We would also like to add a function that will replace the string associated with an object with the string that we pass to it. We let this function be

```
void String::setString(const char * const);
```

Suppose the following statements are executed.

```
String s1("abc");
s1.setString("def");      //replace "abc" by "def"
```

Then the following events should take place when the second statement executes ('s1.cStr' is already pointing at a memory block that contains the string 'abc' and is not NULL).

1. A block of four bytes should be dynamically allocated to accommodate the string "def"
2. The string "def" should get written in that memory block with the null character appended.
3. 's1.cStr' should be made to point at this new block of memory
4. The block of memory at which 's1.cStr' was previously pointing should be deallocated to prevent memory leak.

The formal argument of the 'String::setString()' function is a `const char * const`. The reasons for this have already been discussed under the section on parameterized constructor. We may think that the definition of this function will be the same as that of the constructor. But this is not so. When the constructor starts executing 'cStr' may or may not be NULL (it may contain junk value). But if it is not NULL, it does not mean that it is pointing at a dynamically allocated block of memory. But when the 'String::setString()' function starts executing, if 'cStr' is not NULL, then it is definitely pointing at a dynamically allocated block of memory. Statements to check this condition and to deallocate the memory block and to nullify 'cStr' and to set 'len' to zero should be inserted at the beginning of the 'String::setString()' function. Otherwise a memory leak will occur. Defining the 'String::addChar()' and 'String::setString()' functions is left as an exercise.

Let us think of more such relevant functions that can be added to the class 'String'. There can be a function that will change the value of a character at a particular position in the string at which the pointer of the invoking object points. Moreover, there can be a function that reads the value from a particular position in the string at which the pointer of the invoking object points. These functions can have built-in checks to prevent values from being written to or read from bytes that are beyond the memory block allocated. Again, such a check is not built into character arrays. The application programmer has to put in extra efforts on his/her own to prevent the program from exceeding the bounds of the array.

After we have added all such functions to the class 'String', we will get a new data type that will be safe, efficient, and convenient to use.

Suitably defined constructors and destructors have a vital role to play in the creation of such data types. Together they ensure that

- There are no memory leaks (the destructor frees up unwanted memory).
- There are no run-time errors (no two calls to the destructor try to free up the same block of memory).
- Data is never in an invalid state and domain constraints on the values of data members are never violated.

After such data types have been defined, new data types can be created that extend the definitions of existing data types. They contain the definition of the existing data types and at the same time add more specialized features on their own. This facility of defining new data types by making use of existing data types is known as inheritance. Chapter 5 deals with this feature of OOPS and its implementation in C++.

Summary

Constructors can be used to guarantee a proper initialization of data members of a class. Domain constraints on values of data members can be implemented via constructors.

Constructors are member functions and have the same name as that of the class itself. The compiler creates a zero-argument constructor and a copy constructor if we do not define them. Constructors take parameters and, therefore, can be overloaded. They do not return anything (not even void). The compiler implicitly embeds a call to the constructor for each object that is being created. An explicit call to the constructor for an existing object is forbidden.

If necessary, destructors can be used to guarantee a proper clean up when an object goes out of scope. Destructors are member functions and have the same name as that of the class itself but with the tilde sign prefixed. The compiler creates a destructor if we do not define one. Destructors do not take parameters and, therefore, cannot be overloaded. They do not return anything (not even void). The compiler implicitly embeds a call to the destructor for each object that is going out of scope (being destroyed). An explicit call to the destructor for an existing object is forbidden.

Key Terms

constructors

- called automatically for each object that has just got created

- defined by default
- has the same name as that of the class
- does not return anything

zero-argument constructor

parameterized constructors

copy constructor

destructors

Exercises

1. What are constructors? When are they called? What is their utility?
2. Why should the formal argument of a copy constructor be a reference object?
3. What are destructors? When are they called? What is their utility?
4. Is a destructor necessary for the following class?

```
class Time
{
    int hours, minutes, seconds;
  public:
    /*
      rest of the class Time ... but no more data members
    */
};
```

5. Define a suitable parameterized constructor with default values for the class 'Time' given in question 4.
6. Four member functions are provided by default by the compiler for each class that we define. We have studied three of them in this chapter. Name them.
7. State true or false.

   (i) Memory occupied by an object is allocated by the constructor of its class.
   (ii) Constructors can be used to acquire memory outside the objects.
   (iii) Constructors can be overloaded.
   (iv) A constructor can have a return statement in its definition.
   (v) Memory occupied by an object is deallocated by the destructor of its class.
   (vi) Destructors can be used to release memory that has been acquired outside the objects.
   (vii) Destructors can be overloaded.
   (viii) A destructor can have a return statement in its definition.

8. The copy constructor has been explicitly defined for the class 'String' so that no two objects of the class 'String' end up sharing the same resource, that is, end up with their contained pointers pointing at the same block of dynamically allocated memory. In this case, two such blocks may contain two copies of the same data as a result of the copy constructor, which is perfectly acceptable. However, there are situations where no two objects should share even copies of the same data. If A is a class for whose objects this restriction needs to be applied, then we should ensure that a statement like the second one below should not compile.

```
A A1;
A A2 = A1;
```

How can this objective be achieved? (*Hint:* Member functions are not always public and the copy constructor is a member function.)

