```
}
/*End of multiInheritAmbiguityResolve01.cpp*/
```

Output

show() function of class A called
show() function of class B called

Listing 5.27 Ambiguity resolution by using scope resolution operator

This ambiguity can also be resolved by overriding the multiple inherited base class member as shown in Listing 5.28.

```
/*Beginning of multiInheritAmbiguityResolve02.cpp*/
#include<iostream.h>
class A
{
  public:
    void show()
    {
      cout<<"show() function of class A called\n";
    }
};

class B
{
  public:
    void show()
    {
      cout<<"show() function of class B called\n";
    }
};
class C : public A, public B
{
  public:
    void show()      //override both of the inherited
                     //functions
    {
      cout<<"show() function of class C called\n";
    }
};
void main()
{
  C C1;
  C1.show();    //OK: C::show() called
```

```
}
/*End of multiInheritAmbiguityResolve02.cpp*/
```

Output

show() function of class C called

Listing 5.28 Ambiguity resolution by overriding

We can still call the 'show()' functions of classes A and B with respect to an object of class C by using the scope resolution operator. Let us now replace the 'main()' function with the following one and see the difference.

```
void main()
{
  C1.A::show();
  C1.B::show();
}
```

Output

show() function of class A called
show() function of class B called

Listing 5.29 Calling overridden members by scope resolution operator

- **Diamond-Shaped Inheritance:** Ambiguities can also arise if two or more base classes in turn inherit from a common base class. This is known as diamond-shaped inheritance (see Listing 5.30).

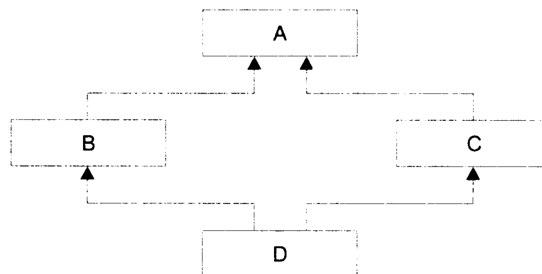

Diagram 5.6 Base classes inheriting from a common base class

```
/*Beginning of multiInheritAmbiguity02.cpp*/
class A
{
  public:
    void show();
};

class B : public A
{};

class C : public A
{};

class D : public B, public C
{};

void main()
{
  D D1;
  D1.show();    //ERROR: ambiguous call to show()
}
/*End of multiInheritAmbiguity02.cpp*/
```

Listing 5.30 Diamond-shaped inheritance

The two previous solutions—using scope resolution operator and overriding—are applicable here also. Nevertheless, a third solution is also available—that of declaring the top base class to be virtual. The ambiguity disappears if we declare class A to be a `virtual` base class of classes B and C. This is demonstrated by the following lines of code.

```
class B : virtual public A
{};
class C : virtual public A
{};
```

Now the call to the 'show()' function with respect to an object of class D is no longer ambiguous.

Multi-level Inheritance

When a class inherits from a derived class, it is known as multi-level inheritance. In other words, a class derives from a class that is in turn derived from another class. Diagram 5.7 depicts multi-level inheritance.

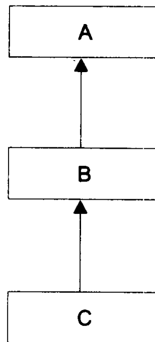

Diagram 5.7 Multi-level inheritance

In Diagram 5.7, class C is derived from class B, which is in turn derived from class A. The syntax for implementing this derivation is as follows.

```
/*Beginning of multiInherit.cpp*/
#include<iostream.h>

class A
{
  public:
    void fA()
    {
      cout<<"fA() called\n";
    }
};

class B : public A //B derived from A
{
  public:
    void fB()
    {
      cout<<"fB() called\n";
    }
};
class C : public B //C derived from B, B derived from A
{
  public:
    void fC()
    {
      cout<<"fC() called\n";
    }
};
```

```
void main()
{
  C C1;
  C1.fA();
  C1.fB();
  C1.fC();
}

/*End of multiInherit.cpp*/
```

Output
fA() called
fB() called
fC() called

Listing 5.31 Multi-level inheritance

Multi-level inheritance can be extended to any level.

Multi-level inheritance is commonly used to implement successive refinement of a data type. For instance, 'Animal' is a more generic class. 'Mammal' is a type of 'Animal'. 'Man' is a type of 'Mammal'.

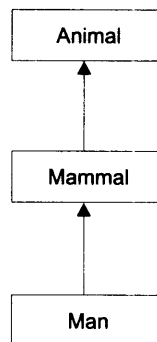

Diagram 5.8 Example of using multi-level inheritance to successively refine a data type

In this example, the data type 'Animal' is successively refined to 'Mammal' and then to 'Man'. The benefit of having intermediate classes, such as the class 'Mammal', is that they can then be used as a base class for some other classes also. For example, the class 'Mammal' can be used as a common base class for classes 'Whale', 'Dog', etc.

Hierarchical Inheritance

In hierarchical inheritance, a single class serves as a base class for more than one derived class. Diagram 5.9 illustrates this.

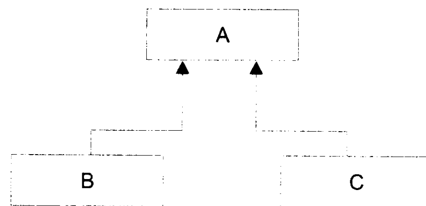

Diagram 5.9 Hierarchical inheritance

In Diagram 5.9, class A is the common base class for classes B and C. This is demonstrated in Listing 5.32.

```
/*Beginning of hierarchicalInherit.cpp*/
#include<iostream.h>

class A
{
  public:
    void fA()
    {
      cout<<"fA() called\n";
    }
};

class B : public A  //derived from A
{
  public:
    void fB()
    {
      cout<<"fB() called\n";
    }
};

class C : public A  //also derived from A
{
  public:
    void fC()
    {
      cout<<"fC() called\n";
    }
};

void main()
{
  B B1;
  C C1;
  B1.fA();
```

```
    B1.fB();
    C1.fA();
    C1.fC();
}
/*End of hierarchicalInherit.cpp*/
```

Output

fA() called
fB() called
fA() called
fC() called

Listing 5.32 Hierarchical inheritance

Hierarchical inheritance is probably the best illustration of the virtues of code reusability. The common features of two or more classes can be put together in a single base class that can then be inherited by those classes. The need to duplicate the common features in more than one class is, thus, eliminated.

As an example, the class 'Mammal' can be a common base class for the classes 'Man', 'Whale', 'Dog', 'Cat', etc. The features that are common to all these derived classes can be placed in the class 'Mammal'. Only the special features may be put in the respective derived classes.

Hybrid inheritance

Hybrid inheritance, as the name indicates, is simply a mixture of all the above kinds of inheritances. Diagram 5.10 illustrates this.

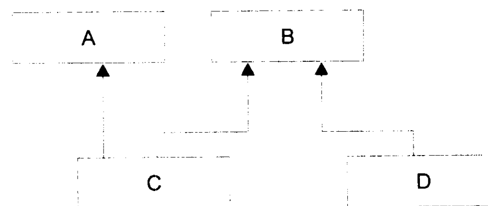

Diagram 5.10 Hybrid inheritance

5.8 Order of Invocation of Constructors and Destructors

Constructors are invoked in the following order:

1. Virtual base class constructors in the order of inheritance
2. Non-virtual base class constructors in the order of inheritance

3. Member objects' constructors in the order of declaration
4. Derived class constructor

Destructors are invoked in the reverse order. Listing 5.33 illustrates this.

```
/*Beginning of cd_order.cpp*/
#include<iostream.h>

class A
{
  public:
    A()
    {
      cout<<"Constructor of class A called\n";
    }
    ~A()
    {
      cout<<"Destructor of class A called\n";
    }
};

class B
{
  public:
    B()
    {
      cout<<"Constructor of class B called\n";
    }
    ~B()
    {
      cout<<"Destructor of class B called\n";
    }
};
class C : virtual public A
{
  public:
    C()
    {
      cout<<"Constructor of class C called\n";
    }
    ~C()
    {
      cout<<"Destructor of class C called\n";
    }
};
```

```
class D : virtual public A
{
  public:
    D()
    {
      cout<<"Constructor of class D called\n";
    }
    ~D()
    {
      cout<<"Destructor of class D called\n";
    }
};

class E
{
  public:
    E()
    {
      cout<<"Constructor of class E called\n";
    }
    ~E()
    {
      cout<<"Destructor of class E called\n";
    }
};
class F : public B, public C, public D
{
  private:
    E Eobj;
  public:
    E()
    {
      cout<<"Constructor of class F called\n";
    }
    ~E()
    {
      cout<<"Destructor of class F called\n";
    }
};

void main()
{
  F Fobj;
}
/*End of cd_order.cpp*/
```

Output

Constructor of class A called
Constructor of class B called

```
Constructor of class C called
Constructor of class D called
Constructor of class E called
Constructor of class F called
Destructor of class F called
Destructor of class E called
Destructor of class D called
Destructor of class C called
Destructor of class B called
Destructor of class A called
```

Listing 5.33 Order of invocation of constructors and destructors

Summary

C++ allows a class to be defined in such a way that it automatically includes member data and member functions of an existing class. As usual, it allows additional member data and member functions to be defined in the new class also. This is called inheritance.

The existing class whose features are being inherited is known as the base class or parent class or super class. The new class that is being defined by inheriting from the base class is known as the derived class or child class or sub-class.

Objects of the derived class contain data members of the derived class as well as the base class. Objects of the base class can call member functions of the derived class as well as those of the base class.

By allowing only the common data members and common member functions in the base class, inheritance enables code reusability and eases code maintenance. Inheritance implements an 'is-a' relationship whereas containership implements a 'has-a' relationship. Friendship is not inherited.

A base class pointer can point at an object of the derived class. But a derived class pointer cannot point at an object of the base class.

Member functions of the base class can be overridden in the derived class. Defining a member function in the derived class in such a manner that its name and signature match those of a base class function is known as function overriding.

Base class members can be initialized to values that are passed to the constructor of the derived class. These values can in turn be passed to the base class constructor.

'Protected' members are inaccessible to non-member functions. But they are accessible to the member functions of their own class and to member functions of the derived classes.

Classes can be derived by the public, protected, and private keywords. Deriving by the public access specifier retains the access level of base class members. Deriving by the protected access specifier reduces the access level of public base class members to protected while the access level of protected and private base class members remains unchanged. Deriving by the private access specifier reduces the access level of public and protected base class members to private while access level of private base class members remains unchanged. The default access specifier for inheritance is 'private'.

In multiple inheritance, a class derives from more than one base class. Multiple inheritance leads to a number of ambiguities. Ambiguity arises if two or more of the base classes have a member of the same name. Ambiguity can also arise if two or more base classes in turn inherit from a common base class. This is known as diamond-shaped inheritance. These ambiguities are resolved by either of the following:

- Using the scope resolution operator and passing the name of the actual owner class to call the function
- Overriding the function of the ultimate base class in the intermediate base class
- Deriving the intermediate base classes by using the virtual keyword

When a class inherits from a derived class, it is known as multi-level inheritance. In hierarchical inheritance, a single class serves as a base class for the derived class(es). Hybrid inheritance is a mixture of all the above kinds of inheritances.

Constructors are invoked in the following order:

- Virtual base class constructors in the order of inheritance
- Non-virtual base class constructors in the order of inheritance
- Member objects' constructors in the order of declaration
- Derived class constructor

Destructors are invoked in the reverse order.

Key Terms

inheritance

base class, parent class, super class

derived class, child class, subclass

data members of base class and objects of the derived class

function members of base class and objects of the derived class

keeping common features in base class for code reusability

base class and derived class pointers

overriding of base class member functions

base class initilization

protected members

deriving by public, protected, and private specifiers

multiple inheritance

- ambiguities in multiple inheritance

multi-level-inheritance

hierarchical inheritance

order of invocation of constructors and destructors

1. What is inheritance? How does it enable code reusability?
2. How does inheritance influence the size and functionality of derived class objects?
3. How does inheritance compare with containership?
4. How does inheritance compare with nesting?
5. Create a global non-member function that has a base class pointer as its formal argument. Call member functions of the base class through the pointer from within this function. Now call the function by passing addresses of the derived class objects.
6. Override one of the base class member functions that have been called from within the function you have defined above, in the derived class. Pass the address of an object of this derived class to the function. Which function gets called—the overridden function of the base class or the overriding function of the derived class?
7. Make a derived class pointer point at an object of the base class by explicit typecasting. Now access a member of the derived class that does not exist in the base class. What happens?
8. Why is it necessary for the derived class constructor to pass values explicitly to the base class constructor for initializing base class members?
9. A base class has data members. However, a class that is derived from it does not. Does the derived class need a constructor? Why?
10. What is the effect of using the protected access specifier on the visibility of a base class member?

11. Will a function of the derived class be able to access a public member of the base class if no access specifier was used to derive the derived class? Why?
12. What are the ambiguities that arise in multiple and diamond-shaped inheritance? How can they be removed?
13. In which order are the constructors and destructors called when an object of the derived class is created?
14. State true or false
    (i) A base class object is usually smaller than an object of its derived class.
    (ii) Inheritance increases the visibility of base class members.
    (iii) The constructor of a virtual base class is called before the constructor of a non-virtual base class.
    (iv) Inheritance implements a 'has-a' relationship.
    (v) A public member of the base class can be called with respect to an object of the derived class in a non-member function if the protected access specifier was used to derive the derived class.
15. Assume that you are building a simplified windows-based drawing program. From a menu, the user would select which type of shape—ellipse or rectangle—he/she wants to draw. After selecting, he/she would drag the mouse pointer from one point of the window to another and the selected shape would get drawn within the enclosing rectangle whose diagonally opposite points coincide with these two points.

    Create a class 'Shape'. Derive two classes—'Ellipse' and 'Rectangle'—from this class. Answer the following questions to arrive at the definitions of the classes:
    (i) Which class/classes should hold the coordinates of the enclosing rectangle as its data members—'Shape', 'Ellipse', 'Rectangle' or all of three?
    (ii) In the chapter on virtual functions and dynamic polymorphism, you would realize that the class 'Shape' should also have functions such as 'draw()' and 'getArea()'. Should these functions have only an empty definition when they are defined as members of the class 'Shape'? Would they have empty definitions when they are defined as members of the classes 'Ellipse' and 'Rectangle'?

Chapter 6

Virtual Functions and Dynamic Polymorphism

OVERVIEW

This chapter deals with one of the most remarkable features of C++: dynamic polymorphism and how virtual functions enable it.

Virtual functions enable the C++ programmer to create reusable code. So far, function overriding has appeared to be an unnecessary feature of C++. This chapter explains why C++ provides the feature of function overriding.

The mechanism by which C++ implements the virtual functions has also been dealt with in this chapter. Pure virtual functions, their need and usage find a prominent place in this chapter.

This chapter also discusses the use of virtual destructors and clone functions.

