UNIT 1 INHERITANCE

Structure **Page No.**

1.0 Introduction 5
1.1 Objectives 5
1.2 Concept of Re-usability 6
1.3 Inheritance 6
  1.3.1 Derived and Base Class 8
  1.3.2 Declaration of Derived Class 9
  1.3.3 Visibility of Class Members 9
1.4 Types of Inheritance 12
1.5 Single Inheritance 12
1.6 Multiple Inheritance 15
1.7 Multi-level Inheritance 17
1.8 Constructors and Destructors in Derived Classes 21
1.9 Summary 26
1.10 Answers to Check Your Progress 27
1.11 Further Readings 34

1.0 INTRODUCTION

There is a great surge of interest today in Object Oriented Programming (OOP) due to obvious reasons. One of the most fundamental concepts of the object-oriented paradigm is inheritance that has profound consequences on the development process. In OOP, it is possible to define a class as inheriting from another. Object Oriented Software Development involves a large number of classes. Many of the classes extension of others. Object Oriented Programming has been widely acclaimed as a technology that will support the creation of re-usable software, particularly because of the inheritance facility. In OOP, inheritance is a re-usability technique. We can show similarities between classes by means of inheritance and describe their similarities in a class which other classes can inherit. Thus, we can re-use common descriptions. Therefore, inheritance is often promoted as a core idea for reuse in the software industry. The abilities of inheritance, if properly used, is a very useful mechanism in many contexts including reuse. This unit starts with a discussion on the re-usability concept. This is followed by the inheritance which is prime feature of object oriented paradigm. It focuses on the standard form of inheritance by extension of a base class with a derived class. Moreover, in this unit, different types of inheritance, the time of its use and methods of its implementation are also discussed. Base classes, derived classes, visibility of class members, and constructors and destructors in derived classes are introduced. Illustrative examples that facilitate understanding of the concept are presented.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- define and understand why we need to study inheritance;
- define and understand the concept of reusability;
- describe an inheritance relationship;
- identify the cases where inheritance is suitable;
- define base classes and derived classes;
- implement different types of inheritance in C ++;

- explain the different types of inheritance;
- understand the need of virtual base class, and
- understand and implement the constructors in derived classes.

1.2 CONCEPT OF RE-USABILITY

Software re-usability is primary attribute of software quality. C++ strongly supports the concept of reusability. C++ features such as classes, virtual functions, and templates allow designs to be expressed so that re-use is made easier (and thus more likely), but in themselves such features do not ensure re-usability. Do we really need re-use? This question can best be answered by an analogy from automobile industry. Consider the design and creation of a new car model. The automotive engineer does not design a new car from scratch. Rather, the engineer borrows from the design of existing cars. For example, the engine design from an existing car may be used in a new model. If the engine design has been used in a previous model, design problems have likely been resolved. Thus, development costs are reduced because a new engine does not need to be designed and tested. Finally, consumer maintenance costs are reduced because machines and others who must maintain the car are already familiar with the operation of the engine.

The American Heritage Dictionary defines quality as “a characteristic or attribute of something”. Re-usability is the degree to which a thing can be reused. Software re-usability represents the ability to use part or the whole system in other systems which are related to the packaging and scope of the functions that programs perform.
Can you tell why do we need to study re-usability? Well, the need for re-usability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before. Do you know what the advantage of re-usability is? There are many advantages of re-usability. They can be applied to reduce cost, effort and time of software development. It also increases the productivity, portability and reliability of the software product.

We can say that re-usability is concerned as to how we can use a system or its part in other systems.

Now, can you tell how the re-use is achieved in program? Re-use in OOP language can be achieved by two ways basically: The first is through class definition-every time a new object of class is defined, we reuse all the code and declarations of that class. This type of re-use can be supported in the function-oriented approach also. The other type of re-use, which is particular to OOP language, can be supported by inheritance. Inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possibly by deriving a new class from the existing one. You have already seen the first type of reusability through class. In the next section, we shall see, how will we achieve the reusability through inheritance?

1.3 INHERITANCE

Inheritance is a prime feature of object oriented programming language. It is process by which new classes called derived classes(sub classes, extended classes, or child classes) are created from existing classes called base classes(super classes, or parent classes). The derived class inherits all the features (capabilities) of the base class and can add new features specific to the newly created derived class. The base class remains unchanged.

Inheritance is a technique of organizing information in a hierarchical form. It is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes.

Inheritance is often referred to as an "is-a" relationship because every object of the class being defined "is" carries an object of the inherited class also.

