squares.
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double x;

    cout.precision(4);
    cout << "      x      sqrt(x)      x^2\n\n";

    for(x = 2.0; x <= 20.0; x++) {
        cout.width(7);
        cout << x << " ";
    }
}

```



```
    cout.width(7);
    cout << sqrt(x) << " ";
    cout.width(7);
    cout << x*x << '\n';
}

return 0;
}
```

This program creates the following table:

x	sqrt(x)	x ²
2	1.414	4
3	1.732	9
4	2	16
5	2.236	25
6	2.449	36
7	2.646	49
8	2.828	64
9	3	81
10	3.162	100
11	3.317	121
12	3.464	144
13	3.606	169
14	3.742	196
15	3.873	225
16	4	256
17	4.123	289
18	4.243	324
19	4.359	361
20	4.472	400

EXERCISES

1. Create a program that prints the natural log and base 10 log of the numbers from 2 to 100. Format the table so the numbers are right justified within a field width of 10, using a precision of five decimal places.
2. Create a function called **center()** that has this prototype:

```
void center(char *s);
```

Have this function center the specified string on the screen. To accomplish this, use the **width()** function. Assume that the screen is 80 characters wide. (For simplicity, you may assume that no string exceeds 80 characters.) Write a program that demonstrates that your function works.

3. On your own, experiment with the format flags and the format functions. Once you become familiar with the C++ I/O system, you will have no trouble using it to format output any way you like.

8.4**USING I/O MANIPULATORS**

There is a second way that you can format information using C++'s I/O system. This method uses special functions called *I/O manipulators*. As you will see, I/O manipulators are, in some situations, easier to use than the **ios** format flags and functions.

I/O manipulators are special I/O format functions that can occur *within* an I/O statement, instead of separate from it the way the **ios** member functions must. The standard manipulators are shown in Table 8-1. As you can see, many of the I/O manipulators parallel member functions of the **ios** class. Many of the manipulators shown in Table 8-1 were added recently to Standard C++ and will be supported only by modern compilers.

To access manipulators that take parameters, such as **setw()**, you must include **<iomanip>** in your program. This is not necessary when you are using a manipulator that does not require an argument.

As stated above, the manipulators can occur in the chain of I/O operations. For example:

```
cout << oct << 100 << hex << 100;  
cout << setw(10) << 100;
```

The first statement tells **cout** to display integers in octal and then outputs 100 in octal. It then tells the stream to display integers in hexadecimal and then outputs 100 in hexadecimal format. The second

Manipulator	Purpose	Input/Output
boolalpha	Turns on boolalpha flag	Input/Output
dec	Turns on dec flag	Input/Output
endl	Outputs a newline character and flushes the stream	Output
ends	Outputs a null	Output
fixed	Turns on fixed flag	Output
flush	Flushes a stream	Output
hex	Turns on hex flag	Input/Output
internal	Turns on internal flag	Output
left	Turns on left flag	Output
noboolalpha	Turns off boolalpha flag	Input/Output
noshowbase	Turns off showbase flag	Output
noshowpoint	Turns off showpoint flag	Output
noshowpos	Turns off showpos flag	Output
noskipws	Turns off skipws flag	Input
nounitbuf	Turns off unitbuf flag	Output
nouppercase	Turns off uppercase flag	Output
oct	Turns on oct flag	Input/Output
resetiosflags(fmtflags <i>f</i>)	Turns off the flags specified in <i>f</i>	Input/Output
right	Turns on right flag	Output
scientific	Turns on scientific flag	Output
setbase(int <i>base</i>)	Sets the number base to <i>base</i>	Input/Output
setfill(int <i>ch</i>)	Sets the fill character to <i>ch</i>	Output
setiosflags(fmtflags <i>f</i>)	Turns on the flags specified in <i>f</i>	Input/Output
setprecision(int <i>p</i>)	Sets the number of digits of precision	Output
setw(int <i>w</i>)	Sets the field width to <i>w</i>	Output
showbase	Turns on showbase flag	Output
showpoint	Turns on showpoint flag	Output
showpos	Turns on showpos flag	Output
skipws	Turns on skipws flag	Input
unitbuf	Turns on unitbuf flag	Output
uppercase	Turns on uppercase flag	Output
ws	Skips leading white space	Input

TABLE 8-1 The Standard C++ I/O Manipulators ▼

statement sets the field width to 10 and then displays 100 in hexadecimal format again. Notice that when a manipulator does not take an argument, such as `oct` in the example, it is not followed by parentheses. This is because it is the address of the manipulator that is passed to the overloaded `<<` operator.

Keep in mind that an I/O manipulator affects only the stream of which the I/O expression is a part. I/O manipulators do *not* affect all streams currently opened for use.

As the preceding example suggests, the main advantages of using manipulators over the `ios` member functions is that they are often easier to use and allow more compact code to be written.

If you wish to set specific format flags manually by using a manipulator, use `setiosflags()`. This manipulator performs the same function as the member function `setf()`. To turn off flags, use the `resetiosflags()` manipulator. This manipulator is equivalent to `unsetf()`.

EXAMPLES

1. This program demonstrates several of the I/O manipulators:

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

int main()
{
    cout << hex << 100 << endl;
    cout << oct << 10 << endl;

    cout << setfill('X') << setw(10);
    cout << 100 << " hi " << endl;

    return 0;
}
```

This program displays the following:

```
64
12
XXXXXXXX144 hi
```

2. Here is another version of the program that displays a table of the squares and square roots of the numbers 2 through 20. This version uses I/O manipulators instead of member functions and format flags.

```

/* This version uses I/O manipulators to display
   the table of squares and square roots. */
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
    double x;

    cout << setprecision(4);
    cout << "      x      sqrt(x)      x^2\n\n";

    for(x = 2.0; x <= 20.0; x++) {
        cout << setw(7) << x << " ";
        cout << setw(7) << sqrt(x) << " ";
        cout << setw(7) << x*x << '\n';
    }

    return 0;
}

```

3. One of the most interesting format flags added by the new I/O library is **boolalpha**. This flag can be set either directly or by using the new manipulators **boolalpha** or **noboolalpha**. What makes **boolalpha** so interesting is that setting it allows you to input and output Boolean values using the keywords **true** and **false**. Normally you must enter 1 for true and 0 for false. For example, consider the following program:

```

// Demonstrate boolalpha format flag.
#include <iostream>
using namespace std;

int main()
{
    bool b;
    cout << "Before setting boolalpha flag: ";

```

```
b = true;
cout << b << " ";
b = false;
cout << b << endl;

cout << "After setting boolalpha flag: ";
b = true;
cout << boolalpha << b << " ";
b = false;
cout << b << endl;

cout << "Enter a Boolean value: ";
cin >> boolalpha >> b; // you can enter true or false
cout << "You entered " << b;

return 0;
}
```

Here is a sample run:

```
Before setting boolalpha flag: 1 0
After setting boolalpha flag: true false
Enter a Boolean value: true
You entered true
```

As you can see, once the **boolalpha** flag has been set, Boolean values are input and output using the words **true** or **false**. Notice that you must set the **boolalpha** flags for **cin** and **cout** separately. As with all format flags, setting **boolalpha** for one stream does not imply that it is also set for another.

EXERCISES

1. Redo Exercises 1 and 2 from Section 8.3, this time using I/O manipulators instead of member functions and format flags.
 2. Show the I/O statement that outputs the value 100 in hexadecimal with the base indicator (the 0x) shown. Use the **setiosflags()** manipulator to accomplish this.
 3. Explain the effect of setting the **boolalpha** flag.
-

As stated earlier, one of the advantages of the C++ I/O system is that you can overload the I/O operators for classes that you create. By doing so, you can seamlessly incorporate your classes into your C++ programs. In this section you learn how to overload C++'s output operator `<<`.

In the language of C++, the output operation is called an *insertion* and the `<<` is called the *insertion operator*. When you overload the `<<` for output, you are creating an *inserter function*, or *inserter* for short. The rationale for these terms comes from the fact that an output operator *inserts* information into a stream.

All inserter functions have this general form:

```
ostream &operator <<(ostream &stream, class-name ob)
{
    // body of inserter
    return stream;
}
```

The first parameter is a reference to an object of type **ostream**. This means that *stream* must be an output stream. (Remember, **ostream** is derived from the **ios** class.) The second parameter receives the object that will be output. (This can also be a reference parameter, if that is more suitable to your application.) Notice that the inserter function returns a reference to *stream*, which is of type **ostream**. This is required if the overloaded `<<` is going to be used in a series of I/O expressions, such as

```
cout << ob1 << ob2 << ob3;
```

Within an inserter you can perform any type of procedure. What an inserter does is completely up to you. However, for the inserter to be consistent with good programming practices, you should limit its operations to outputting information to a stream.

Although you might find this surprising at first, an inserter *cannot* be a member of the class on which it is designed to operate. Here is why: When an operator function of any type is a member of a class, the left operand, which is passed implicitly through the **this pointer**, is the object that generates the call to the operator function. This implies that the left operand is an object of that class. Therefore, if an

overloaded operator function is a member of a class, the left operand must be an object of that class. However, when you create an inserter, the left operand is a stream and the right operand is the object that you want to output. Therefore, an inserter cannot be a member function.

