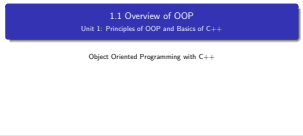
1.1 Overview of OOP

Unit 1: Principles of OOP and Basics of C++

Object Oriented Programming with C++



Lecture 5: Overview of Object-Oriented Programming

- Unit 1: Principles of OOP and Basics of C++
- Topic 1.1: Overview of OOP
- Course: Object Oriented Programming with C++ (DI05011021)

Lecture 5: Overview of Object-Oriented Programming
- Unit 1: Principles of OOP and Basics of C++
- Topic 1.1: Overview of OOP
- Course: Object Oriented Programming with C++ (DI05011021)

Welcome everyone to Lecture 5. Today, we are transitioning from the structural basics of C++ into the core philosophy of the language: Object-Oriented Programming. This lecture is fundamental, as it lays the groundwork for everything we will do in this course. We will contrast OOP with the older procedural style and define the key pillars of OOP.

Procedure-Oriented Programming (POP)

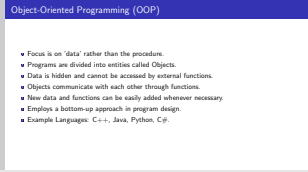
- Focus is on 'doing things' (algorithms/functions).
- Large programs are divided into smaller programs known as functions.
- Data moves openly around the system from function to function.
- Functions transform data from one form to another.
- Employs a top-down approach in program design.
- Example Languages: C, Pascal, FORTRAN.

Procedure-Oriented Programming (POP)
- Focus is on 'doing things' (algorithms/functions).
- Large programs are divided into smaller programs known as functions.
- Data moves openly around the system from function to function.
- Functions transform data from one form to another.
- Employs a top-down approach in program design.
- Example Languages: C, Pascal, FORTRAN.

Before OOP, the dominant paradigm was Procedure-Oriented Programming, or POP. In POP, you view a problem as a sequence of things to be done—reading, calculating, and printing. The primary building block is the function. A major flaw of POP is that data is treated as a secondary citizen. Global data is accessible and modifiable by any function, which makes large programs difficult to manage and debug. It's like having a shared whiteboard where anyone can erase or change the information at any time.

Object-Oriented Programming (OOP)

- Focus is on 'data' rather than the procedure.
- Programs are divided into entities called Objects.
- Data is hidden and cannot be accessed by external functions.
- Objects communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Employs a bottom-up approach in program design.
- Example Languages: C++, Java, Python, C#.



To overcome the limitations of POP, Object-Oriented Programming was invented. OOP treats data as a critical element and does not allow it to flow freely around the system. Instead, OOP ties data closely to the functions that operate on it, protecting it from accidental modification from outside functions. We break the problem down into entities called objects and build up from there—a bottom-up approach.

POP vs. OOP: Key Differences

- **Approach**: POP is Top-down; OOP is Bottom-up.
- **Division**: POP divides into functions; OOP divides into objects.
- **Data Access**: POP has open global data; OOP hides data (Encapsulation).
- **Modifications**: Hard to modify POP code; OOP makes it easy to add new objects and functions.
- **Real-world Modeling**: POP struggles with real-world complexity; OOP excels at modeling real-world entities.

POP vs. OOP: Key Differences
- **Approach**: POP is Top-down; OOP is Bottom-up.
- **Division**: POP divides into functions; OOP divides into objects.
- **Data Access**: POP has open global data; OOP hides data (Encapsulation).
- **Modifications**: Hard to modify POP code; OOP makes it easy to add new objects and functions.
- **Real-world Modeling**: POP struggles with real-world complexity; OOP excels at modeling real-world entities.

Let's summarize the differences. The most crucial distinction is how they handle data. In POP, data is passive and exposed. In OOP, data is active and protected. If you want to model a banking system, POP would have a bunch of functions like 'deposit' and 'withdraw' passing around raw account numbers and balances. OOP will have an 'Account' object that internally manages its balance and exposes safe methods to interact with it.

Basic Concepts of OOP

- The core concepts that define an Object-Oriented Language:
- 1. Objects
- 2. Classes
- 3. Data Abstraction
- 4. Encapsulation
- 5. Inheritance
- 6. Polymorphism
- 7. Dynamic Binding
- 8. Message Passing

Basic Concepts of OOP
- The core concepts that define an Object-Oriented Language:
- 1. Objects
- 2. Classes
- 3. Data Abstraction
- 4. Encapsulation
- 5. Inheritance
- 6. Polymorphism
- 7. Dynamic Binding
- 8. Message Passing

Now that we understand why OOP exists, let's look at its fundamental building blocks. These eight concepts are the 'buzzwords' of OOP, but they are also deeply practical tools. We will go through each of these one by one. Mastering these concepts is equivalent to mastering the philosophy of C++.

