UNIT 1 INHERITANCE

Structure **Page No.**

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:

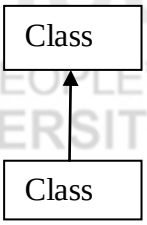


Figure 1.1: Single Inheritance

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.

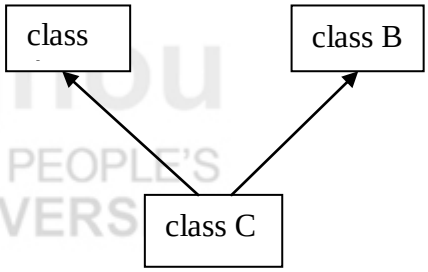


Figure 2.2: Multiple Inheritance

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.

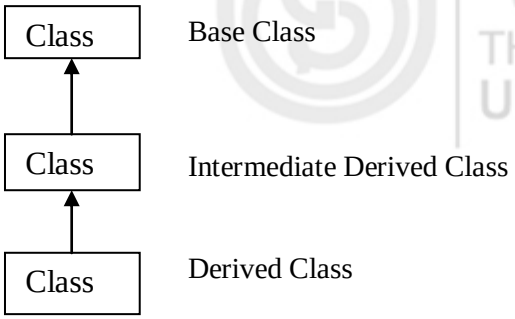


Figure 1.3: Multi-level Inheritance

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()
    {
```

```
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.

UNIT 2 OPERATOR OVERLOADING

Structure **Page Nos.**

2.0 INTRODUCTION

When you create an object (a variable), you give a name to a region of storage. A function is a name for an action. By making up names to describe the system at hand, you create a program that is easier for people to understand and change. Function overloading is one of the defining aspects of the C++ programming language. Not only does it provide support for compile time polymorphism, it also adds flexibility and convenience. In addition, function overloading means that if you have two libraries that contain functions of the same name, they won't conflict as long as the argument lists are different. We'll look at all these factors in detail across this unit.

In C++, you can overload most operators so that they perform special operations relative to classes that you create. When an operator is overloaded, none of its original meanings are lost. Operator overloading allows the full integration of new class type into programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O. Function overloading and operator overloading really aren't very complicated. By the time you reach the end of this unit, you shall learn when to use them and also underlying mechanisms that implement them during compiling and linking.

2.1 OBJECTIVES

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

- explain the need of overloading;
- describe mechanism of function overloading;
- learn the use and application of default argument;

- understand how to overload (redefine) operators to work with new types;
- understand how to convert objects from one class to another;
- learn when to, and when not to, overload operators; and
- learn about several cases of overloaded operators.

2.2 FUNCTION OVERLOADING

Function overloading refers to using the same thing for different purposes. C++ permits the use of different functions with the same name. However such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both. These functions can perform a variety of different tasks. This process of using two or more functions with the same name but differing in the signature is called function overloading. It is only through these differences that the compiler knows which function to call in any given situations.

Overloading of functions with different return types are not allowed.

Consider the following function:

```
int add (int a, int b)
{
return a + b;
}
```

This trivial function adds two integers. However, what if we also need to add two floating point numbers? This function is not at all suitable, as any floating point parameters would be converted to integers, causing the floating point arguments to lose their fractional values.

One way to work around this issue is to define multiple functions with slightly different names:

```
int add (int a, int b)
{
return a + b;
}
double addition (double p, double q)
{
return p + q;
}
```

However, for best effect, this requires that you define a consistent naming standard, remember the name of all the different flavors of the function, and call the correct one (calling add() for addition of double type numbers with integer parameters may produce the wrong result due to precision issues).

Function overloading provides a better solution. Using function overloading, we can declare another add () function that takes double parameters:

double add (double p, double q)

We now have two version of add ():

```
{
return p + q;
}
```

int add (int A, int B); // integer version

double add (double P, double Q); // floating point version

Which version of add () gets called depends on the arguments used in the call — if we provide two ints, C++ will know we mean to call add(int, int). If we provide two floating point numbers, C++ will know we mean to call add (double, double). In fact, we can define as many overloaded add () functions as we want, so long as each add () function has unique parameters.

Consequently, it's also possible to define add () functions with a differing number of parameters:

```
int add (int a, int b, int c)
{
return a + b + c;
}
```

Even though this add () function has 3 parameters instead of 2, because the parameters are different than any other version of add (), this is valid.

Function overloading is one of the most powerful features of C++ programming language. It forms the basis of polymorphism (compile-time polymorphism). Most of the time you'll be overloading the constructor function of a class.

2.2.1 How Function Overloading is Achieved

One thing that might be coming to your mind is, how will the compiler know when to call which function, if there are more than one function of the same name. The answer is, you have to declare functions in such a way that they differ either in terms of the number of parameters or in terms of the type of parameters they take. What that means is, nothing special needs to be done, you just need to declare two or more functions having the same name but either having different number of parameters or having parameters of different types. In overloaded functions, the function call determines which function definition will be executed. The biggest advantage of overloading is that it helps us to perform same operations on different datatypes without having the need to use separate names for each version. For example, an overloaded test() function handles different types of data as shown below:

```
// Declarations
int test (int x);                    //prototype 1
int test (int x, int y);             //prototype 2
double test (int a, double b);       //prototype 3
// Function call
Cout<< test (23);                    //uses prototype 1
Cout<< test (45, 55);                //uses prototype 2
Cout<< test (12, 3.25);          //uses prototype 3
```

Following example illustrates the function overloading:

```
// Example: Function overloading to find the absolute value of any number int, long, float,
double
#include<iostream>
using namespace std;
int abslt(int );
long abslt(long );
float abslt(float );
double abslt(double );
int main()
{
int intgr=-5;
long lng=34225;
float flt=-5.56;
double dbl=-45.6768;
cout<<\" absoulte value of \"<<intgr<<\" = \"<<abslt(intgr)<<endl;
cout<<\" absoulte value of \"<<lng<<\" = \"<<abslt(lng)<<endl;
cout<<\" absoulte value of \"<<flt<<\" = \"<<abslt(flt)<<endl;
cout<<\" absoulte value of \"<<dbl<<\" = \"<<abslt(dbl)<<endl;
}
int abslt(int num)
{
if(num>=0)
return num;
else
 return (-num);
}
long abslt(long num)
{
if(num>=0)
return num;
else return (-num);
}
float abslt(float  num)
{
if(num>=0)
return num;
else return (-num);
}
double abslt(double  num)
{
if(num>=0)
return num;
else return (-num);
}
```

Output

- absoulte value of -5 = 5
- absoulte value of 34225 = 34225
- absoulte value of -5.56 = 5.56
- absoulte value of -45.6768 = 45.6768

The use of overloading may not have reduced the code complexity /size but has definitely made it easier to understand and avoided the necessity of remembering different names for each version function which perform identically the same task.

Example: overloading functions that differ in terms of number of parameters

```
#include<iostream.h>
// function prototype
int func(int i);
int func(int i, int j);
void main(void)
  {
   cout<<func(15);                 //func(int i)is called

   cout<<func(15,15);              //func(int i, int j) is called
  }
  int func(int i)
  {
   return i;
  }
  int func(int i, int j)
  {
   return i+j;
  }
```

Example: overloading functions that differ in terms of types of parameters

```
#include<iostream.h>
 //function prototypes
  int func(int i);
  double func(double i);
  void main(void)
  {
   cout<<func(15);                 //func(int i) is called
   cout<<func(15, 155);            //func(double i) is called
  }
  int func(int i)
  {
   return i;
  }
double func(double i)
  {
   return i;
  }
```

2.2.2 How Function Calls are Matched with Overloaded Functions

Making a call to an overloaded function results in one of three possible outcomes:

1) A match is found. The call is resolved to a particular overloaded function.
2) No match is found. The arguments can not be matched to any overloaded function.
3) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

1) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
void Print(char *szValue);
void Print(int nValue);
Print(10); // exact match with Print(int)
```

Although 10 could technically match Print(char*), it exactly matches Print(int). Thus Print(int) is the best match available.

2) If no exact match is found, C++ tries to find a match through promotion. In the lesson on type conversion and casting, we covered how certain types can be automatically promoted via internal type conversion to other types.

To summarize,

a) Char, unsigned char, and short is promoted to an int.
b) Unsigned short can be promoted to int or unsigned int, depending on the size of an int
c) Float is promoted to double
d) Enum is promoted to int

For example:

```
void Print(char *szValue);
void Print(int nValue);
Print('a'); // promoted to match Print(int)
```

In this case, because there is no Print(char), the char ‘a’ is promoted to an integer, which then matches Print(int).

3) If no promotion is found, C++ tries to find a match through standard conversion. Standard conversions include:

   a) Any numeric type will match any other numeric type, including unsigned (eg. int to float)
   b) Enum will match the formal type of a numeric type (eg. enum to float)
   c) Zero will match a pointer type and numeric type (eg. 0 to char*, or 0 to float)
   d) A pointer will match a void pointer

For example:

```
void Print(float fValue);
void Print(struct sValue);
Print('a'); // promoted to match Print(float)
```

In this case, because there is no Print(char), and no Print(int), the 'a' is converted to a float and matched with Print(float).

Note that all standard conversions are considered equal. No standard conversion is considered better than any of the others.

4) Finally, C++ tries to find a match through user-defined conversion. Although we have not covered classes yet, classes (which are similar to structs) can define conversions to other types that can be implicitly applied to objects of that class.

For example, we might define a class X and a user-defined conversion to int.

