Module 8 : OOP (SOLID, Encapsulation, Abstraction)

By the end of this module, students will be able to:

- Understand the fundamental concepts of Object-Oriented Programming (OOP)

- Transition from procedural C programming to OOP in C++

- Implement classes and objects in C++

- Apply encapsulation principles using access specifiers

- Understand and implement abstraction concepts

- Apply SOLID principles in basic C++ programs

1. Introduction: From Procedural to Object-Oriented Programming

1.1 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around **objects** rather than functions and logic. An object is a data structure that contains both **data** (attributes) and **code** (methods) that operates on that data.

Key Paradigm Comparison:

Aspect	Procedural (C)	Object-Oriented (C++)
Focus	Functions and procedures	Objects and classes
Data & Functions	Separate	Bundled together
Code Organization	By functionality	By entities/objects
Data Protection	Limited (global/local)	Strong (access specifiers)
Code Reuse	Function reuse	Inheritance & polymorphism
Maintenance	Harder for large projects	Easier through modularity

1.2 The Four Pillars of OOP

1. **Encapsulation**: Bundling data and methods that operate on that data within a single unit (class), hiding internal details
2. **Abstraction**: Showing only essential features while hiding implementation details
3. **Inheritance**: Creating new classes from existing classes, promoting code reuse
4. **Polymorphism**: Ability of objects to take many forms, allowing different implementations of the same interface

1.3 Real-World Analogy

Think of a **car**:

- **Object**: Your specific car (e.g., a red Toyota Camry 2020)
- **Class**: The blueprint/design for all Toyota Camry cars
- **Attributes** (data): color, model, year, speed, fuel level
- **Methods** (functions): start(), accelerate(), brake(), turn()

- **Encapsulation**: You don't need to know how the engine works internally; you just use the steering wheel and pedals
- **Abstraction**: The dashboard shows you speed and fuel, hiding complex engine computations

2. C++ Basics: Essential Differences from C

2.1 Basic Syntax Differences

C vs C++ Comparison:

Feature	C	C++
File Extension	`.c`	`.cpp`
Input/Output	`scanf()`, `printf()`	`cin`, `cout`
Header Files	`#include <stdio.h>`	`#include <iostream>`
Namespace	Not used	`using namespace std;`
Comments	`/* */` and `//`	`/* */` and `//` (preferred)
Boolean Type	`int` (0/1)	`bool` (true/false)
String Type	`char[]`	`string` class

2.2 Hello World Comparison

C Program:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

C++ Program:

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

int main() {
    cout << "Hello, World!" << endl;
```

```
    return 0;
}
```

2.3 Input/Output in C++

Basic I/O Operations:

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

int main() {
    int age;
    string name;
    float height;

    // Output (cout with insertion operator <<)
    cout << "Enter your name: ";

    // Input (cin with extraction operator >>)
    cin >> name;

    cout << "Enter your age: ";
    cin >> age;

    cout << "Enter your height (m): ";
    cin >> height;

    // Multiple outputs
    cout << "Name: " << name << endl;
    cout << "Age: " << age << " years" << endl;
    cout << "Height: " << height << " meters" << endl;

    return 0;
}
```

C vs C++ I/O Comparison:

Operation	C	C++
Output integer	`printf("%d", x);`	`cout << x;`

Operation	C	C++
Output float	`printf("%.2f", x);`	`cout << fixed << setprecision(2) << x;`
Output string	`printf("%s", str);`	`cout << str;`
Input integer	`scanf("%d", &x);`	`cin >> x;`
Input string	`scanf("%s", str);`	`cin >> str;`
Multiple inputs	`scanf("%d %d", &a, &b);`	`cin >> a >> b;`

Reading a Full Line:

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

int main() {
    string fullName;

    cout << "Enter your full name: ";
    getline(cin, fullName);  // Reads entire line including spaces

    cout << "Hello, " << fullName << "!" << endl;
    return 0;
}
```

Important Note on cin Buffer:

```
int age;
string name;

cin >> age;          // User enters "25" and presses Enter
// Buffer now contains the newline character

cin.ignore();        // Clear the newline from buffer
getline(cin, name);  // Now can read full line properly
```

3. Classes and Objects

3.1 Understanding Classes and Objects

Definition:

- **Class**: A blueprint or template for creating objects (like a cookie cutter)
- **Object**: An instance of a class (like a cookie made from the cutter)

Real-World Example:

```
Class: Student (blueprint)
    - Attributes: name, ID, GPA
    - Methods: study(), takeExam(), getGPA()

Objects (instances):
    - student1: "Alice", "S001", 3.8
    - student2: "Bob", "S002", 3.5
    - student3: "Charlie", "S003", 3.9
```

3.2 Defining a Class

Basic Class Syntax:

```
class ClassName {
private:
    // Private members (data and functions)
    // Only accessible within the class

public:
    // Public members
    // Accessible from outside the class

protected:
    // Protected members
    // Accessible in derived classes (inheritance)
};
```

Simple Example:

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

class Student {
private:
    string name;
    int id;
    float gpa;

public:
    // Constructor
    Student(string n, int i, float g) {
        name = n;
        id = i;
        gpa = g;
    }

    // Member functions (methods)
    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
        cout << "GPA: " << gpa << endl;
    }

    void study() {
        cout << name << " is studying..." << endl;
    }

    float getGPA() {
        return gpa;
    }

    void setGPA(float newGPA) {
        if (newGPA >= 0.0 && newGPA <= 4.0) {
            gpa = newGPA;
        } else {
            cout << "Invalid GPA!" << endl;
```

```
        }
    }
};

int main() {
    // Creating objects
    Student student1("Alice", 1001, 3.8);
    Student student2("Bob", 1002, 3.5);

    // Using objects
    student1.displayInfo();
    cout << endl;

    student2.study();
    cout << "Bob's GPA: " << student2.getGPA() << endl;

    student2.setGPA(3.7);
    cout << "Updated GPA: " << student2.getGPA() << endl;

    return 0;
}
```

3.3 Procedural vs OOP: A Practical Comparison

Procedural Approach (C):

```
#include <stdio.h>
#include <string.h>

// Separate data structure
struct Student {
    char name[50];
    int id;
    float gpa;
};

// Separate functions
void displayStudent(struct Student s) {
    printf("Name: %s\n", s.name);
    printf("ID: %d\n", s.id);
```

```
    printf("GPA: %.2f\n", s.gpa);
}

void studyStudent(struct Student s) {
    printf("%s is studying...\n", s.name);
}

float getGPA(struct Student s) {
    return s.gpa;
}

int main() {
    struct Student student1;
    strcpy(student1.name, "Alice");
    student1.id = 1001;
    student1.gpa = 3.8;

    displayStudent(student1);
    studyStudent(student1);

    return 0;
}
```

OOP Approach (C++):

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

class Student {
private:
    string name;
    int id;
    float gpa;

public:
    Student(string n, int i, float g) : name(n), id(i), gpa(g) {}

    void display() {
        cout << "Name: " << name << endl;
```

```
        cout << "ID: " << id << endl;
        cout << "GPA: " << gpa << endl;
    }

    void study() {
        cout << name << " is studying..." << endl;
    }

    float getGPA() { return gpa; }
};

int main() {
    Student student1("Alice", 1001, 3.8);

    student1.display();
    student1.study();

    return 0;
}
```

Key Advantages of OOP:

1. **Encapsulation**: Data and functions are bundled together
2. **Data Protection**: Private members prevent unauthorized access
3. **Cleaner Syntax**: Methods are called directly on objects
4. **Better Organization**: Related functionality is grouped together

4. Encapsulation

4.1 What is Encapsulation?

Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit (class), while **restricting direct access** to some of the object's components.

Purpose:

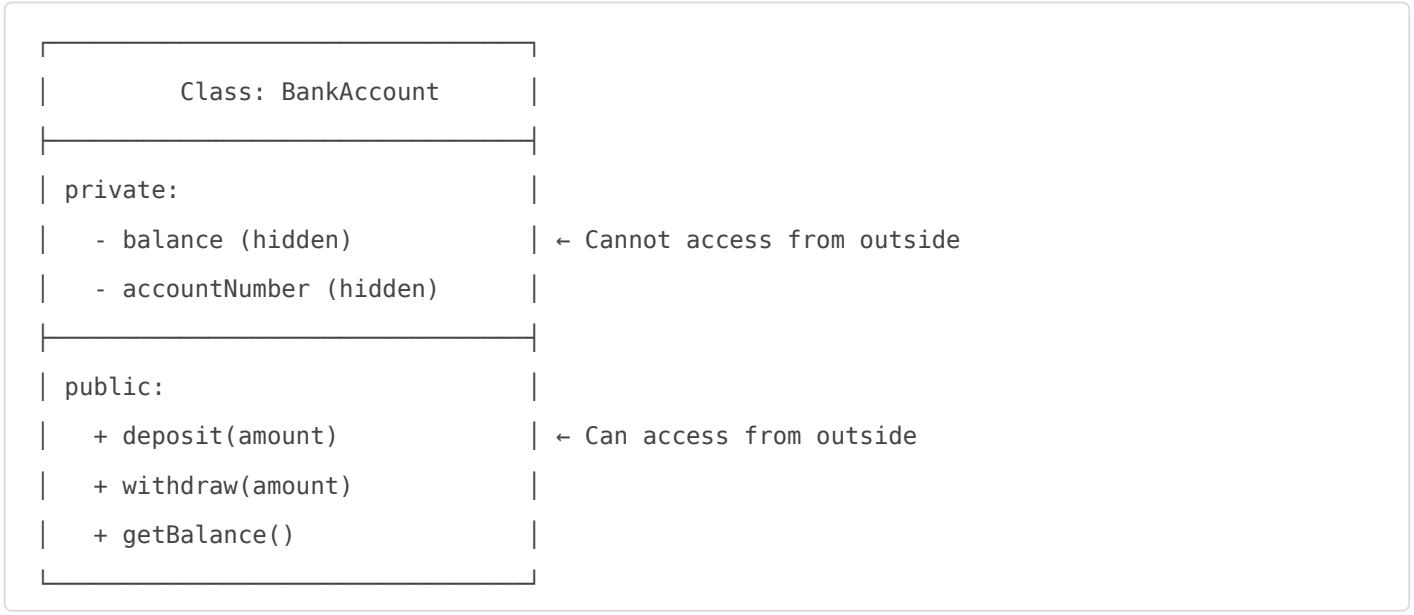
- **Data Hiding**: Protect internal state from unauthorized access
- **Controlled Access**: Provide public methods to access/modify private data
- **Flexibility**: Change internal implementation without affecting external code
- **Validation**: Enforce rules when setting data

4.2 Access Specifiers

Three Access Levels:

Specifier	Access Within Class	Access in Derived Class	Access from Outside
`private`	✓	✗	✗
`protected`	✓	✓	✗
`public`	✓	✓	✓

Visual Representation:



4.3 Implementing Encapsulation

Bad Practice (No Encapsulation):