Chapter 5

Inheritance

OVERVIEW

This chapter discusses inheritance. Inheritance is one of the most important and useful features of the object-oriented programming system.

The chapter begins with an overview of inheritance. Basic concepts such as base class and derived class are discussed. The effects, advantages, and important points of inheritance are also discussed.

The middle portion of the chapter deals with the implications of making a base class pointer point at an object of the derived class and vice-versa. Thereafter, the concept of function overriding is discussed. This is followed by a section on base class initialization in which the method of initializing base class members via constructors of the derived class is discussed.

The protected keyword is an important concept in C++. The protected keyword, along with the public and private keywords, completes the triad of access specifiers provided by C++. A separate section of this chapter elucidates this keyword and the effect of its use in inheritance.

Classes can be derived by public, private, or protected keywords. The effect caused by each of these is different. The current chapter compares this difference in a systematic manner.

Inheritance can be of various types based upon the number of classes derived from a single base class and the number of base classes for a single derived class. All of these types are dealt with in the penultimate section of the chapter.

The chapter ends with a section on the order of invocation of constructors and destructors.

5.1 Introduction to Inheritance

Inheritance is a very useful feature of OOPS that is supported by C++. A class may be defined in such a way that it automatically includes member data and member functions of an existing class. Additionally, member data and member functions may 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 existing class is known as its derived class or child class or sub-class. The syntax for derivation is as follows.

```
class <name of derived class> : <access specifier> <name of
base class>
{
  /*
    definition of derived class
  */
};
```

Suppose a class A already exists. Then a new class B can be derived from class A as follows.

```
class B : public A
{
  /*
    new features of class B
  */
};
```

The public access specifier has been used in the foregoing example. The implications of using the other access specifiers are discussed later in this chapter.

A pointer from the derived class to the base class diagrammatically depicts derivation (see Diagram 5.1).

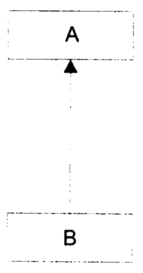

Diagram 5.1 Diagrammatic depiction of inheritance

Effects of Inheritance

Inheritance affects the size and behavior of derived class objects in two ways.

- Obviously, an object of the derived class will contain all data members of the derived class. However, it will contain data members of the base class also. Thus, an object of the derived class will always be larger than an object of the base class. (The only exception to this is when neither the base class nor the derived class has data members. In that case, objects of both the base class as well as the derived class occupy one byte each.)
- Obviously, with respect to an object of the derived class, we can call the public member functions of the derived class in any global non-member function. However, we can call the public member functions of the base class also. (There are exceptions to this. Circumstances under which these exceptions occur are described later in this chapter.)

Listing 5.1 illustrates this.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    void setX(const int=0);
    int getX()const;
};
/*End of A.h*/

/*Beginning of B.h*/
#include"A.h"
class B : public A  //inheriting from A
{
    int y;
  public:
    void setY(const int=0);
    int getY()const;
};
/*End of B.h*/

/*Beginning of inherit.cpp*/
#include<iostream.h>
#include"B.h"
void main()
{
  cout<<sizeof(A)<<endl
      <<sizeof(B)<<endl;
```

x = 4 y = 7

```
  B B1;          //an object of the derived class

  B1.setX(1);   //OK: calling a base class member function
                //with respect to a derived class object
  B1.setY(3);

  cout<<B1.getX()<<endl; //OK: calling a base class member
                         //function with respect to a
                         //derived class object
  cout<<B1.getY()<<endl;
}
/*End of inherit.cpp*/
```

Output

```
4
8
1
3
```

Listing 5.1 Effects of inheritance

Defining the member functions of classes A and B is left as an exercise.

This highly simplified example (Listing 5.1) effectively illustrates the basic mechanisms of inheritance. An object of class B (the derived class) will contain two integers (one from class B and the other from class A). Therefore, its size will be 8. Also, with respect to an object of class B, we can call member functions of class B as well as those of class A.

An object of the derived class will contain the data members of the base class as well as the data members of the derived class. Thus, the size of an object of the derived class will be equal to the sum of sizes of the data members of the base class plus the sum of the sizes of the data members of the derived class.

Inheritance implements an 'is-a' relationship. A derived class is a type of the base class just like an aircraft (derived class) is a type of vehicle (base class). Contrast this to containership that implements a 'has-a' relationship. A class may contain an object of another class or a pointer to a data structure that contains a set of objects of another class. Such a class is known as a container class. For example, an aircraft has one engine or an array of engines.

Another example can be that of a 'manager' class and 'employee' class. A 'manager' (that is, an object of the class 'manager') is an 'employee' (that is, an object of the class 'employee'). Nevertheless, it has some features that are not possessed by all employees.

For example, it may have a pointer to an array of employees that report to him. Derived class object is also a base class object (as shown in the following lines of code).