You can look around and find many real world examples of inheritance like Inheritance between parent and child, employee and manager, person and student, vehicle and light motor vehicle, and animal and mammal etc. Why are we interested in inheritance and how will we use this concept? Well, Let us take the example to understand this concept. Suppose we want to use the classes Employee and Manager in the C++ program. For the time being, let us assume that we do not know the concept of the inheritance. Now, we define the Employee and Manager classes as follows:

```
class Employee
{
public:
int id_number;
char Name_of_Employee[25];
int age;
char Department_Name[25];
int salary;
char Address[25];
// Member functions
void display_name(void);
void display_id(void);
void raise_salary(float percent);
.
.
.
};
class Manager
{
public:
int id_number;
char Name_of_Employee[25];
int age;
char Department_Name[25];
int salary;
char Address[25];
char  Name_of_secretary[25];
Employee *team_members;
void display_name(void);
void display_id(void);
void display_secretary_name(void);
void raise_salary(float percent);
.
.
.
};
```

If you look at the above declarations of the classes Employee and Manager, you can make the observation that there are some common attributes and behavior in Employee and Manager class. We have shown the common attributes and behavior in Manager class again. Now, we introduce the concept of the inheritance. As we know that generally Managers are treated differently from other employees in the organization. Their salary raises are computed differently, they have access to a secretary, they have a group of employee under them and so on. There are some

common data members as well as member functions like name, age, salary, address, display_name(),display_id() etc . This is a kind of situation in which we use the concept of inheritance. Why? Well, we can retain some of what we have already laid down and defined in the Employee class in terms of data members and member functions. Employee and Manager classes are declared as follows:

```
class Employee
{
public:
int id_number;
char Name_of_Employee[25];
int age;
char Department_Name[25];
int salary;
char Address[25];
// Member functions
void display_name(void);
void display_id(void);
void raise_salary(float percent);
.
.
.
};
class Manager :: public Employee
{
public:
char  name_of_secretary[25];
Employee *team_members;
void display_secretary_name(void);
void raise_salary(float percent);
.
.
.
};
```

You can look at the above declaration and observe that we did not declare the common attributes and functions again in the Manager class. Thus, we have reused the previous declarations of data members and functions. We can also observe that we have redefined raise_salary function in the Manager class due to different way to compute salary of the Manager. From the above discussion, we can conclude that what is meant by the application of inheritance and how it is supporting the concept of re-usability by adding additional feature to an existing classes without modifying it.

1.3.1 Derived and Base Class

As we know, when Class A inherits the feature from class B, then Class A is called the derived class and B is called Base class. A derived class extends its features by inheriting the properties (features) from another class called the base class while adding features of its own. In next section, we will see as to how we will define a derived class.

1.3.2 Declaration of Derived Class

The declaration of a derived class shows its relationship with the base class in addition to its own details. The common syntax of declaring a derived class is given as follows:

```
class DerivedClassName : [VisibilityMode] BaseClassName
{
// members of derived class
};
```

The derivation of DerivedClassName from the BaseClassName is indicated by colon (:). The VisibilityMode enclosed within the square bracket is optional. If the VisibilityMode is specified, it must be either public or private or protected. It specifies the features of the base class that are privately derived or publicly derived. There are four possible ways of derivation of derived class which is given below:

```
class DerivedClassName : public BaseClassName          //public derivation
{
//members of derived class
};
class DerivedClassName : private BaseClassName         //private
derivation
{
//members of derived class
};
class DerivedClassName : protected BaseClassName       //protected
derivation
{
//members of derived class
};
class DerivedClassName : BaseClassName                 //private
derivation
{
//members of derived class
 };
```

1.3.3 Visibility of Class Members

There are three visibility modes (visibility modifier). They are private, public and protected. We have already learnt about private and public visibility mode in Unit 3 of Block 1 of this course. Could you tell what the role of these terms in the programs exists? They are actually controllers, used to control the access to members (data members and functions members) of a class.

Why is the different type of visibility mode needed in derivation of a derived class? Well, a class may contain some secret information which we are not interested to share by the derived classes and non-secret information which we are interested to share by the derived class. In nutshell, we can say that visibility mode promotes encapsulation. The visibility of the base class members undergoes modification in a derived class as summarized in Table 1.1.

Table 1.1: Visibility Mode

Base class Visibility	**Derived class Visibility**		
	Private derivation	**Public derivation**	**Protected derivation**
private public protected	Not inherited private private	Not inherited public protected	Not inherited protected protected

From the Table 1.1, you can observe that in derived class declaration, if the visibility mode is private then both 'public members' of the base class as well as 'protected members' of the base class will become private members of the derived class. Therefore, both public and protected member of base class can only be accessed by the member functions of the derived class. They can not be accessed by the objects of the derived class. And private members of the base class will not be inherited. On the other hand, if visibility mode is public, public members of the base class will become public members of the derived class and protected members of the base class will become protected members of the derived class whereas private member of the base class will never become the members of the declared class i.e. it will not be inherited. If the visibility mode is protected then the public and protected members of the base class will become the protected members of the derived class. In this case also, the private members of the base class will not become the member of its derived class. As we have demonstrated that the private members of base class will remain private to the base class whether the base class is inherited publicly or privately or protected by any means. They add to the items of the derived class and they are not directly accessible to the member of a derived class. Derived class can access them through the base class member functions. Consider the following declarations of a base class A and a derived classes B, C, and D to illustrate private and public inheritance.

```
class A
{
private:
int privateA;      // private member of base class A
protected:
int protectedA;  // protected member of base class A
public:
int publicA;       // public member of base class A
int getPrivateA() //public function of base class A
{
return privateA;
}
};

class B: private A          // privately derived class
{
private:
int privateB;
protected:
int protectedB;
public:
int publicB;
void fun1()
{
int b;
b=privateA;       //Won't work: privateA is not accessible
b=getPrivateA(); //OK: inherited member access private data
b=protectedA;   // OK
b=publicA;        // OK
}
};
```

```
class C: public A           // publically derived class
{
private:
int privateC;
protected:
int protectedC;
public:
int publicC;
void fun2()
{
int c;
c=privateA;       //Won't work :privateA is not accessible
c=getPrivateA(); //OK: inherited member access private data
c=protectedA;   // OK
c=publicA;        // OK
}
};
```