```
class BankAccount {
public:
    string accountNumber;
    double balance;  // Anyone can modify this directly!
};

int main() {
    BankAccount acc;
    acc.balance = 1000.0;

    // Problem: Direct modification allows invalid states
    acc.balance = -500.0;  // Negative balance! Bad!
    acc.balance = 999999999.99;  // Unrealistic amount

    return 0;
}
```

Good Practice (With Encapsulation):

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

class BankAccount {
private:
    string accountNumber;
    double balance;
    string ownerName;

public:
    // Constructor
    BankAccount(string accNum, string owner, double initialBalance = 0.0) {
        accountNumber = accNum;
        ownerName = owner;
        balance = (initialBalance >= 0) ? initialBalance : 0.0;
    }
```

```
// Getter methods (accessors)
double getBalance() const {
    return balance;
}

string getAccountNumber() const {
    return accountNumber;
}

string getOwnerName() const {
    return ownerName;
}

// Setter methods with validation (mutators)
bool deposit(double amount) {
    if (amount > 0) {
        balance += amount;
        cout << "Deposited: $" << amount << endl;
        return true;
    }
    cout << "Invalid deposit amount!" << endl;
    return false;
}

bool withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        cout << "Withdrawn: $" << amount << endl;
        return true;
    }
    cout << "Invalid withdrawal or insufficient funds!" << endl;
    return false;
}

void displayInfo() const {
    cout << "Account: " << accountNumber << endl;
    cout << "Owner: " << ownerName << endl;
    cout << "Balance: $" << balance << endl;
}
```

```
};

int main() {
    BankAccount acc("ACC001", "Alice Johnson", 1000.0);

    acc.displayInfo();
    cout << endl;

    acc.deposit(500.0);
    acc.withdraw(200.0);
    acc.withdraw(2000.0);  // Will fail - insufficient funds

    cout << "\nFinal balance: $" << acc.getBalance() << endl;

    // acc.balance = -500;  // ERROR: Cannot access private member

    return 0;
}
```

Output:

```
Account: ACC001
Owner: Alice Johnson
Balance: $1000

Deposited: $500
Withdrawn: $200
Invalid withdrawal or insufficient funds!

Final balance: $1300
```

4.4 Benefits of Encapsulation

1. Data Validation:

```
class Temperature {
private:
    double celsius;

public:
```

```
    void setCelsius(double temp) {
        if (temp >= -273.15) {  // Absolute zero
            celsius = temp;
        } else {
            cout << "Invalid temperature!" << endl;
        }
    }

    double getCelsius() const { return celsius; }
    double getFahrenheit() const { return (celsius * 9.0/5.0) + 32; }
};
```

2. Read-Only Properties:

```
class Person {
private:
    string ssn;  // Social Security Number

public:
    Person(string socialSecNum) : ssn(socialSecNum) {}

    // Only getter, no setter - SSN is read-only
    string getSSN() const { return ssn; }
};
```

3. Internal Implementation Changes:

```
class Circle {
private:
    double radius;
    // We could change to store diameter instead later

public:
    void setRadius(double r) {
        if (r > 0) radius = r;
    }

    double getArea() const {
        return 3.14159 * radius * radius;
    }
```

```
    // External code doesn't need to change if we modify internal storage
};
```

4.5 Const Member Functions

Purpose: Indicate that a method does not modify object state

```
class Rectangle {
private:
    double width, height;

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

    // Const member functions - promise not to modify data
    double getWidth() const { return width; }
    double getHeight() const { return height; }
    double getArea() const { return width * height; }
    double getPerimeter() const { return 2 * (width + height); }

    // Non-const member functions - can modify data
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

int main() {
    const Rectangle rect(5.0, 3.0);  // Const object

    cout << rect.getArea();      // OK - const function
    // rect.setWidth(10);        // ERROR - can't call non-const function on const object

    return 0;
}
```

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

6. SOLID Principles

6.1 Introduction to SOLID

SOLID is an acronym for five design principles that make software designs more understandable, flexible, and maintainable.