```
class X; // with user-defined conversion to int
void Print(float fValue);
void Print(int nValue);
X cValue; // declare a variable named cValue of type class X
Print(cValue); // cValue will be converted to an int and matched to Print(int)
```

Although cValue is of type class X, because this particular class has a user-defined conversion to int, the function call Print(cValue) will resolve to the Print(int) version of the function.

2.3 FUNCTION OVERLOADING AND AMBIGUITY

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be ambiguous. Ambiguous statements are errors, and programs containing ambiguity will not compile.

2.3.1 Ambiguous Matches

If every overloaded function has to have unique parameters, how is it possible that a call could result in more than one match? Because all standard conversions are considered equal, and all user-defined conversions are considered equal, if a function call matches multiple candidates via standard conversion or user-defined conversion, an ambiguous match will result.

For example:

```
void Print(unsigned int nValue);
void Print(float fValue);
Print('p');
Print(10);
Print(1.14);
```

In the case of Print('p'), C++ can not find an exact match. It tries promoting 'a' to an int, but there is no Print (int) either. Using a standard conversion, it can convert 'a' to both an unsigned int and a floating point value. Because all standard conversions are considered equal, this is an ambiguous match.

Print(10) is similar. 10 is an int, and there is no Print(int). It matches both calls via standard conversion.

Print(1.14) might be a little surprising, as most programmers would assume it matches Print(float). But remember that all literal floating point values are doubles unless they have the 'f' suffix. 1.14 is a double, and there is no Print (double). Consequently, it matches both calls via standard conversion.

Ambiguous matches are considered a compile-time error. Consequently, an ambiguous match needs to be disambiguated before your program will compile. There are two ways to resolve ambiguous matches:

1) Often, the best way is simply to define a new overloaded function that takes parameters of exactly the type you are trying to call the function with. Then C++ will be able to find an exact match for the function call.

2) Alternatively, explicitly cast the ambiguous parameter(s) to the type of the function you want to call. For example, to have Print(10) call the Print(unsigned int),

2.4 MULTIPLE ARGUMENTS

If there are multiple arguments, C++ applies the matching rules to each argument in turn. The function chosen is the one for which each argument matches at least as well as all the other functions, with at least one argument matching better than all the other functions. In other words, the function chosen must provide a better match than all the other candidate functions for at least one parameter, and no worse for all of the other parameters.

In the case that such a function is found, it is clearly and unambiguously the best choice. If no such function can be found, the call will be considered ambiguous.

☞ **Check Your Progress 1**

1) What is function overloading?

……

……

……

2) How function calls are matched with overloaded functions?

……

……

……

3) What are the main applications of function overloading?

……

……

……

4) Multiple choice questions:

i) In C++, dynamic memory allocation is accomplished with the operator ____

a) new

b) this

c) malloc( )

d) delete

ii) The operator that cannot be overloaded is

a) ++

b) ::

c) ( )

d) ~

iii) The operator << when overloaded in a class

a) must be a member function

b) must be a non member function

c) can be both (A) & (B) above

d) cannot be overloaded

iv) Identify the operator that is NOT used with pointers

a) ->

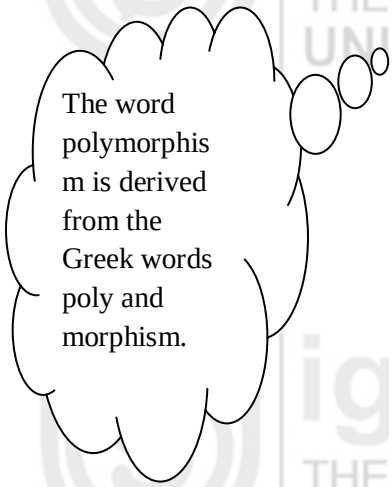
b) &

c) *

d) >>

2.5 OPERATOR OVERLOADING

We already know that a function can be overloaded (same function name having multiple bodies). The concept of overloading a function can be applied to operators as well. For example, in C++ we can multiply two variables of user-defined data type with the same syntax that is applied to the basic data type. This means that C++ has the ability to provide the operators with a special meaning for data type. The



mechanism which provides this special meaning to operators is called operator overloading. The operator overloading feature of C++ is one of the methods of realizing polymorphism. Here, poly refers to many or multiple and morphism refers to actions, i.e. performing many actions with a single operator. Thus operator overloading enables us to make the standard operators, like +, -, * etc, to work with the objects of our own data types. So what we do is, write a function which redefines a particular operator so that it performs a specific operation when it is used with the object of a class. Operator overloading is very exciting feature of C++. The concept of operator overloading can also be applied to data conversion. It enhances the power of extensibility of C++. Thus operator overloading concepts are applied to the following two principle areas:

- Extending capability of operators to operate on user defined data, and
- Data conversion

This session deals with overloading of operators to make Abstract Data Types (ADTS) more natural, and closer to fundamental data types.

2.5.1 Why to Overload Operators

Most fundamental data types have pre-defined operators associated with them. For example, the C++ data type float, together with the operators +, -, *, and /, provides an implementation of the mathematical concepts of an integer. This is purely a convenience to the user of a class. Operator overloading isn't strictly necessary unless other classes or functions expect operators to be defined.

To make a user-defined data type as natural as a fundamental data type, the user-defined data type must be associated with the appropriate set of operators. Operators are defined as either member functions or friend functions. The purpose of operator overloading is to make programs clearer by using conventional meanings for ==, [], +, etc. Operators can be overloaded in any way from those available like globally or on the basis of class by class. While implementing the operator overloading this can be achieved by implementing them as functions. Whether it improves program readability or causes confusion depends on how well you use it. In any case, C++ programmers are expected to be able to use it.

The user can understand the operator notation more easily as compared to a function call because it is closer to the real-life implementation. Thus, by associating a set of meaningful operators, manipulation of an ADT can be done in a conventional and simpler form. Associating operators with an ADT involves overloading them. Although the semantics of any operator can be extend, we can't change its syntax, associativity, precedence etc.

2.5.2 Member vs. Non-member Operators

There are two groups of operators in terms of how they are implemented: member operators and non-member operators. The distinction between the two is the same as it is with methods and functions.

Member operators are operators that are implemented as member functions (methods) of a class.

Non-member operators are operators that are implemented as regular, non-member functions.

Some operators are required to be member operators, others must be non-member operators, and some can be both.

Note: Operator when overloaded is called operator function. Operator function is declared with operator keyword that precedes the operator that has to be overloaded. Again overloaded operator and functions are not the same, but the same way as overloaded functions are distinguished by the number and type of operands, the mechanism is applicable to the operators.

2.5.3 General rules for Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators.

(i) You cannot define new operators, such as **.

(ii) You cannot redefine the meaning of operators when applied to built-in data types.

(ii) Overloaded operators must either be a non static class member function or a global function.

(iii) Obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type Point," expecting 2 to be added to the x coordinate and 3 to be added to the y coordinate.

(iv) Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.

(v) Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.

(vi) If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.

(viii) Overloaded operators cannot have default arguments.

(ix) All overloaded operators except assignment (operator=) are inherited by derived classes.

(x) The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

(xi) Overloading + doesn't overload +=, and similarly for the other extended assignment operators.

(xii) You may not redefine ::, sizeof, ?:, or . (dot).

(xiii) =, [], and -> must be member functions if they are overloaded.

(xiv) ++ and -- need special treatment because they are prefix and postfix operators.

(xv) There are special issues with overloading assignment (=). Assignment should always be overloaded if an object dynamically allocates memory.

Table 2.1 shows the operators which can be overloaded

Table 2.1: List of Operators that can be overloaded

+	-	*	/	%	^	&	|	~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	<<=	>>=	==	!=	<=	>=	&&
||	++	--	,	->*	->	( )	[ ]	new	delete
new[]	delete[]								

Following list shows the operators which can't be overloaded

.	.*	::	sizeof	?:

2.5.4 Why Some Operators can't be Overload

Generally the operators that can't be overloaded are like that because overloading them could and probably would cause serious program errors or it is syntactically not possible. Following section describe the reason for not overloading some of the operators defined in C++ language.

a) **:: (scope resolution)), . (member selection), and .* (member selection through pointer to function)**

For the operators that cannot be overloaded like :: (scope resolution), . (member selection), and .* (member selection through pointer to function), we are quoting from Stroustrup's 3rd edition of 'The C++ Programming Language', section 11.2 (page 263), these operators 'take a name, rather than a value, as their second operand and provide the primary means of referring to members. C++ has no syntax for writing code that works on names rather than values so syntactically these operators can not be overload.

The right hand side of operators **.** (member selection/access) , .* (member selection through pointer to function) and :: (scope resolution) are names of things, e.g. name of a class member. In other words: above three unloaded operators use name instead of operand, so we can`t pass any name (either of variable, class) to any function. We must have to pass the operand for that.

b) **size of operator**

The size of operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you can not overload it with your own runtime code. It is syntactically not possible to do.

c) **? : (conditional operator)**

All operators that can be overloaded must have at least one argument that is a user-defined type. That means you can't overload that operator which has no arguments.

But it does not suite for ? : (conditional operator) as it does not take name as parameter. The reason we cannot overload ? : is that it takes 3 argument rather than 2 or 1. There is no mechanism available by which we can pass 3 parameters during operator overloading.

2.6 DEFINING OPERATOR OVERLOADING

You can overload or redefine the most of built-in operators in C++. These operators can be overloaded globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be member functions or global functions.
An overloaded operator is called an operator function. You declare an operator function with the keyword operator preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

2.6.1 Syntax

Defining an overloaded operator is like defining a function, but the name of that function is operator #, in which # represents the operator that's being overloaded. The number of arguments in the overloaded operator's argument list depends on two factors:

1. Whether it's a unary operator (one argument) or a binary operator (two arguments).
2. Whether the operator is defined as a global function (one argument for unary, two for binary) or a member function (zero arguments for unary, one for binary – the object becomes the left-hand argument).

It may look like following way:

```
Return_type  class_name :: operator op (op_argument_list)
{
Body of function
}
```

Here return_type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator. Operator op is the name of function. They may be also friend functions. Member function has no argument for unary operator and one argument for binary operator. This is because the object used to invoke the member function is passed implicitly and so it available to member function. This case is not with the friend function. Friend function will have one argument for unary operator and two arguments for binary operator. All the arguments may be passed either by value or by reference. Following lines define the steps in overloading the operators

a. Build a class that defines the data type that is going to use in operation of overloading.
b. Declare the operator function *operator op()* in public area of class.
c. Now define the operator function to implement the required operations.

Invoking Operator Function

(i) For member Function

a. For unary operator: **op m** or **m op**
b. For binary operator: **m op n** or **m.operator op (n)**

(ii) For friend Function

a. For unary operator: **operator op (m)**
b. For binary operator: **operator op (m, n)**

Overloading unary operators

To declare a unary operator function as a non static member, you must declare it in the form:
return_type **operator** *op***();**
where *op* is one of the operators listed in the preceding table.
To declare a unary operator function as a global function, you must declare it in the form:
return_type **operator** *op***(** *arg* **);**
Where *op* is described for member operator functions and the *arg* is an argument of class type on which to operate. An overloaded unary operator may return any type.

Some examples of operator overloading:

Unary operator overloading
Example:

```
#include<iostream.h>
#include<conio.h>
Class complex
{
int real, imaginary;
Public:
complex()
{
}
complex(int a, int b)
{
real = a;
imaginary = b;
}
void operator-();
void display()
{
cout<<"Real value"<<real<<endl;
cout<<"imaginary value"<<imaginary<<endl;
}
};
void complex::operator-()
{
real = -real;
imaginary = -imaginary;
}
void main()
{
Clrscr();
complex c1(10,12);
cout<< "real and imaginary value befor operation"<<endl;
c1.display();
c1;        //c1- /*It will give error*/
cout<< "real and imaginary' value after operation"<<endl;
c1.display();
getch();
}
```

Output

Real and imaginary value before operation
10 20
Real and imaginary value after operation
-10 -20

Explanation: In the above example operator, overloading is of unary operator declared in class complex. Outline function make changes values of real and imaginary part by negation. When this is called, it has to be remembered that operand must precede operator otherwise there will be an error. The reason is simple, 'operator' is the function name that is called with operand.

Unary operator overloading using friend function

Example:

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real;
int imaginary;
public:
  complex()                    //default constructor
{
}
complex(int a, int b)
{
        real = a;
        imaginary = b;
}
friend void operator -(); //operator overloading prototype
void display()
 {
cout<<"real value is"<<real<<endl;
cout <<"imaginary value is:"<<imaginary<<endl;
 }
};
//definition of operator overloading function
friend void complex :: operator - (complex &c)
{
        c.real = - c.real;
        c.imaginary = - c.imaginary;
}
void main()
{
clrscr();
complex c1(50,100);
cout<<"real and imaginary value before operation"<<endl;
c1.diaplay();     //calling operator overloading function
-c1;
//c1-;              It will give you an error.
cout<<"real and imaginary value after operator"<<endl;
 c1.display();
 getch();
}
```

Output:

Real and imaginary value before operation

real value is 50 and imaginary value 100

real and imaginary value after operation

real value is -50 and imaginary value -100

2.7.2 Binary Operator Overloading

Example :

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real, imaginary;
public:
complex()
{
}
complex(int a,int b)
{
real = a;
imaginary = b;
}
void operator +(complex c);
};
void complex::operator+(complex c)
{
complex temp;
temp.real = real + c.real;
temp.imaginary = imaginary + c.imaginary;
cout<<"real sum is"<<temp.real<<endl;

}
void main()
{
clrscr();
complex c1(10,20);
complex c2(20,30);
c1+c2;
getch();
}
```

Output

Real sum is 30

Imaginary sum is 50

Explanation: Output is easily predictable, and most of the program is same as above program. Difference lies in void complex :: operator +(complex c) if we compare this statement with our conventional outline statement

<return type > <classname>::<function name> <argument list>

We find that return type is void. Class name is complex. Function name is 'operator+' and it takes one argument which is of class complex type object.

When this function is called in main with statement c1+c2, c1 is object that invokes function 'Operator+' and c2 is passed as argument. Now with in the function real and imaginary parts are c1 will be directly accessible (as it invokes function) while real and imaginary of c2 are taken using formal argument 'complex c'. Here we can find that keyword 'operator' is inserted automatically whenever invoking object is user defined type with operator.

Binary operator overloading using friend function

Example

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real;
int imaginary;
public:
  complex(){ }                    //default constructor
complex(int a, int b)
{
        real = a;
        imaginary = b;
}
friend complex operator +(comlex c1, complex c2);
void display()
 {
cout<<"real value is"<<real<<endl;
cout <<"imaginary value is:"<<imaginary<<endl;
 }
};
complex  operator + (comlex c1, complex c2)
{
        complex tmp;
        tmp.real =  c1.real + c2.real
        tmp.imaginary =  c1.imaginary + c2.imaginary;
        return(tmp);
}
void main()
{
clrscr();
complex c1(10,20);
complex c1(30,50);
ccomlex c3 = c1 + c2;
c3.display();
 getch();
}
```

Output:

real value is : 40

imaginary value is : 70

Unary Operators (Increment and decrement)

The overloaded ++ and – – operators present a dilemma because you want to be able to call different functions depending on whether they appear before (prefix) or after (postfix) the object they're acting upon. The solution is simple, but people sometimes find it a bit complex and confusing at first. When the compiler sees, for example, ++a (a pre-increment), it generates a call to operator ++(a); but when it sees a++, it generates a call to operator ++(a, int.) That is, the compiler differentiates between the two forms by making calls to different overloaded functions.