```
class employee
{
    String name;
    double basic;
    Date doj;
    /*
      rest of the class employee
    */
};

class manager : public employee     //manager is an employee
{
    employee * list;
    /*
      rest of the class manager
    */
};
```

A derived class contains additional data and members and is thus a specialized definition of its base class. Therefore, the process of inheritance is also known as specialization.

Benefits of Inheritance

This process of adding only the additional data members in the derived class has implications. The base class can have a generic common definition. The data and functions that are common to more than one class can be put together in the base class. While only the special ones can be put in each of the derived classes. Thus, inheritance is another feature of C++ that enables code reusability.

Inheritance in Actual Practice

In actual practice, the library programmer defines a certain class and its member functions. Another interested programmer, in order to create his/her application, then inherits from this class and adds only the special data members and the code to handle these additional data members in the derived class.

Base Class and Derived Class Objects

Now, many students of C++ may start believing that objects of the derived class inherit from objects of the base class. This is incorrect as an object of the derived class is not at all related to another simultaneously existing object of the base class.

An object of class A (say 'A1') will occupy four bytes containing only 'x'. Whereas an object of class B (say 'B1') will occupy a different block of eight bytes containing both 'x' and 'y'. (As per the definitions of class A and B given in Listing 5.1)

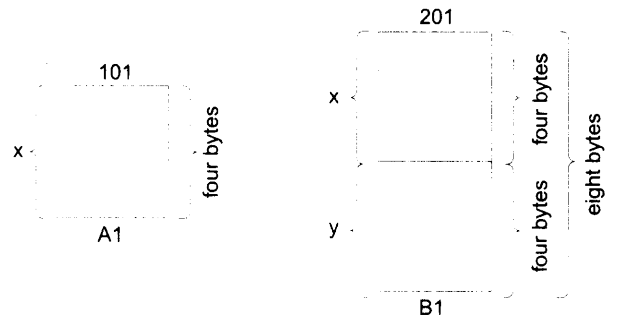

Diagram 5.2 Memory layout of base class and derived class object

Accessing Members of the Base Class in the Derived Class

Only public members of base class can be accessed in the functions of derived class (protected members of the base class can also be accessed; we shall discuss protected members later). But, private members of the base class cannot be accessed.

Suppose in the 'B::setY()' function we write

```
x=y;
```

the compiler will report an error stating that private members of the base class cannot be accessed. (In this case we are trying to access 'x' in a member function of the derived class. But 'x' is a private member of the base class).

But we can access 'A::setX()' and 'A::getX()' functions in the member functions of the derived class because they are public members of the base class. Private members of the base class remain private with respect to member functions of the derived class. The following lines of code demonstrate this.

```
void B::setY(const int q)
{
  y=q;
  setX(y);       //x=y
}
```

This is as it should be. C++ prevents us from accessing private members of the base class in member functions of the derived class to fully implement data security. After all, the base class provider made some members of the base class private because he/she wanted only the member functions of the base class (which he/she has perfected) to access them. If member functions of the derived class are allowed to access private members of the base class, then one cannot identify all statements in the program that access private members of the base class by merely looking at its member and friend functions.

We may argue that the existing set of functions of the base class is sometimes not enough. We would like to create a derived class that supplements the base class by containing those member functions that we feel are *missing* in the base class. For example, suppose a function such as 'String::addChar(`char`)' is not present in the class 'String'. But in this case, the drawback is in the base class itself. It is the base class itself that should be corrected. Inheritance is not used to remove such lacuna. It is used to provide additional data and additional code to work upon the additional data in the derived class. Inheritance is used to add facilities to an existing class without reprogramming it or recompiling it. Thus, it enables us to implement code reusability.

Friendship is not inherited. A class does not become a friend to a class to which its parent is a friend. Listing 5.2 illustrates this.

```
/*Beginning of friendInherit.cpp*/
class B;

class A
{
  friend class B;
  int x;
};

class B
{
  void fB(A * p)
  {
    p->x=0;     //OK: B is a friend of A
  }
};

class C : public B
{
  void fC(A * p)
  {
    p->x=0;     //ERROR: C is not a friend of A
                //despite being derived from a friend
  }
};
/*End of friendInherit.cpp*/
```

Listing 5.2 Friendship is not inherited

5.2 Base Class and Derived Class Pointers

A base class pointer can point at an object of the derived class. However, a derived class pointer cannot point at an object of the base class. To be more precise, a base class pointer can safely point at an object of the derived class without the need for typecasting. But a derived class pointer can be made to point at an object of the base class only forcibly by typecasting. However, this can cause run-time errors.

Note: There are exceptions to the above assertion. The compiler will prevent a base class pointer from pointing at an object of the derived under certain circumstances. These circumstances are described later in this chapter.

First, let us understand why no harm can come by making a base class pointer point at an object of the derived class. To understand this, we consider the classes of Listing 5.3.

```
/*Beginning of A.h*/
class A
{
  public:
    int x;
};
/*End of A.h*/

/*Beginning of B.h*/
#include"A.h"
class B : public A //derived from A
{
  public:
    int y;
};
/*End of B.h*/
```

Listing 5.3 A derived class and its base class

These classes have only data members. There are no member functions and even these member data are public. These classes are given here to initially understand the concepts only. Explanations with classes that have private data members and public member functions are given later.

Now, let us compile the following 'main()' function and see what happens.