Letter	Principle	Core Idea
S	Single Responsibility	A class should have one reason to change
O	Open/Closed	Open for extension, closed for modification
L	Liskov Substitution	Derived classes must be substitutable for base classes
I	Interface Segregation	Many specific interfaces are better than one general interface
D	Dependency Inversion	Depend on abstractions, not concretions

6.2 S - Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

Bad Example (Multiple Responsibilities):

```
// This class has too many responsibilities!
class Employee {
private:
    string name;
    double salary;

public:
    // Responsibility 1: Employee data management
    void setName(string n) { name = n; }
    string getName() { return name; }
    void setSalary(double s) { salary = s; }
    double getSalary() { return salary; }
```

```
    // Responsibility 2: Salary calculation
    double calculateTax() {
        return salary * 0.2;
    }

    double calculateBonus() {
        return salary * 0.1;
    }

    // Responsibility 3: Database operations
    void saveToDatabase() {
        cout << "Saving employee to database..." << endl;
    }

    void loadFromDatabase() {
        cout << "Loading employee from database..." << endl;
    }

    // Responsibility 4: Report generation
    void printPaySlip() {
        cout << "Printing pay slip..." << endl;
    }
};
```

Good Example (Single Responsibility):

```
// Each class has ONE clear responsibility

// 1. Employee data management
class Employee {
private:
    string name;
    string id;
    double salary;

public:
    Employee(string n, string i, double s)
        : name(n), id(i), salary(s) {}
```

```
    string getName() const { return name; }
    string getId() const { return id; }
    double getSalary() const { return salary; }
    void setSalary(double s) { salary = s; }
};

// 2. Salary calculations
class SalaryCalculator {
public:
    double calculateTax(const Employee& emp) {
        return emp.getSalary() * 0.2;
    }

    double calculateBonus(const Employee& emp) {
        return emp.getSalary() * 0.1;
    }

    double calculateNetSalary(const Employee& emp) {
        return emp.getSalary() - calculateTax(emp) + calculateBonus(emp);
    }
};

// 3. Database operations
class EmployeeRepository {
public:
    void save(const Employee& emp) {
        cout << "Saving employee " << emp.getId() << " to database..." << endl;
    }

    Employee* load(string id) {
        cout << "Loading employee " << id << " from database..." << endl;
        return nullptr;  // Simplified
    }
};

// 4. Report generation
class PaySlipGenerator {
private:
    SalaryCalculator calculator;
```

```
public:
    void generatePaySlip(const Employee& emp) {
        cout << "\n===== PAY SLIP =====" << endl;
        cout << "Employee: " << emp.getName() << endl;
        cout << "ID: " << emp.getId() << endl;
        cout << "Gross Salary: $" << emp.getSalary() << endl;
        cout << "Tax: $" << calculator.calculateTax(emp) << endl;
        cout << "Bonus: $" << calculator.calculateBonus(emp) << endl;
        cout << "Net Salary: $" << calculator.calculateNetSalary(emp) << endl;
        cout << "====================" << endl;
    }
};

int main() {
    Employee emp("Alice Johnson", "E001", 5000.0);

    SalaryCalculator calc;
    EmployeeRepository repo;
    PaySlipGenerator payslip;

    payslip.generatePaySlip(emp);
    repo.save(emp);

    return 0;
}
```

Benefits:

- Easier to understand and maintain
- Changes in one responsibility don't affect others
- Easier to test individual components
- Better code organization

6.3 O - Open/Closed Principle (OCP)

Definition: Software entities should be **open for extension** but **closed for modification**. You should be able to add new functionality without changing existing code.

Bad Example (Violates OCP):

```
class Rectangle {
public:
    double width, height;
};

class Circle {
public:
    double radius;
};

// This class needs modification every time we add a new shape
class AreaCalculator {
public:
    double calculateArea(void* shape, string shapeType) {
        if (shapeType == "Rectangle") {
            Rectangle* rect = (Rectangle*)shape;
            return rect->width * rect->height;
        }
        else if (shapeType == "Circle") {
            Circle* circle = (Circle*)shape;
            return 3.14159 * circle->radius * circle->radius;
        }
        // Need to modify this function for every new shape!
        // else if (shapeType == "Triangle") { ... }
        return 0;
    }
};
```

Good Example (Follows OCP):

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

// Abstract base class
class Shape {
public:
    virtual double calculateArea() = 0;
    virtual string getName() = 0;
```

```
    virtual ~Shape() {}
};