The fact that an inserter cannot be a member function might appear to be a serious flaw in C++ because it seems to imply that all data of a class that will be output using an inserter will need to be public, thus violating the key principle of encapsulation. However, this is not the case. Even though inserters cannot be members of the class upon which they are designed to operate, they can be friends of the class. In fact, in most programming situations you will encounter, an overloaded inserter will be a friend of the class for which it was created.

EXAMPLES

1. As a simple first example, this program contains an inserter for the **coord** class, developed in a previous chapter:

```
// Use a friend inserter for objects of type coord.
#include <iostream>
using namespace std;

class coord {
    int x, y;
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

int main()
{
    coord a(1, 1), b(10, 23);
```

C++

```
cout << a << b;
```

```
return 0;
```

```
}
```

This program displays the following:

```
1, 1
```

```
10, 23
```

The inserter in this program illustrates one very important point about creating your own inserters: make them as general as possible. In this case, the I/O statement inside the inserter outputs the values of **x** and **y** to **stream**, which is whatever stream is passed to the function. As you will see in the following chapter, when written correctly the same inserter that outputs to the screen can be used to output to *any* stream. Sometimes beginners are tempted to write the **coord** inserter like this:

```
ostream &operator<<(ostream &stream, coord ob)
{
    cout << ob.x << ", " << ob.y << '\n';
    return stream;
}
```

In this case, the output statement is hard-coded to display information on the standard output device linked to **cout**. However, this prevents the inserter from being used by other streams. The point is that you should make your inserters as general as possible because there is no disadvantage to doing so.

2. For the sake of illustration, here is the preceding program revised so that the inserter is *not* a friend of the **coord** class. Because the inserter does not have access to the private parts of **coord**, the variables **x** and **y** have to be made public.

```
/* Create an inserter for objects of type coord, using
   a non-friend inserter. */
```

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

```
class coord {
public:
    int x, y; // must be public
```



```
coord() { x = 0; y = 0; }
coord(int i, int j) { x = i; y = j; }
};

// An inserter for the coord class.
ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

int main()
{
    coord a(1, 1), b(10, 23);
    cout << a << b;

    return 0;
}
```

3. An inserter is not limited to displaying only textual information. An inserter can perform any operation or conversion necessary to output information in a form needed by a particular device or situation. For example, it is perfectly valid to create an inserter that sends information to a plotter. In this case, the inserter will need to send appropriate plotter codes in addition to the information. To allow you to taste the flavor of this type of inserter, the following program creates a class called **triangle**, which stores the width and height of a right triangle. The inserter for this class displays the triangle on the screen.

```
// This program draws right triangles
#include <iostream>
using namespace std;

class triangle {
    int height, base;
public:
    triangle(int h, int b) { height = h; base = b; }
    friend ostream &operator<<(ostream &stream, triangle ob);
};

// Draw a triangle.
ostream &operator<<(ostream &stream, triangle ob)
{
```

```
int i, j, h, k;

i = j = ob.base-1;
for(h=ob.height-1; h; h--) {
    for(k=i; k; k--)
        stream << ' ';
    stream << '*';

    if(j!=i) {
        for(k=j-i-1; k; k--)
            stream << ' ';
        stream << '*';
    }

    i--;
    stream << '\n';
}
for(k=0; k<ob.base; k++) stream << '*';
stream << '\n';

return stream;
}

int main()
{
    triangle t1(5, 5), t2(10, 10), t3(12, 12);

    cout << t1;
    cout << endl << t2 << endl << t3;

    return 0;
}
```

Notice that this program illustrates how a properly designed inserter can be fully integrated into a "normal" I/O expression. This program displays the following:

```
 *
  **
 ***
****
*****
```

```
          *
         **
        ***
       ****
      *****
     ******
    *******
   ********
  *********
 *****
```

```
          *
         **
        ***
       ****
      *****
     ******
    *******
   ********
  *********
 *****
```

EXERCISES

1. Given the following **strtype** class and partial program, create an inserter that displays a string:

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

```
class strtype {
    char *p;
```



```

C++
int len;
public:
    strtype(char *ptr);
    ~strtype() {delete [] p;}
    friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr)+1;
    p = new char [len];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    strcpy(p, ptr);
}

// Create operator<< inserter function here.

int main()
{
    strtype s1("This is a test."), s2("I like C++.");

    cout << s1 << '\n' << s2;

    return 0;
}

```

2. Replace the **show()** function in the following program with an **inserter** function:

```

#include <iostream>
using namespace std;

class planet {
protected:
    double distance; // miles from the sun
    int revolve; // in days
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
    double circumference; // circumference of orbit

```

```

public:
    earth(double d, int r) : planet(d, r) {
        circumference = 2*distance*3.1416;
    }

    /* Rewrite this so that it displays the information using
       an inserter function. */
    void show() {
        cout << "Distance from sun: " << distance << '\n';
        cout << "Days in orbit: " << revolve << '\n';
        cout << "Circumference of orbit: " << circumference << '\n';
    }
};

int main()
{
    earth ob(93000000, 365);

    cout << ob;

    return 0;
}

```

3. Explain why an inserter cannot be a member function.

8.6

CREATING EXTRACTORS

Just as you can overload the << output operator, you can overload the >> input operator. In C++, the >> is referred to as the *extraction operator* and a function that overloads it is called an *extractor*. The reason for this term is that the act of inputting information from a stream removes (that is, extracts) data from it.

The general form of an extractor function is shown here:

```

istream &operator>>(istream &stream, class-name &ob)
{
    // body of extractor
    return stream;
}

```

Extractors return a reference to **istream**, which is an input stream. The first parameter must be a reference to an input stream. The second parameter is a reference to the object that is receiving input.

For the same reason that an inserter cannot be a member function, an extractor cannot be a member function. Although you can perform any operation within an extractor, it is best to limit its activity to inputting information.

EXAMPLES

1. This program adds an extractor to the **coord** class:

```
// Add a friend extractor for objects of type coord.
#include <iostream>
using namespace std;

class coord {
    int x, y;
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
    friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
    cout << "Enter coordinates: ";
    stream >> ob.x >> ob.y;
    return stream;
}

int main()
{
    coord a(1, 1), b(10, 23);
```



```

cout << a << b;

cin >> a;
cout << a;

return 0;
}

```

Notice how the extractor also prompts the user for input. Although such prompting is not required (or even desired) for most situations, this function shows how a customized extractor can simplify coding when a prompting message is needed.

- Here an inventory class is created that stores the name of an item, the number on hand, and its cost. The program includes both an inserter and an extractor for this class.

```

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

class inventory {
    char item[40]; // name of item
    int onhand; // number on hand
    double cost; // cost of item
public:
    inventory(char *i, int o, double c)
    {
        strcpy(item, i);
        onhand = o;
        cost = c;
    }
    friend ostream &operator<<(ostream &stream, inventory ob);
    friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ": " << ob.onhand;
    stream << " on hand at $" << ob.cost << '\n';

    return stream;
}

istream &operator>>(istream &stream, inventory &ob)

```



```
{
    cout << "Enter item name: ";
    stream >> ob.item;
    cout << "Enter number on hand: ";
    stream >> ob.onhand;
    cout << "Enter cost: ";
    stream >> ob.cost;

    return stream;
}

int main()
{
    inventory ob("hammer", 4, 12.55);

    cout << ob;

    cin >> ob;

    cout << ob;

    return 0;
}
```

EXERCISES

1. Add an extractor to the **strtype** class from Exercise 1 in the preceding section.
 2. Create a class that stores an integer value and its lowest factor. Create both an inserter and an extractor for this class.
-

SKILLS CHECK



Mastery

Skills Check

At this point you should be able to perform the following exercises and answer the questions.

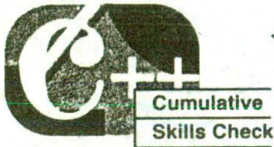
1. Write a program that displays the number 100 in decimal, hexadecimal, and octal. (Use the **ios** format flags.)
2. Write a program that displays the value 1000.5364 in a 20-character field, left justified, with two decimal places, using * as a fill character. (Use the **ios** format flags.)
3. Rewrite your answers to Exercises 1 and 2 so that they use I/O manipulators.
4. Show how to save the format flags for **cout** and how to restore them. Use either member functions or manipulators.
5. Create an inserter and an extractor for this class:

```
class pwr {
    int base;
    int exponent;
    double result; // base to the exponent power
public:
    pwr(int b, int e);
};
```

```
pwr::pwr(int b, int e)
{
    base = b;
    exponent = e;

    result = 1;
    for( ; e; e--) result = result * base;
}
```

6. Create a class called **box** that stores the dimensions of a square. Create an inserter that displays a square box on the screen. (Use any method you like to display the box.)



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Using the **stack** class shown here, create an inserter that displays the contents of the stack. Demonstrate that your inserter works.

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

#define SIZE 10

// Declare a stack class for characters
class stack {
    char stck[SIZE]; // holds the stack
    int tos; // index of top-of-stack
public:
    stack();
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
};

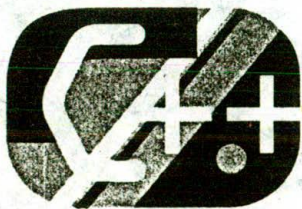
// Initialize the stack
stack::stack()
{
    tos = 0;
}

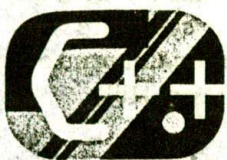
// Push a character.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}
```

```
// Pop a character.
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
}
```

2. Write a program that contains a class called **watch**. Using the standard time functions, have this class's constructor read the system time and store it. Create an inserter that displays the time.
3. Using the following class, which converts feet to inches, create an extractor that prompts the user for feet. Also, create an inserter that displays the number of feet and inches. Include a program that demonstrates that your inserter and extractor work.

