5. Abstraction

5.1 What is Abstraction?

Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object. It focuses on **what** an object does rather than **how** it does it.

Real-World Analogy:

- When you drive a car, you use the steering wheel, pedals, and gear shift
- You don't need to know how the engine, transmission, or brake system work internally
- The car's interface (steering, pedals) is the **abstraction** of complex mechanisms

Abstraction vs Encapsulation:

Aspect	Encapsulation	Abstraction
Focus	Data hiding (how to hide)	Implementation hiding (what to show)
Achieved by	Access specifiers (private/public)	Abstract classes, interfaces
Purpose	Protect data	Simplify complexity
Level	Class level	Design level

5.2 Levels of Abstraction

Example: Coffee Machine

```
// High-level abstraction (user interface)
class CoffeeMachine {
public:
    void makeCoffee() {
        // User just presses button
        grindBeans();
        heatWater();
        brew();
        dispense();
    }

private:
```

```
    // Low-level implementation details (hidden)
    void grindBeans() { /* Complex grinding mechanism */ }
    void heatWater() { /* Temperature control system */ }
    void brew() { /* Pressure and timing control */ }
    void dispense() { /* Dispensing mechanism */ }
};
```

User's Perspective:

```
int main() {
    CoffeeMachine machine;
    machine.makeCoffee();  // Simple interface, complex implementation hidden
    return 0;
}
```

5.3 Implementing Abstraction in C++

Method 1: Using Regular Classes (Practical Abstraction)

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

class EmailService {
private:
    // Complex implementation details hidden
    string smtpServer;
    int port;
    string username;
    string password;

    void connectToServer() {
        cout << "Connecting to SMTP server..." << endl;
        // Complex network code
    }

    void authenticate() {
        cout << "Authenticating..." << endl;
        // Complex authentication logic
    }
```

```
    void encodeMessage(string message) {
        cout << "Encoding message..." << endl;
        // Complex encoding algorithm
    }

    void transmit(string to, string message) {
        cout << "Transmitting to " << to << "..." << endl;
        // Complex transmission protocol
    }

    void disconnectFromServer() {
        cout << "Disconnecting..." << endl;
        // Cleanup code
    }

public:
    EmailService(string server, string user, string pass)
        : smtpServer(server), port(587), username(user), password(pass) {}

    // Simple public interface - abstracts away complexity
    void sendEmail(string recipient, string subject, string body) {
        cout << "\n=== Sending Email ===" << endl;
        connectToServer();
        authenticate();
        encodeMessage(body);
        transmit(recipient, body);
        disconnectFromServer();
        cout << "Email sent successfully!" << endl;
    }
};

int main() {
    EmailService emailer("smtp.gmail.com", "user@example.com", "password");

    // User only needs to call one simple method
    emailer.sendEmail("friend@example.com",
                      "Hello",
                      "Just saying hi!");
```

```
    return 0;
}
```

Output:

```
=== Sending Email ===
Connecting to SMTP server...
Authenticating...
Encoding message...
Transmitting to friend@example.com...
Disconnecting...
Email sent successfully!
```

5.4 Abstract Classes and Pure Virtual Functions

Abstract Class: A class that cannot be instantiated and serves as a base for other classes.

Pure Virtual Function: A virtual function with no implementation, declared with `= 0`.

Syntax:

```
class AbstractClassName {
public:
    virtual void pureVirtualFunction() = 0;  // Pure virtual function
    virtual void anotherFunction() = 0;