// Concrete shapes - extending without modifying existing code
class Rectangle : public Shape {
private:
    double width, height;

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

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

    string getName() override {
        return "Rectangle";
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double calculateArea() override {
        return 3.14159 * radius * radius;
    }

    string getName() override {
        return "Circle";
    }
};

// NEW shape - no modification to existing code needed!
class Triangle : public Shape {
private:
    double base, height;
```

```
public:
    Triangle(double b, double h) : base(b), height(h) {}

    double calculateArea() override {
        return 0.5 * base * height;
    }

    string getName() override {
        return "Triangle";
    }
};

// This class doesn't need modification when adding new shapes
class AreaCalculator {
public:
    void printArea(Shape* shape) {
        cout << shape->getName() << " area: "
             << shape->calculateArea() << endl;
    }

    double getTotalArea(vector<Shape*> shapes) {
        double total = 0;
        for (Shape* shape : shapes) {
            total += shape->calculateArea();
        }
        return total;
    }
};

int main() {
    Rectangle rect(5, 3);
    Circle circle(4);
    Triangle triangle(6, 8);

    AreaCalculator calculator;

    calculator.printArea(&rect);
    calculator.printArea(&circle);
    calculator.printArea(&triangle);
```

```
    vector<Shape*> shapes = {&rect, &circle, &triangle};
    cout << "Total area: " << calculator.getTotalArea(shapes) << endl;

    return 0;
}
```

Benefits:

- Add new features without breaking existing code
- Reduces risk of introducing bugs
- Promotes code reuse through inheritance
- Makes the system more maintainable

6.4 L - Liskov Substitution Principle (LSP)

Definition: Objects of a derived class should be able to replace objects of the base class without breaking the application. In other words, derived classes must be substitutable for their base classes.

Bad Example (Violates LSP):

```
class Bird {
public:
    virtual void fly() {
        cout << "Flying..." << endl;
    }
};

class Sparrow : public Bird {
public:
    void fly() override {
        cout << "Sparrow flying..." << endl;
    }
};

// Ostrich is a bird but can't fly!
class Ostrich : public Bird {
public:
    void fly() override {
        throw runtime_error("Ostrich can't fly!");
```

```
        // This breaks LSP - can't substitute Ostrich for Bird
    }
};

void makeBirdFly(Bird* bird) {
    bird->fly();  // Will crash if bird is an Ostrich!
}
```

Good Example (Follows LSP):

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

// Better abstraction
class Bird {
protected:
    string name;

public:
    Bird(string n) : name(n) {}

    virtual void eat() {
        cout << name << " is eating..." << endl;
    }

    virtual void makeSound() = 0;

    string getName() { return name; }
};

// Separate interface for flying ability
class FlyingBird : public Bird {
public:
    FlyingBird(string n) : Bird(n) {}

    virtual void fly() {
        cout << name << " is flying..." << endl;
    }
};
```

```
// Flying birds
class Sparrow : public FlyingBird {
public:
    Sparrow() : FlyingBird("Sparrow") {}

    void makeSound() override {
        cout << "Chirp chirp!" << endl;
    }
};

class Eagle : public FlyingBird {
public:
    Eagle() : FlyingBird("Eagle") {}

    void makeSound() override {
        cout << "Screech!" << endl;
    }
};

// Non-flying bird - doesn't inherit fly()
class Ostrich : public Bird {
public:
    Ostrich() : Bird("Ostrich") {}

    void makeSound() override {
        cout << "Boom boom!" << endl;
    }

    void run() {
        cout << name << " is running fast..." << endl;
    }
};

class Penguin : public Bird {
public:
    Penguin() : Bird("Penguin") {}

    void makeSound() override {
        cout << "Honk honk!" << endl;
```

```
    }

    void swim() {
        cout << name << " is swimming..." << endl;
    }
};

int main() {
    Sparrow sparrow;
    Eagle eagle;
    Ostrich ostrich;
    Penguin penguin;

    // All birds can make sounds and eat
    Bird* birds[] = {&sparrow, &eagle, &ostrich, &penguin};

    cout << "=== All birds can do these ===" << endl;
    for (Bird* bird : birds) {
        bird->makeSound();
        bird->eat();
        cout << endl;
    }

    // Only flying birds can fly
    cout << "=== Only flying birds ===" << endl;
    FlyingBird* flyingBirds[] = {&sparrow, &eagle};

    for (FlyingBird* bird : flyingBirds) {
        bird->fly();
    }

    // Specialized behaviors
    cout << "\n=== Specialized behaviors ===" << endl;
    ostrich.run();
    penguin.swim();

    return 0;
}
```