The following example overloads the unary operators:

Example

```
#include<iostream>
using namespace std;
//Increment and decrement overloading
class Inc {
        private:
                int count ;
        public:
                Inc() {
                        //Default constructor
                        count = 0 ;       }
                Inc(int C) {
                        // Constructor with Argument
                        count = C ;
                }
                Inc operator ++ () {
                        // Operator Function Definition
                        return Inc(++count);
                }
                Inc operator -- () {
                        // Operator Function Definition
                        return Inc(--count);
                }
                void display(void) {
                        cout << count << endl ;
                }
};
```

```
void  main(void)
 {
        Inc a, b(10), c, d, e(5), f(10);
        cout << "Before using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        ++a;
        b++;
        cout << "After using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        c = ++a;
        d = b++;
        cout << "Result prefix (on a) and postfix (on b)\n";
        cout << "c = ";
        c.display();
        cout << "d = ";
        d.display();
        cout << "Before using the operator --()\n";
        cout << "e = ";
        e.display();
        cout << "f = ";
        f.display();
        --e;
        f--;
        cout << "After using the operator --()\n";
        cout << "e = ";
        e.display();
        cout << "f = ";
        f.display();
}
```

Output:

Before using the operator ++()

a = 0

b = 10

After using the operator ++()

a = 1

b = 11

Result prefix (on a) and postfix (on b)

c = 2

d = 12

Before using the operator --()

e = 5

f = 10

After using the operator - -()

e = 4

f = 9

Note :

When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type int; specifying any other type generates an error.

Overloading assignment operator ('=')

```
class Complex
{
private:
        double real, imag;
public:
        Complex(){
                real = 0;
                imag = 0;
        }
        Complex(double r, double i) {
                real = r;
                imag = i;
        }
        double getReal() const {
                return real;
        }
        double getImag() const {
                return imag;
        }
        Complex & operator=(const Complex &);
};
Complex & Complex::operator=(const Complex& c) {
        real = c.real;
        imag = c.imag;
        return *this;
}
#include <iostream>
int main()
{
        using namespace std;

        Complex c1(5,10);
Complex c2(50,100);
        cout << "c1= " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
```

```
        cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;
        c2 = c1;
        cout << "assign c1 to c2:" << endl;
        cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;
}
```

Output:

```
c1= 5+10i
c2= 50+100i
assign c1 to c2:
c2= 5+10i
```

Overloading the << Operator

Output streams use the << operator for standard types. We can also overload the << operator for our own classes.

Actually, the << is left shift bit manipulation operator. But the ostream class overloads the operator, converting it into an output tool. The cout is an ostream object and that it is smart enough to recognize all the basic C++ types. That's because the ostream class declaration includes an overloaded operator<<() definition for each of the basic types.

Example

```
#include <iostream>
using namespace std;
class Complex
{
private:
        double real, imag;
public:
        Complex(){
                real = 0;
imag = 0;
        }
        Complex(double r, double i) {
                real = r;
                imag = i;
        }
        double getReal() const {
                return real;
        }
        double getImag() const {
                return imag;
        }
        Complex & operator=(const Complex &);
        const Complex operator+(const Complex & );
        Complex & operator++(void);
        Complex  operator++(int);
        /*friend const
                Complex operator+(const Complex&, const Complex&); */
        friend ostream& operator<<(ostream& os, const Complex& c);
};
```

```
Complex & Complex::operator=(const Complex& c) {
        real = c.real;
        imag = c.imag;
        return *this;
}
const Complex Complex::operator+(const Complex& c) {

Complex temp;
        temp.real = this->real + c.real;
        temp.imag = this->imag + c.imag;
        return temp;
}
//pre-increment
Complex & Complex::operator++() {
        real++;
        imag++;
        return *this;
}
//post-increment
Complex Complex::operator++(int) {
          Complex temp = *this;
          real++;
          imag++;
          return temp;
}
/* This is not a member function of Complex class */
/*const Complex operator+(const Complex& c1, const Complex& c2) {
          Complex temp;
          temp.real = c1.real + c2.real;
          temp.imag = c1.imag + c2.imag;
          return temp;
}*/
ostream& operator<<(ostream &os, const Complex& c) {
        os << c.real << '+' << c.imag << 'i' << endl;
        return os;
}
int main()
{
        Complex c1(5,10);
        cout << "c1 = " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
        cout << "Using overloaded << " << endl;
        cout << "c1 = " << c1 << endl;
}
```

Output:

c1 = 5+10i

Using overloaded <<

c1 = 5+10i

Note that we just used:

cout << "c1 = " << c1 << endl;

Note that when we do

cout << c1;

it becomes the following function call:

operator<<(cout, c1);

operator overloading for strings

```
#include<iostream.h>
 #include<string.h>
 #include<conio.h>
class string
   {
      private:
       char str[80];
      public:
       string()  { strcpy(str,"ttt"); }
string(char s[])  { strcpy(str,s); }
       void display()  { cout<<str<<endl; }
           string operator+(string );
   };
string string::operator+(string ss){
      string temp;
      if(strlen(str)+strlen(ss.str)<80)
     {
       strcpy(temp.str,str);
       strcat(temp.str,ss.str);
     }
      else
     {
       cout<<"string overflow"<<endl;
       temp=0;
     }
      return temp;     }
 main()
   {
      clrscr();
      string s1(" Operator");
      string s2(" Overloading");
      string s3;
      s1.display();
      s2.display();
      s3=s1+s2;
      s3.display();
      getch();
      return 0;
   }
```

Output:

Operator

Overloading

Operator Overloading

Operator overloading from String object to basic string

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class string
{
  char *p;
  int len;
  public:
   string()
   {}
   string(char *a)
   {
     len=strlen(a);
     p=new char[len+1];
     strcpy(p,a);
   }
   operatorchar*()
   {
     return(p);
   }
   void display()
   {
     cout<<p;
   }
};
void main()
{
 clrscr();
 string o1="IGNOU";
 cout<<"String of Class type : ";
 o1.display();
 cout<<endl;

 char *str=o1;
 cout<<"String of Basic type : "<<str;
 getch();
}
```

Output:

String of Class type : IGNOU

String of Blass type : IGNOU

Instream and outstream operator overloading

Example

```
Class Complex
{
Public:
Friend istream & operator >>(istream &is, Complex &c2);
Friend ostream & operator <<(ostream &os, Complex &c2);
Private:
Int real,imaginary;
};
Istream& operator >>(istream &is, Complex &c2);
{
Cout<<"enter real and imaginary"<<endl;
Is>>c2.real>>c2.imaginary;
Return(is);
}
Istream& operator <<(ostream &os, Complex &c2)

{
Os<<"the complex number is "<<endl;
Os<<c2.real<<"I"<<c2.imaginary;
Return(os);
}
Void main()
{
Complex c1,c2;
Cin>>c1;
Cout<<c1;
Cin>>c2;
Cout<<c2;
}
```

This operator function have to be declared friends since they have to access the user class and the objects of stream and ostream classes that are sytem defined. Since these operators functions are friend functions, the two objects cin and cout are passed as arguments, along with the objects of the user class. They return the isteam and ostream objects so that the operator can be chained. That is the above two input statements can also be written as,

Cin>>c1>>c2;

Cout<<c1<<c2;

☞ Check Your Progress 2

1) What is concept of operator overloading?

……

……

……

2) What are the basic rules for operator overloading in C++?

……

……

……

3) What are the limitations of Operator overloading and Functional overloading?

……

……

……

4) Multiple choice questions:

i) Which of the following statements is NOT valid about operator overloading?

a) Only existing operators can be overloaded.

b) Overloaded operator must have at least one operand of its class type.

c) The overloaded operators follow the syntax rules of the original operator.

d) none of the above.

ii) The new operator

a) returns a pointer to the variable

b) creates a variable called new

c) obtains memory for a new variable

d) tells how much memory is available

iii) Which of the following operator can be overloaded through friend function?

a) ->

b) =

c) ( )

d) *

iv) Overloading a postfix increment operator by means of a member function takes

a) no argument

b) one argument

c) two arguments

d) three arguments

2.7 SUMMARY

In this unit, we have discussed two important concepts about overloading namely function overloading and operator overloading. As a part of function overloading, you learnt that you can create multiple functions of the same name that work differently depending on parameter type. Function overloading can lower a programs complexity significantly while introducing very little additional risk. Although this particular lesson is long and may seem somewhat complex (particularly the matching rules), in reality function overloading typically works transparently and without any issues. The compiler will flag all ambiguous cases, and they can generally be easily resolved. Operator overloading is common-place among many efficient C++ programmers. Operating overloading allows you to pass different variable types to the same function and produce different results. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type. Operator overloading allows the programmer to define how operators should interact with various data types. It is so because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading. Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables.

2.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

2) a) A match is found. The call is resolved to a particular overloaded function.
b) No match is found. The arguments can not be matched to any overloaded function.
c) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

i) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
void Print(char *szValue);

void Print(int nValue);
```

Print(10); // exact match with Print(int)

Although 10 could technically match Print(char*), it exactly matches Print(int). Thus Print(int) is the best match available.

ii) If no exact match is found, C++ tries to find a match through promotion. In the lesson on type conversion and casting, we covered how certain types can be automatically promoted via internal type conversion to other types.