Consider the following statements:

```
B objb; // objb is a object of class B
C objc; // objc is a object of class C
int x;    // temporary variable x
```

The above statements define the object objb, objc and the integer variable x. Let us consider the statements as follows:

```
x=objb.protectedA;       //Won't work : protectedA is not accessible
x=objb.publicA;           //Won't work : publicA is not accessible
x=objb.getPrivateA();    // Won't work: getPrivateA() is not accessible
```

The above all statements are illegal. Because protectedA, publicA and getPrivateA() each have private accessibility status in the derived class B. However, fun1() of derived class B accesses getPrivateA(), protectedA and publicA. Let us again consider the statements as follows:

```
x=objc.protectedA;       //Won't work : protectedA is not accessible
x=objc.publicA;           //Valid
x=objc.getPrivateA();    // Valid
```

The above first statement is illegal but second and third statements are valid. It is so because in the first statement protected member protectedA of the base class A has protected visibility status in class C. However in the second and third statements both publicA and getPrivateA() have their public visibility status in class C, so they are accessible.

☞ Check Your Progress 1

1) What are the benefits of inheritance? Explain in brief.

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

2) When do we need inheritance?

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

3) Why do we need to study access specifiers?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

4) What are the advantages of re-usability?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

5) Why do we need re-use? Give an analogy to explain your answer.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

1.4 TYPES OF INHERITANCE

In previous section, we discussed inheritance. In this section, we will discuss the types of inheritance. As we know, the derived class inherits some or all of the features from the base class depending on the visibility mode. A derived class can also inherit properties from more than one class or from more than one level. We can classify inheritance into the following type accordingly:

- **Single Inheritance:** Derivation of a class from only one base class is called a single inheritance.
- **Multiple Inheritance:** Derivation of a class from several (two or more) base classes is called multiple inheritance.
- **Multi-level Inheritance:** Derivation of a class from another derived class is called multilevel inheritance.
- **Hierarchical Inheritance:** Derivation of several classes from a single base class is called hierarchical inheritance.
- **Hybird Inheritance:** Derivation of a class involving more than one form of inheritance is called hybrid inheritance.
- **Multi-path Inheritance:** Derivation of a class from other derived classes, which are derived from the same base class is called multi-path inheritance.

In next sections, we shall discuss in detail single inheritance, multiple inheritance and multi-level inheritance.

1.5 SINGLE INHERITANCE

Now, let us discuss about single inheritance. In Single Inheritance, derived class inherits the feature of one base class. If a class is derived from one base class, it is called Single Inheritance.

Figure 1.1 depicts single inheritance:

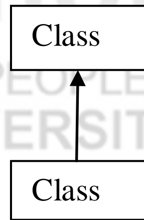

Figure 1.1: Single Inheritance

Class A is the base class and class B is the derived class. The following are the common steps to implement an inheritance. First, declare a base class and second declare a derived class. The syntax of single inheritance of the above figure is given as follows:

A derived class can be declared if its base class is already declared.

```
class A
{
// members of class A
};
class B :[public/private/protected] A
{
// members of class B
};
```

Let us understand the concept of single inheritance with Example1. In the example 1 given below, it is seen that how a single inheritance is implemented:

Example 1: Single Inheritance

```
#include <iostream.h>
class A
{
int a;
public :
int b;
void input_ab(void);
void output_a(void);
int get_a(void);
};
class B : public A
{
int c,d;
public :
void input_c(void);
void display(void);
void sum(void);
};
void A :: input_ab()
{
cout<< "\n Enter the value of a and b :"<<endl;
cin>>a>>b;
}
void A :: output_a()
{
cout<<"\\n The Value of a is :"<<a<<endl;
}
```

```
int A :: get_a()
{
return a;
}
void B :: input_c()
{
cout<< "\n Enter the value of c :"<< endl;
cin>>c;
}
void B :: sum()
{
d=get_a()+b+c;
}
void B :: display()
{
cout<< "\n The value of b is :"<<b<< endl;
cout<< "\n The value of c is :"<<c<< endl;
cout<< "\n The value of d(sum of a,b and c) is :"<<d<< endl;
}
void main()
{
B objb;

objb.input_ab();          //base class member function
objb.input_c();           //derived class member function
objb.output_a();          //base class member function
objb.sum();               //derived class member function
objb.display();           //derived class member function
objb.b=0;                 //objb.a would not work
objb.sum();               //derived class member function
objb.display();           //derived class member function
}
```

The output of the example1 is given below.

```
Enter the value of a and b:
10
20
Enter the value of c:
30
The value of a is : 10
The value of b is : 20
The value of c is : 30
The value of d(sum of a, b and c) is : 60
The value of a is : 10
The value of b is : 0
The value of c is : 30
The value of d(sum of a, b and c) is : 40
```

The above example shows a base class A and a derived class B. The base class A contains one private data member **a**, one public data member **b**, and three public member functions input_ab(), output_a() and get_a(). The class B contains two private data **c** and **d** and three public functions input_c(), display() and sum(). The class B is a derived publicly by class A. Therefore, B inherits all the public members (data and functions) of class A and retains their visibility. Hence, the public members of class A is also a public members of class B. But the private members of class A cannot be inherited by class B. Thus, the derived class B will have more members than what it contains at the time of declaration.