6.1 The Need for Virtual Functions

First, let us consider the following program and its output.

```
/*Beginning of A.h*/
class A
{
  public:
    void show();
};
/*End of A.h*/

/*Beginning of B.h*/
#include"A.h"
class B : public A  //class B derived from class A
{
  public:
    void show();    //function override
};
/*End of B.h*/

/*Beginning of A.cpp*/
#include"A.h"
#include<iostream.h>
void A::show()
{
  cout<<"A\n";
}
/*End of A.cpp*/

/*Beginning of B.cpp*/
#include"B.h"
#include<iostream.h>
void B::show()
{
  A::show();    //calling back the overridden function to
                //logically extend the class definition
  cout<<"B\n";
}
/*End of B.cpp*/
```

Listing 6.1 Overriding member function of base class in the derived class

Now let us consider the following client program.

```
/*Beginning of try1.cpp*/
#include"B.h"
#include<iostream.h>
void main()
{
  A A1;
  B B1;
  A * APtr;
  APtr=&A1;
  APtr->show();      //A::show() called. APtr is of type A*
  APtr=&B1;
  APtr->show();      //A::show() called. APtr is of type A*
}
/*End of try1.cpp*/
```

Output

A
A

Listing 6.2 Calling an overridden function through a pointer of base class type

As we will notice, the base class function is called irrespective of the type of object pointed at by the pointer. Here, the compiler decides which function is to be called by considering the type of the pointer; the type of the object pointed at by the pointer is not considered. The conclusion is that overriding in such cases is ineffective. This can be a serious problem when a client is trying to extend a class hierarchy. Why? Before we try to find an answer, let us realize that calling the function through a reference produces the same effect.

```
/*Beginning of try2.cpp*/
#include"B.h"  //from listing 6.01
#include<iostream.h>
void main()
{
  A A1;
  B B1;
  A &ARef1=A1;
  ARef1.show();      //A::show() called. ARef1 is of type A&
  A &ARef2=B1;
  ARef2.show();      //A::show() called. ARef2 is of type A&
```

```
}
/*End of try2.cpp*/
```

Output

A
A

Listing 6.3 Calling an overridden function through a reference of base class type

Now let us try to understand why the ineffectiveness of overriding can be a major hindrance in the extension of a class hierarchy.

Placing the pointer and the object pointed at by the pointer in the same function as local variables does not make any sense. After all, an object can be as effectively accessed through its name itself. Instead, the pointer appears as a formal argument in function definitions and the address of the object is passed as a parameter to the function calls. Similar comments hold true for the reference variable also. Let us proceed with this piece of knowledge.

Keeping in mind the definitions of classes A and B from Listing 6.1, we have a look at the following definition of function 'abc()' of a class X.

```
/*Beginning of X.h*/
#include"A.h"
class X
{
  public:
    void abc(A*);    //A* is the formal argument
};
/*End of X.h*/

/*Beginning of X.cpp*/
#include"X.h"
void X::abc(A * p)
{
  //some lines of (complicated) code
  p->show();
  //some more lines of (complicated) code
}
/*End of X.cpp*/

/*Beginning of try3.cpp*/
void main()
```

```
{
  X X1;
  A A1;
  B B1;
  X1.abc(&A1); //A::show() will be called
  X1.abc(&B1); //A::show() will be called
}
/*End of try3.cpp*/
```

Output

A
A

Listing 6.4 Calling an overridden function through a pointer of base class type

From our recent study we know that the 'A::show()' function will be called against both of the function calls in the 'main()' function of Listing 6.4.

Now let us take stock of the situation. The library programmer has defined the following:

- The class A
- The 'show()' function of class A
- The class X
- The 'abc()' function of class X

The definitions of the 'A::show()' and 'X::abc()' functions are final and have been put in libraries.

It is expected that a class will get derived from class A. The derived class may override the 'show()' function of class A. The overriding function will add the extra code that is relevant to the derived class. To complete the picture, it will also call back the overridden function 'A::show()'. In this way, the base class function will get successively refined by the overriding functions of the derived classes.

However, as Listing 6.4 shows, such an override has so far appeared ineffective. It is highly desirable that when the address of an object of the derived class B is passed to the 'X::abc()' function, then the 'B::show()' function should be called (see Listing 6.4). If this happens, then the *same* 'X::abc()' function will prove useful irrespective of the type of object whose address is being passed to it. Unfortunately, such an extension of the class hierarchy has so far remained elusive.

We must realize that derived classes such as class B may be defined much after functions such as 'X::abc()' function have been defined.

Moreover, the function 'X::abc()' should work equally well whether the address of an object of the base class A is passed as a parameter to it or the address of an object of any of the derived classes is passed.

In the present situation, it appears necessary to redefine the 'X::abc()' function corresponding to each derived class of class A. That is, the 'X::abc(A *)' function should be copied and redefined with a pointer of the derived class type as a formal argument. For example, 'X::abc(A *), X::abc(B *)', etc. This is certainly impossible because the definition of the 'X::abc()' function will be in some library that is inaccessible to the programmer who is defining the derived classes. It is also extremely cumbersome to redefine the 'X::abc()' function for each of the derived classes. This anyway goes against the principles of code reusability.

Thus, it proves impossible to extend an existing class hierarchy. Function overriding does not produce the desired effects. Virtual functions solve this problem.

6.2 Virtual Functions

Virtual functions provide one of the most useful and powerful features of C++ called dynamic polymorphism.

In order to appreciate the various nuances of dynamic polymorphism, let us first look at a function (shown in Listing 6.5) that returns the sum of factorials of the numbers that belong to a range whose limits are passed to it. The function may have the following definition.

```
long int factorialSum(unsigned int a, unsigned int b)
{
  int i;
  long int sum;
  for(sum=0,i=a;i<=b;i++)
    sum+=factorial(i);
  return sum;
}
```

Listing 6.5 Function to compute sum of factorials

Here, 'factorial()' is a function that returns the factorial of the parameter passed to it. Similarly, the following function returns the sum of cubes of the numbers that belong to the range whose limits are passed to it.

```
long int cubeSum(unsigned int a, unsigned int b)
{
  int i;
  long int sum;
  for(sum=0,i=a;i<=b;i++)
    sum+=cube(i);
  return sum;
}
```

Listing 6.6 Function to compute sum of cubes

The 'cube()' function returns the cube of the number passed as a parameter to it.

Again, the following function returns the sum of logarithms of the numbers that belong to the range whose limits are passed to it.

```
long int logSum(unsigned int a, unsigned int b)
{
  int i;
  long int sum;
  for(sum=0,i=a;i<=b;i++)
    sum+=log(i);
  return sum;
}
```

Listing 6.7 Function to compute sum of logarithms

The 'log()' function returns the logarithm of the number passed as a parameter to it.

A close look at the definitions of these functions reveals that the definitions of the functions are exactly the same except for the name of the inner function they all call. Nevertheless, the similarity in their definitions is striking. It will not be entirely unreasonable on our part to expect that there must be some means of replacing all these functions by a single function. We want to make a generic function that will replace all the above functions. For this, we have to specify a function pointer as an additional formal argument in the function.

```
//a generic sum function
long int genSum(unsigned int a, unsigned int b,
                long int (*p)())
```

```
{
  int i;
  long int sum;
  for(sum=0,i=a;i<=b;i++)
    sum+=(*p)(i);
  return sum;
}
```

Listing 6.8 Generic function to compute summation of series

Next, we can call the function by passing the function whose returned values have to be summed up as the last parameter to this generic function. The following lines of code demonstrate this.

```
x=genSum(1,5,factorial);
x=genSum(3,8,cube);
```

Now, any kind of summation can be carried out by this single generic function, provided the function whose returned values are being summed up returns a `long int` or a value of a compatible type. A very important point to be noted is that the generic sum function is capable of similarly summing up the returned values from a function that may be created well into the future! The function call,

```
(*p)(i),
```

exhibits polymorphic behavior, because while compiling it, it is not known which function will actually be executed. This becomes known only later when the client program that calls the 'genSum()' function is compiled.

We would like to perform a similar feat in C++ also. Let us look at the definition of 'X::abc()' function in Listing 6.4. We would certainly like the 'show()' function of that class to be called whose object's address is passed as its parameter. In other words, we would like to extend the class library as described in the previous section.

If the library programmer, who is defining the base class, expects and suspects overriding of a certain member function and wants to make such an override meaningful, he/she should declare the function as virtual. For declaring a function as virtual, the prototype of the function within the class should be prefixed with the `virtual` keyword. The `virtual` keyword may appear either before or after the keyword specifying the return type of the function. If the function is defined outside the class, then only the prototype should have the `virtual` keyword.

The syntax for declaring a virtual function as follows:

```
virtual <return type> <function name>(<formal arguments>);
```

The following lines of code illustrate how virtual functions are declared.

```
class A
{
  public:
    virtual void show(); //A::show() is virtual
};
```

or

```
class A
{
  public:
    void virtual show(); //A::show() is virtual
};
```

If we define the 'A::show()' function in Listing 6.1 as a virtual function by following this syntax, then the output of Listing 6.2 will be

A
B

instead of

A
A

This means that when the base class pointer points at an object of the derived class and a call is dispatched to an overridden virtual function, then it is the overriding function of the derived class, and not the overridden function of the base class, that is called.

Now, let us take stock of the situation. The library programmer's desire to enable a logical extension of the class library is now fulfilled. Let us go back to Listing 6.4. Even if class A is derived and the 'A::show()' function is overridden in the derived class much after the 'X::abc()' function is defined, the correct function will be called from within the 'X::abc()' function. That is, if the derived class overrides the base class function and the address of the derived class object is passed as a parameter to the 'X::abc()' function, then the overriding function will be called. If no such overriding occurs, the base class function itself will be called.

The function call

```
p->show();
```

in the 'X::abc()' function in Listing 6.4 exhibits polymorphic behavior. Against this function call, the 'show()' function of the base class or the overriding 'show()' function of any of its derived classes will be called. However, this polymorphic behavior is also dynamic in nature. Which function will be ultimately called is not known when the 'X::abc()' function is compiled and put in a library. This is decided only when the client program that calls this function is compiled. We can therefore say that compile time for the client is run time for the library. Therefore, the polymorphic behavior exhibited by virtual functions is also termed as *dynamic polymorphism*.

It is worthwhile to note that functions in the base class usually contain only those statements that are relevant to the base class itself. It is not always possible to provide a complete definition to them, as the base classes are sometimes abstract in nature ('start()' method in the 'Vehicle' class). The overriding functions of the derived class first call back the overridden base class functions and then add the extra statements that complete the definitions with respect to the derived class itself. Having such base class functions as virtual ensures that the client is able to call both the functions in sequence as desired.

Virtual functions of the base classes reappear as virtual in the derived classes also. Again, using the `virtual` keyword while defining the overriding derived class function is optional.

```
/*Beginning of autoVirtual.cpp*/
#include<iostream.h>
class A
{
  public:
    virtual void show()  //A::show() is virtual
    {
      cout<<"A\n";
    }
};
class B : public A       //B derived from A
{
  public:
    void show()          //B::show() is virtual
    {
      cout<<"B\n";
    }
};
class C : public B       //C derived from B
{
  public:
    void show()          //C::show() is virtual
    {
      cout<<"C\n";
    }
};
```

```
void main()
{
  B * BPtr;
  BPtr = new C;
  BPtr->show();
}
/*End of autoVirtual.cpp*/
```

Output

C

Listing 6.9 Virtual functions remain virtual

6.3 The Mechanism of Virtual Functions

Now, let us understand the mechanism of virtual functions. For every base class that has one or more virtual functions, a table of function addresses is created during run time. This table of function addresses is called the virtual table or VTBL in short. The VTBL contains the address of each and every virtual function that has been defined in the corresponding class. Addresses of non-virtual functions do not appear in such tables.

Suppose a class A has two virtual functions—'abc()' and 'def()'.

```
/*Beginning of A.h*/
class A
{
  public:
    virtual void abc();
    virtual void def();
};
/*End of A.h*/
```

Listing 6.10 A class with two virtual functions

During run time the VTBL of class A will be as shown in Diagram 6.1.

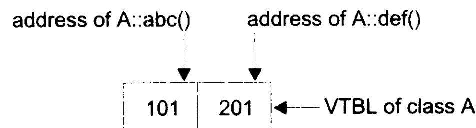

Diagram 6.1 Table of addresses of virtual functions of the base class

Similarly, such a table of addresses of virtual functions will be created for the derived class also. If the derived class does not redefine a certain base class member, then the table will contain the address of the inherited base class virtual function itself. But if a certain base class virtual function is redefined in the derived class, this table will contain the address of the overriding function. Finally, if the derived class defines a new virtual function, then its address will also be contained in the table. Thus, if class B is derived from class A as follows

```
/*Beginning of B.h*/
#include"A.h"
class B : public A
{
  public:
    void def();              //overriding the A::def() function
    virtual void ghi();      //introduces a new virtual
                             //function
};
/*End of B.h*/
```

Listing 6.11 Overriding base class virtual functions and introducing new ones

then the VTBL for class B will appear as follows.

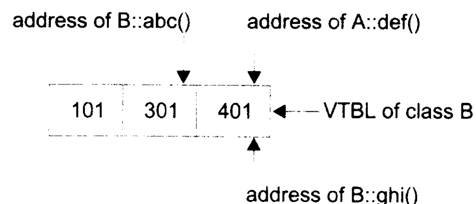

Diagram 6.2 Table of addresses of virtual functions of the derived class

Notice that since the 'A::abc()' function is not overridden in the derived class, its address reappears in the VTBL of class B. On the other hand, since the 'A::def()' function was overridden in the derived class, therefore its address is replaced in the VTBL of class B by the address of the 'B::def()' function. Finally, a new address appears in the VTBL of class B—the address of 'B::ghi()' function which is a newly introduced virtual function in class B.

Finally, every object of a class that has a virtual function contains a pointer to the VTBL of the corresponding class. This pointer is also known as the virtual pointer or VPTR. For example, an object of class A, apart from all other non-static data members, will also have a pointer to the VTBL of class A. This table is depicted in Diagram 6.1. Similarly,

an object of class B, apart from all other non-static data members, will also have a pointer to the VTBL of class B. This table is depicted in Diagram 6.2. Now, whenever a call is dispatched to a virtual function through an object or a reference to an object, or through a pointer to an object, then first of all the value of the VPTR of the object is read. Then the address of the called function from the corresponding VTBL is obtained. Finally, the function is called through the address thus obtained.

Now it is obvious how the virtual functions work. If a base class pointer points at (or a base class reference refers to) an object of the derived class and a virtual function is called with respect to it, then the derived class function will be called if it overrides the base class virtual function. If the base class virtual function is not overridden, then it itself will be called.

Note that it is the table size that varies from class to class (for each class there is only one VTBL). The size of the object does not vary. Only the size of the objects of classes with virtual functions increases uniformly by four bytes due to the presence of the additional pointer (VPTR).

We might wonder as to why C++ supports two types of binding--static and dynamic. Why does it not support dynamic binding only? In other words, why does it not declare all functions virtual by default? The reason is that virtual functions entail a run-time cost in the form of space that is wasted for creating the VTBL and embedding the VPTR in each and every object of the base/derived class. Time is also lost in searching the VTBL for the function address. If none of the member functions of a certain class will be overridden, then making them virtual will unnecessarily incur the above cost. Therefore, C++ allows the programmer to decide whether the member function has to be declared as virtual or not.

6.4 Pure Virtual Functions

From Section 6.3, we already know that it is optional to override virtual functions. A library programmer declares a member function as virtual if he/she expects overriding and wants to make the override effective.

But there are cases where the library programmer would like to enforce an override of the base class virtual functions. Such a case is now described. A call to a base class virtual function has been embedded somewhere in the code with respect to a pointer or reference of base class type. For example, a class A can have a virtual function 'abc()' that is called from a function 'xyz()' of class X.

```
/* Beginning of X.xpp*/
#include"X.h"
void X::xyz(A * p)
```

```
{
  //some lines of (complicated) code
  p->abc();
  //some more lines of (complicated) code
}
/*End of X.cpp*/
```

Listing 6.12 Using virtual functions

Now, 'A::abc()' function may satisfy either of the following descriptions.

- It has no meaningful definition with respect to the base class. For example, a function to rotate the shape cannot be defined in the class 'Shape' itself. The algorithm to rotate the shape is not known since the shape itself is not known.

```
class Shape
{
  public:
    virtual void rotate();
};

void shape::rotate()
{
  //null definition!
}
```

Listing 6.13 Giving blank definition to undefinable virtual function

- It has only a few lines of code, which do not really give it a complete definition. The function is such that it cannot be called in isolation. It can only be called indirectly through the derived class's overriding function that has the necessary code to complete the definition.

Obviously, C++ should provide some mechanism to the library programmer to enforce the desired override. Pure virtual functions provide this mechanism. If even one member function is declared as a pure virtual function, then the corresponding class becomes an Abstract Base Class (ABC in short). A function is declared as a pure virtual function by prefixing its prototype with the `virtual` keyword as before but suffixing it with an 'equal to' sign-and then by a 'zero' (0).

