03. Polymorphism

Concept of Polymorphism

- Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common base class. The term "polymorphism" means "many forms," and it enables methods to do different things based on the object it is acting upon, even if the methods share the same name. This concept facilitates flexibility and the ability to call derived class methods through base class pointers or references.

Compile Time & Run time Polymorphism

- Polymorphism in C++ can be classified into two types: **compile-time polymorphism** and **run-time polymorphism**. These types of polymorphism enable flexibility in how functions or operators are used, allowing the same function or operator to behave differently based on the context.

1. Compile Time

- Compile-time polymorphism, also known as static polymorphism, refers to the ability of the compiler to resolve function calls and operator usage at compile time. This type of polymorphism is achieved through function overloading and operator overloading.

  1. Function Overloading
     - Allows multiple functions with the same name but different parameter lists (different number or types of parameters) to be defined in the same scope. The compiler determines which function to call based on the function signature at compile time.
  2. Operator Overloading
     - Allows defining custom behavior for operators (e.g., `+`, `-`, `*`, `==`) when they are used with user-defined types. The compiler determines which operator function to call based on the types of operands at compile time.

2. Run-time Polymorphism

- Run-time polymorphism, also known as dynamic polymorphism, refers to the ability to resolve function calls at runtime based on the actual object type rather than the type of reference or pointer used to access the object. This type of polymorphism is achieved through virtual functions and inheritance.

  1. Virtual Function
     - Functions declared with the `virtual` keyword in a base class are intended to be overridden in derived classes. At runtime, the appropriate function to call is determined based on the actual object type pointed to or referenced, even if accessed through a base class pointer or reference.
  2. Function Overriding

- Occurs when a derived class provides a specific implementation of a method that is already defined as `virtual` in its base class. The method in the derived class has the same name, return type, and parameters as the method in the base class. The correct method to be invoked is determined at runtime based on the actual object type.

Function Overloading

- Allows multiple functions with the same name to be defined within the same scope, provided they have different parameter lists (different number or types of parameters). The correct function to be called is determined at compile time based on the function signature.

Applications

1. **Multiple Operations with Different Data Types**:
   - Allows defining functions with the same name to handle different data types or numbers of parameters, improving code readability and usability.
2. **Convenience and Flexibility**:
   - Facilitates creating functions that perform similar tasks but with varying inputs, making code easier to maintain and extend.
3. **Compile-Time Efficiency**:
   - Enhances performance by resolving function calls at compile time, leading to faster execution.

Function Overriding

- Occurs when a derived class provides a specific implementation of a method that is already defined as `virtual` in its base class. The method in the derived class has the same name, return type, and parameters as the method in the base class. The correct method to be invoked is determined at runtime based on the actual object type.

Applications

1. **Dynamic Behavior Customization**:
   - Enables derived classes to provide specific implementations of methods defined in a base class, allowing different behaviors based on the object's actual type.
2. **Run-Time Polymorphism**:
   - Supports dynamic method resolution, where the method implementation is chosen at runtime, allowing for flexible and extensible code structures.
3. **Code Extensibility**:

- Allows extending and modifying the behavior of base class methods in derived classes without altering the base class, promoting code reuse and maintainability.

Overloading vs Overriding

OVERLOADING	OVERRIDING
Function Overloading provides multiple definitions of the function by changing signature.	Function Overriding is the redefinition of base class function in its derived class with same signature.
An example of compile time polymorphism.	An example of run time polymorphism.
Function signatures should be different.	Function signatures should be the same.
Overloaded functions are in same scope.	Overridden functions are in different scopes.
Overloading is used when the same function has to behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some different job than the base class function.
A function has the ability to load multiple times.	A function can be overridden only a single time.
In function overloading, we don't need inheritance.	In function overriding, we need an inheritance concept.

Concept of Friend Function

- A **friend function** is a special function in C++ that is not a member of a class but has access to its private and protected members. It is declared within the class using the `friend` keyword and can be either a global function or a member function of another class.
- A friend function in C++ is a non-member function or a member of another class that is granted access to the private and protected members of a class. It is declared within the class using the `friend` keyword and can be used to implement operations that need access to internal class data but do not fit logically as member functions.
- **EXAMPLE**

```
class MyClass {
private:
    int secret;
public:
    MyClass() : secret(42) {}
    friend void revealSecret(MyClass& obj); // Friend function declaration
};

void revealSecret(MyClass& obj) {
```

```
    std::cout << "The secret is " << obj.secret << std::endl;
}
```

Concept of Virtual Function & Pure Function

- **Virtual Function** and **Pure Virtual Function** are key concepts in C++ that support runtime polymorphism and abstract class design. They enable a more flexible and dynamic method resolution mechanism, which is central to object-oriented programming.

1. Virtual Function

- A **virtual function** is a member function in a base class that is declared with the `virtual` keyword. It allows derived classes to provide specific implementations for the function, and the correct function implementation is determined at runtime based on the actual object type.
- **EXAMPLE**

```
class Base {
public:
    virtual void show() {
        std::cout << "Base class show" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show" << std::endl;
    }
};
```

2. Pure Function

- A **pure virtual function** is a virtual function that is declared in a base class and has no implementation in that class. It is specified by assigning `0` to the function in its declaration. Classes containing pure virtual functions are abstract classes and cannot be instantiated directly.
- **EXAMPLE**

```
#include <iostream>

class Shape {
```

```
public:
    virtual void draw() const = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Circle" << std::endl;
    }
};

int main() {
    Shape* s = new Circle();
    s->draw(); // Outputs: Circle
    delete s;
    return 0;
}
```