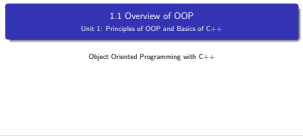
1.1 Overview of OOP

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

Object Oriented Programming with C++



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

- Course: DI05011021 - Object Oriented Programming with C++
- Lecture 9: 1.1 Overview of OOP
- Focus: POP vs OOP, and the Core Pillars of OOP

Unit 1: Principles of OOP and Basics of C++
- Course: DI05011021 - Object Oriented Programming with C++
- Lecture 9: 1.1 Overview of OOP
- Focus: POP vs OOP, and the Core Pillars of OOP

Welcome everyone to Lecture 9. Today marks a very important milestone in this course. We are transitioning from understanding the basic syntax of C++ to grasping the very paradigm that makes C++ so powerful: Object-Oriented Programming, or OOP. By the end of this session, you will understand not just what OOP is, but why it was invented, how it differs from older programming styles, and the foundational pillars that support it.

Learning Objectives

- Contrast Procedure-Oriented Programming (POP) with Object-Oriented Programming (OOP).
- Define Classes and Objects as the fundamental building blocks of OOP.
- Understand Data Abstraction and Encapsulation.
- Explore relationships and behaviors via Inheritance and Polymorphism.
- Explain the runtime mechanics of Dynamic Binding and Message Passing.

Learning Objectives
- Contrast Procedure-Oriented Programming (POP) with Object-Oriented Programming (OOP).
- Define Classes and Objects as the fundamental building blocks of OOP.
- Understand Data Abstraction and Encapsulation.
- Explore relationships and behaviors via Inheritance and Polymorphism.
- Explain the runtime mechanics of Dynamic Binding and Message Passing.

Here is our roadmap for today. We will start by looking backward at Procedure-Oriented Programming to understand the problems early programmers faced. Then, we will introduce OOP as the solution to those problems, systematically going through each of the eight core concepts that define an object-oriented system.

Procedure-Oriented Programming (POP)

- Focus is primarily on 'what to do' (algorithms and functions) rather than data.
- Large programs are divided into smaller, manageable units known as functions.
- Data is openly shared and moves freely around the system from function to function.
- Employs a top-down approach in program design.
- Classic examples of POP languages: C, Pascal, FORTRAN.

Procedure-Oriented Programming (POP)
- Focus is primarily on 'what to do' (algorithms and functions) rather than data.
- Large programs are divided into smaller, manageable units known as functions.
- Data is openly shared and moves freely around the system from function to function.
- Employs a top-down approach in program design.
- Classic examples of POP languages: C, Pascal, FORTRAN.

Before OOP, the primary paradigm was Procedure-Oriented Programming. Think of POP as a recipe: a sequence of steps to be carried out. You have functions that manipulate data to get a final result. However, the data itself is treated as a secondary citizen, existing largely to be processed by the functions.

Limitations of POP

- Data Security: Global data is vulnerable to accidental modification by any function.
- Maintainability: Changes in data types require modifications in all functions that access them.
- Real-world Modeling: Difficult to map real-world physical entities directly into code.
- Code Reusability: Limited ability to reuse existing code seamlessly without duplication.

Limitations of POP
- Data Security: Global data is vulnerable to accidental modification by any function.
- Maintainability: Changes in data types require modifications in all functions that access them.
- Real-world Modeling: Difficult to map real-world physical entities directly into code.
- Code Reusability: Limited ability to reuse existing code seamlessly without duplication.

The lack of data security is the biggest flaw in POP. When a program grows, managing data state across hundreds of functions becomes incredibly difficult. If you change a global data structure, you have to find and update every single function that touches it. This leads to fragile codebases and necessitates a better approach.

Object-Oriented Programming (OOP): The Paradigm Shift

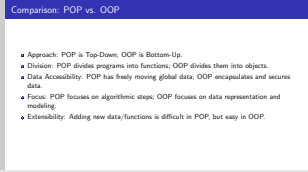
- Focus is primarily on 'data' rather than 'functions'.
- Programs are divided into distinct entities called 'Objects'.
- Functions that operate on the data of an object are tied together in the same data structure.
- Data is hidden and cannot be accessed by external functions.
- Follows a bottom-up approach in program design.