3) a) The use of function overloading is to increase consistency and readability.

b) Overloading provides multiple behaviour to same object with respect to attributes of object.

c) By using function overloading, we can call specific behaviour of that object attributes we set at compiled time. Further, we don't need to write different function name for different action.

d) can develop more than one function with the same name.

e) function overloading exhibits the behavior of polymorphism which helps to get different behaviour, although there will be some link using same name of function. Another powerful use is constructor overloading, which helps to create objects differently and it also helps a lot in inheritance.

4) Multiple Choice Questions

i) (A)

ii) (B)

iii) (C)

iv) (D)

Check Your Progress 2

1) Operator overloading allows existing C++ operators to be redefined so that they work on objects of user-defined classes. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs). When an operator is overloaded, it takes on an additional meaning relative to a certain class. But it can still retain all of its old meanings.

Examples:
a) The operators >> and << may be used for I/O operations because in the header, they are overloaded.

b) In a stack class, it is possible to overload the + operator so that it appends the contents of one stack to the contents of another. But the + operator still retains its original meaning relative to other types of data.

2) The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators, which are covered separately.

You cannot define new operators, such as **.

You cannot redefine the meaning of operators when applied to built-in data types.

Overloaded operators must either be a non-static class member function or a global function.

Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types.

Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.

Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.

If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.

Overloaded operators cannot have default arguments.

All overloaded operators except assignment (operator=) are inherited by derived classes.

The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

3) Function overloading is like you have different functions with the same name but different signatures working differently. So, the compiler can differentiate and find out which function to call depending on the context. In case of operator overloading, you try to create your own functions which are called when the corresponding operator is invoked for the operands.

One important thing to understand is that you can create as many functions as you want with the same name and different signatures so that they can work differently but for a particular class, you cannot overload the operator function based on number of arguments. There is a fundamental reason behind this.

According to the rules, you can not create your own operators and you have to use already available operators. Another thing is, since the operators are already defined for use with built-in types, you can't change their characteristics. For example, the binary operator '+' always takes two parameters, so for this you cannot create a function that takes three parameters. But you can always overload them based on the type of the parameters.

4) Multiple Choice Questions

i) (D)

ii) (C)

iii) (D)

iv) (A)

2.9 FURTHER READINGS

1) *Object-Oriented Programming with C++*, E. Balagurusamy, 2nd edition, TMH, New Delhi, 2001

2) *The C++ Programming Language*, Bjarne Stroustroup, 3rd edition, Addison Wesley, 1997

3) *C++: The Complete Reference*, H. Shildt, 4th edition, TMH, New Delhi, 2004

4) *Mastering C++*, K. R. Venugopal, Rajkumar and T. Ravishankar, TMH, New Delhi, 2004

5) www.learncpp.com/cpp-tutorial

6) http://www.bogotobogo.com/cplusplus

UNIT 3 POLYMORPHISM AND VIRTUAL FUNCTION

Structure **Page Nos.**

3.0 INTRODUCTION

Polymorphism is one of the important features of object-oriented programming. It simply means 'one name multiple forms'. We have already seen the polymorphism concept used in function overloading and operator overloading in unit -2. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation. Polymorphism is the ability to use an operator or function in different ways. The word poly means many, signifies the many uses of these operators and function. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. C++ supports polymorphism both at run-time and at compile-time. Function overloading & operator overloading belongs to compile time polymorphism where as run-time polymorphism can be achieved by the use of both derived classes and virtual functions. In this unit, you will learn about the concept of run time (dynamic) binding, virtual function and pure virtual function in detail.

3.1 OBJECTIVES

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

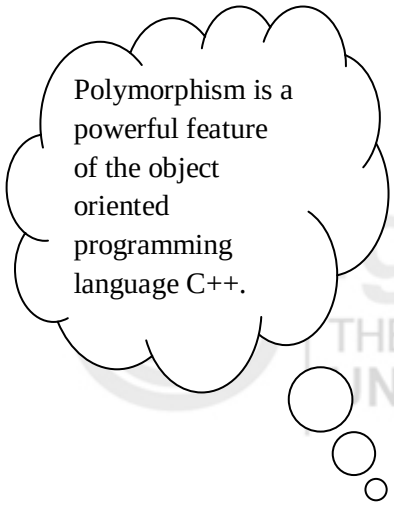
- explain the concept of polymorphism;
- explain the Concepts of dynamic binding;

- learn the concept and application of virtual function;
- use pointers to object;
- explain how to use pointers to derived class;
- understand the concept of pure virtual function, and
- describe the Concepts of Abstract Classes.

3.2 POLYMORPHISM

Polymorphism is the ability to use an operator or method in different ways. Polymorphism gives different meanings or functions to the operators or methods. Poly refers many that signify the many uses of these operators and methods. A single method usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. "Polymorphism is a mechanism that allows you to implement a function in different ways."

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ.



Example of the concept of polymorphism:

- 6 + 10 //The above refers to integer addition.
- The same + operator can be used with different meanings with strings:
- "Technical" + "Training"
- The same + operator can be also used for floating point addition: 7.15 + 3.78

We saw above that a single operator '+' behaves differently in different contexts such as integer, string or float referring the concept of polymorphism. The above concept leads to operator overloading. When the exiting operator or function operates on new data type it is overloaded. C++ also permits the use of different functions with the same name. Such functions have different argument list. The difference can be in terms of number or type of arguments or both. It refers as function overloading. So, we conclude that the concept of operator overloading and function overloading is a branch of polymorphism. Both the concepts have been discussed in unit 2 in detail.

3.2.1 Advantages of Polymorphism

- The biggest advantage of polymorphism is creation of reusable code by programmer's classes once written, tested and implemented can be easily reused without caring about what's written in the case.
- Polymorphic variables help with memory use, in that a single variable can be used to store multiple data types (integers, strings, etc.) rather than declaring a different variable for each data format to be used.

- Applications are Easily Extendable: Once an application is written using the concept of polymorphism, it can easily be extended, providing new objects that conform to the original interface. It is unnecessary to recompile original programs by adding new types. Only re-linking is necessary to exhibit the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, there has always been a need for adding and customizing. By utilizing the concept of polymorphism, time and work effort is reduced in addition to making future maintenance easier.

- It provides easier maintenance of applications.

- It helps in achieving robustness in applications.

3.2.2 Types of Polymorphism

C++ provides three different types of polymorphism:

- Function overloading (for definition and example, see block 2, unit 2)
- Operator overloading (for definition and example, see block 2, unit 2)
- Virtual functions (Defined in section 3.4 of this unit)

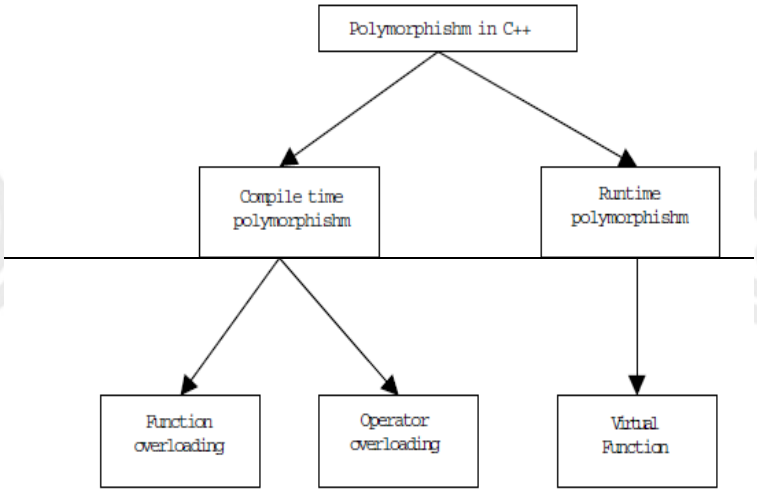


Figure 3.1: Achieving polymorphism

3.3 DYNAMIC BINDING

You know that polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding*, *static binding*, *static linking* or *compile time polymorphism*.