The syntax for declaring a pure virtual function is

```
virtual <return type> <function name>(<formal
                                     arguments>)=0;
```

The following example illustrates this.

```
/*Beginning of A.h*/
class A
{
  virtual void abc()=0;
  /*
    rest of the class A
  */
};
/*End of A.h*/
```

Listing 6.14 Defining a pure virtual function

An abstract base class cannot be instantiated, that is, objects of an abstract base class cannot be declared. Compile-time errors defeat attempts to do so.

```
A A1;      //error
```

The derived class must override all pure virtual functions of the base class or itself get branded as an ABC by the compiler. For example, any class that derives from class A in Listing 6.14 must override the 'A::abc()' function to become a concrete or instantiable class.

What is the utility of an ABC? The utility of an ABC lies in its use as an interface. The library programmer defines the ABC and also some generic functions that implement the general flow of a related algorithm without considering the exact data type on which they will work. An example to illustrate this point follows.

Let us consider an abstract class 'Shape' (see Listing 6.15).

```
/*Beginning of Shape.h*/
class Shape
{
    int x1,y1,x2,y2;                //coordinates of the
                                    //bounding rectangle
  public:
    void setBoundingRect(int,int,int,int);
                                    //set the bounding
                                    //rectangle for the
                                    //shape
    void move(int,int);             //move upper left corner
                                    //of bounding rectangle
```

```
                                        //to specified
                                        //coordinates
    virtual void rotate(float)=0;       //rotate by angle
                                        //specified in the
                                        //parameter
    virtual void shrink(float)=0;       //shrink by percent
                                        //specified in the
                                        //parameter
    virtual void grow(float)=0;         //grow by percent
                                        //specified in the
                                        //parameter
    virtual void hflip()=0;             //flip horizontally
    virtual void vflip()=0;             //flip vertically
    virtual void draw()=0;              //draw the shape
};
/*End of Shape.h*/
```

Listing 6.15 An abstract class

Let us consider a client driver function that the programmer defines to operate upon an object of class 'Shape' or any of its derived classes. This function will flash a shape in a certain sequence. While the actual object that will be flashed is not known when the function is defined, the sequence of operations for carrying out the operation has been decided.

```
/*Beginning of MyWindow.cpp*/
#include"MyWindow.h"
void MyWindow::flash(Shape * p)
{
  p->setBoundingRect(0,0,10,10);
  p->draw();
  p->rotate(90);
  for(int x=0;x<=10;x++)
    p->shrink(5);
  for(int x=0;x<=10;x++)
    p->grow(5);
  p->hlip();
  p->hflip();
  p->vflip();
  p->vflip();
}
/*
  definitions of rest of the functions of class MyWindow
*/
/*End of MyWindow.cpp*/
```

Listing 6.16 Client code to use the abstract class

There might be many more lines of code in the function of Listing 6.16. But the important thing to be remembered is that the class, which will be derived from the class 'Shape' and whose object's address will be passed to this function, might be defined much later. In addition, the same function will work equally well for each such class. Moreover, there can be many such client programs. After all, what is a class library if it does not have plenty of clients! But there is absolutely no need to define such driver functions for each of the derived classes separately. However, every such class will have to define each and every pure virtual function of the class 'Shape'. The abstract nature of the base class ensures this.

Thus, the ABC behaves just like an interface with little or no implementation of its own. The facility that the library programmer gets is that he/she is free to define generic functions without bothering about the implementation details. He/she can enforce all necessary overrides. The advantage for the application programmer is that he/she can derive any class from the ABC, provide his/her own implementations for the derived class and then use the same driver functions like 'MyWindow::flash()' for any of these derived classes.

Abstract Base Classes are also used to build implementation in stages. We know that if a pure virtual function inherited from the base class is not defined in the derived class, it remains a pure virtual function in the derived class. Thus, the derived class also becomes an abstract class.

Let me exemplify this explanation. Suppose there is an abstract class A having a number of pure virtual functions.

```
class A
{
  public:
    virtual void abc()=0;
    virtual void def()=0;
    virtual void ghi()=0;
};

void main()
{
  A A1;    //ERROR!
}
```

Listing 6.17 An abstract base class

A class B is derived from it. This class B overrides and defines only a few of the functions of class A. Thus, class B is also an ABC.

```
class B : public A
{
  public:
    void abc()
    {
      //definition of B::abc() function
    }
};

void main()
{
  B B1;    //ERROR!
}
```

Listing 6.18 Not defining all base class pure virtual functions results in an abstract class

Next, a class C derives from class B. It overrides and defines the remaining pure virtual functions of class A. Thus, class C becomes a concrete class.

```
class C : public B
{
  public:
    void def()
    {
      //definition of C::def() function
    }
    void ghi()
    {
      //definition of C::ghi() function
    }
};

void main()
{
  C C1;    //OK
}
```

Listing 6.19 Defining all the inherited pure virtual functions results in a concrete class

We may ask why all pure virtual functions of class A were not defined in class B itself. The reason is that there is no concrete definition of the remaining functions with respect to class B. It itself serves as a base class for a number of derived classes. Each of the derived classes have a different definition of the pure virtual functions of class A that are left undefined by class B. If class B defines any of these functions, then such functions

will themselves be called when a pointer of class A type points at an object of any of the derived classes of class B and calls are dispatched to the pure virtual function of class A. This is obviously undesirable. Class B will define only those pure virtual functions of class A that can have a suitable meaning with respect to it. Such definitions will be applicable to all its derived classes.

Although concrete classes must provide an implementation to all the pure virtual functions, the abstract data type may provide one as well. The derived class can invoke it by using the scope resolution operator.

```
class A         //An abstract base class
{
  public:
    virtual void abc()=0;
};

void A::abc()   //pure virtual function defined
{
  //definition of A::abc() function
}

class B : public A
{
  public:
    void abc();
};

void B::abc()
{
  A::abc();
  //definition of rest of the B::abc() function
}
```

Listing 6.20 Defining a pure virtual function in the abstract base class itself

The ability to provide an implementation to pure virtual methods allows data types to provide core functionality while still requiring derived classes to provide a specialized implementation. Note that the class remains abstract even if we provide an implementation for its pure virtual function.

As per requirements, a member function can be non-virtual, virtual, or pure virtual.

6.5 Virtual Destructors and Virtual Constructors

We will now study the creation and use of `virtual` constructors and destructors. We will first study `virtual` destructors. This will be followed by a study of `virtual` constructors.

Virtual Destructors

Destructors can be defined as `virtual`. If necessary, destructors must be defined as `virtual`. Why? Let us consider the following snippet (A is a base class and B is a class derived from A).

```
A * APtr;
APtr = new B;
. . . .
. . . .
delete APtr;
```

Listing 6.21 Destroying a derived class object through a base class pointer

Let us consider the last line of Listing 6.22 that deletes the memory occupied by the object of class B at which 'APtr' points. Because 'APtr' is of base class type, only the base class destructor is called with respect to the object before the entire memory occupied by the object is returned to the OS. This can lead to memory leaks apart from other problems. Suppose objects of the derived class B have a pointer. It is possible that this pointer, which is contained in the object at which 'APtr' points, is assigned a dynamically allocated memory block during the lifetime of the object. Although the destructor of class B destroys that memory block to prevent memory leaks, a memory leak will still occur because the destructor of class B is not called.

On the other hand, if the destructor of class A is `virtual`, then against the last line of Listing 6.21, first the destructor of class B will be called, then the destructor of class A will be called. Finally, the entire memory block occupied by the object will be returned to the OS.

The conclusion is that if we expect the use of the `delete` operator on objects of a base class and the presence of pointers in the derived classes, we must declare the destructor of the base class as `virtual`.

An interesting point to be noted is that when a pointer of the base class points at a dynamically created object of the derived class and then deletes the memory occupied by the object, the entire block of memory is deleted. In other words, if the total size of the non-static data members of the base class is 'x' and the total size of the non-static data members of the derived class is 'y', then the total block of size 'x+y' is deleted. This is irrespective of whether the base class destructor is `virtual` or not.

Virtual Constructors

First, let us understand that constructors cannot be `virtual`. Declaring a constructor as `virtual` results in a compile-time error. Why? Consider a class A, and a class B that is derived from A. If the constructor of class A is `virtual`, then in the following statement

```
A * p = new B;
```

the constructor of class B alone will be called. The constructor of class A will not be called. This can lead to trouble. What will happen if class A has a pointer that the constructor correctly initializes? Since the constructor is not called, a rogue pointer will result.

However, the need to *construct virtually* arises very frequently while programming in C++. Let us consider the function in the following lines of code.

```
void abc(A * p)      //A is a class
{
  //definition of abc() function
}
```

For reasons that will be listed later, an exact copy of the object at which 'p' points is required within the 'abc()' function. This means that another object that has the same values as the object at which 'p' points needs to be created within the 'abc()' function. Calling the copy constructor seems to serve the purpose.

```
A * q = new A(*p);
```

or

```
A A1(*p);
```

Listing 6.22 Trying to clone using copy constructor

This will work if the designer of class A has correctly defined the copy constructor and if 'p' points at an object of class A and not at an object of a class that is derived directly or indirectly from class A. If 'p' points at an object of a class that is derived directly or indirectly from class A, the call to the copy constructor as mentioned above will merely create an object of class A. The data members of this object will have the same values as the corresponding data members of the object at which 'p' points. Nevertheless, it will not be of the same type (it will be smaller is size with less data members).

How can this problem be solved? If the designer of class A suspects and expects the need to create copies like this, he/she will define a clone function to do so. Such a function can be defined as follows.

```
class A
{
  public:
    virtual A * clone()
    {
      return new A(*this);
    }
    /*
      definition of class A
    */
};
```

Listing 6.23 A clone function in the base class

Classes that derive from class A will similarly define and override this clone function.

```
class B : public A
{
  public:
    virtual B * clone()
    {
      return new B(*this);
    }
    /*
      definition of class B
    */
};
```

Listing 6.24 A clone function in the derived class

Whenever a clone of the object is required (from within the 'abc()' function as described in Listing 6.24), the clone function is called. The clone object created is subsequently destroyed.

```
void abc(A * p)
{
  . . . .
  . . . .
```

```
    A * q = p->clone();
    . . . .
    . . . .
    delete q;
}
```

Listing 6.25 Using the clone function

Since the 'clone()' function is `virtual`, its correct version is called. Thus, if 'p' points at an object of class A, then another object of class A itself is created which is an exact copy of the object at which 'p' is pointing. And, if 'p' points at an object of a class derived from A, then another object of that same class is created which is an exact copy of the object at which 'p' is pointing. Thus, the 'abc()' function succeeds in obtaining an exact copy of the object at which 'p' is pointing while being unaware of its type.

Since the clone function constructs an object and is also `virtual`, we sometimes call it a *virtual constructor*. But we must remember that there is actually nothing like a *virtual constructor*.

Now let us discuss the need for the clone function. Although there are several examples that highlight this need, the following example alone should suffice.

Let us consider a function that copies and pastes a graphics object to a different place of the window.

```
void MyWindow::copyPaste(const Shape * const p,
                         unsigned int x, unsigned int y)
{
  Shape * q;
  q=p->clone();
  q->move(x,y);
  q->show();
  //code to attach the new object to the list
  //of current objects on the screen
}
```

Listing 6.26 An example to illustrate the use of the clone function

The pointer 'p' points at the object being copied. The variables 'x' and 'y' are the coordinates of the place where the copied object is to be pasted. Although 'p' might point at an object of any of the classes that are derived from the 'Shape' class, the entire operation of copying and pasting works in all cases. The only precondition is that the classes that are derived from the 'Shape' class must define the 'clone()' function, the

'move()' function and the 'show()' function. This is easily ensured by declaring all these functions as `pure virtual` functions in the 'Shape' class. An additional point to be noted in this specific example is that the clone object should not be destroyed. Instead, it should be added to the list of existing objects on the screen.

Summary

Generic code contained in the functions of the base class remains inextensible without the use of virtual functions. Virtual functions make expected overrides in the derived class effective. Marking a base class function as a pure virtual function forces its override in the derived class.

An abstract base class is a class that has at least one pure virtual function. An abstract base class cannot be instantiated. Virtual destructors ensure a proper cleanup operation if the 'delete' operator is applied on a base class pointer that points at a derived class object.

Although we cannot have virtual constructors, clone functions that construct virtually can be used instead. A clone function returns an exact copy of the object at which the base class pointer, with respect to which it is called, points. If the pointer points at a base class object, the clone function creates an object of the base class and returns a pointer to it. If the pointer points at a derived class object, the clone function creates an object of the derived class and returns a pointer to it.

Key Terms

virtual functions

extending class libraties by using virtual functions

pure virtual functions

abstract base class

virtual destructors

clone functions

Exercises

1. What is a virtual function? When is it needed?
2. How does the compiler resolve a call to a virtual function?
3. Suppose a member function of a class has been prototyped as virtual. The function has not been defined. Now, when we instantiate the class, the linker gives an error even if we do not

call the function. Why? (*Hint:* Remember that if a non-virtual function is prototyped and then neither called nor defined, no error is generated.)

4. What is a pure virtual function? When is it needed?
5. State true or false.
   (i) Virtual functions implement static polymorphism.
   (ii) We cannot have a virtual constructor.
   (iii) An abstract base class cannot be instantiated.
   (iv) We cannot define a pure virtual function.
6. Write a program to find out whether a virtual function can be a friend of another class.
7. Create a class 'Shape'. It should have no data members. It should have a pure virtual function 'get_area()'.

   Derive a class 'Rectangle' from the class 'Shape'. It should have two data members—one for holding the width of the rectangle and the other for holding its height. Both of these data members should be of float type. Override the 'Shape::get_area()' function inside this class. This overriding function should return the area of the rectangle. Also, write a constructor for the class.

   Derive another class 'Ellipse' from the class 'Shape'. It should also have two data members—one for holding the length of the major axis of the ellipse and the other for holding the length of its minor axis. Both of these data members should be of float type. Override the 'Shape::get_area()' function inside this class. This overriding function should return the area of the ellipse. Also, write a constructor for the class.

   Create a class 'Canvas'. It should have no data members. Its only member function, 'display()', will have a reference of class 'Shape' type as a formal argument. With this reference, call the 'Shape::get_area()' function inside 'Canvas:: display()' function.

   Finally, write a 'main()' function to utilize these classes. Declare objects of classes 'Rectangle', 'Ellipse', and 'Canvas'. Call the 'Canvas::display()' function first by passing the object of class 'Rectangle' and then by passing the object of class 'Ellipse' to it. Observe the output and ascertain whether the base class function or the derived class function got called.

   Redefine the 'Shape::get_area()' function as a non-virtual function and see the difference in the output.

Chapter 7

Stream Handling

OVERVIEW

This chapter deals with handling streams. It includes a study of classes in the C++ standard library that enable a programmer to handle flow of data to and from the console and also from disk files.

Text and binary mode of handling streams and the distinction between the two forms an important part of the chapter.

The use of classes and their member functions that enable a random access to disk files is discussed. Objects can and in many cases should be made capable of outputting and loading their own data to and from disk files. This chapter tells you how.

Error handling is an important feature expected in any industrial strength software. This chapter discusses the use of error handling functions that pertain to streams.

Manipulators are a handy tool for the C++ programmer. This chapter elucidates the use of many system-defined manipulators and the method of creating user-specific ones.

7.1 Streams

Stream means flow of data. We declare variables in our C++ programs for holding the data temporarily in the memory. Streams are nothing but a flow of data to and from program variables.

Input stream is the flow of data from a file to program variables. The keyboard is also treated as a source of input stream. Output stream is the flow of data to a file from program variables. The monitor is also treated as a target for output stream.