In the above example, we can see that the member functions sum() and display() are not able to access the private data member of class A because it cannot be inherited. However, the data member functions sum() and display() of derived class are able to access the private data of class A through an inherited member function get_a() of class A. In the main part of the program, we can also observe that the object of B can directly access the data member **b** of class A, because data member **b** is publically defined in A.

1.6 MULTIPLE INHERITANCE

Now, let us discuss the multiple inheritance. In multiple inheritance, derived class inherits features from more than one parent classes (base classes). In other way we can say that if a class is derived from more than one parent class (base classes), then it is called multiple inheritance. Figure 2.2 depicts multiple inheritance.

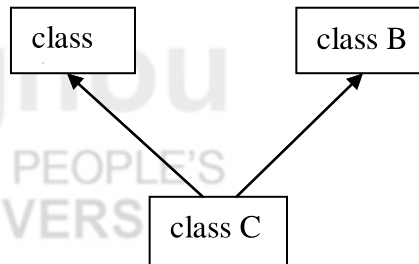

Figure 2.2: Multiple Inheritance

The syntax of declaration of Multiple Inheritance is given below:

```
class A
{
// members of class A
};

class B
{
// members of class B
};

class C :[public/private/protected] A, [public/private/protected] B
{
// members of class C
};
```

In the example 2 given below, it is seen that how multiple inheritance is implemented?

Example 2: Multiple Inheritance

```
#include <iostream.h>
class A
{
int a;
public :
void input_a(void);
void output_a(void);
int get_a(void);
};
class B
{
int b;
public :
void input_b(void);
void output_b(void);
int get_b(void);
};
class C : public A, public B
{
int c,d;
public :
void input_c(void);
void display(void);
void sum(void);
};
void A :: input_a()
{
cout<< "\n Enter the value of a :"<< endl;
cin>>a;
}
void A :: output_a()
{
cout<< "\n The value of a is :"<<a <<endl;
}
int A :: get_a()
{
return a;
}
void B :: input_b()
{
cout<< "\n Enter the value of b :"<< endl;
cin>>b;
}
void B :: output_b()
{
cout<< "\n The value of b is :"<<b<<endl;
}
int B :: get_b()
{
return b;
}
void C :: input_c()
```

```
{
cout<< "\n Enter the value of c :"<< endl;
cin>>c;
}
void C :: sum()
{
d=get_a()+get_b()+c;
}
void C ::display()
{
cout<< "\n The value of c is :"<<c<<endl;
cout<< "\n The value of d (sum of a, b and c) is :"<<d<<endl;
}
void main()
{
C objc;

objc.input_a();          //base class member function
objc.input_b();          //base class member function
objc.input_c();          //derived class member function
objc.output_a();         //base class member function
objc.output_b();         //base class member function
objc.sum();              //derived class member function
objc.display();          //derived class member function
}
```

The output of the example 2 is given below.

Enter the value of a:
10
Enter the value of b:
20
Enter the value of c:
30
The value of a is : 10
The value of b is : 20
The value of c is : 30
The value of d(sum of a, b and c) is : 60

The above example shows multiple inheritance. It contains three classes A, B and C. The class A and class B are parent classes (base classes) and class C is derived class. This class inherits the feature of class A and class B.

1.7 MULTI-LEVEL INHERITANCE

Now, let us discuss the Multi-level inheritance. In multi-level inheritance, the class inherits the feature of another derived class. If a class C is derived from class B which in turn is derived from class A and so on. It is called multi-level inheritance. Figure 1.3 depicts the multi-level inheritance.

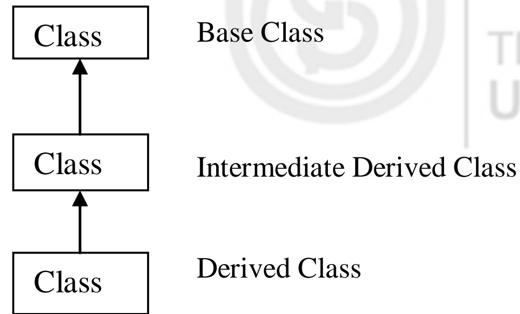

Figure 1.3: Multi-level Inheritance

The syntax of the multi-level inheritance of the above figure is given as follows:

```
class A
{
// members of class A
};

class B : [public/private/protected] A
{
// members of class B
};

class C :[public/private/protected] B
{
// members of class C
};
```

In example 3 given below, it is shown that how multilevel inheritance is implemented.