```
class ft_to_inches {
    double feet;
    double inches;
public:
    void set(double f) {
        feet = f;
        inches = f * 12;
    }
};
```





9

Advanced C++ I/O

chapter objectives

9.1 Creating your own manipulators

9.2 File I/O basics

9.3 Unformatted, binary I/O

9.4 More unformatted I/O functions

9.5 Random access

9.6 Checking the I/O status

9.7 Customized I/O and files



THIS chapter continues the examination of the C++ I/O system. In it you will learn to create your own I/O manipulators and work with files. Keep in mind that the C++ I/O system is both rich and flexible and contains many features. While it is beyond the scope of this book to include all of those features, the most important ones are discussed here. A complete description of the C++ I/O system can be found in my book *C++: The Complete Reference* (Berkeley: Osborne/McGraw-Hill).

The C++ I/O system described in this chapter reflects the one defined by Standard C++ and is compatible with all major C++ compilers. If you have an older or nonconforming compiler, its I/O system will not have all the capabilities described here.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Write a program that displays the sentence "C++ is fun" in a 40-character-wide field using a colon (:) as the fill character.
2. Write a program that displays the outcome of $10/3$ to three decimal places. Use `ios` member functions to do this.
3. Redo the preceding program using I/O manipulators.
4. What is an inserter? What is an extractor?
5. Given the following class, create an inserter and an extractor for it.

```
class date {
    char d[9]; // store date as string: mm/dd/yy
public:
    // add inserter and extractor
};
```

6. What header must be included if your program is to use I/O manipulators that take parameters?

7. What predefined streams are created when a C++ program begins execution?

9.1 CREATING YOUR OWN MANIPULATORS

In addition to overloading the insertion and extraction operators, you can further customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons. First, a manipulator can consolidate a sequence of several separate I/O operations. For example, it is not uncommon to have situations in which the same sequence of I/O operations occurs frequently within a program. In these cases you can use a custom manipulator to perform these actions, thus simplifying your source code and preventing accidental errors. Second, a custom manipulator can be important when you need to perform I/O operations on a nonstandard device. For example, you could use a manipulator to send control codes to a special type of printer or an optical recognition system.

Custom manipulators are a feature of C++ that supports OOP, but they can also benefit programs that aren't object oriented. As you will see, custom manipulators can help make any I/O-intensive program clearer and more efficient.

As you know, there are two basic types of manipulators: those that operate on input streams and those that operate on output streams. In addition to these two broad categories, there is a secondary division: those manipulators that take an argument and those that don't. There are some significant differences between the way a parameterless manipulator and a parameterized manipulator are created. Further, creating parameterized manipulators is substantially more difficult than creating parameterless ones and is beyond the scope of this book. However, writing your own parameterless manipulators is quite easy and is examined here.

All parameterless manipulator output functions have this skeleton:

```
ostream &manip-name(ostream &stream)
{
    // your code here
    return stream;
}
```


Here *manip-name* is the name of the manipulator and *stream* is a reference to the invoking stream. A reference to the stream is returned. This is necessary if a manipulator is used as part of a larger I/O expression. It is important to understand that even though the manipulator has as its single argument a reference to the stream upon which it is operating, no argument is used when the manipulator is called in an output operation.

All parameterless input manipulator functions have this skeleton:

```
istream &manip-name(istream &stream)
{
    // your code here
    return stream;
}
```

An input manipulator receives a reference to the stream on which it was invoked. This stream must be returned by the manipulator.

It is crucial that your manipulators return a reference to the invoking stream. If this is not done, your manipulators cannot be used in a sequence of input or output operations.

EXAMPLES

- As a simple first example, the following program creates a manipulator called **setup()** that sets the field width to 10, the precision to 4, and the fill character to *.

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

ostream &setup(ostream &stream)
{
    stream.width(10);
    stream.precision(4);
    stream.fill('*');

    return stream;
}
```

```
int main()
```

```

cout << setup << 123.123456;

return 0;
}

```

As you can see, **setup** is used as part of an I/O expression in the same way that any of the built-in manipulators would be used.

2. Custom manipulators need not be complex to be useful. For example, the simple manipulators **atn()** and **note()**, shown here, provide a shorter way to output frequently used words or phrases.

```

#include <iostream>
using namespace std;

// Attention:
ostream &atn(ostream &stream)
{
    stream << "Attention: ";
    return stream;
}

// Please note:
ostream &note(ostream &stream)
{
    stream << "Please Note: ";
    return stream;
}

int main()
{
    cout << atn << "High voltage circuit\n";
    cout << note << "Turn off all lights\n";

    return 0;
}

```

Even though they are simple, if used frequently, these manipulators save you from some tedious typing.

3. This program creates the **getpass()** input manipulator, which rings the bell and then prompts for a password:

C++

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

// A simple input manipulator
istream &getpass(istream &stream)
{
    cout << '\a'; // sound bell
    cout << "Enter password: ";

    return stream;
}

int main()
{
    char pw[80];

    do {
        cin >> getpass >> pw;
    } while (strcmp(pw, "password"));

    cout << "Logon complete\n";

    return 0;
}
```

EXERCISES

1. Create an output manipulator that displays the current system time and date. Call this manipulator **td()**.
 2. Create an output manipulator called **sethex()** that sets output to hexadecimal and turns on the **uppercase** and **showbase** flags. Also, create an output manipulator called **reset()** that undoes the changes made by **sethex()**.
 3. Create an input manipulator called **skipchar()** that reads and ignores the next ten characters from the input stream.
-

It is now time to turn our attention to file I/O. As mentioned in the preceding chapter, file I/O and console I/O are closely related. In fact, the same class hierarchy that supports console I/O also supports file I/O. Thus, most of what you have already learned about I/O applies to files as well. Of course, file handling makes use of several new features.

To perform file I/O, you must include the header `<fstream>` in your program. It defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream** and **ostream**. Remember, **istream** and **ostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios** (discussed in the preceding chapter).

In C++, a file is opened by linking it to a stream. There are three types of streams: input, output, and input/output. Before you can open a file, you must first obtain a stream. To create an input stream, declare an object of type **ifstream**. To create an output stream, declare an object of type **ofstream**. Streams that will be performing both input and output operations must be declared as objects of type **fstream**. For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output:

```
ifstream in; // input
ofstream out; // output
fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using the function **open()**. This function is a member of each of the three file stream classes. The prototype for each is shown here:

```
void ifstream::open(const char *filename,
                   openmode mode = ios::in);

void ofstream::open(const char *filename,
                   openmode mode = ios::out | ios::trunc);

void fstream::open(const char *filename,
                  openmode mode = ios::in | ios::out);
```

Here *filename* is the name of the file, which can include a path specifier. The value of *mode* determines how the file is opened. It must

be a value of type **ios::mode**, which is an enumeration defined by **ios** that contains the following values:

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

You can combine two or more of these values by ORing them together. Let's see what each of these values means.

Including **ios::app** causes all output to that file to be appended to the end. This value can be used only with files capable of output.

Including **ios::ate** causes a seek to the end of the file to occur when the file is opened. Although **ios::ate** causes a seek to end-of-file, I/O operations can still occur anywhere within the file.

The **ios::in** value specifies that the file is capable of input. The **ios::out** value specifies that the file is capable of output.

The **ios::binary** value causes a file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations might take place, such as carriage return/linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur. Keep in mind that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode. The only difference is whether character translations take place.

The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed and the file to be truncated to zero length. When you create an output stream using **ofstream**, any preexisting file with the same name is automatically truncated.

The following fragment opens an output file called **test**:

```
ofstream mystream;
mystream.open("test");
```

Since the *mode* parameter to **open()** defaults to a value appropriate to the type of stream being opened, there is no need to specify its value in the preceding example.

If **open()** fails, the stream will evaluate to false when used in a Boolean expression. You can make use of this fact to confirm that the open operation succeeded by using a statement like this:


```

if(!mystream) {
    cout << "Cannot open file.\n";
    // handle error
}

```

In general, you should always check the result of a call to **open()** before attempting to access the file.

You can also check to see if you have successfully opened a file by using the **is_open()** function, which is a member of **fstream**, **ifstream**, and **ofstream**. It has this prototype.

```
bool is_open();
```

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if **mystream** is currently open:

```

if(!mystream.is_open()) {
    cout << "File is not open.\n";
    // ...
}

```

Although it is entirely proper to open a file by using the **open()** function, most of the time you will not do so because the **ifstream**, **ofstream**, and **fstream** classes have constructor functions that automatically open the file. The constructor functions have the same parameters and defaults as the **open()** function. Therefore, the most common way you will see a file opened is shown in this example:

```
ifstream mystream("myfile"); // open file for input
```

As stated, if for some reason the file cannot be opened, the stream variable will evaluate as false when used in a conditional statement. Therefore, whether you use a constructor function to open the file or an explicit call to **open()**, you will want to confirm that the file has actually been opened by testing the value of the stream.

