2. Compile-Time Polymorphism

2.1 Function Overloading

Definition: Multiple functions with the same name but different parameters.

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

class Printer {
public:
    // Overloaded print functions
    void print(int value) {
        cout << "Printing integer: " << value << endl;
    }

    void print(double value) {
        cout << "Printing double: " << value << endl;
    }

    void print(string value) {
        cout << "Printing string: " << value << endl;
    }

    void print(int value, int times) {
        cout << "Printing " << value << " for " << times << " times: ";
        for (int i = 0; i < times; i++) {
            cout << value << " ";
        }
        cout << endl;
    }
};

int main() {
    Printer p;
```

```
    p.print(42);
    p.print(3.14);
    p.print("Hello, World!");
    p.print(7, 3);

    return 0;
}
```

Rules for Function Overloading:

1. Functions must have different parameter lists
2. Return type alone is NOT enough to differentiate
3. Parameter types, number, or order must differ

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

class Example {
public:
    // VALID overloads
    void func(int x) { cout << "int: " << x << endl; }
    void func(double x) { cout << "double: " << x << endl; }
    void func(int x, int y) { cout << "two ints: " << x << ", " << y << endl; }

    // INVALID: Only return type differs
    // int func(int x) { return x; }  // ERROR!

    // VALID: Different parameter order
    void process(int x, double y) { cout << "int, double" << endl; }
    void process(double x, int y) { cout << "double, int" << endl; }
};

int main() {
    Example ex;

    ex.func(10);
    ex.func(3.14);
    ex.func(5, 7);

    ex.process(5, 3.14);
```

```
    ex.process(3.14, 5);

    return 0;
}
```

2.2 Operator Overloading

Definition: Redefining operators to work with user-defined types.

Basic Syntax:

```
return_type operator symbol (parameters) {
    // implementation
}
```

Example: Complex Number Class

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

class Complex {
private:
    double real;
    double imag;

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

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

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

    // Overload * operator
    Complex operator*(const Complex& other) {
```

```
        return Complex(
            real * other.real - imag * other.imag,
            real * other.imag + imag * other.real
        );
    }

    // Overload == operator
    bool operator==(const Complex& other) {
        return (real == other.real && imag == other.imag);
    }

    // Overload << operator (friend function)
    friend ostream& operator<<(ostream& out, const Complex& c) {
        out << c.real;
        if (c.imag >= 0)
            out << " + " << c.imag << "i";
        else
            out << " - " << -c.imag << "i";
        return out;
    }

    void display() {
        cout << *this << endl;
    }
};

int main() {
    Complex c1(3, 4);
    Complex c2(1, 2);

    cout << "c1 = " << c1 << endl;
    cout << "c2 = " << c2 << endl;

    Complex c3 = c1 + c2;
    cout << "c1 + c2 = " << c3 << endl;

    Complex c4 = c1 - c2;
    cout << "c1 - c2 = " << c4 << endl;

    Complex c5 = c1 * c2;
```

```
    cout << "c1 * c2 = " << c5 << endl;

    if (c1 == c2)
        cout << "c1 equals c2" << endl;
    else
        cout << "c1 not equals c2" << endl;

    return 0;
}
```

Common Operators to Overload:

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

class Vector2D {
private:
    double x, y;

public:
    Vector2D(double x = 0, double y = 0) : x(x), y(y) {}

    // Arithmetic operators
    Vector2D operator+(const Vector2D& v) {
        return Vector2D(x + v.x, y + v.y);
    }

    Vector2D operator-(const Vector2D& v) {
        return Vector2D(x - v.x, y - v.y);
    }

    Vector2D operator*(double scalar) {
        return Vector2D(x * scalar, y * scalar);
    }

    // Compound assignment operators
    Vector2D& operator+=(const Vector2D& v) {
        x += v.x;
        y += v.y;
```

```
    return *this;
}

// Unary operators
Vector2D operator-() {
    return Vector2D(-x, -y);
}

// Increment/Decrement
Vector2D& operator++() {  // Prefix
    ++x;
    ++y;
    return *this;
}

Vector2D operator++(int) {  // Postfix
    Vector2D temp = *this;
    ++(*this);
    return temp;
}

// Comparison operators
bool operator==(const Vector2D& v) {
    return (x == v.x && y == v.y);
}

bool operator!=(const Vector2D& v) {
    return !(*this == v);
}

// Subscript operator
double& operator[](int index) {
    if (index == 0) return x;
    return y;
}

// Stream operators
friend ostream& operator<<(ostream& out, const Vector2D& v) {
    out << "(" << v.x << ", " << v.y << ")";
    return out;
```

```
    }

    friend istream& operator>>(istream& in, Vector2D& v) {
        in >> v.x >> v.y;
        return in;
    }
};

int main() {
    Vector2D v1(3, 4);
    Vector2D v2(1, 2);

    cout << "v1 = " << v1 << endl;
    cout << "v2 = " << v2 << endl;

    Vector2D v3 = v1 + v2;
    cout << "v1 + v2 = " << v3 << endl;

    Vector2D v4 = v1 * 2;
    cout << "v1 * 2 = " << v4 << endl;

    v1 += v2;
    cout << "v1 after += v2: " << v1 << endl;

    Vector2D v5 = -v1;
    cout << "-v1 = " << v5 << endl;

    ++v2;
    cout << "++v2 = " << v2 << endl;

    cout << "v1[0] = " << v1[0] << ", v1[1] = " << v1[1] << endl;

    return 0;
}
```

