MTH 4300/4299

Lecture 7: Object-oriented Programming

Jaime Abbariao

Why Object Oriented Programming?

Limitations of Procedural Programming

So far, we've been using **procedural programming**:

- Functions operate on data
- Data and functions are separate
- No way to group related data and functions together
- Difficult to model real-world entities

```
// Procedural approach - separate data and functions
struct Point {
    double x, y;
};

double distance(Point p1, Point p2);
void print(Point p);
Point move(Point p, double dx, double dy);
```

Problems with Procedural Approach

Data and behavior are disconnected:

- Functions can accidentally modify data incorrectly
- No clear ownership of data
- Difficult to maintain as programs grow

Code organization issues:

- Related functions scattered throughout codebase
- No encapsulation or data hiding
- Naming conflicts between similar functions

Real-world modeling:

- Hard to represent complex entities with multiple properties and behaviors
- No natural way to create multiple instances

Object-Oriented Programming (OOP)

OOP organizes code around **objects** rather than functions:

- Objects contain both data (attributes) and functions (methods)
- Objects model real-world entities
- Encapsulation keeps data and methods together
- Better code organization and reusability

Key OOP Principles:

1. **Encapsulation** - bundling data and methods together
2. **Inheritance** - creating new classes based on existing ones
3. **Polymorphism** - using one interface for different underlying forms

Benefits of OOP

Better code organization:

- Related data and functions grouped together
- Clear boundaries between different components
- Easier to understand and maintain

Reusability:

- Create templates (classes) for objects
- Instantiate multiple objects from the same class
- Inherit common functionality

Modeling real-world problems:

- Natural way to represent entities (Student, BankAccount, Car)
- Objects interact like real-world entities

Structs

What are Structs?

A `struct` groups related data together under a single name:

- Combines multiple variables of different types
- Creates a new data type
- Members are accessed using dot notation
- All members are public by default

```
struct Student {
    std::string name;
    int age;
    double gpa;
    std::string major;
};
```

Declaring and Using Structs

```
#include <iostream>
#include <string>

struct Point {
    double x;
    double y;
};

int main() {
    // Create struct instances
    Point p1;               // Uninitialized
    Point p2 = {3.0, 4.0}; // Initialized with values

    // Access members using dot notation
    p1.x = 1.0;
    p1.y = 2.0;

    std::cout << "Point 1: (" << p1.x << ", " << p1.y << ")" <<
```

Declaring and Using Structs (ii)

```
std::endl;
    std::cout << "Point 2: (" << p2.x << ", " << p2.y << ")" <<
std::endl;

    return 0;
}
```

Struct Initialization

Default initialization:

```
Student student1;  // Members have undefined values
```

Aggregate initialization:

```
Student student2 = {"Alice", 20, 3.8, "Computer Science"};
```

Designated initialization (C++20):

```
Student student3 = {
    .name = "Bob",
    .age = 19,
    .gpa = 3.5,
    .major = "Mathematics"
};
```

Functions with Structs

Pass by value (creates a copy):

```
void printStudent(Student s) {
    std::cout << "Name: " << s.name << std::endl;
    std::cout << "Age: " << s.age << std::endl;
    std::cout << "GPA: " << s.gpa << std::endl;
}
```

Pass by reference (more efficient):

```
void updateGPA(Student& s, double newGPA) {
    s.gpa = newGPA;  // Modifies original struct
}

void printStudentConst(const Student& s) {
    std::cout << "Student: " << s.name << std::endl;
    // s.gpa = 4.0;  // Error! Cannot modify const reference
}
```

Structs with Functions (Methods)

Structs can contain functions (called member functions or methods):

```
struct Rectangle {
    double width;
    double height;

    // Member functions
    double area() {
        return width * height;
    }

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

    void display() {
        std::cout << "Rectangle: " << width << " x " << height <<
std::endl;
        std::cout << "Area: " << area() << std::endl;
```

Structs with Functions (Methods) (ii)

```
        std::cout << "Perimeter: " << perimeter() << std::endl;
    }
};
```

Using Struct Methods

```
int main() {
    Rectangle rect = {5.0, 3.0};

    // Call member functions using dot notation
    std::cout << "Area: " << rect.area() << std::endl;
    std::cout << "Perimeter: " << rect.perimeter() << std::endl;

    rect.display();

    // Modify dimensions
    rect.width = 10.0;
    rect.height = 6.0;

    rect.display();

    return 0;
}
```

Constructors in Structs

Constructors initialize struct members:

```
struct Circle {
    double radius;

    // Default constructor
    Circle() {
        radius = 1.0;
    }

    // Parameterized constructor
    Circle(double r) {
        radius = r;
    }

    double area() {
        return 3.14159 * radius * radius;
    }
```

Constructors in Structs (ii)

```
    double circumference() {
        return 2 * 3.14159 * radius;
    }
};
```

```
int main() {
    Circle c1;        // Uses default constructor (radius = 1.0)
    Circle c2(5.0);   // Uses parameterized constructor

    std::cout << "Circle 1 area: " << c1.area() << std::endl;
    std::cout << "Circle 2 area: " << c2.area() << std::endl;

    return 0;
}
```