To close a file, use the member function **close()**. For example, to close the file linked to a stream called **mystream**, use this statement:

```
mystream.close();
```

The **close()** function takes no parameters and returns no value.

You can detect when the end of an input file has been reached by using the **eof()** member function of **ios**. It has this prototype:

C++

`bool eof();`

It returns true when the end of the file has been encountered and false otherwise.

Once a file has been opened, it is very easy to read textual data from it or write formatted, textual data to it. Simply use the `<<` and `>>` operators the same way you do when performing console I/O, except that instead of using `cin` and `cout`, substitute a stream that is linked to a file. In a way, reading and writing files by using `>>` and `<<` is like using C's `fprintf()` and `fscanf()` functions. All information is stored in the file in the same format it would be in if displayed on the screen. Therefore, a file produced by using `<<` is a formatted text file, and any file read by `>>` must be a formatted text file. Typically, files that contain formatted text that you operate on using the `>>` and `<<` operators should be opened for text rather than binary mode. Binary mode is best used on unformatted files, which are described later in this chapter.

EXAMPLES

1. Here is a program that creates an output file, writes information to it, closes the file and opens it again as an input file, and reads in the information:

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

int main()
{
    ofstream fout("test"); // create output file

    if(!fout) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    fout << "Hello!\n";
    fout << 100 << " " << hex << 100 << endl;

    fout.close();
```

```
ifstream fin("test"); // open input file

if(!fin) {
    cout << "Cannot open input file.\n";
    return 1;
}

char str[80];
int i;

fin >> str >> i;
cout << str << ' ' << i << endl;

fin.close();

return 0;
}
```

After you run this program, examine the contents of **test**. It will contain the following:

```
Hello!
100 64
```

As stated earlier, when the << and >> operators are used to perform file I/O, information is formatted exactly as it would appear on the screen.

2. Following is another example of disk I/O. This program reads strings entered at the keyboard and writes them to disk. The program stops when the user enters a \$ as the first character in a string. To use the program, specify the name of the output file on the command line.

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Usage: WRITE <filename>\n";
        return 1;
    }
}
```

C++

```

ofstream out(argv[1]); // output file

if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
}

char str[80];
cout << "Write strings to disk, '$' to stop\n";

do {
    cout << ": ";
    cin >> str;
    out << str << endl;
} while (*str != '$');

out.close();
return 0;
}

```

3. Following is a program that copies a text file and, in the process, converts all spaces into | symbols. Notice how **eof()** is used to check for the end of the input file. Notice also how the input stream **fin** has its **skipws** flag turned off. This prevents leading spaces from being skipped.

```

// Convert spaces to |s.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Usage: CONVERT <input> <output>\n";
        return 1;
    }

    ifstream fin(argv[1]); // open input file
    ofstream fout(argv[2]); // create output file

    if(!fout) {
        cout << "Cannot open output file.\n";
        return 1;
    }
}

```




```

if(!fin) {
    cout << "Cannot open input file.\n";
    return 1;
}

char ch;

fin.unsetf(ios::skipws); // do not skip spaces
while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '|';
    if(!fin.eof()) fout << ch;
}

fin.close();
fout.close();

return 0;
}

```

4. There are a few differences between C++'s original I/O library and the modern Standard C++ library that you should be aware of, especially if you are converting older code. First, in the original I/O library, **open()** allowed a third parameter, which specified the file's protection mode. This parameter defaulted to a normal file. The modern I/O library does not support this parameter.

Second, when you are using the old library to open a stream for input and output using **fstream**, you must explicitly specify both the **ios::in** and the **ios::out** *mode* values. No default value for *mode* is supplied. This applies to both the **fstream** constructor and to its **open()** function. For example, using the old I/O library you must use a call to **open()** as shown here to open a file for input and output:

```

fstream mystream;
mystream.open("test", ios::in | ios::out);

```

In the modern I/O library, an object of type **fstream** automatically opens files for input and output when the *mode* parameter is not supplied.

Finally, in the old I/O system, the *mode* parameter could also include either **ios::nocreate**, which causes the **open()**

function to fail if the file does not already exist, or **ios::noreplace**, which causes the **open()** function to fail if the file does already exist. These values are not supported by Standard C++.

EXERCISES

1. Write a program that will copy a text file: Have this program count the number of characters copied and display this result. Why does the number displayed differ from that shown when you list the output file in the directory?
2. Write a program that writes the following table of information to a file called **phone**:

Isaac Newton, 415 555-3423
Robert Goddard, 213 555-2312
Enrico Fermi, 202 555-1111

3. Write a program that counts the number of words in a file. For simplicity, assume that anything surrounded by whitespace is a word.
 4. What does **is_open()** do?
-

9.3

UNFORMATTED, BINARY I/O

Although formatted text files such as those produced by the preceding examples are useful in a variety of situations, they do not have the flexibility of unformatted, binary files. Unformatted files contain the same binary representation of the data as that used internally by your program rather than the human-readable text that data is translated into by the << and >> operators. Thus, unformatted I/O is also referred to as "raw" I/O. C++ supports a wide range of unformatted file I/O functions. The unformatted functions give you detailed control over how files are written and read.

The lowest-level unformatted I/O functions are **get()** and **put()**. You can read a byte by using **get()** and write a byte by using **put()**. These functions are members of all input and output stream classes, respectively. The **get()** function has many forms, but the most commonly used version is shown here, along with **put()**:

```
istream &get(char &ch);
ostream &put(char ch);
```

The **get()** function reads a single character from the associated stream and puts that value in *ch*. It returns a reference to the stream. If a read is attempted at end-of-file, on return the invoking stream will evaluate to false when used in an expression. The **put()** function writes *ch* to the stream and returns a reference to the stream.

To read and write blocks of data, use the **read()** and **write()** functions, which are also members of the input and output stream classes, respectively. Their prototypes are shown here:

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

The **read()** function reads *num* bytes from the invoking stream and puts them in the buffer pointed to by *buf*. The **write()** function writes *num* bytes to the associated stream from the buffer pointed to by *buf*. The **streamsize** type is some form of integer. An object of type **streamsize** is capable of holding the largest number of bytes that will be transferred in any one I/O operation.

If the end of the file is reached before *num* characters have been read, **read()** simply stops, and the buffer contains as many characters as were available. You can find out how many characters have been read by using the member function **gcount()**, which has this prototype:

```
streamsize gcount();
```

It returns the number of characters read by the last unformatted input operation.

When you are using the unformatted file functions, most often you will open a file for binary rather than text operations. The reason for this is easy to understand: specifying **ios::binary** prevents any character translations from occurring. This is important when the binary representations of data such as integers, **floats**, and pointers

are stored in the file. However, it is perfectly acceptable to use the unformatted functions on a file opened in text mode—as long as that file actually contains only text. But remember, some character translations may occur.

EXAMPLES

1. The next program will display the contents of any file on the screen. It uses the **get()** function.

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Usage: PR <filename>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Cannot open file.\n";
        return 1;
    }

    while(!in.eof()) {
        in.get(ch);
        cout << ch;
    }

    in.close();

    return 0;
}
```

2. This program uses **put()** to write characters to a file until the user enters a dollar sign:

```
#include <iostream>
#include <fstream>
```

```

using namespace std;

int main(int argc, char argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Usage: WRITE <filename>\n";
        return 1;
    }

    ofstream out(argv[1], ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    cout << "Enter a $ to stop\n";
    do {
        cout << ": ";
        cin.get(ch);
        out.put(ch);
    } while (ch!='$');

    out.close();

    return 0;
}

```

Notice that the program uses **get()** to read characters from **cin**. This prevents leading spaces from being discarded.

3. Here is a program that uses **write()** to write a **double** and a string to a file called **test**:

```

#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main()
{
    ofstream out("test", ios::out | ios::binary);

    if(!out) {

```

C++

```

    cout << "Cannot open output file.\n";
    return 1;
}

double num = 100.45;
char str[] = "This is a test";

out.write((char *) &num, sizeof(double));
out.write(str, strlen(str));

out.close();

return 0;
}

```

**Note**

*The type cast to **(char *)** inside the call to **write()** is necessary when outputting a buffer that is not defined as a character array. Because of C++'s strong type checking, a pointer of one type will not automatically be converted into a pointer of another type.*

4. This program uses **read()** to read the file created by the program in Example 3:

```

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

int main()
{
    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    double num;
    char str[80];

    in.read((char *) &num, sizeof(double));
    in.read(str, 14);
    str[14] = '\0'; // null terminate str
}

```



```

cout << num << ' ' << str;

in.close();

return 0;
}

```

As is the case with the program in the preceding example, the type cast inside `read()` is necessary because C++ will not automatically convert a pointer of one type to another.

5. The following program first writes an array of **double** values to a file and then reads them back. It also reports the number of characters read.

```

// Demonstrate gcount().
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    double nums[4] = {1.1, 2.2, 3.3, 4.4 };

    out.write((char *) nums, sizeof(nums));
    out.close();

    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    in.read((char *) &nums, sizeof(nums));

    int i;
    for(i=0; i<4; i++)

```

```
    cout << nums[i] << ' ';

    cout << '\n';

    cout << in.gcount() << " characters read\n";
    in.close();

    return 0;
}
```

EXERCISES

1. Rewrite your answers to Exercises 1 and 3 in the preceding section (Section 9.2) so that they use **get()**, **put()**, **read()**, and/or **write()**. (Use whichever of these functions you deem most appropriate.)
2. Given the following class, write a program that outputs the contents of the class to a file. Create an inserter function for this purpose.