7.2 The Class Hierarchy for Handling Streams

C++ provides us with a library of classes that have the functionality to implement various aspects of stream handling. These classes have been arranged in a hierarchical fashion by using inheritance. The important portion of this hierarchy is depicted in the following figure:

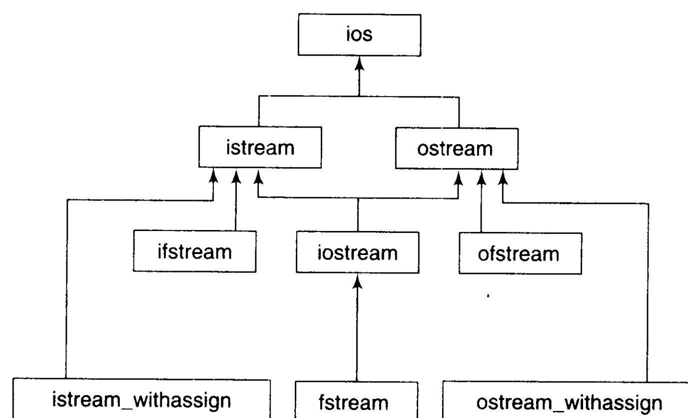

Diagram 7.1 Library classes that handle streams

The class 'ios' is the base class in this hierarchy.

The class 'ostream' is derived from the class 'ios' and handles the general output stream. The insertion operator (<<) is defined and overloaded in the class 'ostream' to handle output streams from program variables to output files.

The class 'ostream_withassign' is derived from the class 'ostream'. 'cout' is an object of the class 'ostream_withassign' and stands for console output. As mentioned earlier, C++ treats all peripheral devices as files. It treats the monitor also as a file (for output stream). The object 'cout' represents the monitor.

Thus the statement

```
cout<<x;
```

translates as:

"Insert the stream from the program variable 'x' into the file called 'cout' (which is nothing but the monitor)."

The class 'istream' is derived from the class 'ios' and handles the general input streams. The extraction operator (>>) is defined and overloaded in the class 'istream' to handle inputs streams from input files to program variables.

The class 'istream_withassign' is derived from the class 'istream'. 'cin' is an object of the class 'istream_withassign' and stands for console input. C++ treats all peripheral devices as files. It treats the keyboard also as a file (for input stream). The object 'cin' represents the keyboard.

Thus the statement

```
cin>>x;
```

translates as:

"Extract the stream from the file (which is nothing but the keyboard) and place it in the program variable 'x'."

The class 'iostream' is derived by multiple inheritance from the classes 'istream' and 'ostream'. It has the functionality to handle both input and output streams.

The class 'ofstream' is derived from the class 'ostream'. It has the functionality to handle output streams to disk files. Objects of the class 'ofstream' represent output files on the disk. Thus, the following piece of code opens a disk file for output (note that the name of the file to be opened is passed as a string to the constructor of the class):

```
ofstream ofile("first.dat");
```

The file 'first.dat' is opened for output in the directory where the executable will run. The entire path of the file to be opened can also be prefixed to the name of the file. Since the insertion operator is defined in the base class of the class 'ofstream', the object 'ofile' can be passed as the left-hand side operand instead of 'cout'.

```
ofile<<x;
```

The above statement translates as:

"Insert the stream from the program variable 'x' into the file 'first.dat'."

The class 'ifstream' is derived from the class 'istream'. It has the functionality to handle input streams from disk files. Objects of the class 'ifstream' represent input files on the disk. Thus, the following piece of code opens a disk file for input (note that the name of the file to be opened is passed as a string to the constructor of the class):

```
ifstream ifile("first.dat");
```

The file 'first.dat' is opened for input in the directory where the executable will run. The entire path of the file to be opened can also be prefixed to the name of the file. Since the extraction operator is defined in the base class of the class 'ifstream', the object 'ifile' can be passed as the left-hand side operand instead of 'cin'.

```
ifile>>x;
```

This statement translates as:

"Extract the stream from the file 'first.dat' and place it in the program variable 'x'."

The class 'fstream' is derived from the class 'iostream'. It has the functionality to handle both input and output streams from and to disk files.

The classes for handling streams to and from disk files are defined in the header file 'fstream.h'. The classes for handling general streams are defined in the header file 'iostream.h'. The header file 'iostream.h' is included in the file 'fstream.h'.

7.3 Text and Binary Input/Output

In this section, the two modes of input/output—text mode and binary mode—will be explained. The difference between them and the suitability of each mode for console I/O and disk I/O will also be explained.

How Data is Stored in Memory?

During run time, the value of a character variable is stored in the memory as the binary equivalent of its ASCII equivalent. But the value of an integer, float, or double type variable is simply stored as its binary equivalent.

For example, if the value of a character type variable is 'A', it is stored in one byte where the bits represent the number '65', which is the ASCII equivalent of 'A', in base 2.

```
01000001
```

But if the value of an integer type variable is '65', it is stored in four bytes where the bits represent the number '65' in base 2.

```
01000001
00000000
00000000
00000000
```

The value of a float and a double type variable is stored in a similar fashion. Thus, the value of a numeric variable (integer, float, or double) is stored in base 2 format in the memory.

Input/Output of Character Data

There is no difference between text mode I/O and binary mode I/O with respect to character type variables. In both modes, the value of the character type data is copied from the memory into the output file as it is and copied from the input file into the memory as it is.

If a function that outputs in binary mode is called to output the value of a character variable, it will copy its value into the output file without transforming its representation in any way. A function that outputs in text mode will output the value of a character variable in the same way.

Input/Output of Numeric Data

A standard library function that outputs numeric data in text mode will reckon that the data, which is in base 2 format in the memory, needs to be output in base 10 (text) format. It will therefore read the value of the source variable from memory, *transform* the representation of the data from the existing base 2 to base 10 and only then copy it to the output file.

Whereas a standard library function that outputs numeric data in binary mode will reckon that the data, which is in base 2 format in the memory, needs to be output in base 2 (binary) format itself. It will therefore read the value of the source variable from memory, *not* transform the data from the existing base 2 to base 10 and simply copy it to the output file.

Further, a standard library function that inputs numeric data in text mode will reckon that the data, which is to be input, exists in base 10 (text) format. It will therefore read the value from the input file, *transform* the representation of the data from the existing base 10 to base 2 and only then copy it to the target variable in memory.

Whereas a standard library function that inputs numeric data in binary mode will reckon that the data, which is to be read, already exists in base 2 (binary) format. It will therefore read the value from the memory, *not* transform the data in any way and simply copy it into the target variable in memory.

Significance of the Difference between Binary mode and Text mode I/O for Numeric Data

Suppose the value of an integer type variable is '65'. The value '65' is stored, in the memory block of four bytes occupied by the variable, in binary mode.

The foregoing data will occupy four bytes with the *same* bit setting in the output disk file, if it is copied by an output function that outputs in binary mode.

If this output function is used to output to the monitor, instead of a disk file, the value 'A' followed by three blank spaces will be displayed. This is because the lowest byte of the variable contains '65', which is the ASCII equivalent of 'A' and the rest of the three bytes have all their bits set to zero. The monitor displays the ASCII equivalent of the value in each byte in the output stream that is supplied to it. But, we would like to see '65' and not 'A' followed by three blank spaces on the monitor.

However, if the same value is copied by an output function that outputs in text mode, this data will occupy two bytes with a different bit setting in the output file. The first byte will have its bits set to represent the character '6' (The ASCII equivalent of '6' will be stored in base 2 format). While the second byte will have its bits set to represent the character '5' (The ASCII equivalent of '5' will be stored in base 2 format). This is because the output function, since it works in text mode, has transformed the representation of the data from base 2 to base 10.

For the same reason, if the same output function is used to output to the monitor, instead of a disk file, the value '65' will be displayed. This is the kind of display we would desire.

An input function that inputs in binary mode will read the first four bytes from an input file if it is asked to input into an integer type variable. It will copy the read value into the memory block of four bytes occupied by the target integer variable *without* transforming the representation of the data. This function will not transform the data because it inputs in binary mode and therefore reckons that the data existing in the input stream is already in base 2 format.

If the input is read from the keyboard instead of a disk file, the function reads the first four bytes from the keyboard and copies the read value into the memory block of four bytes occupied by an integer variable *without* transforming the representation of the data.

Further, a function that reads in text mode, reads the bytes from the input file up to the white space. It reckons that the read value is in base 10 format. It therefore determines the equivalent representation in base 2 format. This produces a value that occupies four bytes. The function then copies these four bytes into the memory block of four bytes occupied by integer variable.

If the input is read from the keyboard instead of a disk file, the function reads the bytes from the keyboard up to the white space and operates upon it in a similar fashion. For example, it the user enters the number '65', the characters '6' and '5' get stored in the keyboard buffer, which represents the input file in this case. The characters '6' and '5' that are stored in two bytes represent the number '65' in base 10 format. The input

function *transforms* this representation into base 2 format and stores the resultant integer value in the four bytes occupied by the target integer variable.

A very important and interesting observation can be made here. *Text mode is suitable for console I/O* because they are in base 10 format with which we are accustomed. For reasons that will be explained shortly, *binary mode is suitable for disk I/O.*

A Note on Opening Disk Files for I/O

In this section, that is, Section 7.3 'Text and binary input/output', we would open files for writing through the constructor of class 'ofstream'. If the file being opened does not exist, it would get created. If it does exist, its contents would get overwritten. For producing a different effect—appending or obtaining errors if the file does not exist—we have to apply the techniques that are explained in sections 7.4 and 7.8. We would also open files for reading through the constructor of class 'ifstream'. If the file being opened does not exist, a run-time error is produced. The technique of handling such errors is explained in section 7.8.

7.4 Text Versus Binary Files

Now let us talk about text files and binary files.

In text files, binary data (numeric data that is stored in base 2 format in memory) is stored in base 10 format. In binary files, the same binary data is stored in the same format (base 2).

Before proceeding further, let us be clear that the files by themselves are neither text files nor binary files. It is the mode in which the data is written into them that defines the nature of the files.

As we have already learnt, when the value of a numeric variable (say an integer) is output in binary mode, it occupies the same number of bytes in the output file as it does in the memory, which is four. Thus, if a number of such values are output in binary mode, the size of the output file will always be a multiple of four. Obviously, we can determine the size of the file and divide it by four to easily find the number of integers (records) that are stored in the file.

We can apply this simple technique to values of other types including objects. Suppose an object consists of an integer (four bytes), a float (four bytes), and a double (eight bytes). This object occupies 16 bytes in the memory. If output in binary mode, it will occupy 16 bytes in the output file also.

The C++ standard library provides functions that input (read from files and write the read values into variables) in binary mode. These functions require the address of the variable whose data needs to be input along with its size. The code inside these functions reads the value from a point in the input file at which a temporary pointer points (more

about this pointer later). The size of the block of bytes this code reads is equal to the supplied size. The code then writes the read data into the memory block whose starting address is equal to the supplied address. Functions that output in binary mode work in a similar fashion. As we can see, binary input functions treat size as the delimiter while reading data from disk files. Therefore, the binary functions that write data into the disks need not insert an artificial delimiter while the data is output.

Let us contrast this to what happens in text mode. In text mode, records stored in the output file are of variable lengths. Again, for our understanding, let us take the case of an integer type variable. Its value is always stored in four bytes in the memory. Suppose it is output to a disk file in text mode. If its value is '1', it will occupy one byte in the output file although it occupies four bytes in the memory. If its value is '11', it will occupy two bytes in the output file although it occupies four bytes in the memory. If its value is '111', it will occupy three bytes in the output file although it occupies four bytes in the memory and so on.

Thus, in case values are output in text mode, size of the output value is not fixed. Hence, size cannot be used as the delimiter by functions that will read the output values in future. But it should be ensured that values that are output in text mode can be read correctly in future. For this, the code that calls a text mode function for output should also insert a delimiter of choice in the output file after every such call. This ensures that another piece of code is able to successfully read this output value.

Choosing a suitable delimiter is certainly an issue. There should be no chance of the delimiting character itself becoming a part of the output value anytime in the future. But this is difficult to guarantee.

There is another difficulty in outputting in text mode. The size of a file does not indicate the number of records stored in it. This is because the size of the records is not fixed.

7.5 Text Input/Output

Text Output

Text output is achieved in C++ by:

- The insertion operator
- The 'put()' function

The Insertion Operator

As we have seen in the first section of this chapter, the insertion operator can be used to output values to disk files. The insertion operator outputs in text mode.

The insertion operator has been defined and overloaded in the class 'ostream'. It takes an object of the class 'ostream' or an object of a class that is derived from the class 'ostream' as its left-hand side operand. As its right-hand side operand, it takes a value of one of the

fundamental data types. It copies the value on its right into the file that is associated with the object on its left. Let us study its action on data of different types. We must keep in mind that the insertion operator has been overloaded differently for each of the data types as follows:

1. Inserting characters into output streams using the insertion operator: A character type value occupies one byte in the memory. If output in text mode by the insertion operator, it occupies one byte in the output file too. The bit setting of both the bytes is identical.

```
/*Beginning of charFileOutput.cpp*/
#include<fstream.h>
void main()
{
  char cVar;
  ofstream ofile("first.dat");
  cVar='A';
  ofile<<cVar;
}
/*End of charFileOutput.cpp*/
```

Listing 7.1 Outputting a character in text mode by using the insertion operator

The last statement of Listing 7.1 copies the value of 'cVar' from memory to the disk file 'first.dat' without transforming its representation in any way.

2. Inserting integers into output streams using the insertion operator: An integer type value occupies four bytes in the memory. As we already know, if output in text mode by the insertion operator, the number of bytes it occupies in the output file depends upon its value.

```
/*Beginning of intFileOutput.cpp*/
#include<fstream.h>
void main()
{
  int iVar;
  ofstream ofile("first.dat");
  iVar=111;
  ofile<<iVar;
}
/*End of intFileOutput.cpp*/
```

Listing 7.2 Outputting an integer in text mode by using the insertion operator

The last statement of Listing 7.2 copies the value of 'iVar' from memory to the disk file 'first.dat' after transforming its representation from base 2 to base 10. The value of 'iVar' will be written in text format (base 10) and will therefore occupy three bytes in the output file. If the value of 'iVar' is '11111' instead of '111', it will occupy five bytes in the output file instead of three.

3. Inserting floats and doubles into output streams using the insertion operator: A float type value occupies four bytes in the memory. As we already know, if output in text mode by the insertion operator, the number of bytes it occupies in the output file depends upon its value.

```
/*Beginning of floatFileOutput.cpp*/
#include<fstream.h>
void main()
{
  float fVar;
  ofstream ofile("first.dat");
  fVar=1.111;
  ofile<<fVar;
}
/*End of floatFileOutput.cpp*/
```

Listing 7.3 Outputting a float in text mode by using the insertion operator

The last statement of Listing 7.3 copies the value of 'fVar' from memory to the disk file 'first.dat' after transforming its representation from base 2 to base 10. The value of 'fVar' will be written in text format (base 10) and will therefore occupy five bytes in the output file. If the value of 'fVar' is '11.111' instead of '1.111', it will occupy six bytes in the output file instead of five.

The insertion operator works in the same way for double type variables.

4. Inserting strings into output streams using the insertion operator: A character array is allocated a fixed number of bytes in the memory during run time. However, the actual string contained in it usually occupies only a part of that memory. For example,

```
char cArr[20]="abcd";
```

The character array 'cArr' will be allocated 20 bytes during run time. But the string inside it will occupy only four bytes. The fifth byte will have the NULL character.

If the value of 'cArr' is output by the insertion operator, it will occupy four bytes in the output file.

```
/*Beginning of charArrFileOutput.cpp*/
#include<fstream.h>
void main()
{
  char cArr[20]="abcd";
  ofstream ofile("first.dat");
  ofile<<cArr;
}
/*End of charArrFileOutput.cpp*/
```

Listing 7.4 Outputting a string in text mode by using the insertion operator

But if 'cArr' contains a string of length five, then five bytes will get written into the output file.

5. Inserting objects into output streams using the insertion operator: If we want to use the insertion operator for inserting objects of a particular class into the output stream, we have to overload it for that class. The concept of operator overloading, its need, and its use are elucidated in the next chapter.

