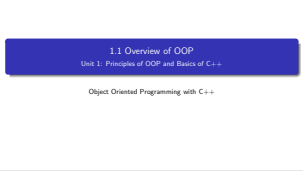
1.1 Overview of OOP

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

Object Oriented Programming with C++



Welcome to Object-Oriented Programming with C++

- Unit 1: Principles of OOP and Basics of C++
- Lecture 1.1: Overview of OOP
- Instructor: Expert C++ Faculty

Welcome to Object-Oriented Programming with C++
- Unit 1: Principles of OOP and Basics of C++
- Lecture 1.1: Overview of OOP
- Instructor: Expert C++ Faculty

Welcome, everyone, to the first lecture of Object-Oriented Programming with C++. Today we are going to lay the conceptual foundation for the entire course. Before we even look at a single line of C++ code, it is absolutely essential to understand the 'why' behind OOP. Why was it invented? What problems does it solve? By the end of today's session, you will understand the fundamental paradigm shift from writing traditional, step-by-step procedures to modeling software as a collection of interacting objects.

The Evolution of Programming Paradigms

- Machine Language & Assembly: Directly manipulating hardware, very complex.
- Unstructured Programming: Spaghetti code, lots of 'goto' statements, hard to maintain.
- Procedure-Oriented Programming (POP): e.g., C, Pascal. Focus on functions and algorithms.
- Object-Oriented Programming (OOP): e.g., C++, Java. Focus on data and objects.
- Software complexity drives the evolution of paradigms.

The Evolution of Programming Paradigms
- Machine Language & Assembly: Directly manipulating hardware, very complex.
- Unstructured Programming: Spaghetti code, lots of 'goto' statements, hard to maintain.
- Procedure-Oriented Programming (POP): e.g., C, Pascal. Focus on functions and algorithms.
- Object-Oriented Programming (OOP): e.g., C++, Java. Focus on data and objects.
- Software complexity drives the evolution of paradigms.

To appreciate OOP, we have to look at history. Early programming was done in machine language or assembly—very close to the hardware. As programs grew, we moved to unstructured languages, which quickly became unmaintainable 'spaghetti code'. Then came Procedure-Oriented Programming, like the C language, which organized code into functions. However, as software systems became massive in the 1980s, POP started to show limitations, especially in managing and protecting data. This necessity birthed Object-Oriented Programming, which revolutionized how we structure code by focusing on 'things' rather than just 'actions'.

Procedure-Oriented Programming (POP)

- Primary focus is on functions (procedures/algorithms) rather than data.
- Large programs are divided into smaller programs known as functions.
- Most functions share global data.
- Data moves freely around the system from function to function.
- Employs a top-down approach in program design.

Procedure-Oriented Programming (POP)
- Primary focus is on functions (procedures/algorithms) rather than data.
- Large programs are divided into smaller programs known as functions.
- Most functions share global data.
- Data moves freely around the system from function to function.
- Employs a top-down approach in program design.

Let's talk about Procedure-Oriented Programming. If you've programmed in C, you've used POP. In POP, you view a problem as a sequence of things to be done: reading, calculating, and printing. The focus is on the 'doing'. The code is chopped up into functions. But here's the catch: to let these functions communicate, we often use global variables. When data is global, it is vulnerable. Any function can accidentally modify it, leading to bugs that are notoriously difficult to track down in large systems.

Object-Oriented Programming (OOP)

- Primary focus is on data rather than procedures.
- Programs are divided into entities known as Objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Follows a bottom-up approach in program design.

Object-Oriented Programming (OOP)
- Primary focus is on data rather than procedures.
- Programs are divided into entities known as Objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Follows a bottom-up approach in program design.

Enter Object-Oriented Programming. OOP flips the script. Instead of asking 'What does this program need to do?', we ask 'What things is this program manipulating?' We focus on the data first. We divide the system into 'Objects'. Crucially, an object contains both its data (attributes) and the functions that operate on that data (methods). By wrapping them together, we can hide the data from the outside world, protecting it from accidental modification. This is a bottom-up approach: you build small, robust objects, and then assemble them to form a complex system.