```
/*Beginning of BasePtr01.cpp*/
#include"B.h"  //from listing 5.03
void main()
{
  A * APtr;
  B B1;
  APtr=&B1;      //line 1 - OK: base class pointer points at
                 //derived class's object
  APtr->x=10;    //line 2 - OK: accessing base class member
                 //through base class pointer
  APtr->y=20;    //line 3 - ERROR: y not found in class A
}
/*End of BasePtr01.cpp*/
```

Listing 5.4 Base class pointer pointing at a derived class object

Line 1 of Listing 5.4 will compile because line 3 will not. A base class pointer can point at an object of the derived class. Let us see why. 'APtr' is of type 'A *'. It is supposed to point at objects of the base class A. Therefore, it cannot access 'y'. There is no member of the name 'y' in class A. The fact that 'APtr' points at an object of the derived class is of no significance. Through 'APtr', it is possible to access 'x' because there is a member of the name 'x' in class A. Although 'APtr' points at 'B1', which occupies eight bytes (four for 'x' and four for 'y'), it is able to access the value contained in only the first four bytes. Thus, 'APtr' cannot access an area in the memory that has not been allocated. Therefore, a pointer of the base class type can safely point at an object of the derived class.

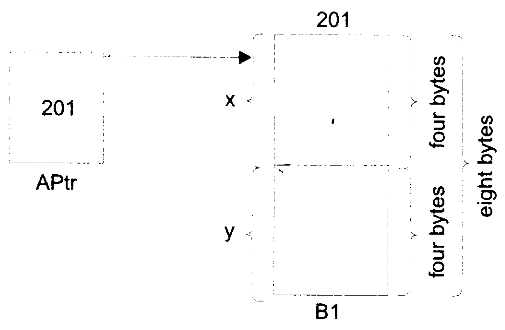

Diagram 5.3 Base class pointer points at an object of the derived class

We will soon realize that making a base class pointer point at an object of the derived class is a very common requirement in C++ programming.

Now let us find out why derived class pointers cannot be made to point at objects of the base class without explicit typecasting, and why, even that is a very unsafe thing to do. Now, let us compile the following 'main()' function and see the result.

```
/*Beginning of DerivedPtr01.cpp*/
#include"B.h"   //from listing 5.3
void main()
{
  A A1;
  B * BPtr;
```

```
    BPtr=&A1;     //line 1 - ERROR. Cannot convert from B* to
                  //A*.
    BPtr->x=10;   //line 2 - OK. Derived class pointer
                  //accesses base class member.

    BPtr->y=20;   //line 3 - OK. Derived class pointer
                  //accesses derived class member.
}
/*End of DerivedPtr01.cpp*/
```

Listing 5.5 Derived class pointer pointing at a base class object

Line 1 of Listing 5.5 will not compile because line 3 will. A derived class pointer cannot point at an object of the base class. Let us see why. 'BPtr' is of type 'B *'. It is supposed to point at objects of the derived class B. Therefore, it can access 'y' also. 'BPtr' is pointing at 'A1', which occupies four bytes only. However, it is able to access the value contained in the next four bytes also. There is a member of name 'y' in class B. Thus, 'BPtr' is able to access an area in the memory that has not been allocated. Therefore, a pointer of the derived class type cannot safely point at an object of the base class.

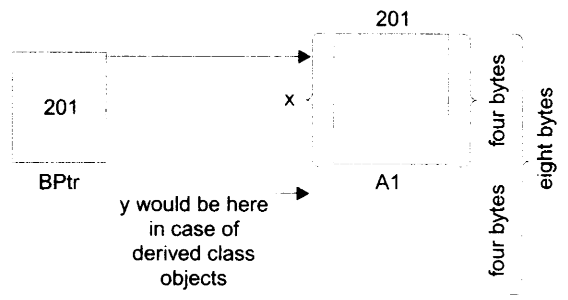

Diagram 5.4 Derived class pointer pointing at an object of the base class

Line 3 of Listing 5.5 would write 20 into the bytes whose addresses are from 205 to 208. But this block has not been allocated for the object at which the pointer points. But this line will compile. The problem is actually in line 1.

However, a derived class pointer *can* be forcibly made to point at an object of the derived class by explicit typecasting. Continuing with classes A and B given in Listing 5.3.

```
/*Beginning of DerivedPtrTypeCast.cpp*/
#include"B.h"  //from listing 5.03
void main()
{
  A A1;
  B * BPtr;          //derived class pointer
  BPtr=(B*)&A1;      //forcible typecasting to make derived
```

```
                        //class pointer point at base class
                        //object
}
/*End of DerivedPtrTypeCast.cpp*/
```

Listing 5.6 Forcible typecasting to make a derived class pointer point at an object of the base class

But explicit address manipulation like this is obviously dangerous. Now let us consider the realistic cases where the classes have private data members and public member functions. The same explanations that have been given above will hold true even if the classes A and B have 'x' and 'y' as private data members, respectively.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    void setx(const int=0);
    /*
      rest of the class A
    */
};
/*End of A.h*/

/*Beginning of B.h*/
#include"A.h"
class B : public A
{
    int y;
  public:
    void sety(const int=0);
    /*
      rest of the class B
    */
};
/*End of B.h*/
```

Listing 5.7 Classes of Listing 5.3 with member functions

The member functions in Listing 5.5 access private data members of their respective classes.

```
/*Beginning of BasePtr02.cpp*/
#include"B.h"  //from listing 5.7
void main()
{
  A * APtr;
  B B1;
  APtr=&B1;          //OK: base class pointer points at
                     //derived class's object
  APtr->setx(10);    //OK: accessing base class member
                     //through base class pointer
  APtr->sety(20);    //ERROR: sety() not a member of class A
}
/*End of BasePtr02.cpp*/
```

