MTH 4300/4299

Lecture 8: Object-oriented Programming; Midterm Review

Jaime Abbariao

Encapsulation

- Bundling data and methods together within a class and controlling access through public/private modifiers

```
class BankAccount {
private:
    double balance;

public:
    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    double getBalance() const {
        return balance;
    }
};
```

Why Encapsulation?

- Data Protection: Prevents invalid states and unauthorized access
- Maintainability: Internal changes don't break external code
- Debugging: Easier to track where data is modified

```
// Without encapsulation - problems:
struct BadAccount {
    double balance;
};

BadAccount acc;
acc.balance = -1000;  // Invalid state!

// With encapsulation - safer:
BankAccount good_acc;
good_acc.deposit(-100);  // Rejected by validation
```

Inheritance

- Creating new classes based on existing classes, inheriting their properties and methods

```
class Animal {
protected:
    std::string name;

public:
    void sleep() { std::cout << name << " is sleeping\n"; }
};

class Dog : public Animal {
public:
    Dog(std::string n) { name = n; }
    void bark() { std::cout << name << " says woof!\n"; }
};
```

Why Inheritance?

- Code Reuse: Avoid duplicating common functionality
- Hierarchical Organization: Models real-world relationships
- Extensibility: Add new features without modifying existing code

```
// Without inheritance - code duplication:
class Dog {
    std::string name;
    void sleep() { /* duplicated code */ }
    void bark() { /* unique code */ }
};

class Cat {
    std::string name;        // Duplicated!
    void sleep() { /* duplicated code */ }
    void meow() { /* unique code */ }
};

// With inheritance - cleaner:
```

Why Inheritance? (ii)

```
// Animal base class handles common behavior
// Dog/Cat focus only on their unique features
```

Polymorphism

- Using a single interface to represent different underlying forms (data types)

```
class Shape {
public:
    virtual double area() = 0;  // Pure virtual function
};

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

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

Polymorphism (ii)

```
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() override { return width * height; }
};
```

Why Polymorphism?

- Flexibility: Same code works with different types
- Extensibility: Add new types without changing existing code
- Simplicity: One interface for multiple implementations

```
// Without polymorphism - rigid code:
void calculateAreas() {
    Circle c(5);
    Rectangle r(3, 4);

    double circle_area = 3.14159 * 5 * 5;
    double rect_area = 3 * 4;
}

// With polymorphism - flexible:
void calculateAreas(std::vector<Shape*> shapes) {
    for (Shape* shape : shapes) {
        std::cout << shape->area() << std::endl;
```

Why Polymorphism? (ii)

```
    }
}
```

Abstraction

- Hiding complex implementation details while exposing only essential features

```
class Calculator {
public:
    double add(double a, double b) { return a + b; }
    double multiply(double a, double b) { return a * b; }

private:
    void validateInput(double x) { /* validation logic */ }
    void logOperation(std::string op) { /* logging logic */ }
};

Calculator calc;
double result = calc.add(5.0, 3.0);
```

Why Abstraction?

- **Simplicity**: Users focus on what, not how
- **Reduced Complexity**: Hide implementation details
- **Interface Stability**: Internal changes don't affect users

```
// Without abstraction - overwhelming:
class BadCalculator {
public:
    double add(double a, double b, bool log, bool validate,
               int precision, std::string format) {
        // User must handle all implementation details!
    }
};

// With abstraction - simple:
class Calculator {
public:
    double add(double a, double b) { return a + b; }
};
```

Why Abstraction? (ii)

```
Calculator calc;
double sum = calc.add(2.5, 3.7);
```

Lab: Object-Oriented Design Exercise

Objective: Design and implement a simple inventory management system using OOP principles.

Part 1: Base Item Class (Encapsulation)

```
class Item {
private:
    std::string name;
    double price;
    int quantity;
public:
    // TODO: Implement constructor, getters, setters
    // TODO: Add validation (price > 0, quantity >= 0)
    virtual double calculateValue() const = 0;
    virtual void displayInfo() const = 0;
};
```

Part 2: Derived Classes (Inheritance)

- `Book`: Add `author` and `isbn` fields

Lab: Object-Oriented Design Exercise (ii)

- `Electronics`: Add `warranty_months` field
- `Clothing`: Add `size` and `material` fields

Part 3: Polymorphism

- Override `calculateValue()` (Electronics: +10% tax, Books: no tax, Clothing: +8% tax)
- Override `displayInfo()` to show category-specific details

Part 4: Inventory Management (Abstraction)