The put() Function

The 'put( )' function is a member of the 'ostream' class. Its prototype is:

```
ostream & ostream :: put(char c);
```

From the prototype, it is obvious that the function can be called with respect to an object of the 'ostream' class or any of the classes that are derived from the 'ostream' class. One such object is 'cout'.

This function copies the character that is passed as a parameter to it into the output file associated with the object with respect to which the function is called. Let us consider the following explanatory program listing:

```
/*Beginning of put.cpp*/
#include<fstream.h>
void main()
{
  ofstream ofile("first.dat");
  ofile.put('a');
}
/*End of put.cpp*/
```

Listing 7.5 The 'put()' function

In Listing 7.5, the put() function is called with respect to the object 'ofile'. This object is associated with the file 'first.dat'. Consequently, the character 'a' is written into the file.

As was mentioned earlier, the 'put()' function can be used with the object 'cout' also (as shown in Listing 7.6).

```
/*Beginning of coutPut.cpp*/
#include<fstream.h>
void main()
{
  cout.put('a');
}
/*End of coutPut.cpp*/
```

Output

a

Listing 7.6 Using the 'put()' function with 'cout' object

The call to the 'put()' function in Listing 7.6 will display the character 'a' on the monitor.

We may wonder what is the difference between using the 'put()' function and the insertion operator. After all we could have used the insertion operator instead of calling the 'put()' function as follows:

```
cout<<'a';
```

The difference between the insertion operator and the 'put()' function is that while the former modifies the format of the output with respect to the manipulators set earlier, the latter simply ignores format manipulator settings. Formatted output is dealt with in one of the later sections of this chapter.

Text Input

Text input is achieved in C++ by:

- The extraction operator
- The 'get()' function
- The 'getline()' function

The Extraction Operator

As we have seen earlier in this chapter, the extraction operator can be used to input values from disk files. The extraction operator inputs in text mode.

The extraction operator has been defined and overloaded in the class 'istream'. It takes an object of the class 'istream' or an object of a class that is derived from the class

'istream' as its left-hand side operand. As its right-hand side operand, it takes a variable of one of the fundamental data types. It copies the value found at the current location in the file that is associated with the object on its left into the variable on its right. Let us study its action on data of different types. We must keep in mind that the extraction operator has been overloaded differently for each of the data types as follows:

1. Extracting characters from input streams using the extraction operator: If the right-hand side operand of the extraction operator is a character type variable, it reads one byte from the input file that is attached with the object on its left and writes it into the variable.

```
/*Beginning of charFileInputText.cpp*/
#include<fstream.h>
void main()
{
  ifstream ifile("first.dat");//Current location is at
                              //the beginning of the file.
                              //Suppose first byte in the
                              //file contains 'A'.
  char cVar;
  ifile>>cVar;
  cout<<cVar;
}
/*End of charFileInputText.cpp*/
```

Output

A

Listing 7.7 Inputting a character in text mode by using the extraction operator

2. Extracting integers from input streams using the extraction operator: If the right-hand side operand of the extraction operator is an integer type variable, it reads bytes from the input file that is attached with the object on its left until it finds a white space. It reckons that the read set of bytes represents an integer in base 10 format. Therefore, the extraction operator converts the read value into base 2 format. Finally, it writes the converted value into the variable.

Suppose the contents of a file 'first.dat' are as follows:

```
11 22 33
```

We must note that there is a space after '11'. The first byte of the file has the ASCII equivalent of the character '1'. The second byte also has the ASCII equivalent of the

character '1'. The third byte has the ASCII equivalent of the character ' ' (space). The fourth byte has the ASCII equivalent of the character '2' and so on.

Now let us consider the following program listing:

```
/*Beginning of intFileInputText.cpp*/
#include<fstream.h>
void main()
{
  ifstream ifile("first.dat");
  int iVar;
  ifile>>iVar;
  cout<<iVar;
}
/*End of intFileInputText.cpp*/
```

Output

11

Listing 7.8 Inputting an integer in text mode by using the extraction operator

As discussed earlier, the extraction operator reads from the file until it finds a white space. Since the third byte contains a white space, it reads the first two bytes only. These two bytes represent the number eleven in base 10 format. The extraction operator converts this into base 2 format. The resultant value is in four bytes. It writes this value into the variable 'iVar'.

3. Extracting floats and doubles from input streams using the extraction operator: Values for float and double type variables are extracted in the same as they are for integer type variables.

4. Extracting strings from input streams using the extraction operator: As in the case of integers, the extraction operator reads from the file until it finds a white space while reading value for a character array.

Suppose the contents of a file 'first.dat' are:

```
abc def ghi
```

We must note that there is a white space after 'c'.

```
/*Beginning of charArrFileInputText.cpp*/
#include<fstream.h>
void main()
```

```
{
  ifstream ifile("first.dat");
  char cArr[20];
  ifile>>cArr;
  cout<<cArr;
}
/*End of charArrFileInputText.cpp*/
```

Output

abc

Listing 7.9 Inputting a character array by using the extraction operator

Obviously, the extraction operator read up to the white space and stored the read value in the character array.

5. Extracting objects from input streams using the extraction operator: If we want to use the extraction operator for extracting objects of a particular class from the input stream, we have to overload it for that class. The concept of operator overloading, its need, and its use are elucidated in a later chapter.

The get() Function

The 'get()' function has been defined in the class 'istream'. It reads one byte from the input file and stores it in the character variable that is passed as a parameter to it.

The prototype of the 'get()' function is as follows:

```
istream & istream :: get(char &);
```

Suppose the contents of a file 'first.dat' are as follows:

```
abcd
```

```
/*Beginning of charFileInputText.cpp*/
#include<fstream.h>
void main()
{
  ifstream ifile("first.dat");
  char cVar;
  ifile.get(cVar);
  cout<<cVar;
}
/*End of charFileInputText.cpp*/
```

Output

a

Listing 7.10 Inputting a character by using the get() function

The getline() Function

The 'getline()' function reads one line from the input file. It has been defined in the class 'istream'.

The prototype of the 'getline()' function is:

```
istream & istream :: getline(char *, int, char = '\n');
```

It takes three parameters. The first parameter is the name of the character array in which the read line will be stored. The second parameter, an integer, signifies the number of bytes that will be read from the input file. The third parameter is the delimiting character whose presence in the stream of bytes that is being read from the input file prevents the 'getline()' function from reading further.

The 'getline()' function reads from the file that is attached with the object with respect to which it has been called till it reads bytes whose total count is one less than the value of the second parameter or till it encounters the delimiting character specified by the third parameter, whichever occurs earlier.

The following listing shows what happens when the 'getline()' function is used to read from the keyboard.

```
/*Beginning of getlineCin.cpp*/
#include<iostream.h>
void main()

{
  char cArr[20];
  cout<<"Enter a string: ";
  cin.getline(cArr,6,'#');
  cout<<"You entered: "<<cArr<<endl;
}
/*End of getlineCin.cpp*/
```

Output
Enter a string: **abcdefgh**<*enter*>
You entered: abcde

Output
Enter a string: **abc#defgh**<*enter*>
You entered: abc

Output
Enter a string: **aa bb cc**<*enter*>
You entered: aa bb

Listing 7.11 Using the getline() function to read from the keyboard

It can be observed that the 'getline()' function reads white spaces also. It is mentioned in the prototype that the 'getline()' function takes a default value, the newline character, for the third parameter. Thus, if the third parameter is not specified, it will continue to read till it encounters the newline character provided the number of bytes it has already read does not exceed the number specified by its second parameter.

The 'getline()' function reads from the keyboard buffer and leaves behind the unread bytes in the buffer itself.

The 'getline()' function works in a similar fashion when it reads from disk files. Suppose the contents of a file 'first.dat' are

```
abcdefgh
```

Now let us consider the following listing:

```
/*Beginning of getlineFile.cpp*/
#include<fstream.h>
void main()

{
  char cArr[20];
  ifstream ifile("first.dat");
  ifile.getline(cArr,6,'#');
  cout<<cArr<<endl;
}
/*End of getlineFile.cpp*/
```

Output
abcde

Listing 7.12 Using the getline() function to read from a disk file

If the contents are

```
abc#def
```

the output would be 'abc'.

Again, if the contents are

```
aa bb cc
```

the output would be 'aa bb'.

7.6 Binary Input/Output

Binary Output—The write() Function

The 'write()' function copies the values of variables from the memory to the specified output file. It works in binary mode.

As we already know, binary mode functions are not concerned about the data type of the variable that is output. They are only interested in the address of the variable (starting point of the block whose data needs to be output) and the size of the variable (total number of bytes to be output). The prototype of the 'write()' function makes this clear:

```
ostream & ostream :: write(const char *, int);
```

The 'write()' function has been defined in the class 'ostream'. It takes two parameters. The first parameter is the address of the variable whose value needs to be outputted. The second parameter is the size of the variable. The 'write()' function writes the value of the variable to the file that is associated with the object with respect to which it has been called.

Let us now discuss how the 'write()' function is used to output data of various types.

1. Inserting characters into output streams using 'write()' function: The following listing illustrates how the 'write()' function can be used to output the value of a character type variable to a disk file.

```
/*Beginning of writeCharDisk.cpp*/
#include<fstream.h>
void main()
{
  ofstream ofile("first.dat");
  char cVar;
  cVar = 'a';
  ofile.write(&cVar,sizeof(char));
}
/*Beginning of writeCharDisk.cpp*/
```

Listing 7.13 Using the 'write()' function to output character type value to a disk file

The following listing illustrates how the 'write()' function can be used to output the value of a character type variable to the monitor.

```
/*Beginning of writeCharConsole.cpp*/
#include<iostream.h>
void main()
{
  char cVar;
  cVar = 'a';
  cout.write(&cVar,sizeof(char));
}
/*End of writeCharConsole.cpp*/
```

Output

a

Listing 7.14 Using the 'write()' function to output character type value to the monitor

It is evident that there is no difference between outputting a character type value in text mode (insertion operator, 'put()' function) and in binary mode ('write()' function). There is no conversion in either case.

2. Inserting integers into output streams using 'write()' function: The following listing illustrates how the 'write()' function can be used to output the value of an integer type variable to a disk file.

```
/*Beginning of writeIntDisk.cpp*/
#include<fstream.h>
void main()
{
  ofstream ofile("first.dat");
  int iVar;
  iVar = 65;
  ofile.write((char *)&iVar,sizeof(int));
}
/*Beginning of writeIntDisk.cpp*/
```

Listing 7.15 Using the 'write()' function to output integer type value to disk file

As we have already discussed, the value contained in the four bytes that are occupied by 'iVar' will get copied to the designated output file without any transformation.

The following listing illustrates how the 'write()' function can be used to output the value of an integer type variable to the monitor.

```
/*Beginning of writeIntConsole.cpp*/
#include<iostream.h>
void main()
{
  int iVar;
  iVar = 65;
  cout.write((char *)&iVar,sizeof(int));
}
/*End of writeIntConsole.cpp*/
```

Output

A

Listing 7.16 Using the write() function to output integer type value to the monitor

It is interesting to understand the output of Listing 7.16. As a result of the second statement of the 'main()' function, the eight bits in the first of the four bytes occupied by 'iVar' are set to represent the binary equivalent of the number '65'. The bits in the remaining three bytes are set to zero. As we know, the monitor shows the ASCII equivalent of each of the bytes that are passed to it. The ASCII equivalent of '65' is 'A'. Hence, we get this output.

It is evident that there is an important difference between outputting an integer type value in text mode (insertion operator, 'put()' function) and in binary mode ('write()' function). In the former case, representation of the value that is read from the memory is transformed from base 2 to base 10 and then copied to the output file. There is no such conversion in the latter case.

3. Inserting floats and doubles into output streams using 'write()' function: Float and double type values are output in the same way in binary mode as integer type values.

4. Inserting strings into output streams using 'write()' function: The following listing illustrates how the 'write()' function can be used to output the value of a character array to a disk file.

```
/*Beginning of writeCharArrDisk.cpp*/
#include<fstream.h>
void main()
{
  ofstream ofile("first.dat");
  char cArr[10]="abcdefgh";
  ofile.write(cArr,sizeof(cArr));
}
/*Beginning of writeCharArrDisk.cpp*/
```

Listing 7.17 Using the write() function to output a string to a disk file

The name of the array that is passed as the first parameter to the 'write()' function represents its starting address. The second parameter represents the size of the memory block whose value is to be written into the output file.

The second parameter that is passed to the 'write()' function in Listing 7.17 evaluates to '10' (the size of the array). For this reason, the entire set of 10 bytes is copied verbatim to the specified output file. This includes the string itself, which is of eight characters, the delimiting NULL character (a single byte with all bits set to zero) that follows the string and one byte at the end with junk value.

If '5' is passed as the second parameter, only the first five bytes of the character array are written into the file.

The following listing illustrates how the 'write()' function can be used to output the value of a character array to the monitor.

```
/*Beginning of writeCharArrConsole.cpp*/
#include<iostream.h>
void main()
{
  char cArr[10] = "abcdefgh";
  cout.write(cArr,strlen(cArr));
}
/*End of writeCharArrConsole.cpp*/
```

Output

abcdefgh

Listing 7.18 Using the write() function to output a string to the monitor

5. Inserting objects into output streams using 'write()' function: The following listing illustrates one of the ways of inserting a class object into output streams in binary mode. In this method, value contained in the memory block that is occupied by a class object is copied to a specified output file.

```
/*Beginning of writeObjectDisk.cpp*/
#include<fstream.h>
class A
{
  /*
    definition of class A
  */
};
```

```
void main()
{
  A A1;
  ofstream ofile("first.dat");
  ofile.write((char *)&A1,sizeof(A));
}
/*End of writeObjectDisk.cpp*/
```

Listing 7.19 Using the write() function to output an object to a disk file

Of course, we will notice that the value of the object is being accessed directly by a non-member function—the 'main()' function. C++ does not prevent a direct access by means of such an explicit typecasting of an object's address. Statements like the following are allowed in C++.

```
char * cPtr = (char *)&A1;
```

A close look at this piece of code reveals a major drawback. Let us consider the case where an object of the class 'String' is used in the above listing instead of the object of the hypothetical class A. In such a case, the value of the pointer that is embedded within the object would get copied to the output file. However, the string that is contained in the memory and at which that pointer is pointing would not get copied.

```
/*Beginning of writeStringDisk.cpp*/
#include<fstream.h>
#include"String.h"  //header file that contains our class
                    //String
void main()
{
  ofstream ofile("first.dat");
  String s1("C++ is a wonderful language");
                    //s1.cStr points at the string
  ofile.write((char *)&s1,sizeof(String));
                    //The value of s1.cStr gets stored
                    //in the file. The string itself does
                    //not get stored.
}
/*End of writeStringDisk.cpp*/
```

Listing 7.20 Problem in using the write() function to output an object with an embedded pointer

If the value that is stored in the file through the program in Listing 7.20 is later read through another program, and stored in an object of the class 'String' declared therein, the pointer in that object would end up pointing at a place where the string itself no longer exists! The string itself would be lost in the memory and the entire purpose of storing the object would get defeated.

Client programs are not supposed to know how the actual data is managed, arranged, organized, and stored internally by the objects they are using (data abstraction). The conclusion is obvious. Objects should be responsible for outputting their own data. This conclusion becomes even more apparent if we consider the case of complex objects such as linked lists, vectors, trees, etc. where the object contains only the pointer to the first node of the data structure while the actual data structure remains outside it.

We will discuss some elementary methods of making objects capable of outputting their own data in one of the later sections of this chapter.

Binary Input—The read() Function

The 'read()' function copies the values from the specified input file to the memory block that is occupied by the target variable. It works in binary mode.

The logic mentioned in the introduction to the 'write()' function holds true in this case also. We can once again conclude that the 'read()' function accepts the address of the variable (starting point of the block into which the read data needs to be input) and the size of the variable (total number of bytes to be input). Accordingly, the prototype of the 'read()' function is as follows:

```
istream & istream :: read(char *, int);
```

The 'read()' function has been defined in the class 'istream'. It takes two parameters. The first parameter is the address of the variable into which the read value needs to be input. The second parameter is the size of the variable. The 'read()' function reads the value for the variable from the file that is associated with the object with respect to which it has been called.