```
class account {
    int custnum;
    char name[80];
    double balance;
public:
    account(int c, char *n, double b)
    {
        custnum = c;
        strcpy(name, n);
        balance = b;
    }
    // create inserter here
};
```

MORE UNFORMATTED I/O FUNCTIONS

In addition to the form shown earlier, there are several different ways in which the **get()** function is overloaded. The prototypes for the three most commonly used overloaded forms are shown here:

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get();
```

The first form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, a newline is found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **get()**. If the newline character is encountered in the input stream, it is *not* extracted. Instead, it remains in the stream until the next input operation.

The second form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **get()**. If the delimiter character is encountered in the input stream, it is *not* extracted. Instead, it remains in the stream until the next input operation.

The third overloaded form of **get()** returns the next character from the stream. It returns EOF if the end of the file is encountered. This form of **get()** is similar to C's **getc()** function.

Another function that performs input is **getline()**. It is a member of each input stream class. Its prototypes are shown here:

```
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

The first form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, a newline character has been found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **getline()**. If the newline

character is encountered in the input stream, it is extracted, but it is not put into *buf*.

The second form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **getline()**. If the delimiter character is encountered in the input stream, it is extracted, but it is not put into *buf*.

As you can see, the two versions of **getline()** are virtually identical to the **get(buf, num)** and **get(buf, num, delim)** versions of **get()**. Both read characters from input and put them into the array pointed to by *buf* until either *num*-1 characters have been read or until the delimiter character or the end of the file is encountered. The difference between **get()** and **getline()** is that **getline()** reads and removes the delimiter from the input stream; **get()** does not.

You can obtain the next character in the input stream without removing it from that stream by using **peek()**. This function is a member of the input stream classes and has this prototype:

```
int peek();
```

It returns the next character in the stream; it returns EOF if the end of the file is encountered.

You can return the last character read from a stream to that stream by using **putback()**, which is a member of the input stream classes. Its prototype is shown here:

```
istream &putback(char c);
```

where *c* is the last character read.

When output is performed, data is not immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to disk before the buffer is full by calling **flush()**. **flush()** is a member of the output stream classes and has this prototype:

```
ostream &flush();
```

Calls to **flush()** might be warranted when a program is going to be used in adverse environments (in situations where power outages occur frequently, for example).

EXAMPLES

1. As you know, when you use `>>` to read a string, it stops reading when the first whitespace character is encountered. This makes it useless for reading a string containing spaces. However, you can overcome this problem by using **getline()**, as this program illustrates:

```
// Use getline() to read a string that contains spaces.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char str[80];

    cout << "Enter your name: ";
    cin.getline(str, 79);

    cout << str << '\n';

    return 0;
}
```

In this program, the delimiter used by **getline()** is the newline. This makes **getline()** act much like the standard **gets()** function.

2. In real programming situations, the functions **peek()** and **putback()** are especially useful because they let you more easily handle situations in which you do not know what type of information is being input at any point in time. The following program gives the flavor of this. It reads either strings or integers from a file. The strings and integers can occur in any order.

```
// Demonstrate peek().
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main()
{
    char ch;
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    char str[80], *p;

    out << 123 << "this is a test" << 23;
    out << "Hello there!" << 99 << "sdf" << endl;
    out.close();

    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    do {
        p = str;
        ch = in.peek(); // see what type of char is next
        if(isdigit(ch)) {
            while(isdigit(*p=in.get())) p++; // read integer
            in.putback(*p); // return char to stream
            *p = '\0'; // null-terminate the string
            cout << "Integer: " << atoi(str);
        }
        else if(isalpha(ch)) { // read a string
            while(isalpha(*p=in.get())) p++;
            in.putback(*p); // return char to stream
            *p = '\0'; // null-terminate the string
            cout << "String: " << str;
        }
        else in.get(); // ignore
```



```

        cout << '\n';
    } while(!in.eof());

    in.close();
    return 0;
}
    
```

EXERCISES

1. Rewrite the program in Example 1 so it uses **get()** instead of **getline()**. Does the program function differently?
2. Write a program that reads a text file one line at a time and displays each line on the screen. Use **getline()**.
3. On your own, think about why there may be cases in which a call to **flush()** is appropriate.

9.5

RANDOM ACCESS

In C++'s I/O system, you perform random access by using the **seekg()** and **seekp()** functions, which are members of the input and output stream classes, respectively. Their most common forms are shown here:

```

istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
    
```

Here **off_type** is an integer type defined by **ios** that is capable of containing the largest valid value that *offset* can have. **seekdir** is an enumeration defined by **ios** that has these values:

Value

ios::beg
 ios::cur
 ios::end

Meaning

Seek from beginning
 Seek from current location
 Seek from end

The C++ I/O system manages two pointers associated with a file. One is the *get pointer*, which specifies where in the file the next input operation will occur. The other is the *put pointer*, which specifies where in the file the next output operation will occur. Each time an input or output operation takes place, the appropriate pointer is automatically sequentially advanced. However, by using the `seekg()` and `seekp()` functions, it is possible to access the file in a nonsequential fashion.

The `seekg()` function moves the associated file's current *get pointer* *offset* number of bytes from the specified *origin*. The `seekp()` function moves the associated file's current *put pointer* *offset* number of bytes from the specified *origin*.

In general, files that will be accessed via `seekg()` and `seekp()` should be opened for binary file operations. This prevents character translations from occurring which may affect the apparent position of an item within a file.

You can determine the current position of each file pointer by using these member functions:

```
pos_type tellg();  
pos_type tellp();
```

Here `pos_type` is an integer type defined by `ios` that is capable of holding the largest value that defines a file position.

There are overloaded versions of `seekg()` and `seekp()` that move the file pointers to the location specified by the return values of `tellg()` and `tellp()`. Their prototypes are shown here.

```
istream &seekg(pos_type position);  
ostream &seekp(pos_type position);
```

EXAMPLES

1. The following program demonstrates the `seekp()` function. It allows you to change a specific character in a file. Specify a file name on the command line, followed by the number of the byte in the file you want to change, followed by the new character. Notice that the file is opened for read/write operations.

```
#include <iostream>  
#include <fstream>
```

```

#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=4) {
        cout << "Usage: CHANGE <filename> <byte> <char>\n";
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    out.seekp(atoi(argv[2]), ios::beg);

    out.put(*argv[3]);
    out.close();

    return 0;
}

```

2. The next program uses **seekg()** to position the get pointer into the middle of a file and then displays the contents of that file from that point. The name of the file and the location to begin reading from are specified on the command line.

```

// Demonstrate seekg().
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Usage: LOCATE <filename> <loc>\n";
        return 1;
    }
}

```


▼
C++

```
ifstream in(argv[1], ios::in | ios::binary);

if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
}

in.seekg(atoi(argv[2]), ios::beg);

while(!in.eof()) {
    in.get(ch);
    cout << ch;
}

in.close();

return 0;
}
```

EXERCISES

1. Write a program that displays a text file backwards. Hint: Think about this before creating your program. The solution is easier than you might imagine.
2. Write a program that swaps each character pair in a text file. For example, if the file contains "1234", then after the program is run, the file will contain "2143". (For simplicity, you may assume that the file contains an even number of characters.)

CHECKING THE I/O STATUS

The C++ I/O system maintains status information about the outcome of each I/O operation. The current status of an I/O stream is described in an object of type **istate**, which is an enumeration defined by **ios** that includes these members:

<u>Name</u>	<u>Meaning</u>
goodbit	No errors occurred.
eofbit	End-of-file has been encountered.
failbit	A nonfatal I/O error has occurred.
badbit	A fatal I/O error has occurred.

For older compilers, the I/O status flags are held in an **int** rather than an object of type **iostate**.

There are two ways in which you can obtain I/O status information. First, you can call the **rdstate()** function, which is a member of **ios**. It has this prototype:

```
iostate rdstate();
```

It returns the current status of the error flags. As you can probably guess from looking at the preceding list of flags, **rdstate()** returns **goodbit** when no error has occurred. Otherwise, an error flag is returned.

The other way you can determine whether an error has occurred is by using one or more of these **ios** member functions.

```
bool bad();
bool eof();
bool fail();
bool good();
```

The **eof()** function was discussed earlier. The **bad()** function returns true if **badbit** is set. The **fail()** function returns true if **failbit** is set. The **good()** function returns true if there are no errors. Otherwise they return false.

Once an error has occurred, it might need to be cleared before your program continues. To do this, use the **ios** member function **clear()**, whose prototype is shown here:

```
void clear(iostate flags = ios::goodbit);
```

If **flags** is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set **flags** to the settings you desire.

EXAMPLES

1. The following program illustrates `rdstate()`. It displays the contents of a text file. If an error occurs, the function reports it by using `checkstatus()`.

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

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Usage: DISPLAY <filename>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    char c;
    while(in.get(c)) {
        cout << c;
        checkstatus(in);
    }

    checkstatus(in); // check final status
    in.close();

    return 0;
}

void checkstatus(ifstream &in)
{
    ios::iostate i;

    i = in.rdstate();

    if(i & ios::eofbit)
        cout << "EOF encountered\n";
}
```



```

else if(i & ios::failbit)
    cout << "Non-Fatal I/O error\n";
else if(i & ios::badbit)
    cout << "Fatal I/O error\n";
}

```

The preceding program will always report at least one "error." After the **while** loop ends, the final call to **checkstatus()** reports, as expected, that an EOF has been encountered.