1. Objects

- The basic run-time entities in an object-oriented system.
- May represent a person, a place, a bank account, a table of data.
- They take up space in memory and have an associated address.
- Consist of State (Data/Properties) and Behavior (Functions/Methods).
- Example: Object 'Car' has State (color, model, speed) and Behavior (accelerate, brake).

1. Objects
- The basic run-time entities in an object-oriented system.
- May represent a person, a place, a bank account, a table of data.
- They take up space in memory and have an associated address.
- Consist of State (Data/Properties) and Behavior (Functions/Methods).
- Example: Object 'Car' has State (color, model, speed) and Behavior (accelerate, brake).

Think of objects as the nouns of your program. They are the actual entities that exist in memory when the program runs. If you are building a student management system, an object would be a specific student, say 'Alice', with her specific roll number and grades. Objects bundle both the state (her grades) and the behavior (calculating her GPA) together.

2. Classes

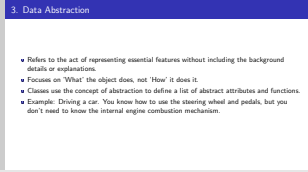
- A class is a blueprint or template for creating objects.
- It defines a user-defined data type.
- Objects are variables of type class.
- Once a class is defined, you can create any number of objects belonging to that class.
- Analogy: A class is like the architectural blueprint of a house; the objects are the actual houses built from that blueprint.

2. Classes
- A class is a blueprint or template for creating objects.
- It defines a user-defined data type.
- Objects are variables of type class.
- Once a class is defined, you can create any number of objects belonging to that class.
- Analogy: A class is like the architectural blueprint of a house; the objects are the actual houses built from that blueprint.

If an object is a specific student like 'Alice', the Class is the general concept of 'Student'. A class doesn't take up memory space for data; it just defines what an object of that type will look like. It specifies what data variables an object will have and what functions it can perform. You define the class once, and then instantiate multiple objects from it.

3. Data Abstraction

- Refers to the act of representing essential features without including the background details or explanations.
- Focuses on 'What' the object does, not 'How' it does it.
- Classes use the concept of abstraction to define a list of abstract attributes and functions.
- Example: Driving a car. You know how to use the steering wheel and pedals, but you don't need to know the internal engine combustion mechanism.



Abstraction is about managing complexity by hiding unnecessary details. When you use the 'cout' object in C++, you don't know the complex low-level code required to put pixels on the screen; you just know that providing a string to it will print the text. That is abstraction. It allows programmers to use complex systems easily.

4. Encapsulation

- The wrapping up of data and functions into a single unit (called class).
- Data is not accessible to the outside world, only those functions which are wrapped in the class can access it.
- Provides 'Data Hiding' and secures data from unintended interference.
- Analogy: A capsule enclosing various medicines.

4. Encapsulation
- The wrapping up of data and functions into a single unit (called class).
- Data is not accessible to the outside world, only those functions which are wrapped in the class can access it.
- Provides 'Data Hiding' and secures data from unintended interference.
- Analogy: A capsule enclosing various medicines.

While abstraction is about hiding complexity (the 'how'), encapsulation is about hiding and protecting the data itself. By keeping data private and only allowing access through specific public functions (often called getters and setters), we prevent the rest of the program from putting the object into an invalid state. This is the primary mechanism OOP uses to fix the global data problem of POP.

Abstraction vs. Encapsulation

- **Abstraction**: Hiding the implementation details (Design level).
- **Encapsulation**: Hiding the data (Implementation level).
- They work together: Encapsulation is the technique used to implement Abstraction.
- You abstract the interface, and you encapsulate the data.

Abstraction vs. Encapsulation
- **Abstraction**: Hiding the implementation details (Design level).
- **Encapsulation**: Hiding the data (Implementation level).
- They work together: Encapsulation is the technique used to implement Abstraction.
- You abstract the interface, and you encapsulate the data.

Students often confuse abstraction and encapsulation because they both involve 'hiding'. Remember this: Abstraction is a design concept. You decide *what* to expose. Encapsulation is the implementation mechanism. You bundle things in a class and make variables private to *achieve* that abstraction.

5. Inheritance

- The process by which objects of one class acquire the properties of objects of another class.
- Supports the concept of hierarchical classification.
- Promotes **Reusability**: You can add additional features to an existing class without modifying it.
- Base Class (Parent) -¿ Derived Class (Child).
- Example: 'Bird' is a base class. 'Sparrow' and 'Penguin' are derived classes that inherit from 'Bird'.