Example 3: Multi-level Inheritance

```
#include <iostream.h>
class A
{
int a;
public :
void input_a(void);
void output_a(void);
int get_a(void);
};
class B : public A
{
int b;
public :
void input_b(void);
void output_b(void);
int get_b(void);
};
class C : public B
{
int c,d;
public :
```

```
void input_c(void);
void display(void);
void sum(void);
};
void A :: input_a()
{
cout<< "\n Enter the value of a :"<< endl;
cin>>a;
}
void A :: output_a()
{
cout<< "\n The value of a is :"<<a<<endl;
}
int A :: get_a()
{
return a;
}
void B :: input_b()
{
cout<< "\n Enter the value of b :"<< endl;
cin>>b;
}
void B :: output_b()
{
cout<< "\n The Value of b is :"<<b<<endl;
}
int B :: get_b()
{
return b;
}
void C :: input_c()
{
cout<< "\n Enter the value of c :"<< endl;
cin>>c;
}
void C:: sum()
{
d=get_a()+get_b()+c;
}
void C ::display()
{
cout<< "\n The value of c is :"<<c<<endl;
cout<< "\n The value of d (sum of a, b and c) is :"<<d<<endl;
}
void main()
{
C objc;
objc.input_a();          // member function of class A
objc.input_b();          // member function of class B
objc.input_c();          // member function of class C
objc.output_a();         // member function of class A
objc.output_b();         // member function of class B
objc.sum();              // member function of class C
objc.display();          // member function of class C
}
```

The output of the example3 is given below.

Enter the value of a:
10
Enter the value of b:
20
Enter the value of c:
30
The value of a is : 10
The value of b is : 20
The value of c is : 30
The value of d(sum of a, b and c) is : 60

The above example shows multi-level inheritance. It contains three classes A, B, and C. The class A is the base class. The class B is derived class. It inherits the features of A. The class C is derived from intermediate derived class B. The class C, after inheritance from A through B, would contain the following members:

```
private :
int c,d;            //own member
public :
void input_a(void);          // inherited from A via B
void output_a(void);         // inherited from A via B
int get_a(void);             // inherited from A via B

void input_b(void);          // inherited from B
void output_b(void);         // inherited from B
int get_b(void);             // inherited from B
void input_c(void);          // own
void display(void);          //own
void sum(void);              //own
```

☞ Check Your Progress 2

1) What are the different forms of inheritance supported by C++?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

2) Write a interactive program in c++ which reads the two integer number a and b then performs the following operation:
   (i) a+b
   (ii) a-b
   (iii) a*b
   (iv) a/b

   Design the function a+b and a-b in a base class and design a*b and a/b in a derived class.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3) What is multi-level inheritance?

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

4) What are the ways to inherit properties of one class into another class?

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

5) What is containership? How does it differ from inheritance?

………………………………………………………………………………………………

………………………………………………………………………………………………

………………………………………………………………………………………………

1.8 CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES

You have already learnt about the constructors and destructors in Unit 4 of Block 1 of this course. As we know, constructors and destructors play an important role in object initialization and remove the resources allocated to the object. Now, we will discuss as to how the constructors and destructors are used in derived classes.

The constructors are used to initialize object's data members and to allocate the required resources such as memory.

Constructors in Derived Classes:

Can you tell when and why we need a constructor in derived class? Well, the derived class need not have a constructor as long as base class has a no-argument constructor. However, if any base class contains a constructor with arguments (one or more), it is necessary for the derived class to have a constructor and pass the arguments to the base class constructors. In inheritance, generally derived class objects are created instead of the base class. Thus, it makes sense for the derived class to have a constructor and pass arguments to the constructor of the base class. When an object of a derived class is created, the constructor of the base class is executed first and later on the constructor of the derived class. Let us consider example 4 given below in which both base and derived class have constructor with parameter.

Example 4: Parametric constructors in Base and Derived classes

```
#include<iostream.h>
class A
{
private:
int a;
 protected:
 int b;
public:
A(int i, int j)
{
a=i;
b=j;
cout<< "A initialized"<<endl;
```

```
}
void display_ab(void)
{
cout<< "\nThe value of a is : "<<a;
cout<< "\nThe value of b is : "<<b;
}
int get_a(void)
{
return a;
}
};
class B
{
private:
int c;
protected:
int d;
public:
B(int i, int j)
{
c=i;
d=j;
cout<< "\nB initialized"<<endl;
}
void display_cd(void)
{
cout<< "\nThe value of c is : "<<c;
cout<< "\nThe value of d is : "<<d;
}
int get_c(void)
{
return c;
}
};

class C : public B, public A
{
int e,f, total;

public:
void C(int m, int n, int o, int p, int q, int r): A(m,n), B(o,p)
{
e=q;
f=r;
cout<< " \nC initialized";
}
void sum(void)
{

total=get_a()+b+get_c()+d+e+f;
}
void display(void)
{
cout<< "\nThe value of e is : "<<e;
cout<< "\nThe value of f is : "<<f;
cout<< "\nThe sum of a,b,c,d,e and f is : "<<total;
```

```
}
};
void main()
{
C objc(10,20,30,40,50,60);

objc.display_ab();
objc.display_cd();
objc.sum();
objc.display();
}
```

The output of the programme given above is:

B initialized
A initialized
C initialized
The value of a is :10
The value of b is :20
The value of c is :30
The value of d is :40
The value of e is :50
The value of f is :60
The sum of a,b,c,d,e,f is :210