Object-Oriented Programming (OOP): The Paradigm Shift
- Focus is primarily on 'data' rather than 'functions'.
- Programs are divided into distinct entities called 'Objects'.
- Functions that operate on the data of an object are tied together in the same data structure.
- Data is hidden and cannot be accessed by external functions.
- Follows a bottom-up approach in program design.

OOP flips the script. Instead of asking 'what actions need to be performed?', we ask 'what real-world entities are we trying to model?'. We bundle the data and the functions that operate on that data into a single, cohesive unit called an object. By hiding the data inside the object, we solve the security and maintainability issues of POP.

Comparison: POP vs. OOP

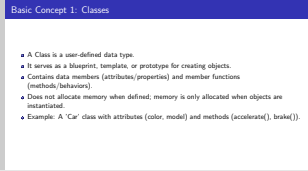
- Approach: POP is Top-Down; OOP is Bottom-Up.
- Division: POP divides programs into functions; OOP divides them into objects.
- Data Accessibility: POP has freely moving global data; OOP encapsulates and secures data.
- Focus: POP focuses on algorithmic steps; OOP focuses on data representation and modeling.
- Extensibility: Adding new data/functions is difficult in POP, but easy in OOP.



Here is a summary comparing the two paradigms. Notice how OOP is essentially a bottom-up approach: we design our base components, or objects, first, and then build the system by having these objects interact. Adding new features in OOP is generally as simple as adding a new object or class, whereas in POP it might require restructuring the whole program.

Basic Concept 1: Classes

- A Class is a user-defined data type.
- It serves as a blueprint, template, or prototype for creating objects.
- Contains data members (attributes/properties) and member functions (methods/behaviors).
- Does not allocate memory when defined; memory is only allocated when objects are instantiated.
- Example: A 'Car' class with attributes (color, model) and methods (accelerate(), brake()).



Let's dive into the core concepts of OOP, starting with Classes. Think of a Class as an architectural blueprint for a house. The blueprint itself isn't a house—you can't live in it—but it defines exactly how the house should be built. In C++, a class defines what data and behaviors an object will eventually have.

Basic Concept 2: Objects

- An Object is a specific instance of a Class.
- It is the basic run-time entity in an object-oriented system.
- Objects take up physical space in memory and have an associated address.
- Objects interact with each other to perform system tasks.
- Example: 'myMustang' could be a specific object instantiated from the 'Car' class.

Basic Concept 2: Objects
- An Object is a specific instance of a Class.
- It is the basic run-time entity in an object-oriented system.
- Objects take up physical space in memory and have an associated address.
- Objects interact with each other to perform system tasks.
- Example: 'myMustang' could be a specific object instantiated from the 'Car' class.

If the Class is the blueprint, the Object is the actual house built from that blueprint. In our running program, objects are the entities that actually do the work. When we create an object, the system allocates memory for it. In a banking system, your specific bank account is an object, instantiated from a general 'Account' class.

Basic Concept 3: Data Abstraction

- Abstraction refers to representing essential features without including background details or complex explanations.
- Classes use abstraction to define a list of abstract attributes and functions.
- Helps manage complexity by hiding unnecessary implementation details from the end user.
- Example: Driving a car. You know how to use the steering wheel and pedals, but you don't need to understand the engine's internal combustion process.

Basic Concept 3: Data Abstraction
- Abstraction refers to representing essential features without including background details or complex explanations.
- Classes use abstraction to define a list of abstract attributes and functions.
- Helps manage complexity by hiding unnecessary implementation details from the end user.
- Example: Driving a car. You know how to use the steering wheel and pedals, but you don't need to understand the engine's internal combustion process.

Abstraction is about showing only what is necessary. When you use a smartphone, you interact with the touchscreen interface. You don't need to understand the complex circuitry or the operating system's kernel to make a call. In OOP, we expose a simple interface to the user of the class while keeping the complicated logic behind the scenes.