2.3 Practical Example: Fraction Class

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

```
class Fraction {
private:
    int numerator;
    int denominator;

    // Helper function to find GCD
    int gcd(int a, int b) {
        if (b == 0) return a;
        return gcd(b, a % b);
    }

    // Simplify the fraction
    void simplify() {
        int g = gcd(abs(numerator), abs(denominator));
        numerator /= g;
        denominator /= g;

        // Keep denominator positive
        if (denominator < 0) {
            numerator = -numerator;
            denominator = -denominator;
        }
    }

public:
    Fraction(int num = 0, int den = 1) : numerator(num), denominator(den) {
        if (denominator == 0) {
            cout << "Error: Denominator cannot be zero!" << endl;
            denominator = 1;
        }
        simplify();
    }

    // Arithmetic operators
    Fraction operator+(const Fraction& f) {
        int num = numerator * f.denominator + f.numerator * denominator;
        int den = denominator * f.denominator;
        return Fraction(num, den);
    }
```

```
Fraction operator-(const Fraction& f) {
    int num = numerator * f.denominator - f.numerator * denominator;
    int den = denominator * f.denominator;
    return Fraction(num, den);
}

Fraction operator*(const Fraction& f) {
    return Fraction(numerator * f.numerator, denominator * f.denominator);
}

Fraction operator/(const Fraction& f) {
    return Fraction(numerator * f.denominator, denominator * f.numerator);
}

// Comparison operators
bool operator==(const Fraction& f) {
    return (numerator == f.numerator && denominator == f.denominator);
}

bool operator<(const Fraction& f) {
    return (numerator * f.denominator < f.numerator * denominator);
}

bool operator>(const Fraction& f) {
    return f < *this;
}

// Stream operators
friend ostream& operator<<(ostream& out, const Fraction& f) {
    if (f.denominator == 1)
        out << f.numerator;
    else
        out << f.numerator << "/" << f.denominator;
    return out;
}

friend istream& operator>>(istream& in, Fraction& f) {
    char slash;
    in >> f.numerator >> slash >> f.denominator;
```

```
        f.simplify();
        return in;
    }
};

int main() {
    Fraction f1(1, 2);  // 1/2
    Fraction f2(3, 4);  // 3/4

    cout << "f1 = " << f1 << endl;
    cout << "f2 = " << f2 << endl;

    Fraction sum = f1 + f2;
    cout << "f1 + f2 = " << sum << endl;

    Fraction diff = f1 - f2;
    cout << "f1 - f2 = " << diff << endl;

    Fraction prod = f1 * f2;
    cout << "f1 * f2 = " << prod << endl;

    Fraction quot = f1 / f2;
    cout << "f1 / f2 = " << quot << endl;

    if (f1 < f2)
        cout << f1 << " is less than " << f2 << endl;
    else
        cout << f1 << " is not less than " << f2 << endl;

    return 0;
}
```

2.4 Stream Operator (Advanced Theory)

The **stream insertion operator (`<<`) is typically overloaded as a friend function** because it **does not logically belong to the class** and **needs access to the class's private data**. Let's break down why.

2.4.1 Reason 1 — The left operand is not the class

When you write:

```
cout << obj;
```

The operator's **left-hand side** is `cout` (an `ostream` object), **not** your class.

So the operator signature must look like:

```
ostream& operator<<(ostream& os, const MyClass& obj);
```

This means:

- It **cannot** be a member of `MyClass` (because then `MyClass` would be the left operand).
- It **must** be a free-standing function.

But this free function still needs to access `obj`'s private data → so you typically declare it as a **friend**.

Example Without Friend

It becomes impossible to access private members:

```
class Point {
private:
    int x, y;   // private
};

ostream& operator<<(ostream& os, const Point& p) {
    os << p.x;   // ERROR — x is private
    return os;
}
```

Because `operator<<` is not a member function, it has **no access** to private members.

2.4.2 Reason 2 — Friend gives access to private data

To solve that, you declare it as a friend:

```
class Point {
private:
    int x, y;

public:
    Point(int x, int y) : x(x), y(y) {}
```

```
    friend ostream& operator<<(ostream& os, const Point& p);
};
```

Now the non-member function has access to private members.

2.4.3 Reason 3 — Makes syntax natural

Overloading as a friend function allows this natural C++ syntax:

```
cout << obj1 << obj2;
```

If it were a member function, you'd need to write something like:

```
obj << cout;   // awkward and reversed operands
```

This is not how streams are meant to be used.

2.4.4 Reason 4 — Works with chaining

Friend/global versions support chaining:

```
cout << a << b << c;
```

Which relies on:

```
return os;
```

So the next `<<` works.

2.4.5 Stream Operator Summary

Reason	Explanation
Left operand is `ostream`	So cannot be a member of your class
Needs access to private data	So it's declared as friend
Natural syntax	Allows `cout << obj` instead of reversed order
Supports chaining	Returning `ostream&` works smoothly

Revision #3
Created 2025-11-25 03:30:53 UTC by DS
Updated 2025-11-25 03:46:38 UTC by DS