The above example shows the three classes A, B and C. The class A have one parametric constructor, class B have also one parametric constructor and class C are derived class and inherits the features of class A and class B. The class C also have a parametric constructor. It is mandatory to have a parametric constructor in class C. Here you can observe that the class b is initialized first, albeit it appears second in the derived constructor because the class B has been declared first in the derived class header before the class A. You can also see that sum() member function of derived class C which is not able to use data members a and c of the base class A and B due to private members of their respective classes. However, it is able to receive b and d due to protected members of their respective classes. Table 1.2 depicts the order of execution of constructors:

Table 1.2: Order of Execution of Constructors

Method of Inheritance	Order of Execution
class B : public A {};	A() : base constructor B() : derived constructor
class C : public B, public A {};	B() : base constructor A() : base constructor C() : derived constructor
class C : public B, virtual A {};	A() : virtual base constructor B() : base constructor C() : derived constructor
class B : public A {}; class C : public B{};	A() : super base constructor B() : base constructor C() : derived constructor

Destructors in Derived Classes:

Unlike constructor in class hierarchy, destructors are invoked in the reverse order of the constructor invocation. Whenever object goes out of scope, the destructor of that

class, whose constructor was executed last while building object of that class, will be executed first. Let us take example 5 given below which shows the order of calling constructors and destructors in Inheritance:

Example 5: Order of calling of Constructors and Destructors in Inheritance

```
#include<iostream.h>
class A
{
protected:
int a,b;
public:
A(int i, int j)
{
a=i;
b=j;
cout<< "A initialized"<<endl;
}
~A()
{
cout<< "\Destructor in base class A"<<endl;
}
void display_ab()
{
cout<< "\nThe value of a is : "<<a;
cout<< "\nThe value of b is : "<<b;
}
};
class B
{
protected:
int c,d;
public:
B(int i, int j)
{
c=i;
d=j;
cout<< "\nB initialized"<<endl;
}
~B()
{
cout<< "\Destructor in base class B"<<endl;
}
void display_cd()
{
cout<< "\nThe value of c is : "<<c;
cout<< "\nThe value of d is : "<<d;
}
};
class C : public B, public A
{
Int e,f, total;

public:
C(int m, int n, int o, int p, int q, int r): A(m,n), B(o,p)
{
```

```
e=q;
f=r;
cout<< " \nC initialized";
}
~C()
{
cout<< "\Destructor in derived class C"<<endl;
}
void sum(void)
{
total=a+b+c+d+e+f;
}
void display(void)
{
cout<< "\nThe value of e is : "<<e;
cout<< "\nThe value of f is : "<<f;
cout<< "\nThe sum of a,b,c,d,e and f is : "<<total<<endl;
}
};
void main()
{
C objc(10,20,30,40,50,60);
objc.display_ab();
objc.display_cd();
objc.sum();
objc.display();
}
```

The output of the program given above is:

```
B initialized
A initialized
C initialized
The value of a is :10
The value of b is :20
The value of c is :30
The value of d is :40
The value of e is :50
The value of f is :60
The sum of a,b,c,d,e,f is :210
Destructor in derived class C
Destructor in base class A
Destructor in base class B
```

The above example shows the three classes A, B and C. The class A have one parametric constructor, class B have also one parametric constructor and C is derived class that inherits the feature of class A and class B. The class C also have a parametric constructor. It is mandatory to have a parametric constructor in class C. Here you can observe that the class B is initialized first, because the class B has been declared first in the derived class header before the class A. You can also see that the constructors are invoked in the order of B(), A() and C() whereas the destructors are invoked in the order of C(), A() and B() which is in reverse order.

☞ **Check Your Progress 3**

1) Explain as to how ambiguity in member access is resolved.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

2) What are base and derived classes? Create a base class called Stack and a derived class called MyStack. Write an interactive program to show the operations of stack.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3) Discuss the cost of inheritance.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

4) Consider an example of declaring the examination result of BCA students of Indira Gandhi National Open University. Design three classes: Student, Exam, and Result. The Student class has data members such as those representing roll number, name, etc. Create the class Exam by inheriting Student class. The Exam class adds fields representing the marks scored in six subjects. Derive the Result from the Exam class, and it has its own fields such as total-marks. Write an interactive program to model this relationship.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

1.9 SUMMARY

Inheritance is one of the prime features of Object Oriented programming language that helps to represent hierarchical relationship between classes. It is technique of building new classes from the existing classes. It facilitates code re-use and extensibility. It helps organize software components into categories and subcategories resulting in classification of software. Classification is the widely accepted use of inheritance of course other mechanisms may also be used for classification. Use of inheritance helps to generate software systems more quickly and easily using reusable components. The syntax of implementing inheritance through base and derived class is discussed. The concept of reusability, constructors and destructors in derived class are also discussed with examples. In this unit, we studied six different forms of inheritance: simple inheritance, multiple inheritance, multilevel inheritance, hybrid inheritance, hierarchical inheritance and multipath inheritance.

1.10 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) There are many benefits that can be derived from the proper use of inheritance. They are code reuse, ease of code maintenance and extension, and reduction in the time to market. The following situations explain benefits of inheritance:

   - When inherited from another class, the code that provides a behavior required in the derived class does not need to be rewritten.
   - Code sharing can occur at several levels.
   - When multiple classes inherit from the same super class, there is a sufficient guarantee that the behavior they inherit will be same in all cases.