Basic Concept 4: Encapsulation

- Encapsulation is the wrapping up of data and functions into a single logical unit (the class).
- Data is not accessible to the outside world; only the functions wrapped inside the class can access it.
- This insulation of data from direct outside access is called 'Data Hiding' or 'Information Hiding'.
- Enforced in C++ using access specifiers: private, protected, and public.

Basic Concept 4: Encapsulation
- Encapsulation is the wrapping up of data and functions into a single logical unit (the class).
- Data is not accessible to the outside world; only the functions wrapped inside the class can access it.
- This insulation of data from direct outside access is called 'Data Hiding' or 'Information Hiding'.
- Enforced in C++ using access specifiers: private, protected, and public.

While abstraction is the concept of hiding complexity, encapsulation is the implementation mechanism that provides security. We encapsulate data by wrapping it inside a class and making it private. The only way to interact with that data is through public functions (methods) provided by the class. This protects the internal state from accidental corruption.

Abstraction vs. Encapsulation

- Abstraction: Focuses on 'what' an object does.
- Abstraction: Represents the outside view of an object (Design level).
- Abstraction: Solves the problem of complexity.
- Encapsulation: Focuses on 'how' an object does it.
- Encapsulation: Represents the inside view of an object (Implementation level).
- Encapsulation: Solves the problem of data security.

Abstraction vs. Encapsulation
- Abstraction: Focuses on 'what' an object does.
- Abstraction: Represents the outside view of an object (Design level).
- Abstraction: Solves the problem of complexity.
- Encapsulation: Focuses on 'how' an object does it.
- Encapsulation: Represents the inside view of an object (Implementation level).
- Encapsulation: Solves the problem of data security.

Students often confuse Abstraction and Encapsulation. Remember this distinction: Abstraction is a design-level concept focused on solving complexity by hiding irrelevant details. Encapsulation is an implementation-level concept focused on solving security by restricting access. Abstraction provides the user interface, while Encapsulation protects the internal state.

Basic Concept 5: Inheritance

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- Supports the concept of hierarchical classification (e.g., Kingdom, Phylum, Class, Order).
- Provides massive code reusability: add additional features to an existing class without modifying the original code.
- Terminology: Base Class (Parent) -¿ Derived Class (Child).
- Example: Class 'Robin' inherits from Class 'Bird', which inherits from Class 'Animal'.

Basic Concept 5: Inheritance
- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- Supports the concept of hierarchical classification (e.g., Kingdom, Phylum, Class, Order).
- Provides massive code reusability: add additional features to an existing class without modifying the original code.
- Terminology: Base Class (Parent) -¿ Derived Class (Child).
- Example: Class 'Robin' inherits from Class 'Bird', which inherits from Class 'Animal'.

Inheritance models the 'is-a' relationship. A Robin 'is a' Bird. Instead of writing the exact same code for flying, eating, and sleeping for every single type of bird, we can write a base class 'Bird' with these features. Specific bird classes then inherit these features and only add the characteristics unique to them.

Basic Concept 6: Polymorphism

- Polymorphism originates from Greek, meaning 'many forms' (poly = many, morph = forms).
- It is the ability of a message or function to be displayed or executed in more than one form.
- An operation may exhibit different behaviors depending upon the types of data used.
- Types in C++: Compile-time (Function/Operator Overloading) and Run-time (Virtual Functions).
- Example: The '+' operator adds numbers (5+5=10) but concatenates strings ('A'+'B'='AB').

Basic Concept 6: Polymorphism
- Polymorphism originates from Greek, meaning 'many forms' (poly = many, morph = forms).
- It is the ability of a message or function to be displayed or executed in more than one form.
- An operation may exhibit different behaviors depending upon the types of data used.
- Types in C++: Compile-time (Function/Operator Overloading) and Run-time (Virtual Functions).
- Example: The '+' operator adds numbers (5+5=10) but concatenates strings ('A'+'B'='AB').

Polymorphism allows us to use a single interface for a general class of actions. The specific action executed is determined by the exact nature of the situation. In C++, we see this through function overloading and operator overloading at compile time, and virtual functions at run time. It makes a system highly flexible and intuitive.

