Lecture 1

Dr. Umair Rehman

Agenda

- Course Overview
- Basics of Object-Oriented Design and Analysis (OODA)
- Object-Oriented Approach vs Functional Programming
- Cohesion and Coupling
- Advantage of the Object-Oriented Approaches

About the Instructor

- Umair Rehman
  - Director of Human-Centered Computing Group (HCCG)
  - Assistance Professor, Computer Science, Western University

- Research Area: Human-centered AI, Games, Human-computer Interaction

CS3307 – About the Course

- By the end of this course, students will be able to design, evaluate, and implement scalable, maintainable software systems using object-oriented principles and design patterns.

Concepts Covered

- Object-Oriented Concepts: Encapsulation, Inheritance, polymorphism, etc
- Design Patterns: Apply, reuse, solve
- UML Diagrams: Class, sequence, use case
- Code Refactoring: Improve, structure, maintainability
- Quality Attributes: Scalability, security, performance
- Design Specifications: Align, requirements, constraints
- Software Evolution: Manage, refactor, enhance quality

Deliverables

- Final Project (80%):
  - Design real-world system
  - Apply OODA in C++Requirements, UML, design patterns
  - Three phases: proposal, intermediate, final
  - Evaluated on design, implementation, testing

- Final Exam (20%):Mix of questions

Due Dates

Deliverable	Due Date	Weight
Project Proposal	October 1, 2025	20%
Intermediate Design and Partial Implementation	October 30, 2025	30%
Final Project Submission	December 15, 2025	30%
Final Exam	December 10, 2025	20%

Objects

- Objects represent real-world entities and consist of two main parts:
  - attributes (also called data members)
  - behaviors (also called member functions or methods)

Classes

- Blueprint or template that defines the properties and behaviors of an object.

- An object is an instance of a class
  - Once a class is defined, you can create objects from it
  - For example, if we have a class Car, objects of this class would be specific cars (like a Ford, Toyota, etc.).

Attributes

- These are variables that store information about the object.
- In C++, you declare attributes inside the class.
  - Example: If Car is a class, attributes could be color, speed, and model.

Methods

- Methods define the behavior of an object, or what actions it can perform.
  - Example: For a `Car` class, methods might include `accelerate()`, `brake(), or getSpeed()`.

Access Specifiers

- Private: Attributes and methods can only be accessed from within the class.
- Public: Attributes and methods can be accessed from outside the class.
- Protected: Used in inheritance; accessible to derived classes but not outside the class.

Constructors and Destructors

- A constructor is a special method that initializes objects of a class.
- A destructor is used to clean up resources when an object is destroyed.

Encapsulation (Next lecture)

- Bundling the data (attributes) and methods (behaviors) into a single unit (the object)

- Restricting direct access to some of the object's components.

Inheritance & Polymorphism (Next Lecture)

- Inheritance allows one class to derive properties and behaviors from another class

- Polymorphism allows one interface to be used for different data types or classes (e.g., function overloading, overriding)

Abstract Class

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

class Animal {   // Abstract class
public:
    virtual void speak() = 0;   // pure virtual (must be overridden)

    void sleep() {              // normal method
        cout << "Sleeping..." << endl;
    }

protected:
    int age;  // data member allowed
};

```

Interfacess

```cpp
class IShape {    // Interface style in C++
public:
    virtual void draw() = 0;    // must be implemented
    virtual double area() = 0;  // must be implemented

    virtual ~IShape() {}        // virtual destructor
};

```

Example

```cpp
#include <iostream>  // Needed for input/output operations
#include <string>    // Needed for using the string data type
class Car {
public:
    // Constructor: Initializes an object with model and year
    Car(std::string carModel, int carYear) {
        model = carModel;  // Assign the carModel parameter to the member variable model
        year = carYear;    // Assign the carYear parameter to the member variable year
    }

    // Method to display the car's details
    void displayInfo() {
        std::cout << "Model: " << model << ", Year: " << year << std::endl;
    }

private:
    std::string model;  // The model of the car (e.g., "Toyota")
    int year;           // The year the car was manufactured (e.g., 2022)
};
```

Example

```
int main() {
    // Create an object of the class Car with specific values
    Car myCar("Toyota", 2022);

