**Experiment No: 3 Date:- 15-10-2020**

**AIM: To study Virtual Functions and Polymorphism**

**THEORY:**

A virtual function is a member function which is declared within a base class and is re-defined (Overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

• Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.

• They are mainly used to achieve Runtime polymorphism.

• Functions are declared with a virtual keyword in base class.

• The resolving of function calls is done at Run-time.

Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer ‘bptr’ contains the address of object ‘d’ of derived class.

Late binding (Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding (Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is print derived class as pointer is pointing to object of derived class) and show () is non-virtual so it will be bound during compile time (output is show base class as pointer is of base type).

A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration.

**Program codes and their Corresponding Outputs**

//Program A- virtual function

#include<iostream>

using namespace std;

class adopt

{

protected: char breed[30];

public: adopt(char \*n)

{

strcpy(breed,n);

}

virtual void display()

{

}

};

class cat:public adopt

{

float price;

int age;

public:cat(char \*b,float p, int a):adopt(b)

{

price=p;

age=a;

}

void display()

{

cout<<"Category:Cat"<<endl;

cout<<"breed:"<<breed<<endl;

cout<<"price:"<<price<<endl;

cout<<"age:"<<age<<" year(s) old"<<endl;

}

};

class dog:public adopt

{

float price;

int age;

public:dog(char \*b, float p, int a):adopt(b)

{

price=p;

age=a;

}

void display()

{

cout<<"Category:Dog"<<endl;

cout<<"breed:"<<breed<<endl;

cout<<"price:"<<price<<endl;

cout<<"age:"<<age<<" year(s) old"<<endl;

}

};

int main()

{

float p;

int age;

char \*b=new char[30];

cout<<"Enter breed of the dog to be given up for adoption"<<endl;

cin.getline(b,30);

cout<<"Enter age of the dog(in years)"<<endl;

cin>>age;

cout<<"Enter price to be adopted for (in Rs.)"<<endl;

cin>>p;

dog d(b,p,age);

while((getchar()!='\n'));

cout<<endl;

cout<<"Enter breed of the cat to be given up for adoption"<<endl;

cin.getline(b,30);

cout<<"Enter age of the cat(in years)"<<endl;

cin>>age;

cout<<"Enter price to be adopted for (in Rs.)"<<endl;

cin>>p;

cat c(b,p,age);

adopt \*a[2];

a[0]=&d;

a[1]=&c;

cout<<endl;

a[0]->display(); cout<<endl<<endl;

a[1]->display();

return 0;

}



//Program B

#include<iostream>

using namespace std;

class base

{

public: virtual void display()=0;

};

void base::display()

{

cout<<"Definition of virtual function display() here"<<endl;

}

class derived:public base

{

public: void display()

{

cout<<"Defintion of derived class function display()"<<endl;

}

};

int main()

{

base \*ptr;

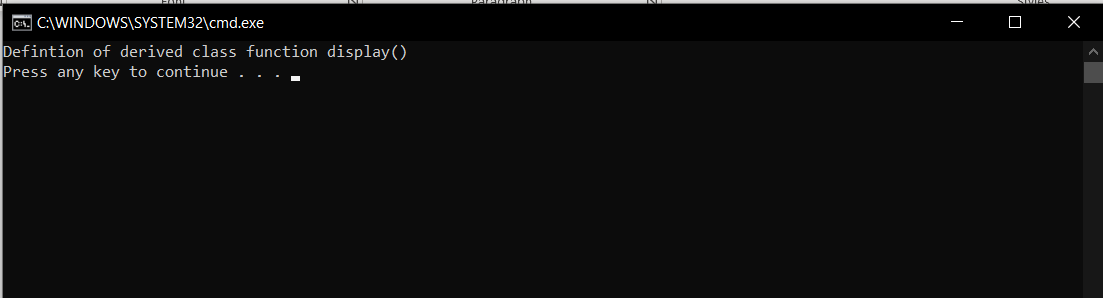
derived a;

ptr=&a;

ptr->display();

return 0;

}



Conclusion: All programs were successfully run and executed to emphasize the working of virtual and pure virtual functions. The following was also observed with respect to these types of functions.

1. **Virtual functions cannot be static and cannot be friend functions of other classes**
2. **Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.**
3. **The prototype of virtual functions should be same in base as well as derived class.**
4. **They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.**
5. **A class may have virtual destructor but it cannot have a virtual constructor.**