Real-World Example:

```
// Bad: Square inheriting from Rectangle violates LSP
class Rectangle {
protected:
    double width, height;

public:
    virtual void setWidth(double w) { width = w; }
    virtual void setHeight(double h) { height = h; }
    double getArea() { return width * height; }
};

class Square : public Rectangle {
public:
    void setWidth(double w) override {
        width = height = w;  // Problem: setting width also sets height!
    }

    void setHeight(double h) override {
        width = height = h;  // Problem: setting height also sets width!
    }
};

// This function expects Rectangle behavior
void processRectangle(Rectangle* rect) {
    rect->setWidth(5);
    rect->setHeight(4);
    // Expected area: 20
    // But if rect is a Square, area will be 16!
    cout << "Area: " << rect->getArea() << endl;
}
```

Better Design:

```
class Shape {
public:
    virtual double getArea() = 0;
    virtual ~Shape() {}
};

class Rectangle : public Shape {
```

```
private:
    double width, height;

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

    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

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

class Square : public Shape {
private:
    double side;

public:
    Square(double s) : side(s) {}

    void setSide(double s) { side = s; }

    double getArea() override {
        return side * side;
    }
};
```

Benefits:

- Ensures polymorphism works correctly
- Prevents unexpected behavior in derived classes
- Makes code more reliable and predictable

6.5 I - Interface Segregation Principle (ISP)

Definition: No client should be forced to depend on methods it does not use. It's better to have many specific interfaces than one general-purpose interface.

Bad Example (Violates ISP):

```
// Fat interface - forces classes to implement methods they don't need
class Worker {
public:
    virtual void work() = 0;
    virtual void eat() = 0;
    virtual void sleep() = 0;
    virtual void attendMeeting() = 0;
    virtual void writeCode() = 0;
    virtual void designUI() = 0;
};

// Robot worker doesn't eat or sleep!
class RobotWorker : public Worker {
public:
    void work() override {
        cout << "Robot working..." << endl;
    }

    void eat() override {
        // Robots don't eat! But forced to implement
        throw runtime_error("Robots don't eat!");
    }

    void sleep() override {
        // Robots don't sleep! But forced to implement
        throw runtime_error("Robots don't sleep!");
    }

    void attendMeeting() override {
        throw runtime_error("Robots don't attend meetings!");
    }

    void writeCode() override {
        cout << "Robot writing code..." << endl;
    }

    void designUI() override {
        throw runtime_error("Robots don't design UI!");
    }
};
```

```
// Manager doesn't write code!
class Manager : public Worker {
public:
    void work() override {
        cout << "Manager working..." << endl;
    }

    void eat() override {
        cout << "Manager eating..." << endl;
    }

    void sleep() override {
        cout << "Manager sleeping..." << endl;
    }

    void attendMeeting() override {
        cout << "Manager attending meeting..." << endl;
    }

    void writeCode() override {
        // Managers don't write code! But forced to implement
        throw runtime_error("Managers don't write code!");
    }

    void designUI() override {
        throw runtime_error("Managers don't design UI!");
    }
};
```

Good Example (Follows ISP):

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

// Segregated interfaces - small, specific interfaces

interface Workable {
public:
```

```
    virtual void work() = 0;
    virtual ~Workable() {}
};

class Eatable {
public:
    virtual void eat() = 0;
    virtual ~Eatable() {}
};

class Sleepable {
public:
    virtual void sleep() = 0;
    virtual ~Sleepable() {}
};

class Codable {
public:
    virtual void writeCode() = 0;
    virtual ~Codable() {}
};

class Designable {
public:
    virtual void designUI() = 0;
    virtual ~Designable() {}
};

class Manageable {
public:
    virtual void attendMeeting() = 0;
    virtual void delegateTasks() = 0;
    virtual ~Manageable() {}
};

// Now classes only implement interfaces they need

class Developer : public Workable, public Eatable, public Sleepable, public Codable {
private:
    string name;
```

```
public:
    Developer(string n) : name(n) {}

    void work() override {
        cout << name << " (Developer) is working..." << endl;
    }

    void eat() override {
        cout << name << " is eating..." << endl;
    }

    void sleep() override {
        cout << name << " is sleeping..." << endl;
    }

    void writeCode() override {
        cout << name << " is writing code..." << endl;
    }
};

class Designer : public Workable, public Eatable, public Sleepable, public Designable {
private:
    string name;

public:
    Designer(string n) : name(n) {}

    void work() override {
        cout << name << " (Designer) is working..." << endl;
    }

    void eat() override {
        cout << name << " is eating..." << endl;
    }

    void sleep() override {
        cout << name << " is sleeping..." << endl;
    }
```