2. This program displays a text file. It uses **good()** to detect a file error:

```

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "PR: <filename>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    while(!in.eof()) {
        in.get(ch);
        // check for error
        if(!in.good() && !in.eof()) {
            cout << "I/O Error...terminating\n";
            return 1;
        }
        cout << ch;
    }

    in.close();

    return 0;
}

```

EXERCISE

1. Add error checking to your answers to the exercises from the preceding section.

9.7**CUSTOMIZED I/O AND FILES**

In the preceding chapter, you learned how to overload the insertion and extraction operators relative to your own classes. In that chapter, only console I/O was performed. However, because all C++ streams are the same, the same overloaded inserter function, for example, can be used to output to the screen or to a file with no changes whatsoever. This is one of the most important and useful features of C++'s approach to I/O.

As stated in the previous chapter, overloaded inserters and extractors, as well as I/O manipulators, can be used with any stream as long as they are written in a general manner. If you "hard-code" a specific stream into an I/O function, its use is, of course, limited to only that stream. This is why you were urged to generalize your I/O functions whenever possible.

EXAMPLES

1. In the following program, the **coord** class overloads the << and >> operators. Notice that you can use the operator functions to write both to the screen and to a file.

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

class coord {
    int x, y;
public:
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
    friend istream &operator>>(istream &stream, coord &ob);
};
```

```
ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ' ' << ob.y << '\n';

    return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
    stream >> ob.x >> ob.y;

    return stream;
}

int main()
{
    coord o1(1, 2), o2(3, 4);
    ofstream out("test");

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    out << o1 << o2;

    out.close();

    ifstream in("test");

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    coord o3(0, 0), o4(0, 0);
    in >> o3 >> o4;

    cout << o3 << o4;

    in.close();

    return 0;
}
```


2. All of the I/O manipulators can be used with files. For example, in this reworked version of a program presented earlier in this chapter, the same manipulator that writes to the screen will also write to a file:

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

// Attention:
ostream &atn(ostream &stream)
{
    stream << "Attention: ";
    return stream;
}

// Please note:
ostream &note(ostream &stream)
{
    stream << "Please Note: ";
    return stream;
}

int main()
{
    ofstream out("test");

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    // write to screen
    cout << atn << "High voltage circuit\n";
    cout << note << "Turn off all lights\n";

    // write to file
    out << atn << "High voltage circuit\n";
    out << note << "Turn off all lights\n";

    out.close();

    return 0;
}
```

EXERCISE

1. On your own, experiment with the programs from the preceding chapter, trying each on a disk file.

SKILLS CHECK

At this point you should be able to perform the following exercises and answer the questions.

1. Create an output manipulator that outputs three tabs and then sets the field width to 20. Demonstrate that your manipulator works.
2. Create an input manipulator that reads and discards all nonalphabetical characters. When the first alphabetical character is read, have the manipulator return it to the input stream and return. Call this manipulator **findalpha**.
3. Write a program that copies a text file. In the process, reverse the case of all letters.
4. Write a program that reads a text file and then reports the number of times each letter in the alphabet occurs in the file.
5. If you have not done so, add complete error checking to your solutions to Exercises 3 and 4 above.
6. What function positions the get pointer? What function positions the put pointer?



Cumulative
Skills Check

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Following is a reworked version of the **inventory** class presented in the preceding chapter. Write a program that fills in the functions **store()** and **retrieve()**. Next, create a small inventory file on disk containing a few entries. Then, using random I/O, allow the user to display the information about any item by specifying its record number.

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

#define SIZE 40

class inventory {
    char item[SIZE]; // name of item
    int onhand; // number on hand
    double cost; // cost of item
public:
    inventory(char *i, int o, double c)
    {
        strcpy(item, i);
        onhand = o;
        cost = c;
    }
    void store(fstream &stream);
    void retrieve(fstream &stream);
    friend ostream &operator<<(ostream &stream, inventory ob);
    friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ": " << ob.onhand;
    stream << " on hand at $" << ob.cost << '\n';
}
```



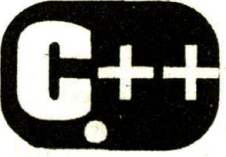
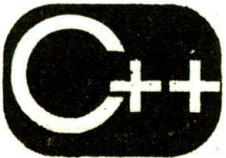
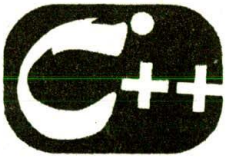
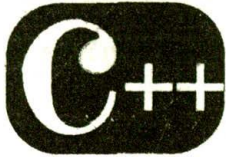
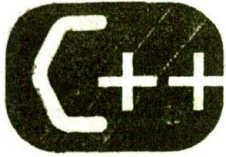
```
    return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
    cout << "Enter item name: ";
    stream >> ob.item;
    cout << "Enter number on hand: ";
    stream >> ob.onhand;
    cout << "Enter cost: ";
    stream >> ob.cost;

    return stream;
}
```

2. As a special challenge, on your own, create a **stack** class for characters that stores them in a disk file rather than in an array in memory.





10

Virtual Functions

chapter objectives

- 10.1** Pointers to derived classes
- 10.2** Introduction to virtual functions
- 10.3** More about virtual functions
- 10.4** Applying polymorphism



C++

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Create a manipulator that causes numbers to be displayed in scientific notation, using a capital E.
2. Write a program that copies a text file. During the copy process, convert all tabs into the correct number of spaces.
3. Write a program that searches a text file for a word specified on the command line. Have the program display how many times the specified word is found. For simplicity, assume that anything surrounded by whitespace is a word.
4. Show the statement that sets the put pointer to the 234th byte in a file linked to a stream called **out**.
5. What functions report status information about the C++ I/O system?
6. Give one advantage of using the C++ I/O functions instead of the C-like I/O system.



THIS chapter examines another important aspect of C++: the virtual function. What makes virtual functions important is that they are used to support run-time polymorphism. Polymorphism is supported by C++ in two ways. First, it is supported at compile time, through the use of overloaded operators and functions. Second, it is supported at run time, through the use of virtual functions. As you will learn, run-time polymorphism provides the greatest flexibility.

At the foundation of virtual functions and run-time polymorphism are pointers to derived classes. For this reason this chapter begins with a discussion of such pointers.

10.1

PPOINTERS TO DERIVED CLASSES

Although Chapter 4 discussed C++ pointers at some length, one special aspect was deferred until now because it relates specifically to virtual functions. The feature is this: A pointer declared as a pointer to a base class can also be used to point to any class derived from that base. For example, assume two classes called **base** and **derived**, where **derived** inherits **base**. Given this situation, the following statements are correct:

```
base *p; // base class pointer

base base_ob; // object of type base
derived derived_ob; // object of type derived

// p can, of course, point to base objects
p = &base_ob; // p points to base object

// p can also point to derived objects without error
p = &derived_ob; // p points to derived object
```

As the comments suggest, a base pointer can point to an object of any class derived from that base without generating a type mismatch error.

Although you can use a base pointer to point to a derived object, you can access only those members of the derived object that were inherited from the base. This is because the base pointer has knowledge only of the base class. It knows nothing about the members added by the derived class.

While it is permissible for a base pointer to point to a derived object, the reverse is not true. A pointer of the derived type cannot be used to access an object of the base class. (A type cast can be used to overcome this restriction, but its use is not recommended practice.)

One final point: Remember that pointer arithmetic is relative to the data type the pointer is declared as pointing to. Thus, if you point a base pointer to a derived object and then increment that pointer, it will not be pointing to the next derived object. It will be pointing to (what it thinks is) the next base object. Be careful about this.

EXAMPLE

1. Here is a short program that illustrates how a base class pointer can be used to access a derived class:

```
// Demonstrate pointer to derived class.
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int i) { x = i; }
    int getx() { return x; }
};

class derived : public base {
    int y;
public:
    void sety(int i) { y = i; }
    int gety() { return y; }
};

int main()
{
    base *p; // pointer to base type
    base b_ob; // object of base
    derived d_ob; // object of derived

    // use p to access base object
    p = &b_ob;
    p->setx(10); // access base object
    cout << "Base object x: " << p->getx() << '\n';

    // use p to access derived object
    p = &d_ob; // point to derived object
    p->setx(99); // access derived object

    // can't use p to set y, so do it directly
    d_ob.sety(88);
    cout << "Derived object x: " << p->getx() << '\n';
    cout << "Derived object y: " << d_ob.gety() << '\n';

    return 0;
}
```

Sub IR 032

Aside from illustrating pointers to derived classes, there is no value in using a base class pointer in the way shown in this example. However, in the next section you will see why base class pointers to derived objects are so important.

EXERCISE

1. On your own, try the preceding example and experiment with it. For example, try declaring a derived pointer and having it access an object of the base class.
-

10.2 INTRODUCTION TO VIRTUAL FUNCTIONS

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function relative to the derived class. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*. When a virtual function is redefined by a derived class, the keyword **virtual** is not needed.

A virtual function can be called just like any other member function. However, what makes a virtual function interesting—and capable of supporting run-time polymorphism—is what happens when a virtual function is called through a pointer. From the preceding section you know that a base class pointer can be used to point to a derived class object. When a base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function

will be executed based upon the *type of object being pointed to* by the pointer. And, this determination is made at *run time*. Put differently, it is the type of the object pointed to at the time when the call occurs that determines which version of the virtual function will be executed. Therefore, if two or more different classes are derived from a base class that contains a virtual function, then when different objects are pointed to by a base pointer, different versions of the virtual function are executed. This process is the way that run-time polymorphism is achieved. In fact, a class that contains a virtual function is referred to as a polymorphic class.

EXAMPLES

1. Here is a short example that uses a virtual function:

```
// A simple example using a virtual function.
#include <iostream>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Using base version of func(): ";
        cout << i << '\n';
    }
};

class derived1 : public base {
public:
    derived1(int x) : base(x) {}
    void func()
    {
        cout << "Using derived1's version of func(): ";
        cout << i*i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) {}
};
```



```

void func()
{
    cout << "Using derived2's version of func(): ";
    cout << i+i << '\n';
}
};

int main()
{
    base *p;
    base ob(10);
    derived1 d_ob1(10);
    derived2 d_ob2(10);

    p = &ob;
    p->func(); // use base's func()

    p = &d_ob1;
    p->func(); // use derived1's func()

    p = &d_ob2;
    p->func(); // use derived2's func()

    return 0;
}

```

base

This program displays the following output:

```

Using base version of func(): 10
Using derived1's version of func(): 100
Using derived2's version of func(): 20

```

The redefinition of a virtual function inside a derived class might, at first, seem somewhat similar to function overloading. However, the two processes are distinctly different. First, an overloaded function must differ in type and/or number of parameters, while a redefined virtual function must have precisely the same type and number of parameters and the same return type. (In fact, if you change either the number or type of parameters when redefining a virtual function, it simply becomes an overloaded function and its virtual nature is lost.) Further, virtual functions must be class members. This is not the case for overloaded functions. Also, while destructor functions can be virtual, constructors cannot. Because of the

differences between overloaded functions and redefined virtual functions, the term *overriding* is used to describe virtual function redefinition.

As you can see, the example program creates three classes. The **base** class defines the virtual function **func()**. This class is then inherited by both **derived1** and **derived2**. Each of these classes overrides **func()** with its individual implementation. Inside **main()**, the base class pointer **p** is declared along with objects of type **base**, **derived1**, and **derived2**. First, **p** is assigned the address of **ob** (an object of type **base**). When **func()** is called by using **p**, it is the version in **base** that is used. Next, **p** is assigned the address of **d_ob1** and **func()** is called again. Because it is the type of the object pointed to that determines which virtual function will be called, this time it is the overridden version in **derived1** that is executed. Finally, **p** is assigned the address of **d_ob2** and **func()** is called again. This time, it is the version of **func()** defined inside **derived2** that is executed.

The key points to understand from the preceding example are that the type of the object being pointed to determines which version of an overridden virtual function will be executed when accessed via a base class pointer, and that this decision is made at run time.

2. Virtual functions are hierarchical in order of inheritance. Further, when a derived class does *not* override a virtual function, the function defined within its base class is used. For example, here is a slightly different version of the preceding program:

```
// Virtual functions are hierarchical.
#include <iostream>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Using base version of func(): ";
        cout << i << '\n';
    }
};
```



```
class derived1 : public base {
public:
    derived1(int x) : base(x) {}
    void func()
    {
        cout << "Using derived1's version of func(): ";
        cout << i*i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) {}
    // derived2 does not override func()
};

int main()
{
    base *p;
    base ob(10);
    derived1 d_ob1(10);
    derived2 d_ob2(10);
    p = &ob;
    p->func(); // use base's func()

    p = &d_ob1;
    p->func(); // use derived1's func()
    p = &d_ob2;
    p->func(); // use base's func()

    return 0;
}
```

This program displays the following output:

```
Using base version of func(): 10
Using derived1's version of func(): 100
Using base version of func(): 10
```

In this version, **derived2** does not override **func()**. When **p** is assigned **d_ob2** and **func()** is called, **base's** version is used because it is next up in the class hierarchy. In general, when a

derived class does not override a virtual function, the base class' version is used.

3. The next example shows how a virtual function can respond to random events that occur at run time. This program selects between **d_ob1** and **d_ob2** based upon the value returned by the standard random number generator **rand()**. Keep in mind that the version of **func()** executed is resolved at run time. (Indeed, it is impossible to resolve the calls to **func()** at compile time.)

```
/* This example illustrates how a virtual function
   can be used to respond to random events occurring
   at run time.
```

```
*/
```

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
class base {
```

```
public:
```

```
    int i;
```

```
    base(int x) { i = x; }
```

```
    virtual void func()
```

```
    {
```

```
        cout << "Using base version of func(): ";
```

```
        cout << i << '\n';
```

```
    }
```

```
};
```

```
class derived1 : public base {
```

```
public:
```

```
    derived1(int x) : base(x) {}
```

```
    void func()
```

```
    {
```

```
        cout << "Using derived1's version of func(): ";
```

```
        cout << i*i << '\n';
```

```
    }
```

```
};
```

```
class derived2 : public base {
```

```
public:
```

```
    derived2(int x) : base(x) {}
```

```
    void func()
```

```
    {
```



```

cout << "Using derived2's version of func(): ";
cout << i+i << '\n';
}
};

int main()
{
    base *p;
    derived1 d_ob1(10);
    derived2 d_ob2(10);
    int i, j;

    for(i=0; i<10; i++) {
        j = rand();
        if((j%2)) p = &d_ob1; // if odd use d_ob1
        else p = &d_ob2; // if even use d_ob2
        p->func(); // call appropriate function
    }

    return 0;
}

```

4. Here is a more practical example of how a virtual function can be used. This program creates a generic base class called **area** that holds two dimensions of a figure. It also declares a virtual function called **getarea()** that, when overridden by derived classes, returns the area of the type of figure defined by the derived class. In this case, the declaration of **getarea()** inside the base class determines the nature of the interface. The actual implementation is left to the classes that inherit it. In this example, the area of a triangle and a rectangle are computed.

```

// Use virtual function to define interface.
#include <iostream>
using namespace std;

class area {
    double dim1, dim2; // dimensions of figure
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
}

```

```
void getdim(double &d1, double &d2)
{
    d1 = dim1;
    d2 = dim2;
}
virtual double getarea()
{
    cout << "You must override this function\n";
    return 0.0;
}
};

class rectangle : public area {
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle : public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    rectangle r;
    triangle t;

    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);

    p = &r;
    cout << "Rectangle has area: " << p->getarea() << '\n';
}
```

```

p = &t;
cout << "Triangle has area: " << p->getarea() << '\n';
return 0;
}

```

Notice that the definition of `getarea()` inside `area` is just a placeholder and performs no real function. Because `area` is not linked to any specific type of figure, there is no meaningful definition that can be given to `getarea()` inside `area`. In fact, `getarea()` must be overridden by a derived class in order to be useful. In the next section, you will see a way to enforce this.

EXERCISES

1. Write a program that creates a base class called `num`. Have this class hold an integer value and contain a virtual function called `shownum()`. Create two derived classes called `outhex` and `outoct` that inherit `num`. Have the derived classes override `shownum()` so that it displays the value in hexadecimal and octal, respectively.
2. Write a program that creates a base class called `dist` that stores the distance between two points in a `double` variable. In `dist`, create a virtual function called `trav_time()` that outputs the time it takes to travel that distance, assuming that the distance is in miles and the speed is 60 miles per hour. In a derived class called `metric`, override `trav_time()` so that it outputs the travel time assuming that the distance is in kilometers and the speed is 100 kilometers per hour.

10.3 MORE ABOUT VIRTUAL FUNCTIONS

As Example 4 from the preceding section illustrates, sometimes when a virtual function is declared in the base class there is no meaningful operation for it to perform. This situation is common because often a

base class does not define a complete class by itself. Instead, it simply supplies a core set of member functions and variables to which the derived class supplies the remainder. When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class *must* override this function. To ensure that this will occur, C++ supports *pure virtual functions*.

A pure virtual function has no definition relative to the base class. Only the function's prototype is included. To make a pure virtual function, use this general form:

```
virtual type func-name( parameter-list ) = 0;
```

The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does not, a compile-time error results. Thus, making a virtual function pure is a way to guarantee that a derived class will provide its own redefinition.

When a class contains at least one pure virtual function, it is referred to as an *abstract class*. Since an abstract class contains at least one function for which no body exists, it is, technically, an incomplete type, and no objects of that class can be created. Thus, abstract classes exist only to be inherited. They are neither intended nor able to stand alone. It is important to understand, however, that you can still create a pointer to an abstract class, since it is through the use of base class pointers that run-time polymorphism is achieved. (It is also permissible to have a reference to an abstract class.)

When a virtual function is inherited, so is its virtual nature. This means that when a derived class inherits a virtual function from a base class and then the derived class is used as a base for yet another derived class, the virtual function can be overridden by the final derived class (as well as the first derived class). For example, if base class B contains a virtual function called `f()`, and D1 inherits B and D2 inherits D1, both D1 and D2 can override `f()` relative to their respective classes.