Listing 5.8 Base class pointer pointing at an object of the derived class

```
/*Beginning of DerivedPtr02.cpp*/
#include"B.h"  //from listing 5.7
void main()
{
  A A1;
  B * BPtr;
  BPtr=&A1;          //ERROR: cannot convert A* to B*
  BPtr->setx(10);    //OK: Derived class pointer accesses
                     //base class member.
  BPtr->sety(20);    //OK: Derived class pointer accesses
                     //derived class member.
}
/*End of DerivedPtr02.cpp*/
```

Listing 5.9 Derived class pointer pointing at an object of the base class

The fact that a base class pointer can point at an object of the derived class should not be surprising. After all, this is exactly what happens when we call a base class function with respect to an object of the derived class.

```
B1.setx(10);
```

Based upon the knowledge we have gained about the 'this' pointer in Chapter 2, we know that the compiler will internally convert the above statement to

```
setx(&B1,10);
```

The address of 'B1' (a derived class object) is passed as a parameter to the function. But the corresponding formal argument in the 'A::setx()' function is the 'this' pointer of type 'A * const'.

```
void setx(A * const this, const int p)
{
  this->x=p;
}
```

Listing 5.10 This pointer in a base class member function points at the derived class invoking object

Obviously, the 'this' pointer points at 'B1', which is an object of the derived class.

5.3 Function Overriding

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. Function overriding results in two functions of the same name and same signature. One of them is in the base class. The other one is in the derived class. An illustrative example follows.

```
/*Beginning of A.h*/
class A
{
  public:
    void show()
    {
      cout<<"show() function of class A called\n";
    }
};
/*End of A.h*/

/*Beginning of B.h*/
#include"A.h"
class B : public A
{
  public:
    void show()      //overriding A::show()
    {
      cout<<"show() function of class B called\n";
    }
};
/*End of B.h*/
```

Listing 5.11 Function overriding

The 'show()' function of class B has overridden the 'show()' function of class A. Consequently, if the 'show()' function is called with respect to an object of the derived class B, the 'show()' function of class B will be called instead of the 'show()' function of class A.

```
/*Beginning of Override01.cpp*/
#include"B.h"
void main()
{
  B B1;
  B1.show();    //B::show() called
}
/*End of Override01.cpp*/
```

Output

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

Listing 5.12 Calling the overriding function

Whenever a function is called with respect to an object of a class, the compiler first searches for the function prototype in the same class. Only if this search fails, the compiler goes up the class hierarchy to look for the function prototype. In Listing 5.11 & 5.12, the 'show()' function of class A was virtually hidden by the 'show()' function of class B.

Of course, the overridden function of the base class will be called if it is called with respect to an object of the base class.

```
/*Beginning of Override02.cpp*/
#include"B.h"
void main()
{
  A A1;
  A1.show();    //A::show() called
}
/*End of Override02.cpp*/
```

Output

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

Listing 5.13 Calling the overridden function with respect to an object of the base class

The overridden base class function can still be called with respect to an object of the derived class by using the scope resolution operator as follows.

```
/*Beginning of Override03.cpp*/
#include"B.h"
void main()
{
  B B1;
  B1.A::show();       //A::show() called
}
/*End of Override03.cpp*/
```

Output

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

Listing 5.14 Calling the overridden function forcibly with respect to an object of the derived class

Function overriding is actually a form of function overloading. Our knowledge of the 'this' pointer immediately makes this clear. The signatures of the overriding function and the overridden function are only apparently the same. They are actually different from each other. The actual prototype of the 'A::show()' function is

```
void show(A * const);
```

On the other hand, the actual prototype of the 'B::show()' function is

```
void show(B * const);
```

The overridden function can be called from the overriding function as follows.

```
void B::show()
{
  A::show();
  /*
    rest of the B::show() function
  */
}
```

The scope resolution operator is necessary to avoid infinite recursion.

But, what is the use of function overriding? Function overriding *appears* to be nothing more than a fancy language construct. Function overriding becomes significant only when the base class function being overridden is virtual. More about virtual functions and how they implement dynamic polymorphism is illustrated in Chapter 6.

5.4 Base Class Initialization

A derived class object is composed of data members of the derived class as well as those of the base class. Often we need to initialize all of these data members while creating an object of the derived class. We must remember that when an object of the derived class is created, the compiler implicitly and inevitably embeds a call to the base class constructor and then the derived class constructor with respect to the object.

Suppose A is the base class and B is its derived class. The statement

```
B B1;
```

is converted into

```
B B1;      //memory allocated for the object
B1.A();    //base class constructor called
B1.B();    //derived class constructor called
```

Destructors are called in the reverse order. As we already know, explicitly calling the constructors and destructors, with respect to an existing object, is prohibited. Now let us look at the following program.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    A(const int=0);
    void setx(const int=0);
    int getx()const;
};
/*End of A.h*/

/*Beginning of A.cpp*/
#include"A.h"

A::A(const int p)
{
  x=p;
}

void A::setx(const int p)
{
  x=p;
}

int A::getx() const
{
  return x;
}
/*End of A.cpp*/

/*Beginning of B.h*/
#include"A.h"

class B : public A
{
    int y;
  public:
    B(const int=0);
    void sety(const int=0);
    int gety()const;
};
/*End of B.h*/