    // Call the displayInfo method to print the car's information
    myCar.displayInfo();

    return 0;
}

```

Object Oriented Design & Analysis

- System is designed using objects
- Represent real-world entities
- Interact with each other to solve a problem
- Structure the system: modular, flexible, and easy to maintain

Designing Classes and Objects

- Start by identifying the key classes and objects that are relevant to the problem.

1. Identify Objects

- Problem domain and identify the entities (objects)

- Real-world objects or concepts:
  - Example: Online shopping system: `Customer, Product, Order, ShoppingCart,` etc.

2. Define Attributes and Methods

- For each object,
  - Determine what information it needs to store (attributes)
  - What actions it needs to perform (methods)

- Example: A Product class might have
  - Attributes like `name, price, and stockQuantity`
  - Methods like `getPrice()` or `updateStock()`

3. Determine Relationships

- Objects often interact with one another
  - Association: Objects know about each other (e.g., a Customer has an Order).
  - Aggregation: One object contains or is composed of other objects (e.g., an Order contains multiple Product objects).
  - Inheritance: One object is a specialized version of another (e.g., `Admin` and `RegularUser` classes inheriting from a User class).

Object Oriented Programming (OOP)

- OOP (Recap)
  - Modeling real-world entities as objects
  - Objects contain both data (attributes) and behavior (methods)
  - Interact with each other to perform tasks
  - It follows the principles of OOP

Object Oriented Programming (OOP)

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

// Object-Oriented Example: Modeling a Bank Account
class Account {
private:
    double balance;   // State (attribute)

public:
    // Constructor
    Account(double initialBalance) : balance(initialBalance) {}

    // Method to deposit money
    void deposit(double amount) {
        balance += amount;  // Modifies state
    }

```

Object Oriented Programming (OOP)

```cpp
    // Method to withdraw money
    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;  // Modifies state
        }
    }

    // Method to check balance
    double getBalance() const {
        return balance;
    }
};

int main() {
    Account myAccount(100.0);  // Create an Account object with $100 initial balance
    myAccount.deposit(50);     // Deposit $50
    myAccount.withdraw(30);    // Withdraw $30

    cout << "Final Balance: $" << myAccount.getBalance() << endl;  // Output final balance
}
```

Functional Programming

- Functional Programming:
  - Focuses on pure functions and immutability
  - Does not modify external state
  - Computation as the evaluation of mathematical functions
  - Emphasizes data flow through functions rather than changing object states

Functional Programming

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

// Functional Programming: Using functions to handle account transactions

// Function to deposit money (returns new balance without modifying input state)
double deposit(double balance, double amount) {
    return balance + amount;  // Return the new balance
}

// Function to withdraw money (returns new balance without modifying input state)
double withdraw(double balance, double amount) {
    if (balance >= amount) {
        return balance - amount;  // Return the new balance
    } else {
        cout << "Insufficient funds!" << endl;
        return balance;  // No change if withdrawal fails
    }
}

```

Functional Programming

```

int main() {
    double balance = 100.0;  // Initial balance

    // Applying functions to simulate deposits and withdrawals
    balance = deposit(balance, 50);   // Deposit $50
    cout << "Balance after deposit: $" << balance << endl;

    balance = withdraw(balance, 30);  // Withdraw $30
    cout << "Balance after withdrawal: $" << balance << endl;

    return 0;
}

```

State and Mutability: OOP

- Objects maintain internal state
- Can be modified by their methods
- Objects interact and modify each other's states
- State of the program evolves over time

- For example, a `BankAccount` object might have methods like `deposit()` and `withdraw()`, which directly change the account balance.

State and Mutability: Functional Programming

- Emphasizes immutability
- Data should not change once it is created

- You create new data structures based on transformations.

- In the same banking scenario, you would return a new account balance after each transaction rather than modifying the existing one.

Modularity and Reuse: OOP

- Reuse is achieved by defining reusable classes.
- Inheritance and polymorphism: new functionality by extending existing classes.

- You model entities as objects, and the methods that operate on these objects are tightly bound to them.

Modularity and Reuse: Functional Programming

- Reuse is achieved by writing pure, reusable functions
- Functions can be combined, passed as arguments, or returned as values (higher-order functions).
- Building software by composing small, reusable functions.

Modularity and Reuse: Functional Programming

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

// A function that adds two numbers
int add(int a, int b) {
    return a + b;
}

// A function that takes two numbers and another function as input
int applyFunction(int x, int y, int (*func)(int, int)) {
    return func(x, y);  // Call the passed function with x and y
}

int main() {
    int result = applyFunction(5, 3, add);  // Pass the 'add' function as an argument
    cout << "Result: " << result << endl;   // Output: Result: 8
    return 0;
}
```

Concurrency: OOP

- Concurrency can be complex because of shared mutable state
- Issues like race conditions, deadlocks, and difficult debugging
- Managing concurrent access
  - Mechanisms like locks or synchronization

Concurrency: Functional Programming

- Concurrency is easier to manage due to immutability
- Data is not modified after creation
- Multiple functions can operate on it without interference

Abstraction and Encapsulation: OOP

- Encapsulation is a key feature (next lecture)
- Objects hide their internal state
- Expose behavior through public methods

