# Infographics

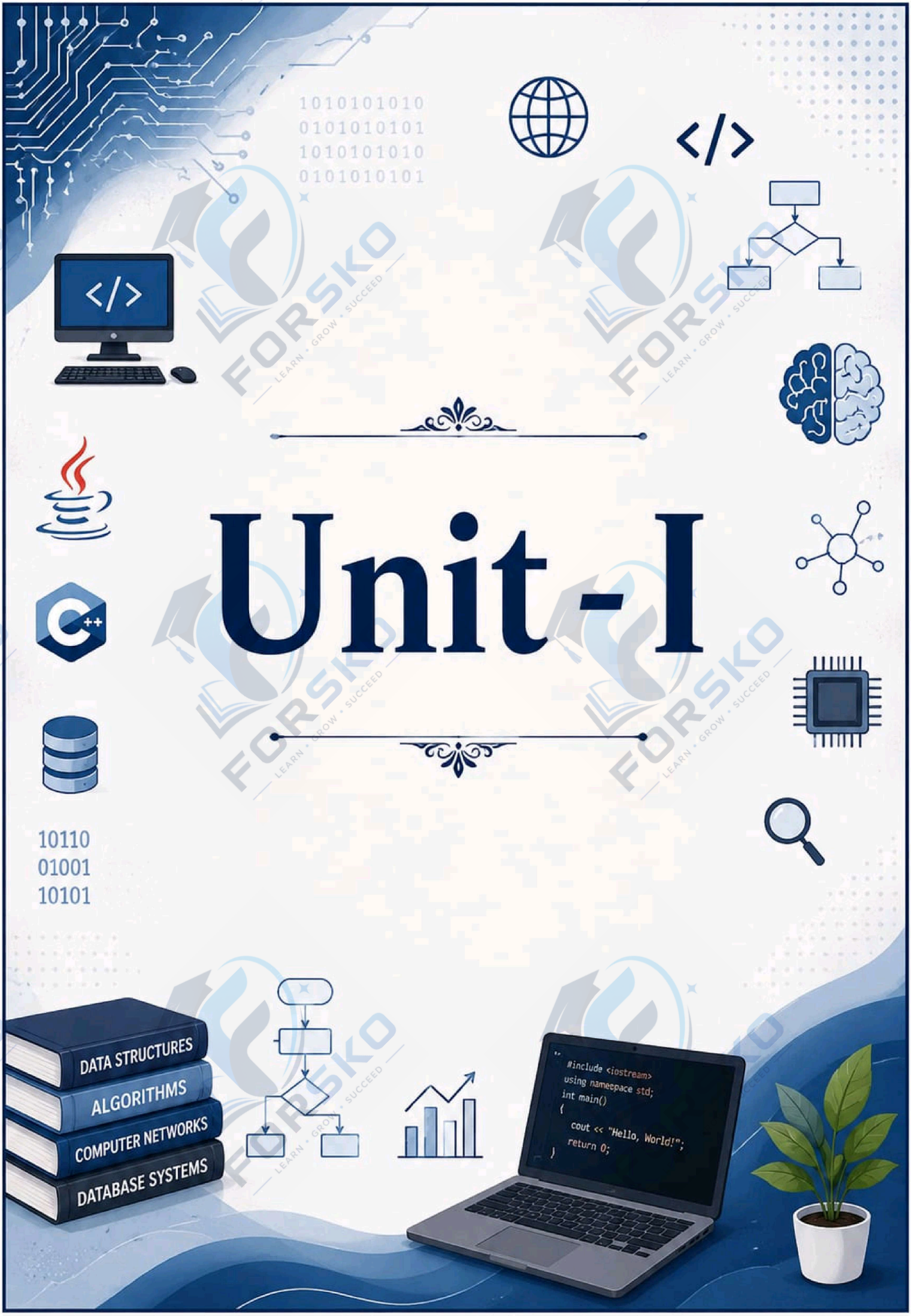
**Visual Learning**
Better understanding through visual content.

**Simplified Concepts**
Complex topics explained in simple and visual form.

**Quick Revision**
Easy to revise important points and diagrams.

**Exam Ready**
Helps in faster revision and better exam preparation.

# Unit-I


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using nameepace std;
int main()
{
cout << "Hello, World!";
return 0;
}

# OOP PARADIGM

## BASIC CONCEPTS

Object Oriented Programming (OOP) is a programming paradigm based on the concept of "Objects", which can contain data and code: data in the form of fields (often known as attributes) and code in the form of procedures (often known as methods).