However, ambiguity creeps in when the base class and the derived class both have a function with same name. For instance, let us consider the following code snippet.

```
Class P
{
int a;
public:
void display() {……} //display in base class
};

Class Q : public P
{
int b;
public:
void display() {……} //display in derived class
};
```

It is also known as *dynamic binding* because the selection of the appropriate function is dynamically at run time.

Since, both the display() functions are same but at in different classes, there is no overloading, and hence early binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the function with the derived class objects.

It would be better if the appropriate function is chosen at the run time. This is known as run time polymorphism. C++ supports *run-time polymorphism* by a mechanism called *virtual function.* It exhibits *late binding.*

As stated earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. Therefore, an essential feature of polymorphism is the ability to refer to objects without any regard to their classes. It implies that a single pointer variable may refer to object of different classes. So, dynamic binding requires pointers to object for its implementation.

In the case of a compiled language, the compiler still doesn't know the actual object type, but it inserts code that finds out and calls the correct function body. When a language implements dynamic binding, there must be some mechanism to determine the type of the object at runtime and call the appropriate member function.

3.4 VIRTUAL FUNCTIONS

You know that polymorphism also refers to the ability to call different functions by using only one type of function call. Suppose a programmer wants to code vehicles of different shapes such as circles, squares, rectangles, etc. one way to define each of these classes is to have a member function for each that makes vehicles of each shape. Another convenient approach the programmer can take is to define a base class named Shape and

then create an instance of that class. The programmer can have array that hold pointers to all different objects of the vehicle followed by a simple loop structure to make the vehicle, as per the shape desired, by inserting pointers into the defined array. This approach leads to different functions executed by the same function call. Polymorphism is used to give different meanings to the same concept. This is the basis for virtual function implementation.

In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply:

1. All different classes must be derived from a single base class. In the above example, the shapes of vehicles (circle, triangle, rectangle) are from the single base class called Shape.

2. The member function must be declared virtual in the base class. In the above example, the member function for making the vehicle should be made as virtual to the base class.

Conclusion of above discussion is that the form of a member function can be changed at runtime. Such member functions are called virtual functions and the corresponding class is called polymorphic class. The objects of the polymorphic class, addressed by pointers, change at runtime and respond differently for the same message. The word 'virtual' means something that does not exist in reality, some sort of imaginary thing. The concept of virtual function is the same as a function, but it does not really exist although it appears in needed places in a program. The object-oriented programming language C++ implements the concept of virtual function as a simple member function, like all member functions of the class. The functionality of virtual functions can be *overridden* in its derived classes.

3.4.1 Function Overriding

To cause late binding to occur for a particular function, C++ requires that you use the **virtual** keyword when declaring the function in the base class. Late binding occurs only with **virtual** functions, and only when you're using an address of the base class where those **virtual** functions exist, although they may also be defined in an earlier base class. To create a member function as **virtual**, you simply precede the declaration of the function with the keyword **virtual**. Only the declaration needs the **virtual** keyword, not the definition. If a function is declared as **virtual** in the base class, it is **virtual** in all the derived classes. The redefinition of a **virtual** function in a derived class is usually called *function overriding*. Notice that you are only required to declare a function **virtual** in the base class. All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism. You *can* use the **virtual** keyword in the derived-class declarations (it does no harm to do so), but it is redundant and can be confusing.

3.4.2 Properties of Virtual Functions

Dynamic Binding Property: Virtual Functions are resolved during run-time or dynamic binding. Virtual functions are also simple member functions. The main difference between a non-virtual C++ member function and a virtual member function is in the way

they are both resolved. A non-virtual C++ member function is resolved during compile time or static binding. Virtual Functions are resolved during run-time or dynamic binding.

- Virtual functions are member functions of a class.
- Virtual functions are declared with the keyword virtual.
- Virtual function takes a different functionality in the derived class.

3.4.3 Definition of Virtual Functions

C++ provides a solution to invoke the exact version of the member function, which has to be decided at runtime using virtual functions. They are the means by which functions of the base class can be overridden by the functions of the derived class. The keyword virtual provides a mechanism for defining virtual functions. When declaring the base class member function, the keyword virtual is used with those functions, which are to be bound dynamically.

The general syntax to declare a virtual function uses the following format:

```
class class_name //This denotes the base class of C++ virtual function
{
public:
virtual return_type member_function_name(arguments) //This denotes the C++ virtual function
{
...
...
}
};
```

Virtual functions should be define in the public section of a class to realize its full potential benefits. When such a declaration is made, it allows to decide which function to be used at runtime, based on the type of object, pointed to by the base pointer rather than the type of the pointer. The examples of virtual functions provided in this unit illustrate the use of base pointer to point to different objects for executing different implementations of the virtual functions.

Note: By default, C++ matches a function call with the correct function definition at compile time. This is called *static binding*. You can specify that the compiler match a function call with the correct function definition at run time; this is called *dynamic binding*. You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

3.4.4 Need of Virtual Function

The first and foremost question which is arises why do we need virtual function? Suppose we do have a list of pointer to objects of a super class in an inheritance hierarchy and we wish to invoke the functions of its derived classes with the help of single list of pointers provided that the functions in super class and sub classes have the same name and signature. That in turn means we want to achieve run time polymorphism. So, let we have brief concept about pointers to derived types.

3.4.4.1 Pointers to derived types

We know that pointer of one type may not point to an object of another type. You shall now learn about the one exception to this general rule: a pointer to an object of a base class can also point to any object derived from that base class. Similarly, a reference to a base class can also reference any object derived from the original base class. In other words, a base class reference parameter can receive an object of types derived from the base class, as well as objects within the base class itself. Let us try to work it out with the following example:

Example 1. //Program without virtual function

```
/*The case when we wish to invoke the child class function with the parent class pointer, but the output is not the same as we expected*/

#include<iostream.h>
#include<conio.h>
class Faculty
{
        int facultyId;
        char facultyName[25];
        char facultyType;
        public:
                void calculateSalary()  //Parent Class Function
                {
                cout<<"\n Calculating the salary of a faculty, no matter the faculty is regular or guest !!";
                }
};

class RegularFaculty : public Faculty
{
        public:
                void calculateSalary()  //Child Class Function
                {
                cout<<"\n Calculating the salary of a regular faculty !!";
                }
};

class GuestFaculty : public Faculty
{
        public:
                void calculateSalary()   //Child Class Function
                {
                cout<<"\n Calculating the salary of a guest faculty !!";
                }
};
```

```
void main()
{
        Faculty *pFaculty;
        RegularFaculty rFaculty;
        GuestFaculty gFaculty;
        clrscr();
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
        //Assigning the address of child class object into parent class pointer
        pFaculty=&gFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
        getch();
}
```

Output:

Calculating the salary of a faculty, no matter the faculty is regular or guest !!

Calculating the salary of a faculty, no matter the faculty is regular or guest !!

Note: But here the output does not come out to be as we expect it to be because here the super class 'Faculty' pointer is having reference of child class objects of RegularFaculty and GuestFaculty classes respectively but when we try to call the derived class function namely the calculateSalary() with the help of this pointer it does not do so. Both the time it is calling the function calculateSalary() of super class only!!

Now look at the following program:

Example 2. //Program with virtual function

```
/* The case when we wish to invoke the child class function
with the parent class pointer and the output comes as expected*/
#include<iostream.h>
#include<conio.h>
class Faculty
{
        int facultyId;
        char facultyName[25];
        char facultyType;
        public:
                virtual void calculateSalary()  //Parent Class Function
                {
                cout<<"\n Calculating the salary of a faculty, no matter the faculty
is regular or guest !!";
                }
};
```

```
class RegularFaculty : public Faculty
{
        public:
                void calculateSalary() //Child Class Function
                {
                cout<<"\n Calculating the salary of a regular faculty !!";
                }
};
class GuestFaculty : public Faculty
{
        public:
                void calculateSalary() //Child Class Function
                {
                cout<<"\n Calculating the salary of a guest faculty !!";
                }
};

void main()
{
        Faculty *pFaculty;
        RegularFaculty rFaculty;
        GuestFaculty gFaculty;
        clrscr();
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
//Assigning the address of child class object into parent class pointer
        pFaculty=&gFaculty;
//Invocation of calculateSalary() function with parent class pointer
        pFaculty->calculateSalary();
        getch();
}
```

Output:

Calculating the salary of a regular faculty !!

Calculating the salary of a guest faculty !!