Abstraction and Encapsulation: Functional Programming

- Does not rely on encapsulation as much
- Through the use of
  - higher-order functions
  - function composition
  - pure functions

Abstraction and Encapsulation: Functional Programming

```
// A function to double a number
int doubleNumber(int x) {
    return x * 2;
}

// A function to square a number
int squareNumber(int x) {
    return x * x;
}

int main() {
    int x = 5;

    // Compose functions: first double the number, then square it
    int result = squareNumber(doubleNumber(x));  // First 5 * 2 = 10, then 10 * 10 = 100
    cout << result << endl;  // Output: 100
}

```

Flexibility and Extensibility: OOP

- Inheritance and polymorphism to extend and modify behavior
- Tightly coupled systems
  - Harder to change certain aspects without affecting others.
- Flexibility comes from object hierarchies and class extension
  - Lead to complexity as the class hierarchy grows.

Flexibility and Extensibility: OOP

```cpp
class Shape {
    virtual void draw() = 0;  // Abstract method
};

class Circle : public Shape {
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

class Square : public Shape {
    void draw() override {
        cout << "Drawing Square" << endl;
    }
};

```

Flexibility and Extensibility: Functional Programming

- Emphasizes composability
- Combine small functions to build more complex behavior
  - Without needing to extend or modify existing code
- Functions are first-class citizens
  - Pass them as arguments to other functions
  - Return them from functions
  - Assign them to variables
  - Store them in data structures (like arrays, lists, etc.)

Flexibility and Extensibility: Functional Programming

```cpp
void drawShape(function<void()> drawFunc) {
    drawFunc();  // Call the passed function to draw
}

int main() {
    drawShape([]() { cout << "Drawing Circle" << endl; });
    drawShape([]() { cout << "Drawing Square" << endl; });
}
```

Typical Use Case: OOP

- Systems that naturally map to real-world entities
  - User interfaces, games, and business systems (e.g., payroll, inventory management)

- Systems where managing state and modeling entities with attributes and behaviors are critical

Typical Use Case: Functional Programming

- Ideal for data transformation, scientific computing, and concurrent processing.

- Common in stateless web apps, reactive systems, and where immutability is crucial.

Object-Oriented Approach vs Functional Programming

Feature	Object-Oriented Approach	Functional Programming
Focus	Objects (data + behavior)	Functions (transformation of data)
State Management	Mutable state within objects	Immutable state, new data structures
Modularity	Classes and object hierarchies	Functions and composition
Concurrency	Complex due to mutable state	Easier with immutability
Abstraction	Objects encapsulate state and behavior	Functions abstract operations
Reuse	Inheritance, polymorphism	Higher-order functions, function composition
Typical Use Cases	UI, business apps, games, systems with entities	Data processing, scientific computing, stateless apps

Cohesion and Coupling in Design

- Two key principles that affect how maintainable, understandable, and flexible your code is.

Cohesion

- Closely related the functions and responsibilities within a single module (class or component)

- The degree to which elements inside a module belong together

- High cohesion is generally desirable

High Cohesion

- Responsibilities are closely related and focused on a single task
- It follows the Single Responsibility Principle (Future Lecture)
- When you need to change something in the system, the change is isolated to one module.

Example of High Cohesion

- In an e-commerce system, a `ShoppingCart` class is cohesive if all its methods `(addItem(), removeItem(), calculateTotal())` are directly related to managing the shopping cart.

- Class is focused solely on cart-related activities

Low Cohesion

- Responsibilities are spread across multiple unrelated tasks.
  - Leads to modules that are harder to maintain
  - Changing one part of the module inadvertently affect other unrelated parts.

Why High Cohesion Is Important:

- Ease of Understanding
- Ease of Maintenance
- Reusability

Coupling

- Degree of dependency between modules.
- Closely two classes or components are connected to each other.
- Low coupling is preferred
  - Changes in one module have minimal impact on others.

Low Coupling

- Low coupling means that modules are mostly independent
- It follows the Dependency Inversion Principle (Future Lectures)

Why Low Coupling Is Important:

- Ease of Maintenance
- Flexibility
- Testing

Cohsesion and Coupling

- High cohesion + low coupling:
  - This is the ideal scenario in software design
- High cohesion + high coupling:
  - Internally well-organized but strong dependency on other modules
- Low cohesion + low coupling:
  - The modules are independent, but they are poorly organized internally.
- Low cohesion + high coupling:
  - Modules are disorganized internally; highly dependent on others

Advantages of OOP: Modularity

- Breaking down a system into smaller, manageable
- Each class is self-contained and performs a specific function
- Modularity makes the system easier to understand, develop, and maintain
- Example:
  - In an e-commerce system, the `ShoppingCart`, `Order`, and `Payment` classes can be developed independently

Advantages of OOP: Reusability

- Promotes code reuse through inheritance and polymorphism
- Class can be reused in other parts of the application
- Create generalized classes and then extend them to handle specific cases by creating subclasses
- Example:
  - A base class `Vehicle` can be reused for different vehicle types like `Car`, `Bike`, and `Truck`.
  - Reuse the base functionality and extend it where needed.