/*Beginning of B.cpp*/
#include"B.h"

B::B(const int q)
{
  y=q;
}

void B::sety(const int q)
{
  y=q;
}

int B::gety() const
{
  return y;
}
/*End of B.cpp*/

/*Beginning of baseinit01.cpp*/
#include"B.h"
#include<iostream.h>

void main()
{
  B B1(20);
  cout<<B1.getx()<<endl
      <<B1.gety()<<endl;
}
/*End of baseinit01.cpp*/
```

Output

```
0
20
```

Listing 5.15 Unsuccessful initialization of base class members

The output is explained by the simple observation that the statement

```
B B1(20);
```

gets converted to the following:

```
B B1;       //memory allocated for the object
B1.A();     //base class constructor called
B1.B(20);   //derived class constructor called
```

As we can see, base class data members of the derived class object got initialized through the base class constructor with the default value being passed to it. Thus, 'B1.y' got initialized to 20 (the value passed). But 'B1.x' got initialized to 0 (the default value). While creating an object of the derived class, we would like to pass a value explicitly to the base class constructor. Thus, in Listing 5.15, the constructor of class B should take not one but two parameters. One of these should be passed to 'y' while the other should be used to initialize 'x'. For this, the prototype and definition of the constructor of class B should be modified as follows.

```
/*Beginning of B.h*/
#include"A.h"

class B : public A
{
  public:
    B(const int=0, const int=0);
  /*
    rest of the class B
  */
}
/*End of B.h*/

/*Beginning of B.cpp*/
#include"B.h"

B::B(const int p,const int q):A(p) //passing value to base
                                   //class constructor
{
  y=q;
}

/*
  Definitions of the remaining member functions of class B
*/

/*End of B.cpp*/
```

Listing 5.16 Modifying the derived class constructor to ensure successful initialization of the base class members

An object of class B can be declared by passing two parameters to its constructor. One of them is assigned to 'x'. The other is assigned to 'y'.

```
/*Beginning of baseinit02.cpp*/
#include"B.h"
#include<iostream.h>

void main()
{
  B B1(10,20);
  cout<<B1.getx()<<endl<<B1.gety()<<endl;
}
/*End of baseinit02.cpp*/
```

Output

```
10
20
```

Listing 5.17 Base class initialization

Again, the output of Listing 5.17 can be explained by noting that due to the modified definition of the constructor of class B, the statement

```
B B1(10,20);
```

gets converted to

```
B B1;      //memory allocated for the object
B1.A(10); //base class constructor called
B1.B(20); //derived class constructor called
```

As per the definition of the derived class constructor in Listing 5.17, the *first* parameter passed to it was in turn passed to the base class constructor. But this is not necessary. Any of the parameters passed to the derived class constructor can be passed to the base class constructor.

5.5 The Protected Access Specifier

Apart from the 'public' and 'private' access specifiers, there is a third access modifier in C++ known as 'protected'. 'Protected' members are inaccessible to non-member functions. However, they are accessible to the member functions of their own class and to member functions of the derived classes. The following example (Listing 5.18) along with its accompanying comments illustrates this.

```
/*Beginning of A.h*/
class A
{
  private:
    int x;
  protected:
    int y;
  public:
    int z;
};
/*End of A.h*/

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

/*Beginning of B.cpp*/
#include"B.h"
void B::xyz()  //member function of derived class
{
  x=1;    //ERROR: private member of base class
  y=2;    //OK: protected member of base class
  z=3;    //OK: public member of base class
}
/*End of B.cpp*/

/*Beginning of protected.cpp*/
#include"A.h"
void main()    //nonmember function
{
  A * Aptr;
  APtr->x=10;  //ERROR: private member
  APtr->y=20;  //ERROR: protected member
  APtr->z=30;  //OK: public member
}
/*End of protected.cpp*/
```

Listing 5.18 Accessing protected members

5.6 Deriving by Different Access Specifiers

Deriving by the 'Public' Access Specifier

Deriving by the public access specifier retains the access level of base class members.

Private Members: Member functions of the derived class cannot access. Member functions of the subsequently derived classes cannot access them. Non-member functions cannot access them.

Protected Members: Member functions of the derived class can access. Member functions of the subsequently derived classes can also access them. Non-member functions cannot access them.

Public Members: Member functions of the derived class can access. Member functions of the subsequently derived classes can also access them. The non-member functions can also access them.

Errors that are encountered while compiling the following program (Listing 5.19) make this evident.

```
/*Beginning of publicInheritance.cpp*/
class A
{
  private:
    int x;
  protected:
    int y;
  public:
    int z;
};
class B : public A  //B is a public derived class of A
{
  public:
    void f1()
    {
      x=1;      //ERROR: private member remains private
      y=2;      //OK: protected member remains protected
      z=2;      //OK: public member remains public
    }
};

class C : public B
{
  public:
    void f2()
    {
      x=1;      //ERROR: private member remains private
      y=2;      //OK: protected member remains protected
      z=2;      //OK: public member remains public
    }
};
```

```
void xyz(B * BPtr)   //non-member function
{
  BPtr->x=10;   //ERROR: private member remains private
  BPtr->y=20;   //ERROR: protected member remains protected
  BPtr->z=30;   //OK: public member remains public
}

/*End of publicInheritance.cpp*/
```

Listing 5.19 Accessing the inherited members of an object of a class derived by public access specifier

A base class pointer can point at an object of a derived class that has been derived by using the public access specifier. Let us redefine the 'xyz()' function from the program in Listing 5.19 as follows and see what happens if we recompile the program.