Let us now discuss how the 'read()' function is used to input data of various types.

1. Extracting characters from input streams using 'read()' function: The following listing illustrates how the 'read()' function can be used to input the value for a character type variable by making it read from a disk file.

Suppose the contents of the file 'first.dat' are:

```
xyz
```

```
/*Beginning of readCharDisk.cpp*/
#include<fstream.h>
void main()
{
  ifstream ifile("first.dat");
  char cVar;
  ifile.read(&cVar,sizeof(char));
  cout<<cVar;
}
/*End of readCharDisk.cpp*/
```

Output

x

Listing 7.21 Using the read() function to input character type value from a disk file

The following listing illustrates how the read() function can be used to input the value of a character type variable by making it read from the keyboard.

```
/*Beginning of readCharConsole.cpp*/
#include<iostream.h>
void main()
{
  char cVar;
  cout<<"Enter a character: ";
  cin.read(&cVar,sizeof(char));
  cout<<cVar;
}
/*End of readCharConsole.cpp*/
```

Output

Enter a character: **a**<*enter*>

a

Listing 7.22 Using the read() function to input character type value from the keyboard

It is evident that there is no difference between inputting a character type value in text mode (extraction operator, 'get()' function) and in binary mode ('read()' function). There is no conversion in either case.

2. Extracting integers from input streams using 'read()' function: The following program illustrates how the 'read()' function can be used to input a value into an integer type variable by making it read from a disk file.

Suppose the first four bytes of a disk file 'first.dat' together contain the binary equivalent of number 64.

```
01000000
00000000
00000000
00000000
. . . .
. . . .
```

```
/*Beginning of readIntDisk.cpp*/
#include<fstream.h>
void main()
{
  ifstream ifile("first.dat");
  int iVar;
  ifile.read((char *)&iVar, sizeof(int));
  cout<<iVar<<endl;
}
/*End of readIntDisk.cpp*/
```

Output

64

Listing 7.23 Using the read() function to input integer type value from a disk file

---

As expected, the 'read()' function reads exactly four bytes from the input file that is associated with its invoking object. This is because the value of the second parameter that has been passed to it is four. It copies the read value into the memory block the address of whose first byte is equal to the first parameter passed to it.

Now let us look at a program in which the 'read()' function is used to read the value for an integer type variable from the keyboard. As per its known characteristics, the 'read()' function is expected to read four bytes from the keyboard, not convert the read bytes in any way and copy them into the four bytes that are occupied by the target integer type variable.

```
/*Beginning of readIntConsole.cpp*/
#include<iostream.h>
void main()
{
  int iVar;
  cout<<"Enter a number in base 2 format: ";
  cin.read((char *)&iVar, sizeof(int));
  iVar = iVar & 0x000000ff;    /*Inputting zeros in the
                               upper 3 bytes of the four
                               bytes of iVar*/
  cout<<iVar<<endl;
}
/*End of readIntConsole.cpp*/
```

Output

Enter a number in base 2 format: **ABCD**<*enter*>
65

Listing 7.24 Using the read() function to input integer type value from the keyboard

The explanation of this program has been left as an exercise for the reader.

3. Extracting floats and doubles from input streams using 'read()' function: Float and double type values are input in the same way in binary mode as integer type values.

4. Extracting strings from input streams using 'read()' function: The following program is a good example for illustrating the use of 'read()' function to read character arrays from a disk file.

Suppose the contents of a file 'first.dat' are as follows:

```
abcdefgh
```

The listing:

```
/*Beginning of readStringDisk.cpp*/
#include<fstream.h>
void main()
{
  ifstream ifile("first.dat");
  char cArr[20] = "12345678";
  ifile.read(cArr,3);
  cout<<cArr<<endl;
```

```
}
/*End of readStringDisk.cpp*/
```

Output
abc45678

Listing 7.25 Using the 'read()' function to input strings from disk files

The number '3' has been passed as the second parameter to the 'read()' function. It therefore reads only three characters from the file that is associated with the object that has called it.

As we know, the name of the array represents the starting address of the memory block occupied by it. Thus, the first parameter passed to the 'read()' function in the Listing 7.25 is the address of the first byte of the memory block that the array occupies. Therefore, the 'read()' function copies the three characters it has already read into the first three bytes of the array.

A similar method can be devised for using the 'read()' function to read character strings from the keyboard.

5. Extracting objects from input streams using 'read()' function: Let us consider the program of Listing 7.19 that was used to output the data of an object into a disk file. The object being output was a simple one, that is, it had no pointers as its data members. Thus, the actual data was stored in the file. Suppose the name of the file is 'first.dat'. The following listing illustrates how the data that was stored in the disk file can be loaded back in an object.

```
/*Beginning of readObj.cpp*/
#include<fstream.h>
class A
{
  /*
    definition of class A
  */
};
void main()
{
  ifstream ifile("first.dat");
  A A1;
  ifile.read((char *)&A1, sizeof(A));
}
/*End of readObj.cpp*/
```

Listing 7.26 Using the 'read()' function to input objects from disk files

Listing 7.26 was simple. Now let us take the case of complex objects, that is, objects having embedded pointers.

After reading the section 'Binary output—the write() function', it is natural to expect that the class of such complex objects will have a suitable function to output the external data structure into disk files. Thus, we can also expect the class to have a function that reads the entire data structure from disk files. Client programs should not and need not take this responsibility. We will discuss the techniques for defining such functions in one of the later sections of this chapter.

7.7 Opening and Closing Files

So far, we have output data to and input data from the same disk file by using two different programs. Data is usually output and input within the same program. For this, it is necessary to close the disk file after one operation before it is opened for another. The 'open()' and 'close()' functions that are provided as members of the library stream handling classes enable us to do this.

The open() Function

So far, we have opened files through the constructors of classes 'ifstream' and 'ofstream'. We can do this by invoking the 'open()' function also. This function has been provided in both of these classes.

The 'open()' function can be called by passing the name of the disk file to be opened as the only parameter.

```
. . . .
ofstream ofile;
ofile.open("first.dat");
. . . .
```

```
. . . .
ifstream ifile;
ifile.open("first.dat");
. . . .
```

A second parameter can also be passed to this function. This parameter is known as the open mode. It is an integer type value. There are a number of integer type constants defined in the stream handling library. Each of these constants, when passed as the second parameter to the 'open()' function, produces a different effect while opening the file. A list of these constants along with their use is given in Table 7.1 .

Table 7.1 Table of Open Mode Bits

Constant	***Meaning***
ios::app	For appending to end of file
ios::ate	For going to end of file on opening
ios::binary	For opening a binary file
ios::in	For opening file for reading only
ios::nocreate	For causing open to fail if the file does not exist
ios::noreplace	For causing open to fail if the file already exists
ios::out	For opening file for writing only
ios::trunc	For deleting contents of the file if it exists

The constructor of the class 'ofstream' and its overridden version of the 'open()' function takes 'ios::out' as the default value for the second parameter. Therefore, the file is opened for writing purpose only.

The constructor of the class 'ifstream' and its overridden version of the 'open()' function take 'ios::in' as the default value for the second parameter. Therefore, the file is opened for reading purpose only.

These constants can be meaningfully combined together to further influence the manner in which the file is opened. Using the bitwise OR operator does this.

```
ofstream ofile;
ofile.open("first.dat", ios::app | ios::nocreate);
```

In this example, the file would be opened for appending. But if the file does not exist already, the operation would fail. The method for detecting such failures is discussed in one of the later sections of this chapter.

The difference between 'ios::app' and 'ios::ate' is discussed in the section on 'seekp()' function.

The close() Function

A currently open file may need to be closed within a program. This need arises when we want to write into a file that we have already opened for reading and vice versa. An open file can be closed by calling the 'close()' function with respect to the object that has been used to open it.

The 'close()' function has been defined in the 'istream' class as well as the 'ostream' class. The following code snippet shows how the 'close()' function is used.

```
. . . .
ostream ofile;
ofile.open("first.dat");
. . . .
. . . .
ofile.close();
. . . .
```

7.8 Files as Objects of the fstream Class

The overloaded version of the 'open()' function for the class 'fstream' does not take a default value for the second parameter. We have to specify explicitly whether we want to open the file for writing or for reading. We can also specify that we want to open the file for both reading as well as writing.

```
. . . .
fstream iofile.
iofile.open("first.dat", ios::in | ios::out);
. . . .
```

In this example, the file will be opened for both reading and writing.

7.9 File Pointers

File pointers are created and maintained for open files during run time. There are two file pointers, the put pointer and the get pointer. The put pointer points at that byte of the open file where the next write operation will be conducted. The get pointer points at that byte of the open file where the next read operation will be conducted.

File pointers can be explicitly manipulated by the use of some functions that have been provided as members of the stream handling classes. An explanation of these functions follows.

The seekp() Function

This function is used to explicitly make the put pointer point at a desired position in the open file. It is important to note that by default the put pointer points at the beginning of the file if it is newly opened for writing. In case an existing file is opened for appending, the put pointer points at its end by default. Also, every write operation pushes forward the put pointer by the number of bytes written.

The 'seekp()' function has been defined in the class 'ostream'. It has two versions.

```
ostream & ostream :: seekp(streampos pos);

ostream & ostream :: seekp(streamoff off,
                                    ios::seek_dir dir);
```

In the first version, the 'seekp()' function takes only one parameter—the absolute position with respect to the beginning of the file. The type 'streampos' is type defined with `long` as the source data type. We must remember that the numbering of the position starts from zero.

In the following example, the put pointer is made to point at the second byte of the file:

```
ofile.seekp(1);      //ofile is an object of class ofstream
```

In the first version, the new position of the put pointer can be specified with respect to the beginning of the file only. But in the second version, the 'seekp()' function takes two parameters—the first parameter is the offset and the second parameter is the position in the open file with respect to which the offset is being specified. The type 'streamoff' is type defined with `long` as the source data type. The type 'ios::seek_dir' is an enumerated type with the following values:

ios::beg—offset will be calculated from the beginning of the file

ios::cur—offset will be calculated from the current position in the file

ios::end—offset will be calculated from the end of the file

In the following example, the put pointer is made to point at the last byte of the file. We must remember that the EOF character is actually the last byte of the file. Thus, in the following example, the put pointer will end up pointing at the last byte that was written into the file that is one byte to the left of the EOF character.

```
ofile.seekp(-1,ios::end);     //ofile is an object of class
                              //ofstream
```

Some more examples follow:

```
ofile.seekp(0,ios::beg); //take the put pointer to the
                         //beginning of the file
ofile.seekp(2,ios::beg); //take the put pointer to the
                         //third byte from the beginning
                         //of the file
ofile.seekp(-2,ios::cur);//take the put pointer two
                         //bytes to the left from the its
                         //current position in the file
ofile.seekp(2,ios::cur); //take the put pointer two bytes
                         //to the right from the its
                         //current position in the file
ofile.seekp(0,ios::end); //take the put pointer to the end
                         //of the file (past the
                         //last byte)
ofile.seekp(-1,ios::end);//take the put pointer to the last
                         //byte of the file
```

Let us now understand the difference between 'ios::app' and 'ios::ate' flags. Both of these flags open the file for appending and make the put pointer point at the end of the opened file by default (past the last existing byte). Neither of the two overwrites an

existing file. But the difference between the two is that while the flag 'ios::ate' allows you to rewind the put pointer and modify the existing contents of the file, the flag 'ios::app' does not allow this. In other words, if the file is opened using 'ios::app' flag, an attempt to use the 'seekp()' function for rewinding the put pointer will fail. The put pointer would continue to point at the end of the file. As bytes are appended to the file, the put pointer, as already mentioned, also moves forward. Thereafter, it cannot be rewound if the file was opened by using the 'ios::app' flag. But in case of 'ios::ate' flag, the put pointer can be rewound.

The tellp() Function

The 'tellp()' function returns the current position of the put pointer. It has been defined in the class 'ostream'.

```
streampos ostream::tellp();
```

In the following example, the current position of the put pointer is determined and stored in a program variable.

```
long pos = ofile.tellp();      //ofile is an object of the
                               //class ofstream
```

The seekg() Function

This function is used to explicitly make the get pointer point at a desired position in the open file. It is important to note that by default the get pointer points at the beginning of the file that is opened for reading. Every read operation pushes forward the get pointer by the number of bytes read.

The 'seekg()' function has been defined in the class 'istream'. It has two versions.

```
istream & istream :: seekg(streampos pos);

istream & istream :: seekg(streamoff off,
                           ios::seek_dir dir);
```

The explanation for these two versions of the 'seekg()' function is similar to the one provided for the corresponding versions of 'seekp()' function.

The tellg() Function

The 'tellg()' function, like the 'tellp()' function, returns the current position of the get pointer. It has been defined in the class 'istream'.

```
streampos istream::tellg();
```

In the following example, the current position of the get pointer is determined and stored in a program variable.

```
long pos = ifile.tellg();      //ifile is an object of the
                               //class ifstream
```

7.10 Random Access to Files

In random access, an intermediate record of a file is accessed directly without sequentially iterating through its neighboring records. We have already studied the tools necessary for accessing a record in a disk file at random, the 'seekp()' and 'seekg()' functions.

Suppose we have output integer type values into a disk file and we want to directly access the *n*th integer. We can do this sequentially by using a loop that starts iterating from the first record. This loop increments a counter after every read and stops when the counter indicates that the (*n*-1)th record has been read. At this point, the pointers would point at the *n*th record. But a more direct approach is to use either the 'seekp()' function or the 'seekg()' function, as the need may be, as follows:

```
iofile.seekp((n-1)*sizeof(int), ios::beg);   //iofile is an
                                             //object of
                                             //the class
                                             //fstream
```

This statement causes the file pointers to point at the *n*th record. At this point, if the write operation is conducted, the *n*th record would get modified.

Note that the technique works only if the size of all the records that are stored in the file is equal. This is possible only if binary data is stored in binary mode.

The size of the file and the number of records can also be found out very easily.

```
iofile.seekp(0,ios::end);      //iofile is an object of the
                               //class fstream
long lSize = iofile.tellp();
int iNoOfRec = lSize/sizeof(int);
```

In this example, the pointer is first forced to the end of the file. The current position of the pointer, since it points just past the last byte and the byte numbering starts from zero, denotes the size of the file in bytes. Dividing this size by the size of each record gives the number of records. Again, we must note that the technique works only if the size of all the records that are stored in the file is equal. This is possible only if binary data is stored in binary mode.

7.11 Object Input/Output Through Member Functions

We have realized that classes that have pointers that point at externally held data should also have the necessary functionality to output and input their data. Client programs of such classes should not be burdened with the responsibility of knowing how the data stored in the objects of such classes is organized.

Let us provide the 'String' class, which has been our running example so far, with the functionality to write its data into and read its data from disk files.

```
/*Beginning of string.h*/
#include<fstream.h>
class String
{
  public:
    . . . .
    . . . .
    void diskOut(ostream &);
    void diskIn(istream &);
    /*
      definition of the rest of class String
    */
};
/*End of string.h*/

/*Beginning of string.cpp*/
#include"string.h"
void String::diskOut(ostream & fout)
{
  fout.write((char *)&len, sizeof(int)); //output the
                                         //string's length
  for(int i = 0;i<len;i++) //output the string
  {
    fout.put(cStr[i]);
  }
}
void String::diskIn(istream & fin)
{
  String temp;
  fin.read((char *)&temp.len, sizeof(int)); //input the
                                            //string's length
  temp.cStr = new char[temp.len+1];
  for(int i = 0;i<temp.len;i++) //input the string
    fin.get(temp.cStr[i]);
  temp.cStr[I]='\0';
  *this = temp;
}
```

```
/*
  definition of the rest of the functions of class String
*/
/*End of string.cpp*/

/*Beginning of strDiskMain.cpp*/
#include"string.h"
void main()
{
  String s1;
  s1.setcStr("abcd");

  ofstream ofile("C:\\string.dat");
  s1.diskOut(ofile);
  ofile.close();

  String s2;
  ifstream ifile("C:\\string.dat");
  s2.diskIn(ifile);
  cout<<s2.getcStr()<<endl;
  ifile.close();
}
/*End of strDiskMain.cpp*/
```

Output

abcd

Listing 7.27 Input/output of objects through member functions

7.12 Error Handling

Every object of the class 'istream', 'ostream' or of a class that is derived from one of these two classes contains three flags that indicate state of the next byte in the associated file. These flags are:

- **eofbit**—becomes true if the end of file is encountered
- **failbit**—becomes true if the read/write operation fails (This in turn can be due to various reasons that are described shortly.)
- **badbit**—becomes true if the file being read is corrupt beyond recovery

The eof() Function

The 'eof()' function returns true whenever the file pointer encounters the end of file mark while reading the file that has been opened through the calling object. Whenever a stream library function, while reading from an input file, reaches the end of file mark, it sets the value of 'eofbit' to true.

```
while(!ifile.eof()) //read till end of file
{
  //statements to read from the file and operate upon the
  //read value
}
```

Note that the 'eof()' function returns the result of a past read. It does not look ahead before returning the result. Therefore, the test for end of file is given at the beginning of the loop.

