# CS2810 OOAIA

# Polymorphism

---

## Example Code Snippets

1. **Basic Function Overloading**

```cpp
#include <iostream>

// Demonstrates compile-time polymorphism through function overloading
class Calculator {
public:
    int add(int x, int y) {
        return x + y;
    }

    double add(double x, double y) {
        return x + y;
    }

    int add(int x, int y, int z) {
        return x + y + z;
    }
};

int main() {
    Calculator calc;
    std::cout << "Addition of integers: " << calc.add(3, 4) << std::endl;
```

```
    std::cout << "Addition of doubles: " << calc.add(3.5,
2.5) << std::endl;
    std::cout << "Addition of three integers: " <<
calc.add(1, 2, 3) << std::endl;
    return 0;
}
```

2. **Operator Overloading**

```
#include <iostream>

// Shows polymorphic behavior through operator overloading
class Complex {
    double real, imag;
public:
    Complex(double r, double i): real(r), imag(i) {}

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

    void print() const {
        std::cout << real << " + " << imag << "i" <<
std::endl;
    }
};

int main() {
    Complex c1(2.3, 3.4), c2(1.2, 4.5);
    Complex c3 = c1 + c2;
    c3.print();  // Output: 3.5 + 7.9i
    return 0;
}
```

3. **Virtual Functions (Runtime Polymorphism)**

```cpp
#include <iostream>

// Demonstrates runtime polymorphism using virtual functions
class Shape {
public:
    virtual double area() const = 0;  // Pure virtual function
    virtual ~Shape() {} // Virtual destructor for proper cleanup
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {}
    double area() const override {
        return 3.14159 * radius * radius;
    }
};

class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h): width(w), height(h) {}
    double area() const override {
        return width * height;
    }
};

int main() {
    Shape* shape1 = new Circle(5);
    Shape* shape2 = new Rectangle(4, 6);

    std::cout << "Circle Area: " << shape1->area() <<
```

```
std::endl;
    std::cout << "Rectangle Area: " << shape2->area() <<
std::endl;

    delete shape1;
    delete shape2;
    return 0;
}
```

4. **Smart Pointer Overloading**

```
#include <iostream>

// Shows smart pointer behavior through operator
overloading
class SmartPtr {
    int* ptr;
public:
    explicit SmartPtr(int* p): ptr(p) {}

    ~SmartPtr() { delete ptr; }

    // Overload dereference operator
    int& operator*() {
        return *ptr;
    }

    // Overload arrow operator
    int* operator->() {
        return ptr;
    }
};

int main() {
    SmartPtr sp(new int(42));
    std::cout << "Value: " << *sp << std::endl;
```

```
    return 0;
}
```

5. **Function Object (Functor)**

```
#include <iostream>

// Demonstrates polymorphic behavior through function
objects
class Multiplier {
    int factor;
public:
    explicit Multiplier(int f): factor(f) {}

    int operator()(int x) const {
        return x * factor;
    }
};

int main() {
    Multiplier times2(2);
    std::cout << "2 * 5 = " << times2(5) << std::endl;
    return 0;
}
```

6. **Template Polymorphism**

```
#include <iostream>

// Shows compile-time polymorphism using templates
template<typename T>
class Container {
    T data;
public:
    Container(T val) : data(val) {}

    void store(T val) {
```

```
        data = val;
    }

    T retrieve() const {
        return data;
    }

    void display() const {
        std::cout << "Stored data: " << data << std::endl;
    }
};

int main() {
    Container<int> intContainer(10);
    intContainer.display();

    Container<std::string> strContainer("Hello");
    strContainer.display();

    return 0;
}
```

## Practice Problems

1. **Shape Hierarchy (Difficulty: Easy)**
   Create a class hierarchy for geometric shapes with a **base class** Shape that provides **pure virtual functions** for calculating **area** and **perimeter**. Implement **derived classes** Rectangle **and** Circle that override these functions appropriately.

   - The Rectangle class should take **width and height** as parameters.
   - The Circle class should take **radius** as a parameter.

- Use **runtime polymorphism** to compute and display the area and perimeter dynamically.

2. **Operator Overloading (Difficulty: Medium)**
   Implement a Fraction **class** that supports arithmetic operations:
   - Overload the +, -, *, and / operators for fraction arithmetic.
   - Overload the << operator for outputting fractions in the form a/b.
   - Implement **error handling** to prevent division by zero.
   - Ensure the fraction is stored in **simplified form** (e.g., 4/6 should be stored as 2/3).
   - Demonstrate the class functionality in a main() function.

3. **Smart Container with Iterators (Difficulty: Hard)**
   Design a **template-based container class** that supports the following:
   - **Dynamic resizing** when new elements are added.
   - Overloaded **indexing operators ([])** for element access with **bounds checking**.
   - Implementation of an **iterator** that allows range-based for loops and standard iteration.
   - Provide functions to **add, remove, and retrieve elements** from the container.

4. **Function Objects for Sorting (Difficulty: Medium)**
   Create a **sorting utility** that can sort an array using different comparison strategies, implemented as **function objects (functors)**.
   - Implement at least **three different sorting criteria**, such as:
     1. **Ascending order**
     2. **Descending order**
     3. **Custom order** (e.g., even numbers before odd numbers)
   - Allow the sorting function to accept a **custom comparator object**.
   - Demonstrate the sorting utility with different strategies.

5. **Expression Tree (Difficulty: Hard)**
   Implement an **expression tree** to evaluate mathematical expressions.

- Use **polymorphism** to create a base class ExpressionNode, with derived classes for:
    - **Operators (+, -, *, /)**
    - **Operands (constants and variables)**
- Implement a function to **evaluate the tree recursively**.
- Allow expressions to be **constructed dynamically** and evaluated at runtime.
- Include a mechanism for **variable substitution**, where variables can be assigned values before evaluation.