POP vs. OOP: A Direct Comparison

- Focus: POP focuses on functions; OOP focuses on data.
- Program Division: POP uses functions; OOP uses objects.
- Data Security: POP data is freely moving/global (less secure); OOP data is hidden (highly secure).
- Approach: POP is Top-Down; OOP is Bottom-Up.
- Real-world Modeling: POP is poor at real-world modeling; OOP excels at modeling real-world entities.
- Examples: POP (C, Pascal); OOP (C++, Java, Python).

POP vs. OOP: A Direct Comparison
- Focus: POP focuses on functions; OOP focuses on data.
- Program Division: POP uses functions; OOP uses objects.
- Data Security: POP data is freely moving/global (less secure); OOP data is hidden (highly secure).
- Approach: POP is Top-Down; OOP is Bottom-Up.
- Real-world Modeling: POP is poor at real-world modeling; OOP excels at modeling real-world entities.
- Examples: POP (C, Pascal); OOP (C++, Java, Python).

This slide summarizes the paradigm shift. In POP, functions are the kings, and data just serves them, often globally. In OOP, Data is the king, and it keeps its own loyal functions to protect and manage it. If you want to simulate a bank, using OOP, you create a 'Customer' object and an 'Account' object. It maps perfectly to how we perceive the real world. In POP, you'd just have a bunch of disconnected arrays and functions like 'calculateInterest()'. OOP provides a much more secure and scalable architecture.

Basic Concept 1: Objects

- Objects are the basic runtime entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data, etc.
- An object is an instance of a class.
- Objects take up space in memory and have an associated address.
- When a program is executed, the objects interact by sending messages to one another.

Basic Concept 1: Objects
- Objects are the basic runtime entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data, etc.
- An object is an instance of a class.
- Objects take up space in memory and have an associated address.
- When a program is executed, the objects interact by sending messages to one another.

Now let's dive into the core concepts of OOP, starting with Objects. An object is a real-world entity that exists at runtime. Think of your physical smartphone. It has data (battery level, screen brightness, contacts) and it has functions or behaviors (make a call, lower volume). In software, an object is a chunk of memory that holds state and behavior. When your C++ program runs, it is essentially bringing these objects to life in RAM, and they start talking to each other to get work done.

Basic Concept 2: Classes

- A class is a blueprint, template, or prototype from which objects are created.
- It is a user-defined data type.
- Once a class has been defined, we can create any number of objects belonging to that class.
- Analogy: A class is the architectural blueprint of a house; objects are the actual houses built from that blueprint.

Basic Concept 2: Classes
- A class is a blueprint, template, or prototype from which objects are created.
- It is a user-defined data type.
- Once a class has been defined, we can create any number of objects belonging to that class.
- Analogy: A class is the architectural blueprint of a house; objects are the actual houses built from that blueprint.

If an object is a specific smartphone, a Class is the engineering blueprint that defines what that model of smartphone is. A class does not take up memory for its data until an object is created. It is purely a definition, a new user-defined data type. You define a 'Car' class once, specifying that it has wheels and an engine, and then you can instantiate thousands of 'Car' objects in your program, each with its own specific color and speed. Classes organize our code.

Basic Concept 3: Data Abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attributes.
- Real-world example: Driving a car. You know how to use the steering wheel, accelerator, and brakes.
- You do not need to know the internal mechanism of the engine or the fuel injection system to drive.

Basic Concept 3: Data Abstraction
- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attributes.
- Real-world example: Driving a car. You know how to use the steering wheel, accelerator, and brakes.
- You do not need to know the internal mechanism of the engine or the fuel injection system to drive.

Abstraction is about managing complexity by hiding unnecessary details. When you drive a car, you don't need an engineering degree in combustion engines. You interact with a simple interface: the steering wheel and pedals. The car 'abstracts' away the complex engine mechanics. In C++, we design classes similarly. We provide the user of our class with simple functions (like 'depositMoney') and hide the complex database queries and network calls happening behind the scenes.

Basic Concept 4: Encapsulation

- Encapsulation is the wrapping up of data and functions into a single unit (called a class).
- It is the most striking feature of a class.
- The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- This insulation of data from direct access by the program is called 'data hiding' or 'information hiding'.

Basic Concept 4: Encapsulation
- Encapsulation is the wrapping up of data and functions into a single unit (called a class).
- It is the most striking feature of a class.
- The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- This insulation of data from direct access by the program is called 'data hiding' or 'information hiding'.