EXAMPLES

1. Here is an improved version of the program shown in Example 4 in the preceding section. In this version, the function `getarea()` is declared as pure in the base class `area`.

```
// Create an abstract class.
#include <iostream>
using namespace std;

class area {
    double dim1, dim2; // dimensions of figure
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double &d2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea() = 0; // pure virtual function
};

class rectangle : public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle : public area {
```

C++

```
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    rectangle r;
    triangle t;

    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);

    p = &r;
    cout << "Rectangle has area: " << p->getarea() << '\n';

    p = &t;
    cout << "Triangle has area: " << p->getarea() << '\n';

    return 0;
}
```

Now that **getarea()** is pure, it ensures that each derived class will override it.

2. The following program illustrates how a function's virtual nature is preserved when it is inherited:

```
// Virtual functions retain their virtual nature when inherited.
#include <iostream>
using namespace std;

class base {
public:
    virtual void func()
    {
        cout << "Using base version of func()\n";
    }
}
```



```

class derived1 : public base {
public:
    void func()
    {
        cout << "Using derived1's version of func()\n";
    }
};

// Derived2 inherits derived1.
class derived2 : public derived1 {
public:
    void func()
    {
        cout << "Using derived2's version of func()\n";
    }
};

int main()
{
    base *p;
    base ob;
    derived1 d_ob1;
    derived2 d_ob2;

    p = &ob;
    p->func(); // use base's func()

    p = &d_ob1;
    p->func(); // use derived1's func()

    p = &d_ob2;
    p->func(); // use derived2's func()

    return 0;
}

```

In this program, the virtual function `func()` is first inherited by **derived1**, which overrides it relative to itself. Next, **derived2** inherits **derived1**. In **derived2**, `func()` is again overridden.

Because virtual functions are hierarchical, if **derived2** did not override `func()`, when `d_ob2` was accessed, **derived1**'s `func()` would have been used. If neither **derived1** nor **derived2** had overridden `func()`, all references to it would have been routed to the one defined in **base**.

EXERCISES

1. On your own, experiment with the two example programs. Specifically, try creating an object by using **area** from Example 1 and observe the error message. In Example 2, try removing the redefinition of **func()** within **derived2**. Confirm that, indeed, the version inside **derived1** is used.
2. Why can't an object be created by using an abstract class?
3. In Example 2, what happens if you remove only the redefinition of **func()** inside **derived1**? Does the program still compile and run? If so, why?

10.3

APPLYING POLYMORPHISM

Now that you know how to use a virtual function to achieve run-time polymorphism, it is time to consider how and why to use it. As has been stated many times in this book, polymorphism is the process by which a common interface is applied to two or more similar (but technically different) situations, thus implementing the "one interface, multiple methods" philosophy. Polymorphism is important because it can greatly simplify complex systems. A single, well-defined interface is used to access a number of different but related actions, and artificial complexity is removed. In essence, polymorphism allows the logical relationship of similar actions to become apparent; thus, the program is easier to understand and maintain. When related actions are accessed through a common interface, you have less to remember.

There are two terms that are often linked to OOP in general and to C++ specifically. They are *early binding* and *late binding*. It is important that you know what they mean. Early binding essentially refers to those events that can be known at compile time. Specifically, it refers to those function calls that can be resolved during compilation. Early bound entities include "normal" functions, overloaded functions, and nonvirtual member and friend functions. When these types of functions are compiled, all address information necessary to call them is known at compile time. The main advantage of early binding (and the reason that it is so widely used) is that it is very efficient. Calls to

functions bound at compile time are the fastest types of function calls. The main disadvantage is lack of flexibility.

Late binding refers to events that must occur at run time. A late bound function call is one in which the address of the function to be called is not known until the program runs. In C++, a virtual function is a late bound object. When a virtual function is accessed via a base class pointer, the program must determine at run time what type of object is being pointed to and then select which version of the overridden function to execute. The main advantage of late binding is flexibility at run time. Your program is free to respond to random events without having to contain large amounts of "contingency code." Its primary disadvantage is that there is more overhead associated with a function call. This generally makes such calls slower than those that occur with early binding.

Because of the potential efficiency trade-offs, you must decide when it is appropriate to use early binding and when to use late binding.

EXAMPLES

1. Here is a program that illustrates "one interface, multiple methods." It defines an abstract list class for integer values. The interface to the list is defined by the pure virtual functions **store()** and **retrieve()**. To store a value, call the **store()** function. To retrieve a value from the list, call **retrieve()**. The base class **list** does not define any default methods for these actions. Instead, each derived class defines exactly what type of list will be maintained. In the program, two types of lists are implemented: a queue and a stack. Although the two lists operate completely differently, each is accessed using the same interface. You should study this program carefully.

```
// Demonstrate virtual functions.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class list {
public:
    list *head; // pointer to start of list
    list *tail; // pointer to end of list
```


▼ C++

```
list *next; // pointer to next item
int num; // value to be stored
```

```
list() { head = tail = next = NULL; }
virtual void store(int i) = 0;
virtual int retrieve() = 0;
};
```

```
// Create a queue-type list.
```

```
class queue : public list {
public:
    void store(int i);
    int retrieve();
};
```

```
void queue::store(int i)
```

```
{
    list *item;

    item = new queue;
    if(!item) {
        cout << "Allocation error.\n";
        exit(1);
    }
    item->num = i;

    // put on end of list
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
    if(!head) head = tail;
}
```

```
int queue::retrieve()
```

```
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // remove from start of list
    i = head->num;
```

```
p = head;
head = head->next;
delete p;

return i;

// Create a stack-type list.
class stack : public list {
public:
    void store(int i);
    int retrieve();
};

void stack::store(int i)
{
    list *item;

    item = new stack;
    if(!item) {
        cout << "Allocation error.\n";
        exit(1);
    }
    item->num = i;

    // put on front of list for stack-like operation
    if(head) item->next = head;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // remove from start of list
    i = head->num;
    p = head;
    head = head->next;
```

C++

```
delete p;

return i;
}

int main()
{
    list *p;

    // demonstrate queue
    queue q_ob;
    p = &q_ob; // point to queue

    p->store(1);
    p->store(2);
    p->store(3);

    cout << "Queue: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();

    cout << '\n';

    // demonstrate stack
    stack s_ob;
    p = &s_ob; // point to stack

    p->store(1);
    p->store(2);
    p->store(3);

    cout << "Stack: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();

    cout << '\n';

    return 0;
}
```

2. The **main()** function in the list program just shown simply illustrates that the list classes do, indeed, work. However, to

begin to see why run-time polymorphism is so powerful, try using this **main()** instead:

```
int main()
{
    list *p;
    stack s_ob;
    queue q_ob;
    char ch;
    int i;

    for(i=0; i<10; i++) {
        cout << "Stack or Queue? (S/Q): ";
        cin >> ch;
        ch = tolower(ch);
        if(ch=='q') p = &q_ob;
        else p = &s_ob;
        p->store(i);
    }

    cout << "Enter T to terminate\n";
    for(;;) {
        cout << "Remove from Stack or Queue? (S/Q): ";
        cin >> ch;
        ch = tolower(ch);
        if(ch=='t') break;
        if(ch=='q') p = &q_ob;
        else p = &s_ob;
        cout << p->retrieve() << '\n';
    }

    cout << '\n';

    return 0;
}
```

This **main()** illustrates how random events that occur at run time can be easily handled by using virtual functions and run-time polymorphism. The program executes a **for** loop running from 0 to 9. Each iteration through the loop, you are asked to choose into which type of list—the stack or the queue—you want to put a value. According to your answer, the base pointer **p** is set to point to the correct object and the current value of **i** is stored. Once the loop is finished, another

loop begins that prompts you to indicate from which list to remove a value. Once again, it is your response that determines which list is selected.

While this example is trivial, you should be able to see how run-time polymorphism can simplify a program that must respond to random events. For instance, the Windows operating system interfaces to a program by sending it messages. As far as the program is concerned, these messages are generated at random, and your program must respond to each one as it is received. One way to respond to these messages is through the use of virtual functions.

EXERCISES

1. Add another type of list to the program in Example 1. Have this version maintain a sorted list (in ascending order). Call this list **sorted**.
 2. On your own, think about ways in which you can apply run-time polymorphism to simplify the solutions to certain types of problems.
-

SKILLS CHECK



At this point you should be able to perform the following exercises and answer the questions.

1. What is a virtual function?
2. What types of functions cannot be made virtual?
3. How does a virtual function help achieve run-time polymorphism? Be specific.

4. What is a pure virtual function?
5. What is an abstract class? What is a polymorphic class?
6. Is the following fragment correct? If not, why not?

```
class base {
public:
    virtual int f(int a) = 0;
    // ...
};

class derived : public base {
public:
    int f(int a, int b) { return a*b; }
    // ...
};
```

7. Is the virtual quality inherited?
8. On your own, experiment with virtual functions at this time. This is an important concept and you should master the technique.



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Enhance the list example from Section 10.4, Example 1, so that it overloads the `+` and `--` operators. Have the `+` store an element and the `--` retrieve an element.
2. How do virtual functions differ from overloaded functions?
3. On your own, reexamine some of the function overloading examples presented earlier in this book. Determine which can be converted to virtual functions. Also, think about ways in which a virtual function can solve some of your own programming problems.