Explanation

See the magic of virtual function. There is a slight change in the above program the function calculateSalary() in super class is declared to be virtual and the output is turned to be as per our expectations, we are having the pointer 'pFaculty' to super class but when the references of child class objects namely the rFaculty and gFaculty to this pointer we see that with the pointer having the reference of rFaculty the function of child class regularFaculty is called where as when it contains the referenc of gFaculty then the function of child class GuestFaculty is called.

Let we have one more example to clarify the concept:

The vital reason for having a virtual function is to implement a different functionality in the derived class.

For example: a Make function in a class Vehicle may have to make a Vehicle with red color. A class called FourWheeler, derived or inherited from Vehicle, may have to use a blue background and 4 tires as wheels. For this scenario, the Make function for FourWheeler should now have a different functionality from the one at the class called Vehicle. This concept is called Virtual Function.

Example 3.

```
class Vehicle //This denotes the base class of C++ virtual function
{
public:
virtual void Make() //This denotes the C++ virtual function
{
cout << "Member function of Base Class Vehicle Accessed" << endl;
}
};
```

After the virtual function is declared, the derived class is defined. In this derived class, the new definition of the virtual function takes place. When the class FourWheeler is derived or inherited from Vehicle and defined by the virtual function in the class FourWheeler, it is written as:

```
#include <iostream>
using namespace std;
class Vehicle   //This denotes the base class of C++ virtual function
{
public:
      virtual void Make()   //This denotes the C++ virtual function
      {
            cout << "Member function of Base Class Vehicle Accessed" << endl;
      }
};
class FourWheeler : public Vehicle
{
public:
      void Make()
      {
            cout << "Virtual Member function of Derived class FourWheeler Accessed"
<< endl;
      }
};
void main()
{
      Vehicle *a, *b;
      a = new Vehicle();
      a->Make();
      b = new FourWheeler();
      b->Make();
}
```

Explanation

In the above example, it is evident that after declaring the member functions Make() as virtual inside the base class Vehicle, class FourWheeler is derived from the base class Vehicle. In this derived class, the new implementation for virtual function Make() is placed.

In this example, the member function is declared virtual and the address is bounded only during run time, making it dynamic binding and thus the derived class member function is called.

To achieve the concept of dynamic binding in C++, the compiler creates a v-table each time a virtual function is declared. This v-table contains classes and pointers to the functions from each of the objects of the derived class. This is used by the compiler whenever a virtual function is needed.

3.4.5 Rules for Virtual Functions

- The virtual functions must be the members of some class.
- A class member function can be declared to be virtual by just specifying the keyword 'virtual' in front of the function declaration. The syntax of declaring a virtual function is as follows:

  virtual <return type> <function name><(argument list)>

  {//Function Body}
- Virtual Functions enables derived (sub) class to provide its own implementation for the function already defined in its base (super) class.
- Virtual Functions give power to the derived class functions to override the function in its base class with the same name and signature.
- Virtual Functions can't be static members.
- Only the functions that are members of some class can be declared as virtual that means we can't declare regular functions or friend functions as virtual.
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- If one will call the virtual function with the pointer having the reference to the base class object then the function of the base class will be called for sure.
- The corresponding functions in the derived class must agree with the virtual function's name and signature that means both must have same name and signature.

3.4.6 Limitations of Virtual Functions

- The function call takes slightly longer due to the virtual mechanism, and it also makes it more difficult for the compiler to optimize because it doesn't know exactly which function is going to be called at compile time.
- In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.
- Virtual functions will usually not be inlined.
- Size of object increases due to virtual pointer.

3.5 PURE VIRTUAL FUNCTIONS

Remember that a class containing pure virtual functions can't be used to declare any objects of its own.

Pure Virtual Functions are the specific type of virtual functions. A virtual function with no function body is called pure virtual function i.e. a pure virtual function is a function declared in a base class that has no definition relative to base class. A pure virtual function purely exists in base class only to be overridden by the derived class functions. A base class only specifies that there is a function with this name and signature which will be implemented by any of derived class or by some other derived class in the same inheritance hierarchy. Such classes are also called *abstract base class*.

3.5.1 Syntax of pure virtual function

The syntax of declaring a pure virtual function is as follows:
virtual <return type> <function name><(argument list)> =0;

//Program with pure virtual function

```
#include<iostream.h>
#include<conio.h>

class Faculty
{
        int facultyId;
        char facultyName[25];
        char facultyType;
        public:
virtual void calculateSalary()=0;  //Pure Virtual Function in parent class
};
class RegularFaculty:public Faculty
{
        public:
                void calculateSalary() //Child Class Function
        {
                        cout<<"\n Calculating the salary of a regular faculty !!";
                }
};
class GuestFaculty:public Faculty
{
        public:
                void calculateSalary() //Child Class Function
                {
                        cout<<"\n Calculating the salary of a guest faculty !!";
                }
};
void main()
{
        Faculty *pFaculty;
        RegularFaculty rFaculty;
        GuestFaculty gFaculty;
        clrscr();
```

```
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
        pFaculty->calculateSalary(); //Invocation of calculateSalary() function with
        parent class  pointer
        //Assigning the address of child class object into parent class pointer

        pFaculty=&gFaculty;
        pFaculty->calculateSalary(); //Invocation of calculateSalary() function with
        parent class pointer
        getch();
}
```

Output:

```
Calculating the salary of a regular faculty !!
Calculating the salary of a guest faculty !!
```

This is the same faculty salary calculation program that is introduced in the discussion of virtual functions but there is a change here, the function calculateSalary() in base class 'Faculty' is declared to be pure virtual with no function definition. This function is implemented by two concrete children of Faculty class namely the 'ReguarFaculty' and 'GuestFaculty'.

Then in main, we have pointer to base class Faculty namely the 'pFaculty' that is assigned by the references of the child class objects 'rFaculty' and 'gFaculty' one by one and is used to call the calculateSalary() function and it is clear from the output that one time it calls up the function in RegularFaculty and the function in GuestFaculty the other time.

3.5.2 Characteristics of Pure Virtual Functions

- A class member function can be declared to be pure virtual by just specifying the keyword 'virtual' in front and putting '=0' at the end of the function declaration.
- Pure virtual function itself do nothing but acts as a prototype in the base class and gives the responsibility to a derived class to define this function.
- As pure virtual functions are not defined in the base class thus a base class can not have its direct instances or objects that means a class with pure virtual function acts as an abstract class that cannot be instantiated but its concrete derived classes can be.
- We cannot have objects of the class having pure virtual function but we can have pointers to it that can in turn hold the reference of its concrete derived classes.
- Pure virtual functions also implements run time polymorphism as the normal virtual functions do as binding of functions to the appropriate objects here is also delayed up to the run time, that means which function is to invoke is decided at the run time.
- Pure virtual functions are meant to be overridden.
- Only the functions that are members of some class can be declared as pure virtual that means we cannot declare regular functions or friend functions as pure virtual.

- The corresponding functions in the derived class must agree be compatible with the pure virtual function's name and signature that means both must have same name and signature.
- For abstract class, pure virtual function is must.
- The pure virtual functions in an abstract base class are never implemented. Because no objects of that type are ever created, there is no reason to provide implementations, and the ADT (Abstract Data Type) works purely as the definition of an interface to objects which derive from it.
- It is possible, however, to provide an implementation to a pure virtual function. The function can then be called by objects derived from the ADT, perhaps to provide common functionality to all the overridden functions.

3.5.3 Abstract Classes

Abstract classes act as a container of general concepts from which more specific classes can be inherited. Thus an abstract class is one that is not used to create any object of its own but it solely exists to act as a base class for the other classes that means the abstract class must be a part of some inheritance hierarchy.

An abstract class can further be illuminated through following points:

- An abstract class can not be instantiated that means abstract classes can not have their own instances but their child or derived classes may have their own instances provided the child class itself is not an abstract class.
- Though objects of an abstract class cannot be created, however, one can use pointers and references to abstract class types.
- A class should contain at least one pure virtual function to be called as abstract. Pure virtual functions can be declared with the keyword virtual and =0 syntax at the end of function declaration statement.
- If a class is made abstract by giving a pure virtual function then it must be inherited by a child class of it that provides the implementation of the pure virtual function.
- If a class inherits an abstract class and does not provide the implementation of the pure virtual function then the child class itself should declare the function as pure virtual that means the child class will be an abstract class as well.
- The signature of the function declared as pure virtual in base class must strictly agree with the signature of the function in child class that implements the pure virtual function.