The fail() Function

The 'fail()' function returns true if the file could not be opened for any reason. Whenever the 'open()' function fails to open a file, it sets the 'failbit' to true.

One reason that causes the 'open()' function to fail is the non-existence of the file that is being opened for reading or writing by using the 'ios::nocreate' flag.

```
ifstream ifile;
ifile.open("first.dat",ios::in | ios::nocreate);
if(ifile.fail())
{
  cout<<"File does not exist for reading\n";
  /*
    statements to take corrective action
  */
}
```

Another reason can be that the file is being opened for writing by using the 'ios::noreplace' flag but it already exists.

```
ofstream ofile;
ofile.open("first.dat", ios::out | ios::noreplace);
if(ofile.fail())
{
  cout<<"File already exists ... overwrite (y/n)?";
  /*
    statements to record user's response and take
    appropriate action
  */
}
```

Some more reasons that cause the read/write operation to fail follow:

- The file being opened for writing is read only.
- There is no space on the disk.
- The file being opened for writing is in a disk that is write-protected.

The bad() Function

The 'bad()' function returns true whenever a function that is reading from a file encounters a serious I/O error. Under such circumstances, the value of the 'badbit' flag gets set to true. It is best to abort I/O operations on the stream in this situation.

The clear() Function

The 'clear()' function is used to clear the bits returned by the 'bad()' function. This is necessary under a number of circumstances. The following listing illustrates one such circumstance.

```
/*Beginning of clearEof.cpp*/
#include<fstream.h>
void main()
{
  fstream iofile("first.dat",ios::in | ios::app);
  char cArr[100];
  int i=0;
  while(!iofile.eof())
  {
    iofile.get(cArr[i++]);
  }
  iofile.clear();
  for(int j=0;j<i;j++)
    iofile.put(cArr[j]); //append the contents of the file
                         //to itself
}
/*End of clearEof.cpp*/
```

Listing 7.28 The clear() function

We must note that the use of 'clear()' function was necessary in Listing 7.28. After the while loop ends, the 'eofbit' flag becomes true. Any further write operation on the file will fail if the 'clear()' function is not used.

7.13 Manipulators

Manipulators are used to format the output. C++ provides some pre-defined manipulators. The programmer can create his own application-specific manipulators too.

Manipulators can be inserted in an output stream just like values are inserted for output.

```
out << manip1 << manip2 << value1 << manip3 << value2;
```

In this example, 'out' is an object of the class 'ostream' or any of its derived classes. cout can also be used in place of 'out' to format the output to the monitor.

Pre-defined Manipulators

C++ provides a number of handy manipulators that are pre-defined in the header file 'iomanip.h'. Therefore, programs that use these manipulators must include this header file.

Some of the most commonly used pre-defined manipulators are listed in Table 7.2.

Table 7.2 Pre-defined manipulators

Manipulator	*Use*
setw(int w)	Set the field width to w
setprecision(int d)	Set the floating point precision to d
setfill(int c)	Set the fill character to c
setiosflags(long f)	Set the format flag to f
resetiosflags(long f)	Clear the flag specified by f

The setw() Manipulator

The 'setw()' manipulator takes an integer type variable as its only parameter. This parameter specifies the width of the column within which the next output will be output. If the value that is output after this manipulator is passed in the 'insertion' stream occupies less number of bytes than the specified parameter, then extra space will be created in the column that will contain the output value. These extra spaces will be padded by blanks or by the character that is passed as a parameter to the 'setfill()' function.

An example code snippet follows:

```
cout << 123 << endl;
cout << setw(3) << 10;
```

Output

```
123
 10
```

It is obvious that there is a blank space on the left of '10' in the second line of this output.

The 'setw()' manipulator has to be used separately for each item to be displayed.

```
cout << setw(5) << 10 << setw(5) << 234 << endl;
```

No truncation of data occurs if the parameter that is passed to the 'setw()' function is not sufficient to hold the data that is output subsequently. Instead, the padding requirement implied by the 'setw()' function is ignored.

```
cout << 123 << endl;
cout << setw(3) << 10000;
```

Output

```
123
10000
```

The setprecision() Manipulator

By default, C++ displays the values of float and double type with six digits after the decimal point. However, we can pass the number of digits we want to display after the decimal point as a parameter to the 'setprecision()' manipulator.

```
cout << setprecision(3)
     << sqrt(3) << endl
     << 1.14159 << endl;
```

Output

```
1.732
1.142
```

We must notice how the second output got rounded off to the nearest number.

Unlike the 'setw()' manipulator, the 'setprecision()' manipulator retains its effect even after outputting a value.

The setfill() Manipulator

By default, the 'setw()' manipulator pads any extra spaces it finds in the column that it has created with blank spaces. However, we can also specify the padding character by passing it as a parameter to the 'setfill()' manipulator.

```
cout << setfill('*')
     << setw(5) << 10
     << setw(5) << 234
     << endl;
```

Output

```
***10**234
```

The setiosflags() Manipulator

The 'setiosflags()' manipulator is also used to format the manner in which the output data is displayed. Two important parameters that it takes are 'ios::showpos' and 'ios::showpoint'.

The 'ios::showpos' flag, when passed as a parameter to the 'setiosflags()' manipulator, ensures that the positive sign is prefixed to numeric data when they are displayed.

```
cout << setiosflags(ios::showpos) << 10;
```

Output

```
+10
```

The 'ios::showpoint' flag, when passed as a parameter to the 'setiosflags()' manipulator, ensures that if the number of significant digits in the value being output is less than that specified by the 'setprecision()' manipulator, then the extra spaces obtained thereby are filled with zeros.

```
cout << setprecision(3)
     << 2.5 <<endl
     << setiosflags(ios::showpoint)
     << 2.5 << endl;
```

Output

```
2.5
2.500
```

The second line in the output highlights the effect of the 'setiosflags()' manipulator.

The resetiosflags() Manipulator

This manipulator cancels the effect of the parameter that was passed to an earlier call to the 'setiosflags()' manipulator. The output of the following code snippet shows how.

```
cout << setprecision(3)
     << 2.5 << endl
     << setiosflags(ios::showpoint)
     << 2.5 << endl
```

```
        << resetiosflags(ios::showpoint)
        << 2.5 << endl;
```

Output

```
2.5
2.500
2.5
```

User-defined manipulators

It is possible to create requirement-specific manipulators too. A programmer can create a manipulator to satisfy his specific needs. He/she can do this by defining a function as follows:

```
ostream & <manipulator> (ostream & out)
{
  //statements
  return out;
}
```

An example of a user-defined manipulator follows:

```
ostream & currency (ostream & out)
{
  out << "$. ";
  return out;
}
```

Now if we write

```
cout << currency << 20;
```

the output would be

```
$ 20
```

User-defined manipulators enable modularity. A user-defined manipulator can be used throughout an application to format the output in a uniform manner. If a change is required, it needs to be carried out at only one place—the definition of the manipulator—and again the change occurs uniformly throughout the application.

Summary

Streams are nothing but a flow of data to and from program variables. Input stream is the flow of data from a file on the permanent storage medium to program variables. The keyboard is also treated as source of input stream. Output stream is the flow of data to a file on the permanent storage medium from program variables. The monitor is also treated as target for output stream.

C++ provides us with a hierarchy of classes that have the functionality to implement various aspects of stream handling. The class 'ios' is the base class in this hierarchy.

The class 'ostream' is derived from the class 'ios' and handles the general output stream. The insertion operator (<<) is defined and overloaded in the class 'ostream' to handle output streams from program variables to output files.

The class 'ostream_withassign' is derived from the class 'ostream'. 'cout' is an object of the class 'ostream_withassign'. 'cout' stands for **c**onsole **out**put. As mentioned earlier, C++ treats all peripheral devices as files. It treats the monitor also as a file (for output stream). The object 'cout' represents the monitor.

The class 'istream' is derived from 'ios' and handles the general input streams. The extraction operator (>>) is defined and overloaded in the class 'istream' to handle inputs streams from input files to program variables.

The class 'istream_withassign' is derived from the class 'istream'. 'cin' is an object of the class 'istream_withassign'. 'cin' stands for **c**onsole **in**put. C++ treats all peripheral devices as files. It treats the keyboard also as a file (for input stream). The object 'cin' represents the keyboard.

The class 'iostream' is derived by multiple inheritance from the classes 'istream' and 'ostream'. It has the functionality to handle both input and output streams. The class 'ofstream' is derived from the class 'ostream'. It has the functionality to handle output streams to disk files. The class 'ifstream' is derived from the class 'istream'. It has the functionality to handle input streams from disk files. The class 'fstream' is derived from the class 'iostream'. It has the functionality to handle both input and output streams from and to disk files.

In text mode output, numeric data that exists in base 2 format in the memory variables, is first converted to base 10 format before being output. In binary mode, no such conversion occurs.

In text mode input, numeric data that is being input into a memory variable is reckoned to be in base 10 format. Therefore, it is first converted into base 2 format and then stored in the target memory variable.

The 'insertion' operator is used to output data in text mode. The 'put()' function is used to output a single character at a time. The 'extraction' operator is used to input data in text mode. The 'get()' function is used to input a single character at a time.

The 'write()' function is used to output data in binary mode. The 'read()' function is used to input data in binary mode.

Apart from the constructors of the library classes, the 'open()' function can also be used to open files. The first parameter that the constructor and the 'open()' function take is the name of the file. The second parameter specifies the open mode. Destructors of library stream classes close the files associated with them anyway. But the 'close()' function can be used to explicitly close files.

File pointers can be manipulated by the 'seekp()' and 'seekg()' functions. Their current positions can be determined by the 'tellp()' and 'tellg()' functions. An intermediate record can be directly accessed by using the 'seekp()' or 'seekg()' function to make the file pointer jump to a specific byte in the file. It is mandatory to use member functions for outputting and inputting data in case of complex classes.

Every object of the class 'istream', 'ostream' or of a class that is derived from either of these two classes, contains three flags that indicate state of the next byte in the associated file. These flags are:

- **eofbit**—becomes true if the end of file is encountered (The 'eof()' function returns the state of the **eofbit** flag.)
- **failbit**—becomes true if the read/write operation fails (The 'fail()' function returns the state of the **failbit** flag.)
- **badbit**—becomes true if the file being read is corrupt beyond recovery (The 'bad()' function returns the state of the 'badbit' flag.)

The 'clear()' function is used to clear the bits described above.

Manipulators are used to format the output. C++ provides some pre-defined manipulators. The programmer can create his own application-specific manipulators too.

Key Terms

streams

standard stream handling classes of C++

- ios
- ostream
- ostream_withassign
- istream
- istream_withassign
- iostream

- ofstream
- ifstream
- fstream

text mode input/output

cout

insertion operator

put() function

cin

extraction operator

get() function

write() function

read() function

open() function

close() function

seekp() function

seekg() function

tellp() function

tellg() function

eof() function and the eofbit flag

fail() function and the failbit flag

bad() function and the badbit flag

clear() function

manipulators

Exercises

1. Briefly describe the class hierarchy provided by C++ for stream handling.
2. State true or false.
   (i) 'cout' is an object of the class 'ostream_withassign'.
   (ii) The insertion operator (<<) is defined and overloaded in the class 'istream'.

(iii) The header file 'iostream.h' is included in the file 'fstream.h'.
(iv) The insertion operator outputs in binary mode.

3. What are text mode and binary mode input/output? What are their corresponding strengths and weaknesses?
4. What is the difference between a text file and a binary file?
5. Why should read operation on a file take place in the same mode in which the write operation has occurred? Explain.
6. How are values of various types output to disk files by using the insertion operator?
7. Describe the 'read()' and 'write()' functions, their prototype, use, and the way they input and output data.
8. How can a file be opened for both reading and writing?
9. What is the difference between opening a file using the constructor of the 'stream' class and the 'open()' function.
10. Describe how the contents of a disk file can be randomly accessed in C++.
11. Describe the circumstances under which each of the flags—'eofbit', 'failbit', and 'badbit'—becomes true.
12. Describe the use of the following manipulators:
    setw()
    setprecision()
    setfill()
    setiosflags()
    resetiosflags()
13. How can a programmer define his/her own manipulators?
14. Write a program to obtain as many integers from the user as he/she wants and write them into a disk file. After the user has finished entering the integers, read them from the file and display them on the monitor.
15. Create a class whose objects would hold linked lists of integers. Apart from the regular features of linked lists, the objects would also have the necessary functionality to download their data into a specified disk file and to upload their data from specified disk files. The application would be menu-driven. The user will have the option to save the linked list and to 'save as' the linked list.

Hint:

Create two classes as follows:

```
class intNode
{
    int data;
    intNode * Next;
  public:
    //functions to set and get the data members
};

class intList
{
    intNode * head;
  public:
    //functions to add, delete, modify, save and load
};
```

16. Write a manipulator that prefixes a currency symbol to the output value. For this, the manipulator should read the symbol from a disk file.

Chapter 8

Operator Overloading, Type Conversion, New Style Casts, and RTTI

OVERVIEW

Operator overloading is an extremely interesting feature of C++. It is not only interesting and exciting, but also an essential tool for the class designer. This chapter explains the following:

- the concept of operator overloading,
- the support provided by C++ for operator overloading,
- the need to overload operators,
- rules for operator overloading,
- use and misuse of operator overloading, and
- pitfalls in operator overloading.

The initial sections of the chapter give an overview of operator overloading. They contain only the skeleton code to illustrate the concepts without burdening the reader with the intricacies of the exact code. The exact code to overload various operators for various classes is dealt with in the later sections.

Type conversions from basic type to class type, from class type to basic type, and from one class type to another are also dealt with in this chapter.

C++ provides the following four new style cast operators to replace the use of the old error prone and difficult to detect C style casts:

- dynamic_cast
- static_cast
- reinterpret_cast
- const_cast

RTTI (Run Time Type Information) enables the programmer to find the type of object at which a pointer points during run time. Apart from the 'dynamic_cast' operator, C++ provides the 'typeid' operator for implementing RTTI.

The chapter ends with an explanation of new style cast operators and RTTI.

8.1 Operator Overloading

Let us first understand the meaning of operator overloading and how this useful feature of the C++ language is implemented.

Basic Definition of Operator Overloading

Overloading an operator means programming an operator to work on operands of types it has not yet been designed to operate. For instance, the addition operator can work on operands of type `char`, `int`, `float`, and `double`. However, if 's1', 's2', and 's3' are objects of the class 'String', which we have defined earlier, then the following statement

```
s3 = s1 + s2
```

will not compile unless the creator of class 'String' explicitly overloads the 'addition' operator to work on objects of his class. The method of implementing such overloading is described next.

Overloading Operators—The Syntax

Operators are overloaded by writing operator-overloading functions. These functions are either member functions or friend functions of that class whose objects are intended as operands of the overloaded operator. Operator overloading functions are very similar to the member functions and friend functions we have been reading about all along. The only thing peculiar about them is their name. The names of operator-overloading functions are composed of the keyword `operator` followed by the symbol of the operator being overloaded.