```
class Inventory {
private:
    std::vector<Item*> items;
public:
    void addItem(Item* item);
    double getTotalValue() const;
    void displayByCategory() const;
};
```

Midterm 1 Review

Exam Format: 4 questions, 40 points total (October 9th, 2025)

All the questions are focused on problem-solving.

What to expect

- Cheat sheet (up to 3 pages back-and-front)
- No electronic devices.
- No personal belongings at desk
- No bathroom breaks. If you leave, you can't resume your exam.

Key Topics to Review

- **Enums and Control Flow**: Defining enumerated types, switch statements, function parameters
- **String Processing**: Parsing, tokenization, character operations, string building
- **Recursion**: Base cases, recursive calls, problem-solving strategies
- **Arrays and Memory**: 1D/2D operations, indexing patterns, in-place modifications

Enums and Control Flow

- **Defining Enums**: Create custom data types with named constants
- **Switch Statements**: Handle multiple enum values efficiently
- **Function Parameters**: Pass enums to control function behavior

```
enum Operation { ADD, SUBTRACT, MULTIPLY };

double calculate(double a, double b, Operation op) {
    switch (op) {
        case ADD: return a + b;
        case SUBTRACT: return a - b;
        case MULTIPLY: return a * b;
        default: return 0;
    }
}
```

Practice: Grade Calculator

Write a function that converts numerical scores to letter grades using enums.

```
enum Grade { A, B, C, D, F };
Grade calculateGrade(int score);
```

Requirements:

- A: 90-100, B: 80-89, C: 70-79, D: 60-69, F: below 60
- Handle invalid scores (negative or > 100)

Sample: `calculateGrade(85)` returns `B`

String Processing

- **Parsing**: Breaking strings into components (words, characters)
- **Character Operations**: Checking, comparing, transforming individual chars
- **String Building**: Constructing new strings from processed data

```
std::string reverseWords(const std::string& text) {
    std::string result = "";
    std::string word = "";

    for (char c : text) {
        if (c == ' ') {
            result = word + " " + result;
            word = "";
        } else {
            word += c;
        }
    }
```

String Processing (ii)

```
    return word + " " + result;
}
```

Practice: Word Counter

Write a function that counts words in a string and returns the count.

```
int countWords(const std::string& text);
```

Requirements:

- Words are separated by spaces
- Handle multiple consecutive spaces
- Empty string returns 0

Sample: `countWords("hello  world")` returns `2`

Recursion

- **Base Case**: Condition that stops recursion (prevents infinite loops)
- **Recursive Case**: Function calls itself with modified parameters
- **Problem Decomposition**: Break large problems into smaller subproblems

```
int factorial(int n) {
    if (n <= 1) return 1;         // Base case
    return n * factorial(n - 1); // Recursive case
}

int fibonacci(int n) {
    if (n <= 1) return n;                    // Base case
    return fibonacci(n-1) + fibonacci(n-2);  // Recursive case
}
```

Practice: Piecewise Function

Implement this mathematical piecewise function recursively:

```
f(n) = { 1,                  if n ≤ 0
       { n,                  if n = 1 or n = 2
       { f(n-1) + f(n-2) + f(n-3), if n > 2
```

```
int piecewiseFunction(int n);
```

Requirements:

- Use recursion (no loops)
- Follow the mathematical definition exactly

Sample: `piecewiseFunction(4)` returns `6` (f(3) + f(2) + f(1) = 3 + 2 + 1)

Arrays and Memory

- **Indexing**: Accessing elements with `array[index]`
- **2D in 1D**: Representing matrices as `array[row * cols + col]`
- **In-place Operations**: Modifying arrays without extra memory

```
// 2D matrix in 1D array
void printMatrix(int arr[], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            std::cout << arr[i * cols + j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Practice: Array Reversal

Write a function that reverses an array in-place.

```
void reverseArray(int arr[], int size);
```

Requirements:

- Modify the original array (no extra arrays)
- Handle arrays of any size

Sample: `[1, 2, 3, 4]` becomes `[4, 3, 2, 1]`