5. Inheritance
- The process by which objects of one class acquire the properties of objects of another class.
- Supports the concept of hierarchical classification.
- Promotes **Reusability**: You can add additional features to an existing class without modifying it.
- Base Class (Parent) -¿ Derived Class (Child).
- Example: 'Bird' is a base class. 'Sparrow' and 'Penguin' are derived classes that inherit from 'Bird'.

Inheritance is arguably the most powerful feature for code reuse. If you have a class that works perfectly, and you need a new class that does almost the same thing but with a few additions, you don't copy-paste the code. You inherit it. The derived class gets all the non-private attributes and methods of the base class automatically. This forms an 'is-a' relationship. A Sparrow *is a* Bird.

6. Polymorphism

- A Greek term meaning 'ability to take more than one form'.
- An operation may exhibit different behaviors in different instances.
- The behavior depends upon the types of data used in the operation.
- Types: Compile-time (Function/Operator Overloading) and Run-time (Virtual Functions).
- Example: The '+' operator can add integers (5+4) or concatenate strings ('hello' + 'world').

6. Polymorphism
- A Greek term meaning 'ability to take more than one form'.
- An operation may exhibit different behaviors in different instances.
- The behavior depends upon the types of data used in the operation.
- Types: Compile-time (Function/Operator Overloading) and Run-time (Virtual Functions).
- Example: The '+' operator can add integers (5+4) or concatenate strings ('hello' + 'world').

Polymorphism allows you to use the same interface for different underlying forms (data types). Imagine a function named 'draw()'. If you call it on a Circle object, it draws a circle. If you call it on a Square object, it draws a square. The same function name results in different behaviors depending on the object it is acting upon.

7. Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response.
- Dynamic binding (or late binding) means the code associated with a given procedure call is not known until the time of the call at run-time.
- Heavily associated with polymorphism and inheritance.
- C++ uses 'virtual functions' to achieve dynamic binding.

7. Dynamic Binding
- Binding refers to the linking of a procedure call to the code to be executed in response.
- Dynamic binding (or late binding) means the code associated with a given procedure call is not known until the time of the call at run-time.
- Heavily associated with polymorphism and inheritance.
- C++ uses 'virtual functions' to achieve dynamic binding.

When the compiler translates your code, it normally links a function call directly to a specific memory address where the function lives (static binding). But with dynamic binding, the exact function to call isn't known until the program is actually running. The program looks at the *actual type* of the object at runtime and calls the appropriate function. We will explore this deeply when we cover virtual functions later in the course.

8. Message Passing

- An object-oriented program consists of a set of objects that communicate with each other.
- Message passing involves specifying the name of the object, the name of the function (message), and the information to be sent.
- Format: `object.method(parameters);`
- Example: `employee.computeSalary(hoursWorked);`

8. Message Passing
- An object-oriented program consists of a set of objects that communicate with each other.
- Message passing involves specifying the name of the object, the name of the function (message), and the information to be sent.
- Format: object.method(parameters);
- Example: employee.computeSalary(hoursWorked);

Objects don't exist in isolation; they interact. In OOP, we don't say we 'call a function on an object'. We prefer to say we 'send a message to an object'. The message is the function name, the data we send are the parameters, and the object is the recipient. The object receives the message and executes its corresponding method.

Benefits of OOP

- Code Reusability (through inheritance).
- Data Security (through encapsulation).
- Easier Troubleshooting and Maintenance (modular structure).
- Flexibility (through polymorphism).
- Better modeling of complex, real-world problems.

Benefits of OOP
- Code Reusability (through inheritance).
- Data Security (through encapsulation).
- Easier Troubleshooting and Maintenance (modular structure).
- Flexibility (through polymorphism).
- Better modeling of complex, real-world problems.

To wrap up, why do we use OOP? It's not just to make things more complicated! OOP makes large-scale software development possible. It allows teams to work on different classes independently. It allows us to reuse existing code, secure our data from bugs, and mentally map the software directly to the real-world problem we are trying to solve.

Questions & Review

- Can you name the 4 core pillars of OOP? (Abstraction, Encapsulation, Inheritance, Polymorphism)
- What is the main difference between a Class and an Object?
- How does Encapsulation improve software security?
- Any questions?

Questions & Review
- Can you name the 4 core pillars of OOP? (Abstraction, Encapsulation, Inheritance, Polymorphism)
- What is the main difference between a Class and an Object?
- How does Encapsulation improve software security?
- Any questions?

Let's take a moment to review. Who can define abstraction in their own words? How about the difference between a class and an object? (Pause for student answers). Great. Are there any other questions about the concepts we covered today?