The syntax for member functions that overload a given operator is as follows:

```
class <class_name>
{
  <return_type> operator <op> (<arg_list>); //prototype
};

<return_type> <class_name> :: operator <op> (<arg_list>)
//definition
{
  //function body
}
```

Member functions that overload operators can be `private`, `protected`, or `public`. The prototype of the operator-overloading function specifies a return type (as do the 'normal' member functions). The keyword `operator` follows the return type. This in turn

is followed by the symbol of the operator being overloaded. Finally, a pair of parentheses containing the formal arguments is specified (as do the 'normal' member functions).

The syntax for a friend function that overloads a given operator is as follows:

```
class <class_name>
{
  friend <return_type> operator <op> (<arg_list>);
//prototype
};

<return_type> operator <op> (<arg_list>) //definition
{
  //function body
}
```

We already know that a friend function takes one argument more that the member function that serves the same purpose (because the invoking object appears as an explicit parameter to the friend function whereas in member functions it is passed as an implicit parameter). The same holds true in case of operator-loading functions.

The following examples will help in clarifying this syntax.

Suppose we want to overload the 'addition' operator (+) so that it can take objects of the class 'String' that we defined earlier. The exact syntax for this (in case of member function) would be as follows.

```
/*Beginning of String.h*/
class String
{
  public:
    String operator + (const String &) const; //prototype
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
String String :: operator + (const String & ss) const
                    //definition
{
  //function body
}
/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

```
/*Beginning of SomeProgram.cpp*/
#include"String.h"
void f()  //some function
{
  String s1,s2,s3;
  /*
    rest of the function f()
  */
  s3 = s1 + s2;
  /*
    rest of the function f()
  */
}
/*End of SomeProgram.cpp*/
```

Listing 8.1 Defining and using operator-overloading function as a member function

We can notice that the function has been declared as a public member of the class. This is because the operator will usually be used in its overloaded form within the non-member functions. The reasons for the return type and signature of this function will be discussed later. Moreover, the techniques of defining such functions will be demonstrated later.

If this function were to be declared as a friend, then the syntax would be as follows.

```
/*Beginning of String.h*/
class String
{
  friend String operator + (const String &,
                            const String &); //prototype
    /*
      rest of the class String
    */
};
/*End of String.h*/
/*Beginning of String.cpp*/
#include"String.h"
String operator + (const String & ss1, const String & ss2)
                    //definition
{
  //function body
}
/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

```
/*Beginning of SomeProgram.cpp*/
#include"String.h"
void f()  //some function
{
  String s1,s2,s3;
  /*
    rest of the function f()
  */
  s3 = s1 + s2;
  /*
    rest of the function f()
  */
}
/*End of SomeProgram.cpp*/
```

Listing 8.2 Defining operator-overloading function as a friend function

How Does the Compiler Interpret the Operator-Overloading Functions?

It is important to understand how the compiler interprets operator-overloading functions.

The statement

```
s3 = s1 + s2;  //s1, s2 and s3 are objects of the class
               //String
```

is interpreted as

```
s3 = s1.operator + (s2);
```

If the operator-overloading function has been declared as a member function, then this interpretation is satisfied. Otherwise, the statement is interpreted as

```
s3 = operator + (s1, s2);
```

If the operator-overloading function has been declared as a friend function, then this interpretation is satisfied. Otherwise, the compiler reports an error to the effect that the given operator has not been overloaded for the class. It is interesting to note the compiler does not say that invalid operands have been passed to the operator!

So far, we have seen that the operators have been overloaded within the classes using member functions or friend functions. These functions are compiled and stored in the library.

We have also seen that the overloaded operators have been used within the applications using their usual syntax. As described in this section, the compiler first converts the statements where the overloaded operators are used. However, we must note that the

operator-overloading functions can also be called directly from within the application programs (the way the compiler finally interprets it). Operator-overloading functions can be called directly as follows,

```
s3 = s1.operator + (s2);        //in case of member function
```

or

```
s3 = operator + (s1, s2);       //in case of friend function
```

The benefit of overloading the operator will not be felt if the overloaded operators are directly called in this manner. (In that case, they can be very well replaced by ordinary member functions.) Moreover, we must note that only the name of the operator-overloading function is unusual (it contains the keyword `operator`). Otherwise, the operator-overloading functions are implemented just like ordinary member, non-member, or friend functions.

Why Overload using Friend Functions?

We might wonder why friend functions are used to overload operators. After all, member functions seem to serve the purpose. In order to understand this, let us consider two classes A (which we have defined) and B (an existing class or an intrinsic data type). We realize that for some reason only an object of class A will be added to an object of class B to get another object of class A. This will be done as follows.

```
a2 = b1 + a1;          //a1, a2 are objects of class A, b1 is
                       //an object of class B
```

An object of class B will not be added to an object of class A. Objects of class B will appear on the left of the 'addition' operator and not on the right. We will soon realize that such restrictions can and do exist. Statements such as the one that follow will not be written.

```
a2 = a1 + b1;          //a1, a2 are objects of class A, b1 is
                       //an object of class B
```

Further let us assume (rather accept) that we have no means of modifying the definition of class B. (This is a perfectly acceptable restriction. We cannot somebody else's class definition. Class definitions are provided in read-only header files and definitions of member functions in libraries.) Now, if we define the operator-overloading function as a member function of class A as follows, the first of the two preceding statements will not compile.

```
class A
{
  public:
    A operator + (const B &);
};
```

The compiler will interpret the statement

```
a2 = b1 + a1;
```

first as

```
a2 = b1.operator + (a1);
```

and then as

```
a2 = operator + (b1,a1);
```

The prototype of the member function satisfies neither of these two interpretations. The compiler will naturally throw an error. Declaring the operator-overloading function as a friend function with an object of class B as the first formal argument solves the problem.

```
class A
{
  public:
    friend A operator + (const B &, const A &);
                          //prototype
};

A operator + (const B & bb, const A & aa)     //definition
{
  //function body
}
```

Listing 8.3 Operator overloading using friend function

It is interesting to note that the compiler throws an ambiguity error if both member function and friend function are used to overload an operator. This is because both of them will satisfy calls to the overloaded operator. The compiler will certainly be in no position to decide with which function such a call is to be resolved.

Overview of Overloading Unary and Binary Operators

Member functions that overload unary operators take no operands. This is because apart from the calling object, no other parameter is passed to the operator and the calling object is passed as an implicit parameter to the object. Friend functions that overload unary operators will naturally take one parameter since the calling object will be passed as an explicit parameter to it.

Similarly, member functions that overload binary operators will take one parameter. This is because apart from the calling object, another value will be passed to the operator as an operand (binary operators take two operands). The calling object will itself be passed to the function as an implicit parameter. Again, friend functions that overload binary operators will take one operand more, that is, two operands. We can very well explain this.

Why are Operators Overloaded?

Let us now find out the need to overload operators. After all, the operator-overloading functions can be so easily substituted by member functions or friend functions with ordinary but meaningful and relevant names. For example, the operator-overloading function to overload the 'addition' operator (+) for objects of the class 'String' can be easily replaced by a member function of a proper name.

```
class String
{
  public:
    //String operator + (const String &);
    String add(const String &);     //prototype
};

String String :: add(const String & ss) //definition
{
  //function body
}

void f()  //some function
{
  String s1,s2,s3;
  /*
    rest of the function f()
  */
  s3 = s1.add(s2);
  /*
    rest of the function f()
  */
}
```

Listing 8.4 Using an ordinary member function to substitute an operator-overloading function

The definition of the 'String :: add()' function can be the same as the operator- overloading function to overload the 'addition' operator (+).

However, operator overloading becomes mandatory under the following circumstances:

- Objects of the class acquire resources dynamically during run time and no two objects should share the same copy of the resource.
- Objects of the class acquire some resources dynamically during run time and no two objects should share even different copies of the resource.
- Objects need to be passed as parameters in function templates and the operators being used on template class objects within the template functions should work in the same way on objects of the class.
- The default action of the dynamic memory management operators (`new` and `delete`) are unsuitable for the class being designed.
- Change in the implementation of the class forces an undesirable change in its interface in turn necessitating the rewriting and recompiling of the application programs.
- Overloading provides better readability of code. Although this is a somewhat weak reason, it, nevertheless, is a factor that can be considered. The statement

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

is much more readable than say a statement such as

```
o2 = o1.pre_fix_increment();
```

Let us understand these circumstances one by one. For understanding the first case, let us reconsider the class 'String'. Let us try to visualize what happens at the end of the following block of code.

```
String s1("abc"), s2;
s2 = s1;
```

Listing 8.5 Undesirable default action of the assignment operator

As a result of the second statement in Listing 8.5, the following scenario emerges.

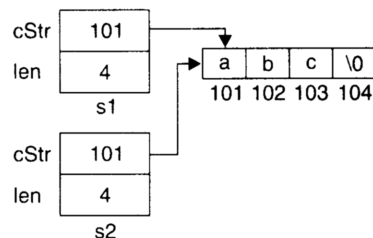

Diagram 8.1 Diagram depicting the drawback in the default action of the assignment operator

As a result of the second statement in Listing 8.5, the pointers embedded in both the objects point at the same dynamically allocated memory block. The default action of the assignment operator simply copies the value of the pointer embedded in 's1' into the pointer embedded in 's2'.

The problems that arise out of such a situation have already been discussed in Chapter 4. We will notice that the same undesirable situation arose due to the initial absence of a suitable copy constructor in the class 'String'. This had prompted us to define a suitable copy constructor for the class 'String'. The same factors dictate that a suitable function to overload the assignment operator be defined for the class 'String'. Instead of the default action of the assignment operator, execution of this function will take place when statements such as the second one in Listing 8.5 are executed.

For understanding the second circumstance where operator overloading is mandatory, let us imagine that there is a class whose objects should not share even separate copies of dynamically allocated resources. This means that statements such as the following one should not compile at all.

```
o1 = o2;   //o1, o2 are objects of the said class
```

Here the solution is quite simple. We just declare the function to overload the assignment operator in the private section of the class. Any use of the assignment operator within a non-member function will launch a call to this operator-overloading function. Since the function is private, such a call will throw a compile-time error. As desired, the use of the assignment operator will be prevented. However, what would happen if we inadvertently use the assignment operator within a member function or a friend function? The private nature of the function will not be enough to prevent such a call. However, even such calls can be prevented by not defining the function to overload the assignment operator. This trick will make the linker throw an error.

To understand the third circumstance where operator overloading is mandatory, we require the knowledge of function templates, which are discussed in the next chapter.

Now, let us understand the fourth circumstance. The 'new' operator does a number of things by default, some, or all of which might be undesirable for the class being designed.

By default, the 'new' operator throws an exception if it fails to allocate the amount of memory requested (exceptions are dealt with in one of the later chapters). However, this default action of the 'new' operator may be unsuitable for the class being designed. In response to this out-of-memory condition, the class designer might instead need to call one of the member functions of the class. Only overloading the 'new' operator can fulfill this need.

Also by default, the 'new' operator not only allocates the amount of memory requested, it also stores the amount of memory allocated in the memory itself. This enables the 'delete' operator (if it is called) to find out the size of the memory allocated so that it can

then deallocate the same amount of memory (see Chapter 3). However, in memory critical applications, such expenditure of memory might be prohibitive. If the class designer knows that the same amount of memory will be allocated whenever the 'new' operator is called, he/she can cleverly prevent this wastage of memory. Again, only overloading the 'new' operator can do this.

Further, by default, the 'new' operator simply allocates memory for the object whose type is passed as an operand to it. However, the class designer would not want that the class should ever have more than one object. He/she may want that an object should be created only when the 'new' operator is called for the very first time. Subsequent calls to the 'new' operator should not create more objects. Instead, such subsequent calls should merely return the address of the object that was created in response to the first call to the 'new' operator.

The last circumstance that mandates operator overloading is self-explanatory.

Rules for Operator Overloading

The following rules must be observed while overloading operators.

- **New operators cannot be created:** New operators (such as **) cannot be created. For example, the following piece of code will produce a compile-time error.

```
class A
{
  public:
    void operator ** ();
};
```

Listing 8.6 An illegal attempt to create a new operator

- **Meaning of existing operators cannot be changed:** Any operator overloading function (member or friend) should take at least one operand of the class of which it is a member or friend. Thus, it is not possible to change the manner in which an existing operator works on operands of fundamental types (`char`, `int`, `float`, `double`).

  In case of member functions, this condition is automatically enforced because the address of the calling object is implicitly passed as a parameter to it. However, in case of friend functions, the library programmer needs to take extra care. For example, the following piece of code (Listing 8.7) will not compile.

```
class A
{
  public:
    friend int operator + (int, int);    //ERROR: will not
                                          //compile
};
```

Listing 8.7 An illegal attempt to modify the behavior of operators on intrinsic types

As we can see, by ensuring that at least one operand of an operator-overloading function must be of the class type, the compiler ensures that the meanings of the existing operators cannot be changed. If the code in Listing 8.7 had compiled, the statement

```
z = x + y;      //x, y, z are integer type
```

could have invoked the 'operator + ()' function of the class A. Of course, this is undesirable.

- **Some of the existing operators cannot be overloaded:** The following operators cannot be overloaded:

  :: (scope resolution)
  . (member selection)
  .* (member selection through pointer to member)
  ?: (conditional operator)
  sizeof (finding the size of values and types)
  typeid (finding the type of object pointed at)

- **Some operators can be overloaded using non-static member functions only:** The following operators can be overloaded using non-static member functions alone.

  = (Assignment operator)
  () (Function operator)
  [] (Subscripting operator)
  -> (Pointer-to-member access operator)

  These operators cannot be overloaded using friend functions or static functions.

- **Number of arguments that an existing operator takes cannot be changed:** Operator-overloading functions should take the same number of parameters that the operator being overloaded ordinarily takes. For example, the division operator

takes two arguments. Hence, the following class definition causes a compile-time error 'operator / takes too few arguments' for the operator-overloading function.

```
class A
{
  public:
    void operator / ();
};
```

Listing 8.8 An illegal attempt to modify the number of arguments that an operator takes by default

- **Overloaded operators cannot take default arguments:** The following class definition causes a compile-time error 'operator/cannot take default arguments' for the operator-overloading function.

```
class A
{
  public:
    void operator / (int = 0);
};
```

Listing 8.9 An illegal attempt to assign a default value to an argument of an operator-overloading function

Finally, we must note that it is highly imprudent to modify the values of the operands that are passed to the operator-overloading functions. To appreciate this point better, let us consider the function to overload the 'addition' operator for the class 'String'.

```
class String
{
    char * cStr;
    long int len;
  public:
    String operator + (String &);
};
```

The library programmer may mistakenly write some statements to modify the value of the implicit or the explicit parameter of the 'String :: operator + ()' function.

```
String String :: operator + (String & ss)
{
  /*
    rest of the function String :: operator + ()
  */
  this->cStr = NULL;        // BUG: left-hand parameter
                            //changed!
  /*
    rest of the function String :: operator + ()
  */
  ss.cStr = NULL;           //BUG: right hand parameter
                            //changed!
  /*
    rest of the function String :: operator + ()
  */
}
```

Listing 8.10 Modifying the left-hand side and the right-hand side operands of the addition operands in the function to overload it

To guard against this mishap, the operator-overloading function can be declared as follows.

```
class String
{
    char * cStr;
    long int len;
  public:
    String operator + (const String &) const;
};
```

Listing 8.11 Making necessary use of the 'const' keyword to prevent bugs

Neither of the statements in Listing 8.10 that have bugs will compile.

Let us now see how operators are actually overloaded.

8.2 Overloading the Various Operators

Overloading the Increment and the Decrement Operators (Prefix and Postfix)

Let us recollect the class 'Distance'. We can overload the increment operator for objects of the class. What would we like such a function to do? If 'd1' and 'd2' are objects of the class 'Distance', then the following statement

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