While abstraction is the concept of hiding complexity, Encapsulation is the physical mechanism that achieves it. Encapsulation is binding the data (variables) and the functions (methods) that manipulate them into a single secure capsule—the class. It creates a protective shield. Because the data is locked inside, we prevent external code from accidentally corrupting it. In C++, we use access specifiers like 'private' and 'public' to enforce encapsulation. This is what fundamentally solves the 'global data' problem of POP.

Basic Concept 5: Inheritance

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- It supports the concept of hierarchical classification.
- Promotes Reusability: We can add additional features to an existing class without modifying it.
- Example: A 'Robin' is a part of the class 'Flying Bird', which is a part of the class 'Bird'.

Basic Concept 5: Inheritance
- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- It supports the concept of hierarchical classification.
- Promotes Reusability: We can add additional features to an existing class without modifying it.
- Example: A 'Robin' is a part of the class 'Flying Bird', which is a part of the class 'Bird'.

Inheritance is how OOP achieves extreme code reusability. Imagine you have a class called 'Vehicle' with generic attributes like speed and weight. Now you need a 'Car' and a 'Truck'. Instead of writing them from scratch, 'Car' and 'Truck' can inherit from 'Vehicle'. They automatically get all of Vehicle's properties, and you only need to write the new, specific code for them (like trunk space for a Car, or payload capacity for a Truck). It creates a logical, parent-child hierarchy in your software.

Basic Concept 6: Polymorphism

- Polymorphism is a Greek term that means the 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 ($5 + 5 = 10$) or concatenate strings ('Hello' + 'World' = 'HelloWorld').

Basic Concept 6: Polymorphism
- Polymorphism is a Greek term that means the 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 ($5 + 5 = 10$) or concatenate strings ('Hello' + 'World' = 'HelloWorld').

Polymorphism means 'many forms'. It allows us to use a single interface for different underlying forms (data types). Think of the 'draw()' command. If I send the 'draw' message to a Circle object, it draws a circle. If I send exactly the same 'draw' message to a Triangle object, it draws a triangle. The command is the same, but the object determines how to execute it. In C++, this is implemented through overloading and virtual functions, which we will explore deeply later in the course.

Basic Concept 7: Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding (or late binding) means that the code associated with a given procedure call is not known until the time of the call at runtime.
- It is closely associated with polymorphism and inheritance.
- Allows a pointer of a base class to call the overridden function of a derived class.

Basic Concept 7: Dynamic Binding
- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding (or late binding) means that the code associated with a given procedure call is not known until the time of the call at runtime.
- It is closely associated with polymorphism and inheritance.
- Allows a pointer of a base class to call the overridden function of a derived class.

Dynamic binding is an advanced concept that makes polymorphism work at runtime. In traditional compiling, the compiler links a function call to a specific block of memory immediately. That's static binding. But with Dynamic Binding, the compiler doesn't know exactly which function will be called until the program is actually running. The decision is deferred to runtime based on the actual object being manipulated. This provides incredible flexibility in designing plug-and-play software architectures.

Basic Concept 8: Message Passing

- An OOP program consists of a set of objects that communicate with each other.
- The process of programming involves:
- 1. Creating classes that define objects and their behaviors.
- 2. Creating objects from class definitions.
- 3. Establishing communication among objects.
- A message for an object is a request for execution of a procedure.

Basic Concept 8: Message Passing
- An OOP program consists of a set of objects that communicate with each other.
- The process of programming involves:
- 1. Creating classes that define objects and their behaviors.
- 2. Creating objects from class definitions.
- 3. Establishing communication among objects.
- A message for an object is a request for execution of a procedure.

Finally, how do these objects actually do work? Through Message Passing. In OOP, you don't 'call a function', you 'send a message to an object'. A message involves the name of the object, the name of the function (message), and any information needed (parameters). For example, `Employee.calculateSalary(month)`. The `Employee` is the object, `calculateSalary` is the message, and `month` is the parameter. This models how entities interact in the real world.