```
    void designUI() override {
        cout << name << " is designing UI..." << endl;
    }
};

class Manager : public Workable, public Eatable, public Sleepable, public Manageable {
private:
    string name;

public:
    Manager(string n) : name(n) {}

    void work() override {
        cout << name << " (Manager) is working..." << endl;
    }

    void eat() override {
        cout << name << " is eating..." << endl;
    }

    void sleep() override {
        cout << name << " is sleeping..." << endl;
    }

    void attendMeeting() override {
        cout << name << " is attending meeting..." << endl;
    }

    void delegateTasks() override {
        cout << name << " is delegating tasks..." << endl;
    }
};

class RobotWorker : public Workable, public Codable {
private:
    string model;

public:
    RobotWorker(string m) : model(m) {}
```

```
    void work() override {
        cout << model << " (Robot) is working 24/7..." << endl;
    }

    void writeCode() override {
        cout << model << " is writing code..." << endl;
    }
    // No eat() or sleep() - robots don't need these!
};

int main() {
    Developer dev("Alice");
    Designer des("Bob");
    Manager mgr("Carol");
    RobotWorker robot("R2D2");

    cout << "=== Developer ===" << endl;
    dev.work();
    dev.writeCode();
    dev.eat();

    cout << "\n=== Designer ===" << endl;
    des.work();
    des.designUI();
    des.sleep();

    cout << "\n=== Manager ===" << endl;
    mgr.work();
    mgr.attendMeeting();
    mgr.delegateTasks();

    cout << "\n=== Robot ===" << endl;
    robot.work();
    robot.writeCode();
    // robot.eat();  // Doesn't exist - good!

    return 0;
}
```

Benefits:

- Classes only depend on methods they actually use
- More flexible and maintainable code
- Easier to understand class responsibilities
- Reduces coupling between classes

6.6 D - Dependency Inversion Principle (DIP)

Definition:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Bad Example (Violates DIP):

```
// Low-level modules (concrete implementations)
class MySQLDatabase {
public:
    void connect() {
        cout << "Connecting to MySQL..." << endl;
    }

    void executeQuery(string query) {
        cout << "Executing MySQL query: " << query << endl;
    }
};

// High-level module directly depends on low-level module
class UserService {
private:
    MySQLDatabase database;  // Direct dependency on concrete class!

public:
    void getUser(int id) {
        database.connect();
        database.executeQuery("SELECT * FROM users WHERE id = " + to_string(id));
    }

    // If we want to switch to PostgreSQL, we must modify this class!
};
```

Good Example (Follows DIP):

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

// Abstraction (high-level interface)
class Database {
public:
    virtual void connect() = 0;
    virtual void executeQuery(string query) = 0;
    virtual ~Database() {}
};

// Low-level modules implement the abstraction
class MySQLDatabase : public Database {
public:
    void connect() override {
        cout << "Connecting to MySQL database..." << endl;
    }

    void executeQuery(string query) override {
        cout << "MySQL executing: " << query << endl;
    }
};

class PostgreSQLDatabase : public Database {
public:
    void connect() override {
        cout << "Connecting to PostgreSQL database..." << endl;
    }

    void executeQuery(string query) override {
        cout << "PostgreSQL executing: " << query << endl;
    }
};

class MongoDatabase : public Database {
public:
    void connect() override {
        cout << "Connecting to MongoDB..." << endl;
```

```
    }

    void executeQuery(string query) override {
        cout << "MongoDB executing: " << query << endl;
    }
};

// High-level module depends on abstraction, not concrete class
class UserService {
private:
    Database* database;  // Depends on abstraction!

public:
    // Dependency injection through constructor
    UserService(Database* db) : database(db) {}

    void getUser(int id) {
        database->connect();
        database->executeQuery("SELECT * FROM users WHERE id = " + to_string(id));
    }

    void saveUser(string name, string email) {
        database->connect();
        database->executeQuery("INSERT INTO users (name, email) VALUES ('" +
                               name + "', '" + email + "')");
    }

    // Can easily switch database implementation!
    void setDatabase(Database* db) {
        database = db;
    }
};

int main() {
    // Create different database implementations
    MySQLDatabase mysql;
    PostgreSQLDatabase postgres;
    MongoDatabase mongo;

    // Inject dependency (database) into high-level module
```

```
    cout << "=== Using MySQL ===" << endl;
    UserService userService1(&mysql);
    userService1.getUser(1);
    userService1.saveUser("Alice", "alice@example.com");

    cout << "\n=== Switching to PostgreSQL ===" << endl;
    UserService userService2(&postgres);
    userService2.getUser(2);

    cout << "\n=== Switching to MongoDB ===" << endl;
    UserService userService3(&mongo);
    userService3.getUser(3);

    return 0;
}
```

Another Example: Payment Processing

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

// Abstraction
class PaymentProcessor {
public:
    virtual bool processPayment(double amount) = 0;
    virtual string getProcessorName() = 0;
    virtual ~PaymentProcessor() {}
};

// Concrete implementations
class CreditCardProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        cout << "Processing $" << amount << " via Credit Card..." << endl;
        return true;
    }

    string getProcessorName() override {
        return "Credit Card";
```

```
    }
};

class PayPalProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        cout << "Processing $" << amount << " via PayPal..." << endl;
        return true;
    }

    string getProcessorName() override {
        return "PayPal";
    }
};

class BitcoinProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        cout << "Processing $" << amount << " via Bitcoin..." << endl;
        return true;
    }

    string getProcessorName() override {
        return "Bitcoin";
    }
};

