📘 Polymorphism in C++

Overview

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different types to be treated as objects of a common base type. In C++, polymorphism enables flexibility and reusability in code by allowing the same interface to behave differently for different underlying data types.

There are two main types of polymorphism in C++:

1. **Compile-time polymorphism** (also known as static polymorphism)
2. **Run-time polymorphism** (also known as dynamic polymorphism)

---

1. Compile-Time Polymorphism

The decision about which function to call is made at **compile time**. This is achieved through **function overloading** and **operator overloading**.

Function Overloading

For a function to be overloaded, the methods...

- must be different on the parameters
  - the number of parameters are different OR
  - the type of parameters are different
- must have the same exact name (casing is critical)

> NOTE: it is NOT enough that the functions are different on the return type. The return type can be different but it is not enough to make an overload. The parameters have to be different.

Example: Function Overloading

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

class Print {
public:
    void show(int i) {
        cout << "Integer: " << i << endl;
    }

    void show(double d) {
        cout << "Double: " << d << endl;
    }

    void show(string s) {
        cout << "String: " << s << endl;
    }
};

int main() {
    Print p;
    p.show(5);
    p.show(3.14);
    p.show("Hello");
    return 0;
}
```

Operator Overloading in C++

Operator overloading in C++ allows developers to redefine the behavior of operators (like +, -, ==, etc.) for user-defined types (classes or structs). This makes objects of those types behave more like built-in types, improving code readability and usability.

Why Use Operator Overloading?

- To provide intuitive operations for custom classes (e.g., adding two Complex numbers using +).
- To make code more expressive and maintainable.
- To enable polymorphic behavior for operators.

Key Points

1. Syntax: Operator overloading is done by defining a special function using the keyword operator followed by the operator symbol.

```cpp
return_type operator<symbol>(parameter_list);
```

2. Rules

- You cannot create new operators; only overload existing ones.
- At least one operand must be a user-defined type.
- Some operators cannot be overloaded (e.g., ::, .?, sizeof).
- Overloading does not change operator precedence or associativity.

3. Types of overloading

- Member function: Operates on the current object (this).
- Non-member function: Often used when the left operand is not the class type.

Example: Operator Overloading

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

class Complex {
private:
    double real, imag;

public:
    // Constructor
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Overload '+' operator
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    // Display function
    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3.0, 4.0);
    Complex c2(1.5, 2.5);

    Complex sum = c1 + c2;  // Uses overloaded operator+
    sum.display();          // Output: 4.5 + 6.5i

    return 0;
}
```

Explanation of Example

- operator+ is defined as a member function.
- It takes a const Complex& parameter to avoid unnecessary copying.
- Returns a new Complex object representing the sum.
- In main(), c1 + c2 invokes the overloaded operator, making the syntax natural.

Benefits

- Improves code readability (c1 + c2 instead of c1.add(c2)).
- Makes custom types behave like built-in types.

---

2. Run-Time Polymorphism

This is achieved using **inheritance** and **virtual functions**. The function call is resolved at runtime using **dynamic dispatch**.

Key Concepts:

- **Base class pointer/reference** can point to **derived class objects**.
- `virtual`
  - Declares a function in the **base class** that can be **overridden** in derived classes.
  - Enables **dynamic dispatch** via a **vtable** (virtual table).
  - Ensures that the correct function is called for an object, even when accessed through a base class pointer or reference.
- `override`
  - Used in the **derived class** to indicate that a function is **intended to override** a virtual function from the base class.
  - Helps catch errors at compile time (e.g., mismatched function signatures).

🧱 Syntax and Structure

Step 1: mark the base method with the `virtual` keyword

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

- `virtual` tells the compiler to support dynamic dispatch for `display()`.

Step 2: override the method in the derived class

> NOTE: the method signature in the derived MUST match the signature in the base

```cpp
class Derived : public Base {
public:
    void display() const override; //optionally add 'override'
};
void Derived::display() const {
    std::cout << "Derived display()" << std::endl;
}
```

- `override` ensures this method matches a virtual method in the base class.

🧪 Full Example

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

class Animal {
public:
    virtual void speak() {
        cout << "Animal speaks " << endl;
    }
};

class Dog : public Animal {
public:
    void speak() override {
        //FULL override: do NOT call the base version
        cout << "Dog barks" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() override {
        //EXTENSION override: call the base version
        Animal::speak();
        cout << "Cat meows" << endl;
    }
};

void makeSound(Animal* a) {
    a->speak();  // Dynamic dispatch
}

int main() {
    Dog d;
    Cat c;

    makeSound(&d);  // Output: Dog barks
    makeSound(&c);  // Output: Animal speaks Cat meows

    return 0;
}
```

🧭 Virtual/Override Summary

Keyword	Used In	Purpose
`virtual`	Base class	Enables dynamic dispatch
`override`	Derived class	Ensures correct overriding of base method

Why Use Polymorphism?

- **Extensibility**: Easily add new classes with minimal changes to existing code.
- **Maintainability**: Reduces code duplication and improves readability.
- **Flexibility**: Write more generic and reusable code.

🧰 Best Practices

- Always use `override` in derived classes to avoid subtle bugs.
- Use `const` correctness if the base method is `const`.
- Prefer using **references or pointers** to base class for polymorphism.

Summary

Type	Mechanism	Binding Time	Example
Compile-time	Function/Operator Overloading	Compile Time	`show(int)`, `show(double)`
Run-time	Virtual Functions	Run Time	`speak()` in `Animal` class

🎯 Quiz!

Here's a short quiz on the topic: quiz

Footer Separator

🔭 Markdown Viewer

How to view the markdown files in a browser...

- Markdown Viewer

🧠 Lecture Practices

Here are the lecture Practices...

- Day 7
- Day 8
- Day 9

🔍 Lecture Quizzes

Here are the lecture quizzes...

- Day 7
- Day 8
- Day 9

Weekly Topics

Here are the topics for the week...

- Classes
- Structs
- Fields
- Getters and Setters
- Constructors
- Instances
- Inheritance
- Polymorphism
- Pointers
- Upcasting
- Misc. Concepts
- 4 Pillars of OOP