Modern C++: Polymorphism and Its Types (Compile-time vs Run-time)

Polymorphism is one of the fundamental pillars of Object-Oriented Programming (OOP) in C++. It allows objects of different classes to be treated as objects of a common base class. Polymorphism helps in writing flexible and maintainable code. In C++, polymorphism can be categorized into two main types: **Compile-time (Static)** and **Run-time (Dynamic)** polymorphism. Both types offer different mechanisms and use cases.

This article will explore each type in detail, along with examples to clarify the concepts.

---

1. Compile-Time Polymorphism (Static Polymorphism)

Compile-time polymorphism is determined during the compilation process. This type of polymorphism is achieved using **function overloading**, **operator overloading**, and **templates**. The compiler decides which function or operator to call at compile time based on the function signature.

1.1 **Function Overloading**

Function overloading allows multiple functions with the same name but different parameters. The compiler selects the appropriate function based on the number or types of arguments passed to the function.

Example:

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

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
```

```
    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Calculator calc;
    cout << "Addition of two integers: " << calc.add(10, 20) << endl;
    cout << "Addition of two doubles: " << calc.add(10.5, 20.5) << endl;
    cout << "Addition of three integers: " << calc.add(10, 20, 30) << endl;
    return 0;
}
```

In the example above, the function `add` is overloaded to handle different types and numbers of parameters.

1.2 Operator Overloading

C++ allows the overloading of most operators, such as `+`, `-`, `*`, etc., to work with user-defined data types. Operator overloading provides a way to define how operators behave for custom classes.

Example:

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

class Complex {
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : 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() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3.5, 2.5), c2(1.5, 3.0);
    Complex c3 = c1 + c2;  // Calls the overloaded + operator
    c3.display();
    return 0;
```

```
}
```

Here, the `+` operator is overloaded to add two complex numbers.

1.3 **Templates (Generic Programming)**

Templates allow for writing generic code that works with any data type. This is another form of compile-time polymorphism.

Example:

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

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << "Addition of integers: " << add<int>(10, 20) << endl;
    cout << "Addition of doubles: " << add<double>(10.5, 20.5) << endl;
    return 0;
}
```

In this example, the template function `add` can handle both integers and floating-point numbers.

---

2. Run-time Polymorphism (Dynamic Polymorphism)

Run-time polymorphism is determined during program execution. This is achieved through **inheritance** and **virtual functions**. In C++, a base class can define a function as `virtual`, which allows derived classes to override this function. The correct function to call is determined at runtime based on the object type.

2.1 **Inheritance and Virtual Functions**

A base class may have virtual functions, which can be overridden in derived classes. When using pointers or references to base classes, the appropriate derived class function is called at runtime.

Example:

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

class Animal {
public:
    virtual void sound() const {
```

```
        cout << "Some generic animal sound" << endl;
    }
};

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

class Cat : public Animal {
public:
    void sound() const override {
        cout << "Meow" << endl;
    }
};

void makeSound(const Animal& animal) {
    animal.sound();  // Calls the appropriate derived class function
}

int main() {
    Dog dog;
    Cat cat;

    makeSound(dog);  // Output: Bark
    makeSound(cat);  // Output: Meow

    return 0;
}
```

Here, `sound` is a virtual function. The correct version of the function is called at runtime based on the actual type of the object.

2.2 **Pure Virtual Functions and Abstract Classes**

In some cases, a base class may declare a pure virtual function, making the class **abstract**. This forces derived classes to provide an implementation for the function.

Example:

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

class Shape {
public:
    virtual void draw() const = 0;  // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle" << endl;
```

```
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a rectangle" << endl;
    }
};

void renderShape(const Shape& shape) {
    shape.draw();
}

int main() {
    Circle circle;
    Rectangle rectangle;

    renderShape(circle);     // Output: Drawing a circle
    renderShape(rectangle);  // Output: Drawing a rectangle

    return 0;
}
```

In this example, the `Shape` class defines a pure virtual function `draw`, making it an abstract class. The derived classes, `Circle` and `Rectangle`, provide their own implementations.

Key Differences between Compile-time and Run-time Polymorphism

Aspect	Compile-time Polymorphism	Run-time Polymorphism
Determination	Decided during compilation	Decided during program execution
Mechanism	Achieved using function overloading, operator overloading, and templates	Achieved using virtual functions and inheritance
Performance	Generally faster as it's resolved at compile time	Slightly slower due to dynamic binding
Flexibility	Less flexible, depends on function signatures	More flexible, allows dynamic behavior
Use Cases	Simple cases with known types at compile-time	When polymorphic behavior is needed at runtime

Conclusion

Polymorphism in C++ is a powerful feature that enhances code reusability, flexibility, and maintainability. Understanding both compile-time and run-time polymorphism allows you to write efficient and dynamic code. **Compile-time polymorphism** (via function overloading, operator overloading, and templates) is resolved during compilation, making it faster but less flexible. **Run-time polymorphism** (via virtual functions and inheritance) is determined during execution, providing flexibility for complex, dynamic behavior in OOP.

Both types have their advantages, and knowing when and how to use each is essential for writing robust and scalable applications in modern C++.