// High-level module depends on abstraction
class ShoppingCart {
private:
    double total;
    PaymentProcessor* paymentProcessor;

public:
    ShoppingCart() : total(0), paymentProcessor(nullptr) {}

    void addItem(double price) {
        total += price;
        cout << "Item added. Current total: $" << total << endl;
    }
```

```
    void setPaymentMethod(PaymentProcessor* processor) {
        paymentProcessor = processor;
        cout << "Payment method set to: " << processor->getProcessorName() << endl;
    }

    void checkout() {
        if (paymentProcessor == nullptr) {
            cout << "Error: No payment method selected!" << endl;
            return;
        }

        cout << "\n=== Checkout ===" << endl;
        cout << "Total amount: $" << total << endl;

        if (paymentProcessor->processPayment(total)) {
            cout << "Payment successful!" << endl;
            total = 0;
        } else {
            cout << "Payment failed!" << endl;
        }
    }
};

int main() {
    CreditCardProcessor creditCard;
    PayPalProcessor paypal;
    BitcoinProcessor bitcoin;

    ShoppingCart cart;

    cart.addItem(29.99);
    cart.addItem(49.99);
    cart.addItem(15.50);

    cout << "\n--- Paying with Credit Card ---" << endl;
    cart.setPaymentMethod(&creditCard);
    cart.checkout();

    cout << "\n--- New purchase ---" << endl;
```

```
    cart.addItem(99.99);
    cout << "\n--- Paying with PayPal ---" << endl;
    cart.setPaymentMethod(&paypal);
    cart.checkout();

    cout << "\n--- Another purchase ---" << endl;
    cart.addItem(199.99);
    cout << "\n--- Paying with Bitcoin ---" << endl;
    cart.setPaymentMethod(&bitcoin);
    cart.checkout();

    return 0;
}
```

Benefits:

- Easy to switch implementations
- Reduces coupling between modules
- More testable code (can inject mock dependencies)
- Promotes code reuse and flexibility

7. Constructors and Destructors

7.1 Constructors

Purpose: Special member function that initializes an object when it's created.

Types of Constructors:

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

class Student {
private:
    string name;
    int id;
    float gpa;

public:
    // 1. Default Constructor
    Student() {
        name = "Unknown";
        id = 0;
        gpa = 0.0;
        cout << "Default constructor called" << endl;
    }

    // 2. Parameterized Constructor
    Student(string n, int i, float g) {
        name = n;
        id = i;
        gpa = g;
        cout << "Parameterized constructor called" << endl;
    }

    // 3. Constructor with Default Parameters
    Student(string n, int i = 0, float g = 0.0) {
```

```
        name = n;
        id = i;
        gpa = g;
    }

    // 4. Copy Constructor
    Student(const Student& other) {
        name = other.name;
        id = other.id;
        gpa = other.gpa;
        cout << "Copy constructor called" << endl;
    }

    void display() {
        cout << "Name: " << name << ", ID: " << id << ", GPA: " << gpa << endl;
    }
};

int main() {
    Student s1;                         // Default constructor
    Student s2("Alice", 101, 3.8);      // Parameterized constructor
    Student s3 = s2;                    // Copy constructor
    Student s4(s2);                     // Copy constructor (explicit)

    s1.display();
    s2.display();
    s3.display();

    return 0;
}
```

7.2 Member Initializer List

Preferred way to initialize member variables:

```
class Rectangle {
private:
    double width;
    double height;
```

```
    const int id;  // const members must use initializer list

public:
    // Using initializer list (more efficient)
    Rectangle(double w, double h, int i) : width(w), height(h), id(i) {
        // Constructor body
    }

    // Alternative (less efficient - assigns after construction)
    // Rectangle(double w, double h) {
    //     width = w;
    //     height = h;
    // }
};
```

7.3 Destructors

Purpose: Special member function called when an object is destroyed, used for cleanup.

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

class FileHandler {
private:
    string filename;
    bool isOpen;

public:
    // Constructor
    FileHandler(string fname) : filename(fname), isOpen(false) {
        cout << "Opening file: " << filename << endl;
        isOpen = true;
    }

    // Destructor
    ~FileHandler() {
        if (isOpen) {
            cout << "Closing file: " << filename << endl;
            isOpen = false;
```

```
        }
    }

    void writeData(string data) {
        if (isOpen) {
            cout << "Writing to " << filename << ": " << data << endl;
        }
    }
};

int main() {
    {
        FileHandler file1("data.txt");
        file1.writeData("Hello World");

        // Destructor automatically called when file1 goes out of scope
    }

    cout << "After block" << endl;

    return 0;
    // Destructor called for any remaining objects
}
```

Output:

```
Opening file: data.txt
Writing to data.txt: Hello World
Closing file: data.txt
After block
```