Day 107/180

Polymorphism 📑

Ques. Difference between Compile time Polymorphism and Runtime Polymorphism.

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

In C++, polymorphism can be divided into two types:

1. Compile-time Polymorphism
2. Run-time Polymorphism

Compile Time Polymorphism in C++

In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments. At compile-time, c++ knows which method to call by checking the method signatures. So this is called compile-time polymorphism or static or early binding.

Compile-time polymorphism in C++ is achieved through,

1. Function Overloading
2. Operator Overloading

Example of Compile-time Polymorphism

```
class CompileTimePloymorphismExample {
     // First display function
    void display() {
       System.out.println("In Display without parameter");
    }
    // Second display function but with argument
    void display(String value) {
       System.out.println("In Display with parameter" + value);
    }
}
public class Main {
    public static void main(String args[]) {
        CompileTime obj = new CompileTime();
        obj.display();
        obj.display("Polymorphism");
    }
}
```

Run Time Polymorphism in C++

Whenever an object is bound with the functionality at runtime, this is known as runtime polymorphism. The runtime polymorphism can be achieved by method overriding. The decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type. It is also called dynamic or late binding.

Run Time Polymorphism can be exhibited by:

1. Method Overriding using Virtual Functions

Method Overriding

Method overriding refers to the process of creating a new definition of a function in a derived class that is already defined inside its base class. Some rules that must be followed while overriding a method are:

1. Method names must be the same.
2. Method parameters must be the same.

Virtual Function

1. Virtual Function is a member function that is declared as virtual in the base class and it can be overridden in the derived classes that inherit the base class.
2. Virtual functions are generally declared in the base class and are typically defined in both the base and derived classes.

Example of Run-time Polymorphism

```
// C++ Code to illustrate Virtual Function
#include <iostream>
using namespace std;

// base class
class Base {
public:
    // virtual function
    virtual void print() { cout << "base"; }
};

// derived class
class Derived : public Base {
public:
    // method overriding
    void print() override { cout << "derived"; }
};

// driver code
int main()
{
    Base* obj = new Derived();
    obj->print();
    delete obj;
    return 0;
}
```

Difference Between Compile Time And Run Time Polymorphism

Compile-Time Polymorphism	Run-Time Polymorphism
It is also called Static Polymorphism.	It is also known as Dynamic Polymorphism.
In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.	In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type.
Function calls are statically bound.	Function calls are dynamically bound.
Compile-time Polymorphism can be exhibited by: 1. Function Overloading 2. Operator Overloading	Run-time Polymorphism can be exhibited by Function Overriding.
Faster execution rate.	Comparatively slower execution rate.

Inheritance is not involved.	Involves inheritance.

Ques. Overload this Operator (* Multiply) and Unary Operator (- Minus), with your own Example.

```cpp
#include <iostream>

class Number {
private:
    int value;

public:
    // Constructor
    Number(int val) : value(val) {}

    // Getter for value
    int getValue() const {
        return value;
    }

    // Overloading * (Multiply) operator
    Number operator*(const Number& other) const {
        return Number(value * other.value);
    }

    // Overloading unary - (Minus) operator
    Number operator-() const {
        return Number(-value);
    }
};

// Overloading << operator for easy printing
std::ostream& operator<<(std::ostream& os, const Number& num) {
    os << num.getValue();
    return os;
}

int main() {
    // Example usage
    Number a(5);
    Number b(3);

    // Overloaded * (Multiply) operator
    Number resultMultiply = a * b;
    std::cout << a << " * " << b << " = " << resultMultiply << std::endl;

    // Overloaded unary - (Minus) operator
    Number resultMinus = -a;
    std::cout << "Negative of " << a << " = " << resultMinus <<
std::endl;

    return 0;
}
```

Ques. Create a Virtual Function Example in C++ with your own example.

```
// C++ Code to illustrate Virtual Function
#include <iostream>
using namespace std;

// base class
class Base {
public:
     // virtual function
     virtual void print() { cout << "base"; }
};

// derived class
class Derived : public Base {
public:
     // method overriding
     void print() override { cout << "derived"; }
};

// driver code
int main()
{
     Base* obj = new Derived();
```

```
    obj->print();
    delete obj;
    return 0;
}
```

____ END ____