Himalaya College of Engineering

Advanced C++ Programming Lab Report

Lab 7: Virtual Functions in C++

Prepared By : Nawnit Paudel (HCE081BEI024)

Subject : Object-Oriented Programming (OOP)

Program : Bachelor of Electronics, Communication and Information Engineering

Institution : Himalaya College of Engineering

OBJECTIVE

- To understand the concept of Virtual Functions in C++.

BACKGROUND THEORY

Virtual Functions:

Virtual functions in C++ are a fundamental concept in Object-Oriented Programming (OOP) that enable **runtime polymorphism**. They allow you to define a function in a base class that can be overridden by derived classes and then call the correct overridden version of the function based on the actual type of the object at runtime, even if you're using a pointer or reference to the base class.

Types of Virtual Functions:

A. **Pure Virtual Functions:** These are the types of virtual functions that do not include any declaration within it rather it is used to further override by base classes. These functions when in parent class themselves do not have any operation/execution in the code.

Syntax: class

Base { public:

virtual void functionName() = 0; //Pure Virtual function declaration };

B. **Virtual Functions:** These are the general virtual functions which may or may not have any declaration and are meant to be overridden by child classes where the actual use of virtual function is implemented.

Syntax: class

Base { public:

virtual void functionName() {

// Virtual function declaration

};

Rules of Using Virtual Functions

1. Virtual functions cannot be declared as static: Since static functions belong to the class rather than an object instance, they do not support runtime polymorphism and therefore cannot be virtual.

2. The function signature (prototype) of the virtual function must be identical in both the base and derived classes. This ensures proper overriding and correct function dispatch at runtime.

3. Virtual functions are typically accessed through pointers or references to the base class. This mechanism enables runtime polymorphism, allowing the correct derived class function to be invoked.

4. Virtual functions can be declared as friend functions of other classes. However, friend functions are not members of the class and cannot themselves be virtual.

5. Overriding a virtual function in the derived class is optional. If the derived class does not override the virtual function, the base class version is called at runtime.

6. Constructors cannot be virtual. Object construction happens before the object type is fully established, so virtual dispatch is not possible. However, destructors can and should be declared virtual to ensure proper cleanup of derived class objects when deleted through base class pointers.

Example:

```
#include<iostream>

using namespace std;

class Shape { public:

   virtual void draw() = 0; // Pure virtual function

}; class Circle : public Shape {

public:

   void draw() override {        cout <<

"Drawing a Circle" << endl;

   } }; class Rectangle : public

Shape { public:

   void draw() override {        cout << "Drawing

a Rectangle" << endl;     } }; int main() {

   Shape* shape1 = new Circle();

Shape* shape2 = new Rectangle();
shape1->draw();     shape2->draw();
delete shape1;     delete shape2;     return
0; }
```

LAB ASSIGNMENTS

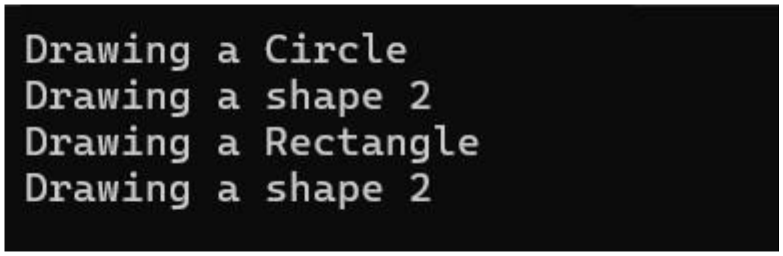
1. Create a base class Shape with a virtual function draw() that prints "Drawing Shape". Derive two classes, Circle and Rectangle, each overriding draw() to print "Drawing Circle" and "Drawing Rectangle", respectively. In the main function:

- Create Circle and Rectangle objects.
- Use a Shape* pointer to call draw() on both objects to show polymorphic behavior.
- Create a version of Shape without the virtual keyword for draw() and repeat the experiment. Compare outputs to explain why virtual functions are needed.
- Use a Circle* pointer to call draw() on a Circle object and compare with the base class pointer's behavior.

Source Code:

```
#include <iostream>
using namespace std;
class Shape{
public:
   virtual void Draw(){         cout <<
"Drawing a shape" << endl;
    }
    void Draw2(){         cout << "Drawing a
shape 2" << endl;
    } }; class Circle : public
Shape{     public:
    void Draw() override{         cout <<
"Drawing a Circle" << endl;
    }
    void Draw2() {         cout << "Drawing a
Circle 2" << endl;     } }; class Rectangle :
public Shape{     public:
    void Draw() override{         cout <<
"Drawing a Rectangle" << endl;
    }
    void Draw2() {         cout << "Drawing a
Rectangle 2" << endl;
    } }; int main() {     Shape
*shape;     shape = new
Circle();     shape->Draw();
shape->Draw2();     shape =
new Rectangle();     shape-
>Draw();     shape-
>Draw2();
    return 0;
}
```

