UNIT 3 POLYMORPHISM AND VIRTUAL FUNCTION

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.

Polymorphism is a powerful feature of the object oriented programming language C++.

- 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)
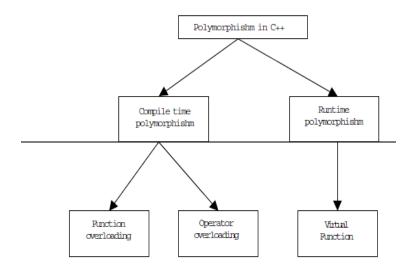
Figure 3.1: Achieving polymorphism

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.

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

3.5   PURE VIRTUAL FUNCTIONS

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
{
```

```cpp
        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