2) Inheritance is suitable where the following situation arises:

   - Whenever there are similarities between two or more classes, you can apply inheritance.
   - If a new class to be defined has certain features in addition to the features of an existing class, you can use inheritance and code only the additional features of this new class.
   - If the relationship between the classes is Is_a, Is_a_Kind_Of, or Is_Link or is Type_Of, you can select inheritance.
   - The hierarchical relationship creates a relationship tree with specialized type branching off from more generalized types. Inheritance is advisable when generalization is fixed and does not require any modification or change.
   - The most common use of inheritance is for specialization.

3) Access specifiers are used to control the accessibility of data members and member functions of class. It helps classes to prevent unwanted exposure of members (data and functions) to outside world.

4) There are many advantages of re-usability. They can be applied to reduce cost, effort and time of software development. It also increases the productivity, portability and reliability of the software product.

5) Because of the high development costs, software should be reusable. If many millions of dollars are to be spent to develop a software system, it makes sense to make its component flexible so they can be re-used when developing a new software system. Re-use can improve reliability, reduce development costs, and improve maintainability. Let us take an analogy of Automobile industry to understand the need of re-usability. Consider the design and creation of a new car model. The automotive engineer does not design a new car from scratch. Rather, the engineer borrows from the design of existing cars. For example, the engine design from an existing car may be used in a new model. If the engine design has been used in a previous model, design problems have likely been resolved. Thus development costs are reduced because a new engine does not need to be designed and tested. Finally, consumer maintenance costs are reduced because machines and

others who must maintain the car are already familiar with the operation of the engine.

Check Your Progress 2

1) There are six forms of inheritance supported by c+ +, namely
   (i) Single Inheritance
   (ii) Multiple Inheritance
   (iii) Multi-level Inheritance
   (iv) Hybrid Inheritance
   (v) Multi-path Inheritance
   (vi) Hierarchical Inheritance

2)

```
#include <iostream.h>
    #include<stdlib.h>
    class A
    {
    int a,b;
    public :
    void input_ab(void);
    int get_a(void);
    int get_b(void);
    int add(void);
    int sub(void);
    };
    class B : public A
    {
    public :
    int mul(void);
    int div(void);
    void display(int opt, int res);
    };
    void A :: input_ab()
    {
    cout<< "\n Enter the value of a and b :"<<endl;
    cin>>a>>b;
    }
    int A :: get_a()
    {
    return a;
    }
    int A :: get_b()
    {
    return b;
    }
    int A :: Add()
    {
    return (a+b);
    }
    int A :: Sub()
    {
    return (a-b);
    }

    void B :: Mul()
    {
```


ignou THE PEOPLE'S UNIVERSITY

```
return(get_a()*get_b());
}
void B :: Div()
{
return(get_a()/get_b());
}
void B :: display(int choice, int result)
{
cout<< "\n The value of a is :"<<a<< endl;
cout<< "\n The value of b is :"<<b<< "endl;
switch(choice)
{
case 1 : cout << "\n The sum of a and b is :" <<result<<endl;
 break;
case 2 : cout << "\n The subtraction of a and b is :" <<result<<endl;
 break;
case 3 : cout << "\n The multiplication of a and b is :" <<result<<endl;
 break;
case 4 : cout << "\n The division of a and b is :" <<result<<endl;
 break;
}
}
void main()
{
B objb;
int choice;
int result;
objb.input_ab();
while (1)
{
cout <<" Operations on two numbers …"<<endl;
cout <<" 1. Addition"<<endl;
cout <<" 2. Subtraction"<<endl;
cout <<" 3. Multiplication"<<endl;
cout <<" 4. Division"<<endl;
cout <<" 5. Quit"<<endl;
cout <<" Enter choice:"<<endl;
cin>>choice;
switch(choice)
{
case 1 : result=objb.add();
objb.display(choice,result);
 break;
case 2 : result=objb.sub();
objb.display(choice,result);
 break;
case 3 : result=objb.mul();
objb.display(choice,result);
 break;
case 4 : if (b!=0)
{
result=objb.div();
objb.display(choice,result);
}
else
cout<< "\nDivide by zero error:"<<endl;
 break;
```

```
case 5 : exit(1);
 break;
default :
cout <<" Bad option selected"<<endl;
continue;
}
}
}
```

3) Derivation of a class from another derived class is called multi-level inheritance. The multi-level inheritance mechanism can be extended to any levels.

4) There are two ways to inherit properties of one class into another class as follows:
   (i) Inheritance
   (ii) Object Composition

5) The use of objects in a class as data members is referred to as object composition. Thus, we can say that an object can be collection of many other objects. This relationship is called has-a relationship or containership. This relationship is also called nesting of objects. In many situations, inheritance and containership relationships can serve the same purpose. Containership does not provide flexibility of ownership. Inheritance relationship is simpler to implement and offers a clearer conceptual framework.

Check Your Progress 3

1) Ambiguity is a problem that surfaces in certain situations involving multiple inheritance. Consider the following cases:

   - Base classes having functions with the same name.
   - The class derived from these base classes is not having a function with the name as those of its base classes.
   - Members of a derived class or its objects referring to a member whose name is the same as those of base classes.