Advantages of OOP: Encapsulation

- Control over the data
  - Accessed or modified only in an intended way.

- Protect the object's state from outside interference
  - Reducing bugs and unintended side effects

- Example:
  - A `BankAccount` class may expose methods like `deposit()` and `withdraw()`, but the `balance` is kept private.

Advantages of OOP: Maintainability

- Separation of concerns
  - Each class or object is responsible for a specific part of the system.
  - When a bug or a new feature is identified, changes are often isolated to a single class.
  - Modifying one part of the system typically doesn't affect others, making the system easier to maintain over time.
- Example:
  - If a bug is found in the Payment class of an online store, it can be fixed without affecting the rest of the application, such as `ShoppingCart` or `Order`.

Advantages of OOP: Extensibility

- Open for extension but closed for modification, following the Open/Closed Principle (OCP) (Next Lecture)
- Extend functionality by adding new classes or methods without changing existing code
- Easier to add new features and scale the system as requirements evolve without disrupting existing functionality

Advantages of OOP: Polymorphism

- Objects of different classes to be treated as objects of a common superclass.
- Flexibility in designing systems that can operate on objects in a generic way
- Example:
  - A `PaymentMethod` interface can have different implementations like `CreditCardPayment, PayPalPayment, or CryptoPayment`.
  - All payment methods polymorphically, allowing new payment types to be added without changing the payment processing logic.

Advantages of OOP: Inheritance

- Inherit properties and methods from other classes, which encourages code reuse and reduces duplication

- Allows for the creation of hierarchical relationships

- Example:
  - In a game, a base class Character might define common attributes like health, strength, and methods like `move()` or `attack()`.
  - Specific characters like Warrior or Mage can inherit from Character and add their own unique attributes or abilities.

Advantages of OOP: Team Collaboration

- Team collaboration by clearly defining interfaces and class responsibilities.

- Different team members or teams can work on separate parts of the system

- Example:
  - One team might be responsible for developing the User class and its associated logic, while another team handles the Transaction class

Advantages of OOP: Security and Access Control

- You can restrict access to certain parts of the system by using access modifiers

- Example:
  - In a banking application, sensitive operations such as transferring money between accounts may be handled by a `BankTransaction` class that limits access to certain methods based on user roles, thus preventing unauthorized access.

Advantages of OOP: Scaling

- Isolate different parts of the system, it's easier to scale individual components of the system without affecting others.

- Example:
  - In a microservices architecture, each service (e.g., UserService, PaymentService) can be treated as an object with well-defined interfaces

  - If one part of the system (e.g., PaymentService) needs to handle more traffic, it can be scaled independently without affecting other services

Advantages of OOP: Debugging and Testing

- Debugging and testing easier because classes are self-contained units
  - Tested independently

- Mock objects or dependency injection
  - Test individual classes in isolation

- Example:
  - In a web application, you can test the Order class in isolation by mocking external dependencies like the Payment or Shipping systems

Course Project (80%)

- Develop a software system that solves a moderately complex problem using the core concepts of Object-Oriented Design (OOD)
  - Requirements Gathering and Analysis
  - UML System Modeling
  - Applying Design Patterns
  - Full System Implementation in C++

Deliverable 1: Proposal (20%)

- Due: Oct 1, 2025
  - Problem Statement
  - System Features
  - Design Approach
  - Initial UML Class Diagram

Deliverable 2: Intermediate Design and Partial Implementation (30%)

- Due: Oct 30, 2025
  - Complete UML Diagrams
  - Use Case Descriptions
  - Partial Implementation in C++
  - Design Rationale Document

Deliverable 3: Final Submission (30%)

- Due: Dec 15, 2025
  - Complete System Implementation
  - Final UML Diagrams
  - Comprehensive Documentation

Submission Guidelines

- All deliverables submitted through Brightspace and Github.
- UML diagrams must be submitted in PDF format
- Ensure that your documentation is clear and concise

Final Exam (20%) D

- Due: Dec 10, 2025
  - Mix of multiple-choice, short answer, and problem-solving questions.
  - Ability to apply the concepts and techniques discussed throughout the course
  - Lots and Lots of Refactoring ….writing code with hand

Recap

- Course Overview
- Basics of Object-Oriented Design and Analysis (OODA)
- Object-Oriented Approach vs Functional Programming
- Cohesion and Coupling
- Advantage of the Object-Oriented Approaches