Example

```
//Program with abstract class having pure virtual functions
/* Complete Faculty Salary Calculation program in action*/
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Faculty  //Abstract class having pure virtual function
{
        protected:
                int facultyId;
                char facultyName[25];
char facultyType;
        public:
                //Pure Virtual Functions in parent class
                virtual float calculateSalary()=0;
                virtual void showDetails()=0;
};
class RegularFaculty:public Faculty
{
        float basic,da,hra,tax;
        public:
        RegularFaculty(int id,char name[])
                {
                        facultyId=id;
                        strcpy(facultyName,name);
                        facultyType='R';
                }
        setSalaryParameters(float b,float d,float h,float t)
                {
                        basic=b;
                        da=d;
                        hra=h;
                        tax=t;
                }

        float calculateSalary() //Child Class Function
{
                        return((basic+da+hra)-tax);
                }

        void showDetails()
                {
                        cout<<"\n Id:"<<facultyId;
                        cout<<"\n Name:"<<facultyName;
                        cout<<"\n FacultyType: Regular";
                }
};
class GuestFaculty:public Faculty
{
```

```
        int noOfLectures;
        float perLectureRemuneration;
        public:
        GuestFaculty(int id,char name[])
                {
                        facultyId=id;
                        strcpy(facultyName,name);
                        facultyType='G';
                }
        setSalaryParameters(int nol,float plr)
                {
                        noOfLectures=nol;
                        perLectureRemuneration=plr;
                }
        float calculateSalary() //Child Class Function
                {
                        return(noOfLectures*perLectureRemuneration);
                }
        void showDetails()
                {
                cout<<"\n Id:"<<facultyId;
                cout<<"\n Name:"<<facultyName;
                cout<<"\n FacultyType: Guest";
                }
};
void main()
{
        float sal;
        Faculty *pFaculty;
        RegularFaculty rFaculty(1,"Ram");
        GuestFaculty gFaculty(2,"Shyam");
        clrscr();
        rFaculty.setSalaryParameters(1500,550.65,250.5,120);
        //Assigning the address of child class object into parent class pointer
        pFaculty=&rFaculty;
        //Invocation of calculateSalary() function with parent class pointer
        sal=pFaculty->calculateSalary();
        //Invocation of showDetails() function with parent class pointer
        pFaculty->showDetails();
        cout<<"\n Salary:"<<sal<<endl;
        gFaculty.setSalaryParameters(20,150.50);
//Assigning the address of child class object into parent class pointer
        pFaculty=&gFaculty;
        //Invocation of calculateSalary() function with parent class pointer
        sal=pFaculty->calculateSalary();
        //Invocation of showDetails() function with parent class pointer
        pFaculty->showDetails();
        cout<<"\n Salary:"<<sal;
        getch();
}
```

Output:

Id:1

Name:Ram

FacultyType: Regular

Salary:2181.149902

Id:2

Name:Shyam

FacultyType: Guest

Salary:3010

Explanation

Here in this example the class 'Faculty' is an abstract class as it is having two pure virtual functions named calculateSalary and showDetails. The implementation of these pure virtual functions is provided by the concrete subclasses of the class Faculty namely the RegularFaculty and GuestFaculty.

In the main function we do have objects of the two concrete subclasses and we assign the address of these objects in the pointer variable of super class type i.e. Faculty (as we cannot have direct objects of base class but we can have pointer to the abstract class type that can contain the references to the objects of its concrete child class objects) and when the functions are invoked by this pointer variable, the invocation or calling of function is bound with the exact function definition with the help of the type of reference that the pointer variable is containing. When it contains the reference of object of RegularFaculty class then the calcuateSalary and showDetails of RegularFaculty class are invoked and when it contains the reference of object of GuestFaculty class then the calcuateSalary and showDetails of GuestFaculty class are invoked and this binding is delayed upto the run time that's why it is termed as late binding or dynamic binding.

☞ Check Your Progress 1

1) Describe the concept of polymorphism

……

……

……

2) What is dynamic binding?

……

……

……

3) Explain the pointers to object with the help of an example.

……

……

……

4) How Virtual functions call up is maintained?

……

……

……

5) Explain briefly the importance of pure virtual function in the software development paradigm.

……

……

……

6) Multiple choice questions:

i) RunTime Polymorphism is achieved by _____

a) friend function

b) virtual function

c) operator overloading

d) function overloading

ii) Pure virtual functions

a) have to be redefined in the inherited class.

b) cannot have public access specification.

c) are mandatory for a virtual class.

d) None of the above

iii) Use of virtual functions implies

a) overloading.

b) overriding.

c) static binding.

d) dynamic binding.

iv) A *virtual* class is the same as

a) an abstract class

b) a class with a virtual function

c) a base class

d) none of the above.

v) A pointer to the base class can hold address of

a) only base class object

b) only derived class object

c) base class object as well as derived class object

d) None of the above

vi) A pure virtual function is a virtual function that

a) has no body

b) returns nothing

c) is used in base class

d) both (A) and (C)

3.6 SUMMARY

Polymorphism simply defines the concept of one name having more than multiple forms. It has two types of time compile time and run time. In compile time, an object is bound to its function call at compile time. In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports the run time polymorphism with the help of virtual function by using the concept of dynamic binding. Dynamic binding requires use of pointers to objects. Pointers to objects of a base class type are compatible with pointers to objects of a derived class. Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. Such virtual functions (equated to zero) are called pure virtual functions. A class containing such pure function is called an abstract class.

3.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Polymorphism in biology means the ability of an organism to assume a variety of forms.

Polymorphism is the ability to use an operator or function in different ways. Poly implies multiple that signify the many uses of these operators and methods. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. Polymorphism is two type, compile time polymorphism (operator & function overloading) and run-time polymorphism (virtual function)

2) Dynamic binding is a mechanism that is use to implement run-time polymorphism. Dynamic binding change the form of function at run time. In this binding, the objects of the class, addressed by pointers, change at run-time and respond differently for the same message. Such mechanism requires postponement of binding of a function call to the member function until run-time.

3) By pointers you can access the class members. Pointer can also point to object created by a class. Consider the following statement:

element el;

Where element is a class and el is an object of that class. IN similar way we can define a pointer *el_pointer* of type *element* as follows:

Element *el_pointer;

Pointers to objects are useful in crteating objects at run time. Let we explain it more broadly with the help of an example.

Example: //pointers to objects

```
#include <iostream.h>

Class element
{

int id;
float price;

public:

void input(int p, int q)
{
id=p;
price=q;
}

Void display(void)
{

Cout<< "ID :" << id<< "\n";
 Cout<< "PRICE :" << price<< "\n";

}
};

 Const int limit = 3;
```

```
main()

{
        int *a = new element[limit];
        int *b = a;
        int m, i;
        float n;
        for(i=0; i<limit; i++)
        {
        cout<< "Input ID and price of element" <<i+1;
                cin >> m  >> n;
                a->input(m, n);
                a++;
        }
        for(i=0; i<limit; i++)
        {
        cout<< "ELEMENT"  << i+1<<  "\n";
                b-> display();
                        b++;
                }
                }
```

Output:

Input ID and price of element 1 40 342.25

Input ID and price of element 2 11 250.50

Input ID and price of element 3 101 500

ELEMENT 1

ID : 40

PRICE : 342.25

ELEMENT 2

ID : 11

PRICE : 250.50

ELEMENT 3

ID : 101

PRICE : 500

Explanation

In above program we created space **dynamically** for three objects of equal size. The statement

int *a = new element[limit]

allocates enough memory for an array of 3 objects of **element** in the object structure and assign the address of the memory space to pointer **a.** The object pointer also use an object pointerto access the public members of an object.

4) Through Look up tables added by the compiler to every class image. This also leads to performance penalty.

5) Its normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serve as a placeholder. Such functions are called 'do-nothing' functions.

A 'do-nothing' function may be defined as follows:

virtual void show ( ) = 0;

Such functions are called pure virtual functions.

6) Multiple Choice Questions

I-(B)

II-(A)

III-(D)

IV-(D)

V-(C)

VI-(D)

3.8 FURTHER READINGS

1) *The C++ Programming Language*, Bjarne Stroustroup, 3rd edition, Addison Wesley, 1997

2) *C++: The Complete Reference*, H. Shildt, 4th edition, TMH, New Delhi, 2004

3) *Mastering C++*, K. R. Venugopal, Rajkumar and T. Ravishankar, TMH, New Delhi, 2004

4) *Object-Oriented Programming with C++*, E. Balagurusamy, 2nd edition, TMH, New Delhi, 2001

5) www.learncpp.com/cpp-tutorial

6) http://www.bogotobogo.com/cplusplus

7) http://www.gamespp.com/c/introductionToCppMetrowerksLesson11.html