## CORE BUILDING BLOCKS

### 01 CLASS

A class is a blueprint or template for creating objects. It defines the data (attributes) and behavior (methods) that the objects of the class will have.

```
class Car
{
   // attributes
   // methods
}
```

### 02 OBJECT

An object is an instance of a class. It represents a real-world entity and has its own identity, state (attributes) and behavior (methods).

```
Car c1 = new Car();
```

### 03 MEMBER

Members are the properties and functions defined inside a class.

- **Data Members (Attributes/Fields)**
- **Member Functions (Methods)**

## FOUR PILLARS OF OOP

### 01 ENCAPSULATION

Binding data (variables) and methods together in a single unit (class) and restricting direct access to some of the object's components.
It helps in data hiding and protects the data from unauthorized access.

### 02 ABSTRACTION

Hiding the internal details and showing only the essential features of an object to the user.
It reduces complexity and helps in focusing on what the object does, not how it does it.

### 03 INHERITANCE

A mechanism where a new class acquires the properties and behaviors (methods and fields) of an existing class.
It promotes code reusability and establishes an "is-a" relationship.

### 04 POLYMORPHISM

The ability of an object to take many forms.
The same method or operation can behave differently based on the object that invokes it.

## RELATIONSHIPS IN OOP

**Association**
A general relationship between two independent classes.

**Aggregation**
A special form of association where one class has a weak ownership of another.

**Composition**
A strong form of aggregation where one class completely owns the other.

**Dependency**
One class depends on another class for its functionality.

## OOP vs PROCEDURAL PROGRAMMING

| Feature | OOP | Procedural Programming |
|---|---|---|
| Approach | Bottom-up | Top-down |
| Focus | Objects | Procedures/Functions |
| Data Security | High (Data Hiding) | Low |
| Reusability | High (Inheritance) | Low |
| Maintainability | Easy | Difficult |
| Modularity | High | Low |
| Real World Mapping | Easy | Difficult |

## BENEFITS OF OOP

- Code Reusability
- Modularity
- Data Security
- Easy Maintenance
- Real World Modeling
- Flexibility and Scalability

OOP makes software development more efficient, scalable and easy to maintain by organizing code using objects and classes.

# OOP PARADIGM

## FEATURES, ADVANTAGES, APPLICATIONS OF OOP

## 1 FEATURES OF OOP

### 1. CLASS
A class is a blueprint or template that defines the properties (data members) and behaviors (methods) of objects.

### 2. OBJECT
An object is an instance of a class that has identity, state (attributes) and behavior (methods).

### 3. ENCAPSULATION
Binding data and methods together in a single unit (class) and restricting direct access to some components.

### 4. INHERITANCE
A mechanism where a new class acquires the properties and behaviors of an existing class.

### 5. ABSTRACTION
Hiding the internal details and showing only the essential features to the user.

### 6. POLYMORPHISM
The ability of an object to take many forms. The same interface behaves differently based on the object.

### 7. DYNAMIC BINDING
The method call is resolved at runtime based on the actual object.

### 8. MODULARITY
The program is divided into independent modules (classes) which makes development and maintenance easier.

## 2 ADVANTAGES OF OOP

**1. Reusability of Code**
Code can be reused through inheritance and class libraries.

**2. Modularity**
Program is divided into classes and objects, making it easy to manage.

**3. Security**
Data hiding (Encapsulation) protects data from unwanted access.

**4. Scalability**
OOP makes it easy to extend the functionality of programs.

**5. Easy Maintenance**
Changes in one class do not affect other classes.

**6. Real World Modeling**
OOP represents real world entities more naturally.

## 3 APPLICATIONS OF OOP

**1. Software Development**
Used in large scale software systems, IDEs, compilers, databases etc.

**2. Game Development**
Helps in creating realistic characters, objects and environments.

**3. Banking Systems**
Used in ATM systems, online banking, transaction processing systems.

**4. E-commerce Applications**
Used in shopping carts, product catalogs, payment systems etc.

**5. Mobile Applications**
Used in Android apps, iOS apps and cross-platform applications.

**6. Real-time Systems**
Used in robotics, simulation systems, embedded systems etc.

**7. Healthcare Systems**
Used in hospital management, patient records, appointment systems.

**8. Data Analysis & AI Applications**
Used in data mining, machine learning, neural networks etc.

OOP helps in building robust, flexible and maintainable applications by organizing code using objects and classes that model real world entities.