Basic Concept 7: Dynamic Binding

- Binding refers to the linking of a procedure call to the specific code to be executed.
- Static Binding: The compiler links the function call at compile-time.
- Dynamic (Late) Binding: The code associated with a procedure call is not known until run-time.
- Heavily associated with polymorphism and inheritance.
- Implemented in C++ using 'Virtual Functions'.

Basic Concept 7: Dynamic Binding
- Binding refers to the linking of a procedure call to the specific code to be executed.
- Static Binding: The compiler links the function call at compile-time.
- Dynamic (Late) Binding: The code associated with a procedure call is not known until run-time.
- Heavily associated with polymorphism and inheritance.
- Implemented in C++ using 'Virtual Functions'.

In traditional POP programming, the compiler figures out exactly which function to call when it compiles the code. This is static binding. Dynamic binding defers this decision until the program is actually running. If you have a generic pointer to an Animal, but at runtime it points to a Dog, dynamic binding ensures the Dog's specific 'speak' function is called.

Basic Concept 8: Message Passing

- Objects communicate with one another by sending and receiving information, similar to how people pass messages.
- A 'message' sent to an object is essentially a request for the execution of a procedure (method).
- Message passing involves three elements: the name of the object, the name of the function, and the parameters.
- Example syntax: employeeObject.calculateSalary(hoursWorked);

Basic Concept 8: Message Passing
- Objects communicate with one another by sending and receiving information, similar to how people pass messages.
- A 'message' sent to an object is essentially a request for the execution of a procedure (method).
- Message passing involves three elements: the name of the object, the name of the function, and the parameters.
- Example syntax: employeeObject.calculateSalary(hoursWorked);

In OOP, objects are not isolated entities. They interact to build a complete system. When one object wants another object to perform a task, it invokes a method of that target object. This is termed 'message passing'. The sender doesn't need to know how the receiver executes the task, only that it will respond to the message correctly.

Benefits of Object-Oriented Programming

- Reusability: Classes can be built, tested, and shared across multiple projects seamlessly.
- Security: Data hiding prevents unauthorized access and accidental state modification.
- Maintainability: A modular, object-based structure makes debugging and upgrading much easier.
- Scalability: Large-scale projects remain manageable by dividing them into interacting objects.
- Mapping: Provides a highly intuitive way to map real-world problems directly into code.

Benefits of Object-Oriented Programming
- Reusability: Classes can be built, tested, and shared across multiple projects seamlessly.
- Security: Data hiding prevents unauthorized access and accidental state modification.
- Maintainability: A modular, object-based structure makes debugging and upgrading much easier.
- Scalability: Large-scale projects remain manageable by dividing them into interacting objects.
- Mapping: Provides a highly intuitive way to map real-world problems directly into code.

Why do we go through the trouble of learning the OOP paradigm? Because it scales beautifully. As software systems grow into millions of lines of code, OOP keeps the codebase manageable. We can assign different classes to different teams of developers, and we can update a class without breaking the entire system, provided the interface remains intact.

Summary of 1.1 Overview of OOP

- POP focuses on algorithmic functions, which leads to structural issues in large applications.
- OOP shifts the focus to modeling real-world entities using encapsulated data and behaviors.
- Classes serve as blueprints; Objects serve as runtime instances.
- Abstraction and Encapsulation provide architectural simplicity and data security.
- Inheritance and Polymorphism enable code reuse and flexible interface design.
- Dynamic Binding and Message Passing allow for complex, runtime object interactions.

Summary of 1.1 Overview of OOP
- POP focuses on algorithmic functions, which leads to structural issues in large applications.
- OOP shifts the focus to modeling real-world entities using encapsulated data and behaviors.
- Classes serve as blueprints; Objects serve as runtime instances.
- Abstraction and Encapsulation provide architectural simplicity and data security.
- Inheritance and Polymorphism enable code reuse and flexible interface design.
- Dynamic Binding and Message Passing allow for complex, runtime object interactions.

To wrap up, today we covered the philosophical shift from Procedure-Oriented to Object-Oriented Programming. We defined the eight foundational pillars: Classes, Objects, Abstraction, Encapsulation, Inheritance, Polymorphism, Dynamic Binding, and Message Passing. Mastering these concepts is crucial, as they will dictate how you architect C++ software for the rest of your career.