Putting It All Together: A Real-World Analogy

- Class: The architectural blueprint of a bank.
- Object: Your specific bank account.
- Encapsulation: The bank vault protecting your money.
- Abstraction: The ATM interface (hides the complex banking software).
- Inheritance: 'Savings Account' and 'Checking Account' inherit from 'Generic Account'.
- Polymorphism: Swiping a card can mean 'debit' for one card and 'credit' for another.

Putting It All Together: A Real-World Analogy
- Class: The architectural blueprint of a bank.
- Object: Your specific bank account.
- Encapsulation: The bank vault protecting your money.
- Abstraction: The ATM interface (hides the complex banking software).
- Inheritance: 'Savings Account' and 'Checking Account' inherit from 'Generic Account'.
- Polymorphism: Swiping a card can mean 'debit' for one card and 'credit' for another.

To summarize, think of a banking system. The blueprint of an account is the Class. Your specific account is the Object. The fact that I cannot directly change your balance from my computer is Encapsulation. The fact that you just press 'Withdraw' on an ATM without knowing how the mainframe works is Abstraction. Both Savings and Checking accounts sharing standard features is Inheritance. And different cards processing differently when swiped is Polymorphism.

Advantages of OOP

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- The principle of data hiding helps build secure programs.
- Software complexity can be easily managed.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.

Advantages of OOP
- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- The principle of data hiding helps build secure programs.
- Software complexity can be easily managed.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.

Why go through the trouble of learning this? Because OOP offers massive advantages for real-world software engineering. By reusing code via inheritance, we save thousands of hours. By hiding data, we prevent disastrous bugs. Because we model code after the real world, large, complex systems are much easier for teams of programmers to conceptualize and manage. C++, being a powerful OOP language, allows you to reap all these benefits while maintaining high performance.

Lecture Summary

- We explored the evolution from Procedure-Oriented to Object-Oriented Programming.
- POP focuses on functions; OOP focuses on data and objects.
- We defined the core components: Objects (instances) and Classes (blueprints).
- We learned the 4 pillars: Abstraction (hiding complexity), Encapsulation (data hiding), Inheritance (reusability), and Polymorphism (many forms).
- We touched on Dynamic Binding and Message Passing.

Lecture Summary
- We explored the evolution from Procedure-Oriented to Object-Oriented Programming.
- POP focuses on functions; OOP focuses on data and objects.
- We defined the core components: Objects (instances) and Classes (blueprints).
- We learned the 4 pillars: Abstraction (hiding complexity), Encapsulation (data hiding), Inheritance (reusability), and Polymorphism (many forms).
- We touched on Dynamic Binding and Message Passing.

That wraps up our introductory lecture. We have covered a lot of theoretical ground today. You now know the difference between POP and OOP. You know that classes are blueprints and objects are real instances in memory. And most importantly, you should now be familiar with the four pillars: Abstraction, Encapsulation, Inheritance, and Polymorphism. These terms will be your vocabulary for the rest of your software engineering career.

Questions & Answers

- Any questions about POP vs OOP?
- Are the definitions of Class and Object clear?
- Do you have questions about the 4 main pillars (Abstraction, Encapsulation, Inheritance, Polymorphism)?

Questions & Answers
- Any questions about POP vs OOP?
- Are the definitions of Class and Object clear?
- Do you have questions about the 4 main pillars (Abstraction, Encapsulation, Inheritance, Polymorphism)?

I'll now open the floor to any questions. Are there any analogies that didn't quite make sense? Is the distinction between abstraction and encapsulation clear? Don't worry if you don't fully grasp how to write the code for these yet; right now, the conceptual understanding is what matters.

Next Lecture Preview

- Lecture 1.2: Introduction to C++ Language
- Structure of a C++ Program.
- Writing our first 'Hello World' in C++.
- Understanding I/O streams: cin and cout.
- Basic data types and variables in C++.

Next Lecture Preview
- Lecture 1.2: Introduction to C++ Language
- Structure of a C++ Program.
- Writing our first 'Hello World' in C++.
- Understanding I/O streams: cin and cout.
- Basic data types and variables in C++.

In our next lecture, we will move from theory to practice. We will look at the history of C++, understand the basic structure of a C++ program, and write our very first piece of code. We will learn how to take input from a user and display output to the screen using standard streams. Make sure to review today's concepts, as we will slowly start mapping them to actual C++ syntax in the coming weeks.