Output:



```
Drawing a Circle
Drawing a shape 2
Drawing a Rectangle
Drawing a shape 2
```

2. Create an abstract base class Animal with a pure virtual function speak() and a virtual destructor. Derive two classes, Dog and Cat, each implementing speak() to print "Dog barks" and "Cat meows", respectively. Include destructors in both derived classes that print "Dog destroyed" and "Cat destroyed". In the main function:

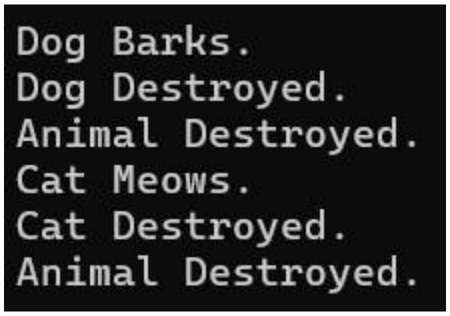
- Attempt to instantiate an Animal object (this should fail).

- Create Dog and Cat objects using Animal* pointers and call speak().
- Delete the objects through the Animal* pointers and verify that derived class destructors are called.
- Modify the Animal destructor to be non-virtual, repeat the deletion, and observe the difference.

Source Code:

```
#include  <iostream>

using namespace std;

class Animal{

   public:

   virtual void speak() = 0;

virtual~Animal(){          cout << "Animal

Destroyed." << endl;

   };

}; class Dog : public

Animal{

   public:

   void speak() override{          cout

<< "Dog Barks." << endl;

   }

   ~Dog(){         cout << "Dog

Destroyed." << endl;

   }

};
class Cat : public Animal{

public:

   void speak() override{          cout

<< "Cat Meows." << endl;

   }
   ~Cat(){         cout << "Cat
Destroyed." << endl;
   } }; int main(){
Animal *animal;
animal = new Dog();
animal->speak();
delete animal;
animal = new Cat();
animal->speak();
delete animal;     return
0;
}
```

Output:



```
Dog Barks.
Dog Destroyed.
Animal Destroyed.
Cat Meows.
Cat Destroyed.
Animal Destroyed.
```

3. Create a base class Employee with a virtual function getRole() that returns a string "Employee". Derive two classes, Manager and Engineer, overriding getRole() to return "Manager" and "Engineer", respectively. In the main function:

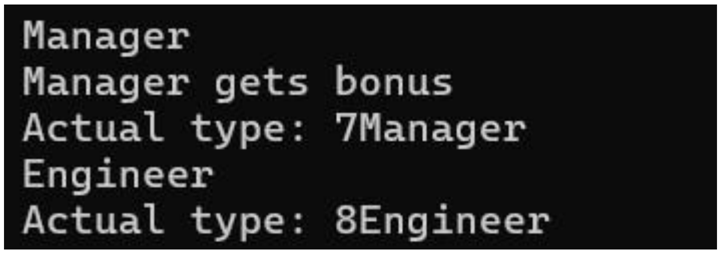
- Create an array of Employee* pointers to store Manager and Engineer objects.
- Iterate through the array to call getRole() for each object.
- Use dynamic_cast to check if each pointer points to a Manager, and if so, print a bonus message(e.g., "Manager gets bonus").
- Use typeid to print the actual type of each object.

Source Code:

```
#include <iostream>
#include <typeinfo>
```

```
using namespace std;

class Employee{

    public:.

    virtual void getRole(){

cout<<"Employee"<<endl;

    }

};

class Manager : public Employee{

    public:

    void getRole() override{

cout<<"Manager"<<endl;

    }

}; class Engineer : public

Employee{

    public:

    void getRole() override{

 cout<<"Engineer"<<endl;

    }
}; int main(){     Employee *e[2];     e[0] =

new Manager();     e[1] = new Engineer();

for(int i=0;i<2;i++){         e[i]->getRole();

if(dynamic_cast<Manager*>(e[i])) {

cout << "Manager gets bonus" << endl;

        }

         cout << "Actual type: " << typeid(*e[i]).name() << endl;

    }

    for(int x=0;x<2;x++){
        delete e[x];
    }
return 0;
}
```

Output:



```
Manager
Manager gets bonus
Actual type: 7Manager
Engineer
Actual type: 8Engineer
```

DISCUSSION

In this lab we were able to understand the core concept of virtual functions where we learnt about overriding functions, pure and virtual functions. We also learnt about dynamic cast, typeid and typeinfo got information on interpreting cast and many more. It was very difficult to understand these concepts but with the help of online mediums on the internet I was able to understand these concepts.

CONCLUSION

From this lab, we can conclude that the use of virtual functions enhances code maintainability, reduces the need for rewriting code, and facilitates polymorphism which is a core concept in OOP.