Experiment No- 6 **Date-**

Aim – To study Virtual functions and Polymorphism (Runtime Polymorphism)

Theory –

Runtime Polymorphism (Dynamic Polymorphism)

Runtime polymorphism, also known as dynamic polymorphism, is a concept in object-oriented programming that allows a method to perform different tasks based on the object that it is called upon. It's called runtime because the decision about which method to invoke is made during program execution.

The most common way to achieve runtime polymorphism in languages like C++ or is through method overriding and the use of pointers (or references) to the base class.

Key Characteristics:

1. **Inheritance:** Runtime polymorphism requires a base class and at least one derived class that overrides a method of the base class.
2. **Overriding**: A derived class redefines a function of the base class. This enables the derived class to have specific functionality for the function.
3. **Pointers and References:** A base class pointer or reference can refer to derived class objects. This allows for the method resolution to happen at runtime.
4. **Virtual Functions**: In C++, the base class function should be declared with the virtual keyword to enable runtime polymorphism.

Virtual Functions in C++

A **virtual function** is a member function in a base class that you expect to override in derived classes. When you use a base class pointer or reference to call a virtual function, C++ ensures that the correct function of the derived class is called, based on the actual object type, not the type of reference or pointer used.

This feature enables **runtime polymorphism**, allowing different objects to be treated uniformly while ensuring that their specific behavior is correctly invoked.

SYNTAX

```
class Base {
public:
   virtual void functionName();  // Declares a virtual function
   virtual ~Base();           // Virtual destructor
};
```

Advantages of Virtual Functions

1. **Runtime Polymorphism**: Allows dynamic method resolution for the correct function to be called during program execution.
2. **Code Reusability**: Promotes reuse of code with common interfaces for different derived types.
3. **Decoupling**: Helps achieve low coupling between code modules, enhancing maintainability.
4. **Uniformity**: Allows treating derived class objects through base class pointers, promoting a unified interface.
5. **Dynamic Binding**: Supports late binding, making the program adaptable for new derived types without altering the base class logic.

Disadvantages of Virtual Functions

1. **Performance Overhead**: Function calls through the virtual table are slightly slower compared to normal function calls.
2. **Memory Consumption**: Each object with virtual functions has an associated virtual table pointer (vptr), which adds memory overhead.
3. **Complexity**: Adds complexity due to the need for careful management of function overrides.
4. **Requires Virtual Destructors**: If not used correctly, such as lacking a virtual destructor, it can lead to resource management issues, particularly in derived classes.
5. **Debugging Difficulty**: Debugging code that uses runtime polymorphism can be challenging because function calls are resolved at runtime, not compile time.

Pure Virtual Functions in C++

A **pure virtual function** is a virtual function that is declared within a base class but has no implementation in that class. The purpose of a pure virtual function is to provide an interface that derived classes must implement, making the base class abstract.

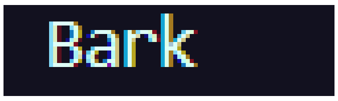
A class containing at least one pure virtual function is known as an **abstract class**. This means you cannot create an instance of an abstract class; you can only use it as a base class for other classes.

[A] Write a C++ program to understand virtual functions in C++

PROGRAM –

```
#include <iostream>
using namespace std;

class Animal {
public:
   virtual void sound() {  // Virtual function
      cout << "Generic animal sound" << endl;
   }
};

class Dog : public Animal {
public:
   void sound() override {  // Overridden function
      cout << "Bark" << endl;
   }
};

int main() {
   Animal *animal = new Dog();  // Base class pointer to
derived class object
   animal->sound();          // Output: Bark (runtime
polymorphism)

   delete animal;
   return 0;
}
```

OUTPUT-



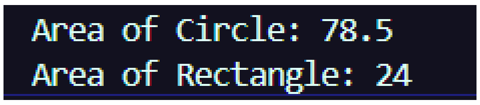
```
Bark
```

[B] Write a C++ program to understand pure virtual functions in C++

PROGRAM –

```
#include <iostream>
using namespace std;

// Abstract Base class
class Shape {
public:
   // Pure virtual function
   virtual void area() = 0;

   virtual ~Shape() {}  // Virtual destructor to ensure proper
cleanup
};

// Derived class 1
class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) : radius(r) {}

   void area() override {
      cout << "Area of Circle: " << 3.14 * radius * radius <<
endl;
   }
};

// Derived class 2
class Rectangle : public Shape {
private:
   double length, width;
public:
   Rectangle(double l, double w) : length(l), width(w) {}
```

OUTPUT-



```
Area of Circle: 78.5
Area of Rectangle: 24
```

```
    void area() override {
        cout << "Area of Rectangle: " << length * width <<
endl;
    }
};

int main() {
    // Array of base class pointers
    Shape* shapes[2];

    // Assign derived class objects
    shapes[0] = new Circle(5.0);
    shapes[1] = new Rectangle(4.0, 6.0);

    // Call area() for each shape
    for (int i = 0; i < 2; i++) {
        shapes[i]->area();
    }

    // Clean up
    for (int i = 0; i < 2; i++) {
        delete shapes[i];
    }

    return 0;
}
```