# OOP PARADIGM

## STRUCTURED VS OOP, TRENDING OOP LANGUAGES

OOP (**Object Oriented Programming**) organizes software around objects that combine data (attributes) and behavior (methods), making systems more modular, scalable, reusable and easier to maintain.

## 1 STRUCTURED VS OOP

| Aspect | Structured Programming | Object Oriented Programming (OOP) |
|---|---|---|
| Focus | Focuses on procedure (functions) and logic. | Focuses on objects that combine data and behavior. |
| Design Approach | Top-down approach. | Bottom-up approach. |
| Program Structure | Program is divided into functions. | Program is divided into classes and objects. |
| Data & Functions | Data is separate from functions and can be accessed by any function. | Data and functions are bundled together in a class and data is accessed through methods. |
| Data Security | Less secure. Data can be accessed by any part of the program. | More secure due to data hiding (Encapsulation). |
| Reusability | Function reusability is limited. | High reusability through inheritance and polymorphism. |
| Maintainability | Difficult to maintain in large programs. | Easy to maintain and modify. |
| Scalability | Not suitable for large and complex systems. | Highly scalable for large and complex systems. |
| Examples | C, Pascal | Java, C++, Python, C# |
| Real World Modeling | Difficult to represent real world entities. | Easy to model real world entities using objects and classes. |

## 2 TRENDING OOP LANGUAGES

### 1. JAVA
Platform independent, robust, secure and widely used in enterprise applications, Android development, etc.

### 2. C++
High performance, used in system/software development, games, embedded systems, etc.

### 3. PYTHON
Simple, readable and versatile. Used in web development, AI, data science, automation, etc.

### 4. C#
Developed by Microsoft. Used in Windows applications, games (Unity), web and enterprise apps.

### 5. PHP
Used mainly for web development. Works well with databases and server-side scripting.

### 6. JavaScript
Used for web development (frontend and backend with Node.js). Event-driven and versatile.

### 7. TypeScript
Superset of JavaScript. Adds static typing and is widely used in large scale web applications.

### 8. Kotlin
Modern language for Android development. Interoperable with Java and concise.

### 9. Swift
Developed by Apple. Used for iOS, macOS, watchOS and tvOS applications.

### 10. Go
Concurrent, fast and efficient. Used in cloud services, APIs and backend systems.

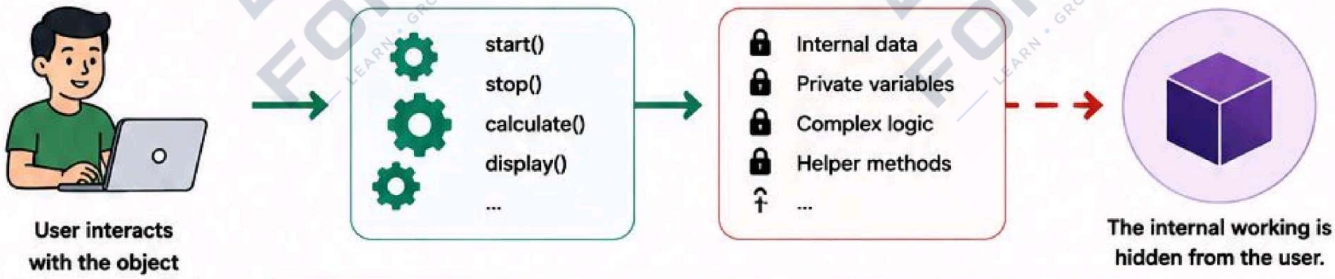
**SUMMARY** OOP makes programs modular, reusable, secure and easy to maintain. It is widely used in modern software development and is supported by many powerful programming languages.

# OOP CONCEPTS

## DATA ABSTRACTION

Data Abstraction is the process of **hiding** the implementation details (internal data and complexity) and showing only the **essential features** or functionality to the user.
It helps in reducing complexity and focusing on **what** an object does, not **how** it does it.

## 1. HOW DATA ABSTRACTION WORKS




Only **what** is necessary is shown to the user, not **how** it is implemented.