```
void xyz()         //non-member function
{
  B B1;            //line 1: An object of a public derived
                   //class
  B1.z=100;        //line 2: OK. Can access public member of
                   //a base class through an object of a
                   //public derived class.
  A * APtr;        //line 3
  APtr=&B1;        //line 4: OK. Can make a base class pointer
                   //point at an object of a public derived
                   //class.
  Aptr->z=100; //line 5. OK. Can access inherited public
                   //member of the base class through a base
                   //class pointer.
}
```

Listing 5.20 A base class pointer can point at an object of the public derived class

Line 4 of Listing 5.20 will compile successfully because lines 2 and 5 will. Line 2 will compile successfully because 'z' is a public member of the base class A and class B is derived from class A by using the public access specifier. In this case, the base class pointer would access the object of a public derived class in a way (line 5 of Listing 5.20) that is anyway permitted when the object is accessed by using the name of the object itself (line 2 of Listing 5.20).

Therefore, the C++ compiler does not prevent a base class pointer from pointing at an object of the derived class if the public access specifier has been used to derive the class.

Deriving by the 'Protected' Access Specifier

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.

Private Members: Member functions of the derived class cannot access. Member functions of the subsequently derived classes cannot access them. Non-member functions cannot access them.

Protected Members: Member functions of the derived class can access. Member functions of the subsequently derived classes can also access them. Non-member functions cannot access them.

Public Members: Member functions of the derived class can access. Member functions of the subsequently derived classes can also access them. Non-member functions cannot access them.

Errors encountered while compiling the following program (Listing 5.21) demonstrate this.

```
/*Beginning of protectedInheritance.cpp*/
class A
{
  private:
    int x;
  protected:
    int y;
  public:
    int z;
};

class B : protected A     //B is a protected derived class
                          //of A
{
  public:
    void f1()
    {
      x=1;      //ERROR: private member remains private
      y=2;      //OK: protected member remains protected
      z=2;      //OK: public member becomes protected
    }
};

class C : public B
{
  public:
    void f2()
```

```
      {
        x=1;        //ERROR: private member remains private
        y=2;        //OK: protected member remains protected
        z=2;        //OK: protected member remains protected
      }
};
void xyz(B * BPtr)  //non-member function
{
  BPtr->x=10;   //ERROR: private member remains private
  BPtr->y=20;   //ERROR: protected member remains protected
  BPtr->z=30;   //ERROR: public member becomes protected
}
/*End of protectedInheritance.cpp*/
```

Listing 5.21 Accessing the inherited members of an object of a class derived by protected access specifier

A base class pointer cannot point at an object of a derived class that has been derived by using the protected access specifier. Let us redefine the 'xyz()' function from the program in Listing 5.21 as follows and see what happens if we recompile the program.

```
void xyz()        //non-member function
{
  B B1;           //line 1: An object of a protected derived
                  //class
  B1.z=100;       //line 2: ERROR. Cannot access public member
                  //of a base class through an object of a
                  //protected derived class.
  A * APtr;       //line 3
  APtr=&B1;       //line 4: ERROR. Cannot make a base class
                  //pointer point at an object of a protected
                  //derived class.
  Aptr->z=100;    //line 5. OK. Can access public
                  //member of the base class through a base
                  //class pointer.
}
```

Listing 5.22 A base class pointer cannot point at an object of the protected derived class

Line 4 of Listing 5.22 will not compile because line 2 will not compile and line 5 will compile. Line 2 will not compile because although 'z' is a public member of the base class A, class B is derived from class A by using the protected access specifier. In this case, the base class pointer might access the object of a protected derived class in a way (line 5 of Listing 5.22) that is not permitted when the object is accessed by using the name of the object itself (line 2 of Listing 5.22).

(Therefore, the C++ compiler prevents a base class pointer from pointing at an object of the derived class if the protected access specifier has been used to derive the class.)

Deriving by the 'Private' Access Specifier

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.

Private Members: Member functions of the derived class cannot access. Member functions of the subsequently derived classes cannot access them. Non-member functions cannot access them.

Protected Members: Member functions of the derived class cannot access. Member functions of the subsequently derived classes cannot access them. Non-member functions cannot access them.

Public Members: Member functions of the derived class cannot access. Member functions of the subsequently derived classes cannot access them. Non-member functions cannot access them.

Errors encountered while compiling the following program (Listing 5.23) demonstrate this.

```
/*Beginning of privateInheritance.cpp*/
class A
{
  private:
    int x;

  protected:
    int y;
  public:
    int z;
};

class B : private A //B is a private derived class of A
{
  public:
    void f1()
    {
      x=1;      //ERROR: private member remains private
      y=2;      //ERROR: protected member becomes private
      z=2;      //ERROR: public member becomes private
    }
};
```

```
class C : public B
{
  public:
    void f2()
    {
      x=1;      //ERROR: private member remains private
      y=2;      //ERROR: private member remains private
      z=2;      //ERROR: private member remains private
    }
};

void xyz(B * BPtr)  //non-member function
{
  BPtr->x=10;  //ERROR: private member remains private
  BPtr->y=20;  //ERROR: protected member becomes private
  BPtr->z=30;  //ERROR: public member becomes private
}

/*End of privateInheritance.cpp*/
```

Listing 5.23 Accessing the inherited members of an object of a class derived by private access specifier

A base class pointer cannot point at an object of a derived class that has been derived by using the private access specifier. Let us redefine the 'xyz()' function from the above program (Listing 5.23) as follows and see what happens if we recompile the program.