Classes

Structs vs Classes

In C++, `struct` and `class` are almost identical:

- Both can have constructors, destructors, and member functions
- Both support inheritance and polymorphism
- Only difference: default access level

Struct: Members are `public` by default **Class:** Members are `private` by default

Access Specifiers

Control visibility of class members:

`public` - accessible from anywhere **`private`** - accessible only within the class **`protected`** - accessible within class and derived classes

```
class BankAccount {
private:
    double balance;      // Hidden from outside access
    std::string accountNumber;

public:
    // Constructor
    BankAccount(std::string accNum, double initialBalance) {
        accountNumber = accNum;
        balance = initialBalance;
    }

    // Public methods to access private data
    double getBalance() const {
```

Access Specifiers (ii)

```
        return balance;
    }

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

    bool withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            return true;
        }
        return false;
    }
};
```

Encapsulation

Encapsulation bundles data and methods together and controls access:

```
class Temperature {
private:
    double celsius;

public:
    Temperature(double c = 0.0) : celsius(c) {}

    // Getters (accessors)
    double getCelsius() const { return celsius; }
    double getFahrenheit() const { return celsius * 9.0/5.0 + 32.0; }
    double getKelvin() const { return celsius + 273.15; }

    // Setters (mutators) with validation
    void setCelsius(double c) {
        if (c >= -273.15) {  // Absolute zero check
            celsius = c;
        }
```

Encapsulation (ii)

```
    }

    void setFahrenheit(double f) {
        setCelsius((f - 32.0) * 5.0/9.0);
    }
};
```

Constructor Initialization Lists

More efficient way to initialize members:

```
class Student {
private:
    std::string name;
    int age;
    double gpa;
    std::vector<std::string> courses;

public:
    // Constructor with initialization list
    Student(const std::string& n, int a, double g)
        : name(n), age(a), gpa(g) {
        // Constructor body can be empty or contain additional logic
    }

    // Constructor with default parameters
    Student(const std::string& n = "Unknown", int a = 18)
        : name(n), age(a), gpa(0.0) {}
```

Constructor Initialization Lists (ii)

```
    void addCourse(const std::string& course) {
        courses.push_back(course);
    }

    void displayInfo() const {
        std::cout << "Name: " << name << ", Age: " << age
                  << ", GPA: " << gpa << std::endl;
        std::cout << "Courses: ";
        for (const auto& course : courses) {
            std::cout << course << " ";
        }
        std::cout << std::endl;
    }
};
```

Const Member Functions

Functions that don't modify object state:

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

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

    // Const member functions - don't modify the object
    double getWidth() const { return width; }
    double getHeight() const { return height; }
    double area() const { return width * height; }
    double perimeter() const { return 2 * (width + height); }

    // Non-const member functions - can modify the object
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
    void scale(double factor) {
```

Const Member Functions (ii)

```
        width *= factor;
        height *= factor;
    }
};
```

Static Members

Shared by all instances of a class:

```
class Counter {
private:
    static int count;  // Shared by all Counter objects
    int id;

public:
    Counter() {
        count++;
        id = count;
    }

    ~Counter() {
        count--;
    }

    int getId() const { return id; }
    static int getCount() { return count; }  // Static function
```

Static Members (ii)

```
};

// Definition of static member (outside class)
int Counter::count = 0;
```

```
int main() {
    std::cout << "Count: " << Counter::getCount() << std::endl; // 0

    Counter c1, c2, c3;
    std::cout << "Count: " << Counter::getCount() << std::endl; // 3

    {
        Counter c4;
        std::cout << "Count: " << Counter::getCount() <<
std::endl; // 4
    } // c4 destroyed here

    std::cout << "Count: " << Counter::getCount() << std::endl; // 3
```

Static Members (iii)

```
    return 0;
}
```

Lab

Part 1: Basic Student Struct

Create a `Student` struct with the following requirements:

1. **Data members:**
   - `name` (string)
   - `studentId` (int)
   - `gpa` (double)
   - `major` (string)
2. **Member functions:**
   - `displayInfo()` - prints all student information
   - `updateGPA(double newGPA)` - updates the GPA with validation (0.0-4.0)
   - `isHonorStudent()` - returns true if GPA >= 3.5

Part 2: Course Class with Encapsulation

Create a `Course` class with proper encapsulation:

1. **Private data members:**
   - `courseCode` (string)
   - `courseName` (string)
   - `credits` (int)
   - `maxStudents` (int)
   - `enrolledStudents` (int)
2. **Public methods:**
   - Constructor with parameters
   - Getters for all private members
   - `enrollStudent()` - increments enrolled count if space available
   - `dropStudent()` - decrements enrolled count if > 0
   - `isCourseFull()` - returns true if at capacity
   - `getAvailableSpots()` - returns remaining capacity

Part 3: Enhanced Student Class

Enhance the Student struct to become a proper class:

1. **Convert to class with private members**
2. **Add constructor with initialization list**
3. **Add vector of enrolled courses**
4. **Add methods:**
   - `enrollInCourse(const std::string& courseCode)`
   - `getEnrolledCourses()`
   - `getTotalCredits()`
   - `calculateSemesterGPA()`