    void regularFunction() {
        // Regular implementation
    }
};
```

Complete Example:

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

// Abstract base class - defines interface
class Shape {
protected:
```

```
    string color;

public:
    Shape(string c) : color(c) {}

    // Pure virtual functions - must be implemented by derived classes
    virtual double getArea() = 0;
    virtual double getPerimeter() = 0;
    virtual void displayInfo() = 0;

    // Regular function with implementation
    string getColor() { return color; }
    void setColor(string c) { color = c; }
};

// Concrete class 1: Circle
class Circle : public Shape {
private:
    double radius;

public:
    Circle(string c, double r) : Shape(c), radius(r) {}

    // Implementing abstract methods
    double getArea() override {
        return 3.14159 * radius * radius;
    }

    double getPerimeter() override {
        return 2 * 3.14159 * radius;
    }

    void displayInfo() override {
        cout << "Circle [Color: " << color << ", Radius: " << radius << "]" << endl;
        cout << "Area: " << getArea() << endl;
        cout << "Perimeter: " << getPerimeter() << endl;
    }
};
```

```
// Concrete class 2: Rectangle
class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(string c, double w, double h)
        : Shape(c), width(w), height(h) {}

    double getArea() override {
        return width * height;
    }

    double getPerimeter() override {
        return 2 * (width + height);
    }

    void displayInfo() override {
        cout << "Rectangle [Color: " << color
             << ", Width: " << width << ", Height: " << height << "]" << endl;
        cout << "Area: " << getArea() << endl;
        cout << "Perimeter: " << getPerimeter() << endl;
    }
};

int main() {
    // Shape shape("red");  // ERROR: Cannot instantiate abstract class

    Circle circle("Red", 5.0);
    Rectangle rect("Blue", 4.0, 6.0);

    circle.displayInfo();
    cout << endl;
    rect.displayInfo();

    // Polymorphic behavior
    Shape* shapes[2];
    shapes[0] = &circle;
    shapes[1] = &rect;
```

```
    cout << "\n=== Using Polymorphism ===" << endl;
    for (int i = 0; i < 2; i++) {
        shapes[i]->displayInfo();
        cout << endl;
    }

    return 0;
}
```

5.5 Interface-Like Classes in C++

Pure Abstract Class (Interface):

```
// Interface - all methods are pure virtual
class Drawable {
public:
    virtual void draw() = 0;
    virtual void erase() = 0;
    virtual ~Drawable() {}  // Virtual destructor
};

class Movable {
public:
    virtual void moveUp() = 0;
    virtual void moveDown() = 0;
    virtual void moveLeft() = 0;
    virtual void moveRight() = 0;
    virtual ~Movable() {}
};

// Class implementing multiple interfaces
class GameCharacter : public Drawable, public Movable {
private:
    int x, y;
    string name;

public:
    GameCharacter(string n, int posX, int posY)
```

```
        : name(n), x(posX), y(posY) {}

    // Implement Drawable interface
    void draw() override {
        cout << "Drawing " << name << " at (" << x << ", " << y << ")" << endl;
    }

    void erase() override {
        cout << "Erasing " << name << endl;
    }

    // Implement Movable interface
    void moveUp() override { y++; }
    void moveDown() override { y--; }
    void moveLeft() override { x--; }
    void moveRight() override { x++; }
};

int main() {
    GameCharacter hero("Hero", 10, 20);

    hero.draw();
    hero.moveRight();
    hero.moveUp();
    hero.draw();

    return 0;
}
```

5.6 Benefits of Abstraction

1. Simplification:

```
// User doesn't need to know implementation details
DatabaseConnection db("localhost", "mydb", "user", "pass");
db.connect();
db.executeQuery("SELECT * FROM users");
db.close();
```

2. Flexibility:

```
class PaymentProcessor {
public:
    virtual void processPayment(double amount) = 0;
};

class CreditCardProcessor : public PaymentProcessor {
    void processPayment(double amount) override {
        // Credit card specific implementation
    }
};

class PayPalProcessor : public PaymentProcessor {
    void processPayment(double amount) override {
        // PayPal specific implementation
    }
};
```

3. Maintainability:

- Change implementation without affecting user code
- Add new implementations without modifying existing code

Revision #1
Created 2025-11-09 13:17:14 UTC by DS
Updated 2025-11-09 13:17:53 UTC by DS