Polymorphism in C++

Polymorphism is a core concept in object-oriented programming (OOP) and refers to the ability of a function, method, or operator to behave in multiple ways depending on the context. In C++, polymorphism is achieved through **inheritance** and **dynamic dispatch**, enabling methods in derived classes to override base class methods and allow objects of different types to be treated uniformly.

Types of Polymorphism in C++

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

1. **Compile-time Polymorphism** (also known as **Static Polymorphism**)
2. **Runtime Polymorphism** (also known as **Dynamic Polymorphism**)

---

1. Compile-time Polymorphism (Static Binding)

Compile-time polymorphism is resolved during compilation. In C++, the most common examples of compile-time polymorphism are:

- **Function Overloading**
- **Operator Overloading**

Function Overloading

Function overloading allows multiple functions with the same name but different parameter lists (either different number of parameters or different types). The function to be invoked is determined at compile time based on the arguments passed to it.

Example of function overloading:

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

class Calculator {
public:
    // Function to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded function to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Calculator calc;
```

```
    cout << "Sum of 2 numbers: " << calc.add(5, 10) << endl;        // Output: 15
    cout << "Sum of 3 numbers: " << calc.add(5, 10, 15) << endl;    // Output: 30

    return 0;
}
```

In this example, the `add` function is overloaded with two versions: one takes two parameters, and the other takes three. The compiler determines which version to call based on the number of arguments provided during the function call.

Operator Overloading

Operator overloading allows you to define or modify the behavior of operators for user-defined data types (like classes).

Example of operator overloading:

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

class Complex {
private:
    int real, imag;

public:
    Complex(int r, int i) : real(r), imag(i) {}

    // Overload '+' operator to add two complex numbers
    Complex operator+(const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;  // Calls overloaded operator+
    c3.display();           // Output: 4 + 6i

    return 0;
}
```

Here, the `+` operator is overloaded for the `Complex` class, allowing two complex numbers to be added using the `+` operator.

---

2. Runtime Polymorphism (Dynamic Binding)

Runtime polymorphism is resolved during execution and is primarily achieved through **method overriding** in the context of inheritance and virtual functions. In C++, runtime polymorphism is made possible by declaring the base class method as `virtual`, allowing the appropriate derived class method to be invoked at runtime based on the actual object type.

Method Overriding

In method overriding, a subclass provides its own implementation of a method that is already defined in its superclass. The method is marked as `virtual` in the base class, and the correct method is selected based on the actual object type at runtime.

Example of runtime polymorphism:

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

class Animal {
public:
    // Virtual function in base class
    virtual void sound() {
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
    // Overriding the sound method in Dog class
    void sound() override {
        cout << "Dog barks" << endl;
    }
};

class Cat : public Animal {
public:
    // Overriding the sound method in Cat class
    void sound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog();  // Animal pointer, but Dog object
    Animal* animal2 = new Cat();  // Animal pointer, but Cat object

    animal1->sound();  // Output: Dog barks
    animal2->sound();  // Output: Cat meows

    delete animal1;
    delete animal2;
```

```
    return 0;
}
```

In this example:

- The `sound()` method is marked as `virtual` in the base class `Animal`.
- The `Dog` and `Cat` classes override this method to provide their specific implementations.
- At runtime, when `sound()` is called on the `Animal` pointer, the correct method is invoked based on the actual object type (`Dog` or `Cat`).

Virtual Functions and Dynamic Binding

The key to enabling runtime polymorphism in C++ is the use of **virtual functions**. When a function is marked as `virtual`, it tells the compiler that it should use dynamic dispatch to select the appropriate method based on the actual object type, not just the reference type. This is known as **dynamic binding**.

Key Points to Remember

- **Compile-time polymorphism** is resolved at compile time and includes **function overloading** and **operator overloading**.
- **Runtime polymorphism** is resolved at runtime and involves **method overriding** using **virtual functions**.
- To achieve runtime polymorphism, a base class method must be marked as `virtual`, and the derived class should override this method.
- **Virtual destructors** are important in C++ when using polymorphism, as they ensure proper cleanup of dynamically allocated memory when an object is deleted through a base class pointer.

Benefits of Polymorphism in C++

- **Code Reusability**: Polymorphism promotes code reuse by allowing the same interface to be used for different object types.
- **Flexibility**: Polymorphism enables more flexible and extensible code, allowing new classes to be introduced without changing the existing codebase.
- **Maintainability**: It simplifies the code by allowing common functionality to be handled through base class references or pointers, making the code easier to maintain and extend.

Conclusion

Polymorphism is a powerful feature in C++ that enables flexibility and extensibility in object-oriented design. By allowing a single function or method to operate on different types, polymorphism helps developers write cleaner, more maintainable, and reusable code. Understanding both **compile-time** and **runtime polymorphism** is crucial for designing robust object-oriented systems in C++.