```
void xyz()         //non-member function
{
  B B1;            //line 1: An object of a private derived
                   //class

  B1.z=100;        //line 2: ERROR. Cannot access public member
                   //of a base class through an object of a
                   //private derived class.
  A * APtr;        //line 3
  APtr=&B1;        //line 4: ERROR. Cannot make a base class
                   //pointer point at an object of a private
                   //derived class.
  Aptr->z=100;  //line 5. OK. Can access public
                   //member of the base class through a base
                   //class pointer.
}
```

Listing 5.24 A base class pointer cannot point at an object of the private derived class

Line 4 of Listing 5.24 will not compile because line 2 will not compile and line 5 will compile. Line 2 will not co mpile because although 'z' is a public member of the base class A, class B is derived from class A by using the private access specifier. In this case, the base class pointer might access the object of a private derived class in a way (line 5 of Listing 5.24) that is not permitted when the object is accessed by using the name of the object itself (line 2 of Listing 5.24).

Therefore, the C++ compiler prevents a base class pointer from pointing at an object of the derived class if the private access specifier has been used to derive the class.

The default access specifier for inheritance is 'private'. The following declarations are equivalent:

```
class B : private A //B is a private derived class of A
{
  /*
    definition of class B
  */
};

class B : A    //B is still a private derived class of A
{
  /*
    definition of class B
  */
};
```

5.7 Different Kinds of Inheritance

Multiple Inheritance

In multiple inheritance, a class derives from more than one base class.

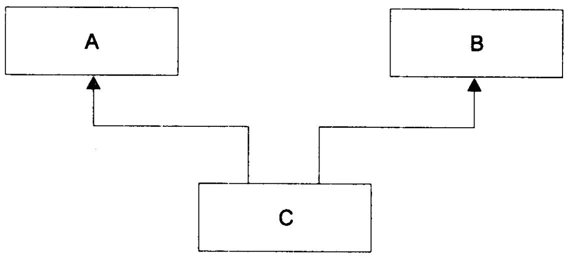

Diagram 5.5 Multiple inheritance (class C derived from classes A and B)

The general syntax for multiple inheritance is as follows:

```
class <name of derived class>
: <access specifier> <name of first base class>,
  <access specifier> <name of second base class>,
  <access specifier> <name of third base class> ...
{
  /*
    definition of derived class
  */
};
```

An illustrative example follows (Listing 5.25). We must note that for each of the base classes, a different access specifier can be used.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    void setx(const int=0);
    int getx()const;
};
/*End of A.h*/

/*Beginning of B.h*/
class B
{
    int y;
  public:
    void sety(const int=0);
    int gety()const;
};
/*End of B.h*/

/*Beginning of C.h*/
#include"A.h"
#include"B.h"

class C : public A, public B  //multiple inheritance
{
    int z;
  public:
    void setz(const int=0);
    int getz()const;
};
/*End of C.h*/
```

```
/*Beginning of multi.cpp*/
#include"C.h"

void main()
{
  C C1;
  cout<<sizeof(C1)<<endl;
  C1.setx(10);
  C1.sety(20);
  C1.setz(30);
  cout<<C1.getx()<<endl
      <<C1.gety()<<endl
      <<C1.getz()<<endl;
}
/*End of multi.cpp*/
```

Output

```
12
10
20
30
```

Listing 5.25 Multiple inheritance

An object of a class defined by multiple inheritance contains not only the data members defined in the derived class, but also the data members defined in all of the base classes. Thus, the size of such an object is equal to the sum of the sizes of the data members of all the base classes plus the sum of the sizes of the data members of all of the derived classes. Hence, the size of an object of the class C in Listing 5.25 is 12.

Moreover, with respect to such an object, it is possible to call the member functions of not only the derived class, but also the member functions of all the base classes. Therefore, in Listing 5.25, the member functions of classes A, B, and C have been called with respect to 'C1'.

Ambiguities in Multiple Inheritance

Multiple inheritance leads to a number of ambiguities, namely, identical members in more than one base class and diamond-shaped inheritance.

- **Identical Members in More than One Base Class:** The first ambiguity arises if two or more of the base classes have a member of the same name. This is illustrated in Listing 5.26.

```
/*Beginning of multiInheritAmbiguity.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
{};

void main()
{
  C C1;
  C1.show();    //ERROR! ambiguous call to show()
}
/*End of multiInheritAmbiguity.cpp*/
```

Listing 5.26 Ambiguity due to identical member being in more than one base class

In Listing 5.26, the compiler will not be able to decide which of the two 'show()' functions it has to call. This ambiguity can be resolved by using the scope resolution operator. We can replace the 'main()' function of Listing 5.26 with the following one and see the difference.

```
/*Beginning of multiInheritAmbiguityResolve01.cpp*/
void main()
{
  C1.A::show();      //OK: show() function of class A called
  C1.B::show();      //OK: show() function of class B called
```