Notes on Unit- V
Polymorphism, Virtual Function & Pure Virtual Function

Unit No. 5

Page No.: 1
Date:

Polymorphism

Introduction :-

Polymorphism is one of the crucial feature of object oriented programing language. It simply means one name with multiple forms. Basically the concept of polymorphism is implimented by using the overloaded functions and operators.

The overloaded member functions are selected for invoking by matching the arguments both datatype & number. This information is known to the compiler at the compile time and their fore compiler is able to select the appropeate function for a perticular call at the compile time itself. This concept is called as early binding static binding or static linking. The same concept is also known as compile time polymorphism. early binding simply means that an object is bound to its function call at

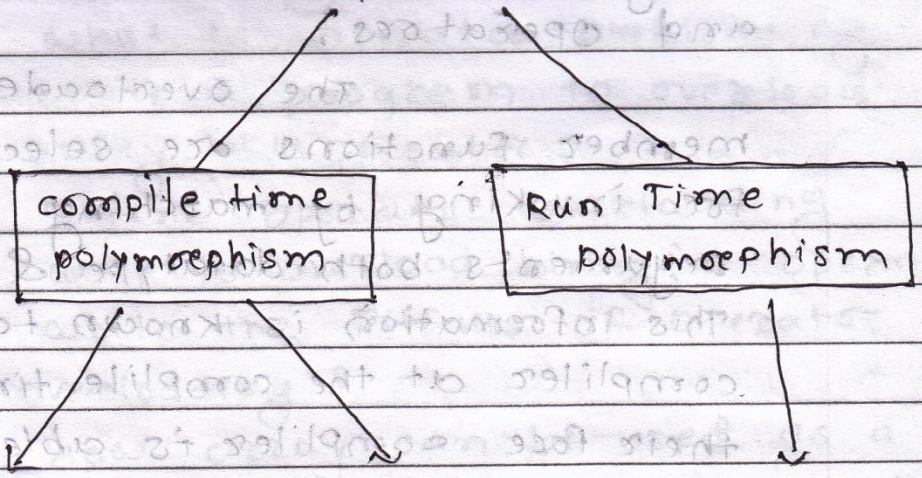
(*) what is polymorephism ? Explain its different types ?

compile time.

(*) Types of polymorephism (*)

In object oriented programing c++ polymorephism is basically divided into tao main types. They are,

(1) compile time polymorephism.
(2) Run time polymorephism.



(*) compile time polymorephism.

The information is known to the compiler at the compile time of the program and the compiler is able to select the appropeate function for a perticular call at the compile

time it self. This concept is known as compile time polymorphism.

The concept of compile time polymorphism is implimented basically by using the concepts of function overloading and operator overloading. simply we can say that tao achieve compile time polymorphism it basically uses the concepts of function as well as operator overloading.

(*) Run time polymorphism :-

c++ steongly suppoet the machinism known as virtual function. To achieve the concept of run time polymorphism At Run time the class object are under consider-ation, when the appropiate function is envoke, since the function is linked with a peeticulae class. much lattee after the compilation, this process is know as late binding. The same concept is also known as dynamic binding because the selection of the appropate function

is done dynamically at run time dynamic binding is one of the powerfull feature of c++ which uses the concept of pointers to object.

Pointers to derived classes:-

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes.

e.g. if B is a base class and D is a derived class from B, then a pointer declared as a pointer to B can also be a pointer to D.

Consider the following declarations:

```
B *cptr; // Pointer to class B type variable
B b.     // base object
D d;     // derived object
cptr = &b; // cptr points to object b
```

We can make cptr to point to the object d as follows:

```
cptr = &d; // cptr points to object d
```

This is perfectly valid with C++ because d is an object derived from the class B.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class.

Imp (SM) ★

Virtual Functions :-

Defn When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration.

When a function is made virtual C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

Following program clears the idea about virtual function.

One important point to remember is that, we must access virtual functions through the use of a pointer declared as a pointer to base class.

Run time Polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

```
// Pgm for Virtual Function
#include<iostream.h>
#include<conio.h>
class Base
{
 public:
 virtual void display()
 {
  cout << "\n Base class is invoked";
 }
};
class Derived: public Base
{
 public: void display()
 {
  cout << "\n Derived Class is invoked"<<endl;
 }
};
void main()
{
        clrscr();
        cout<<"\n\t\t\t ***** OUTPUT *****";
        Base *a;    //pointer of base class
        Derived b;     //object of derived class
         a = &b;
        a->display();
        getch();
}
```

******* OUTPUT *******

Derived Class is invoked

Rules for Virtual Functions:

Following are the rules for virtual functions.

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of derived object, the reverse is not true. That is, we cannot use a pointer to a derived class to access an object of the base type.

Imp * (5M) **Pure Virtual Functions :-**

Defn A pure virtual function is a function declared in a base class that has no definition relative to the base class.
In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Pure virtual function is also called as 'do-nothing' function.
Thus pure virtual function (i.e. do-nothing function) may be defined as follows:

```
virtual void display() = 0;
```

Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.
Following program clears the idea about pure virtual function.

—x—

```
// Pgm for Pure Virtual Function
#include<iostream.h>
#include<conio.h>
class Base
{
   public:
   virtual void show( ) = 0;
};
class Derived : public Base
{
   public: void show()
   {
      cout << "\n Derived class is derived from the base class.";
   }
};
void main()
{
   clrscr();
   cout<<"\n\t\t\t ***** OUTPUT *****";
   Base *bptr;
   Derived d;
   bptr = &d;
   bptr->show();
   getch();
}
```

******* OUTPUT *******

Derived class is derived from the base class.

******** BEST OF LUCK ********