   These situations create ambiguity in deciding which of the base class's function has to be referred. This problem is resolved by using the scope resolution operator which is given as follows:

   ObjectName.BaseClassName::MemberName(...).

2) Inheritance is a property by which one class inherits the feature of another class. The class which inherits the feature from another class is called derived class and class from which another class takes feature is called base class.

```
#include <iostream.h>
#include<stdlib.h>
#define Max_Size 5            //Maximum stack size
class Stack
{
protected :
int stack[Max_Size];
int top;
public :
Stack (void)
```

```
void push (int item);
void pop (int &item);
};
class MyStack : public Stack
{
public :
int push(int item);
int pop(int &item);
void stackContent(void);
};
Stack::Stack()
{
top=-1;              //Stack empty
}
void Stack::push(int item)
{
top++;
stack[top]=item;
}
void Stack::pop(int &item)
{
item=stack[top];
top--;
}
int MyStack :: push(int item)
{
If (top<Max_Size-1)
{
Stack::push(item);
return 1;                              //push operation successful
}
cout<< "\n Stack Overflow :"<< endl;
return 0;
}
int MyStack :: pop(int &item)
{
If (top>=0)
{
Stack::pop(item);
return 1;                              //push operation successful
}
cout<< "\n Stack Underflow :"<< endl;
return 0;
}
void MyStack :: stackContent(void)
{
int stop;
stop=top;
for (int i=0; i<=stop;i++)
cout<< ":"<<stack[i];
}
void main()
{
MyStack stack;
int choice;
int item;
while (1)
```

```
{
cout << "\nStack Operation …"<<endl;
cout << "\n1. Item to push?"<<endl;
cout << "2. Item to pop"<<endl;
cout << "3. Quit"<<endl;
cout << "Enter choice:"<<endl;
cin>>choice;
switch(choice)
{
case 1 :cout<< "Enter the item:"<<endl;
cin>>item;
cout<< "\n Stack content before push operation:";
stack.stackContent();
if ((stack.push(item))==1)
{
cout<< "\n Stack content after push operation:";
stack.stackContent();
}
 break;
case 2 : cout<< "\n Stack content before pop operation:";
stack.stackContent();
if ((stack.pop(item))==1)
{
cout<< "\n Stack content after pop operation:";
stack.stackContent();
cout<< "popped item:" <<item;
}
 break;
case 3 : exit(1);
 break;
default :
cout <<" Bad option selected"<<endl;
continue;
}
}
}
```

3) Despite the advantages of inheritance, it incurs compiler overhead. In inheritance relationship, there are certain members in the base class that are not at all used; however, data space is allocated to them. This necessitates the need for specialized inheritance which is complex to develop. The following are some of the perceived costs of inheritance:

- Inherited methods, which must be prepared to deal with arbitrary subclasses, are often slower than specialized codes.
- Message passing by its very nature is a more costly than the invocation of simple procedures.
- Albeit object oriented programming is often touted as a solution to the problem of software complexity, overuse or improper use of inheritance can simply transfer one form of complexity to another form.

4)

```
#include<iostream.h>
      #include<string.h>
      class Student
      {
      int roll_no;
      char name[25];
      public:
      void ReadStudentData(void);
      void DisplayStudentData(void);
      };
      class Exam :public Student
      {
      protected:
      int marks[6];
      public :
      void ReadExamMarks(void);
      void DisplayExamMarks(void);
      };
      class Result : public Exam
      {
      int total_marks;
      public :
      void Display(void);
      };
      void Student :: ReadStudentData()
      {
      cout<<"\n Enetr the Name:"<<endl;
      cin>>name;
      cout<< "\n Enter the Roll No.:"<<endl;
      cin>>roll_no;
      }
      void Student :: DisplayStudentData()
      {
      cout<<"\n Name :"<<name<<endl;
      cout<<"\n Roll No. :"<<roll_no;
      }
      void Exam::ReadExamMarks()
      {
      cout <<"\nEnter Marks :"<<endl;
      for (int i=0; i<6; i++)
      {
      cout<<"\n Marks scored in subject"<<i+1<<"<Max:100>"<<endl;
      cin>>marks[i];
      }
      }
      void Exam::DisplayExamMarks()
      {
      for (int i=0; i<6; i++)
      cout<<"\n Marks scored in subject"<<i+1<<":"<<marks[i];
      }
      void Result::Display()
      {
total_marks=0;
for (int i=0; i<6; i++)
total_marks=total_marks+marks[i];
cout<<"\n Total Marks scored in six subjects :"<<total_marks;
}
```

```
void main()
{
Result objr;
objr.ReadStudentData();
objr.ReadExamMarks();
objr.DisplayExamMarks();
objr.Display();
}
```

1.11 FURTHER READINGS

1) B. Stroustrup, *The C++ Programming Language*, Third Edition, Pearson/Addison-wesley Publication, 1997.

2) K. R. Venu Gopal, Raj Kumar Buyya, T Ravishankar, *Mastering C++*, Tata-McGraw-Hill Publishing Company Limited, New Delhi, 2004.

3) E. Balagurusamy, *Object Oriented Programming with C++*, Tata Mc-Graw-Hill Publishing Company Limited, New Delhi, 2001.

4) N. Barkakati, *Object Oriented Programming in C++*, Prentice-Hall of India.