## 2. ACHIEVED USING

**Abstract Classes**
Declares abstract methods (without body) that must be implemented by subclasses.

**Interfaces**
Declares methods that the class must implement. **(100% abstraction)**

**Access Modifiers**
Use of private and protected to hide data and methods from outside the class.

## 3. EXAMPLE (JAVA)

```java
// Abstract class
abstract class Shape {
    // Abstract method (no body)
    abstract double area();
}

// Subclass
class Circle extends Shape {
    private double radius;   // Hidden data
    public Circle(double r) {
        radius = r;
    }
    // Implementation hidden from user
    public double area() {
        return 3.1416 * radius * radius;
    }
}

// Using the object
class Test {
  public static void main(String[] args) {
     Shape s = new Circle(5);
     System.out.println("Area: " + s.area());
  }
}
```

**What the user sees:**

- Shape s = new Circle(5);
- s.area();

---

**What the user does NOT see:**

- radius
- How area is calculated
- Internal logic

## 4. BENEFITS

**Reduces Complexity**
Hides complex implementation details from the user.

**Focus on What, Not How**
Users focus on what an object does instead of how it does.

**Increases Flexibility**
Internal implementation can be changed without affecting the user.

**Improves Security**
Hides sensitive data and prevents unauthorized access.

**Better Maintainability**
Easier to maintain and update the code.

## 5. REAL LIFE ANALOGY

**What you see:**

- ✓ Steering
- ✓ Accelerator
- ✓ Brake
- ✓ Dashboard

**What you don't see:**

- ✗ Engine internal parts
- ✗ Wiring
- ✗ Fuel injection system
- ✗ Complex machinery

You only use the features; the internal working is hidden from you.

**SUMMARY** Data Abstraction is a key OOP concept that hides unnecessary details and exposes only the essential features, making programs simpler, secure, flexible and easy to maintain.

OOP CONCEPT

# ENCAPSULATION: CLASSES AND OBJECTS

Encapsulation is the process of **bundling data (variables)** and **methods (functions)** that operate on the data into **a single unit (class)** and **restricting direct access** to some of the object's components. It helps in **data hiding** and protects the object's state from invalid or unauthorized access.

## 1 WHAT IS ENCAPSULATION?

**Data (Variables)**
- private data
- protected data
- sensitive data

+

**Methods (Functions)**
- set() methods (to update data)
- get() methods (to read data)

→

**Encapsulation**

Data and methods are wrapped together in a single unit – Class.

→

**Object**

Objects are created from classes and can access only what is allowed.

Encapsulation hides internal details and shows only the necessary parts.

## 2 CLASSES AND OBJECTS

### Class (Blueprint)

A class is a blueprint or template for creating objects. It contains variables and methods.

```
class Student {
    private int rollNo;      // private data (hidden)
    private String name;

    // public methods (interface)
    public void setRollNo(int r) {
        rollNo = r;          // update data
    }
    public int getRollNo() {
        return rollNo;       // read data
    public void setName(String n) {
        name = n;
    }
    public String getName() {
        return name;
}
```

### Object (Instance)

An object is an instance of a class. It uses the class members through public methods.

```
Student s1 = new Student();
```

```
s1.setRollNo(101);          s1.setName("Faizan");
(Update data)               (Update data)

int r = s1.getRollNo();     String n = s1.getName();
(Read data)                 (Read data)
```

Direct access to data is not allowed:

```
// s1.rollNo = 101;  ✗ (Not allowed)
// System.out.println(s1.name);  ✗ (Not allowed)
```

## 3 HOW ENCAPSULATION ACHIEVES DATA HIDING

### Without Encapsulation

Data can be accessed and modified directly.

```
class Student {
   int rollNo;  // public
   String name; // public
}

Student s = new Student();
s.rollNo = 101;     // allowed
s.name = "Faizan"; // allowed
```

→ **Encapsulation (Data Hiding)**

Protects data by restricting direct access and using methods.

→

### With Encapsulation

Data is hidden and accessed only through methods.

```
class Student {
   private int rollNo;
   private String name;
   public void setRollNo(int r) { rollNo = r; }
   public int getRollNo() { return rollNo; }
}
Student s = new Student();
s.setRollNo(101);      // allowed
int r = s.getRollNo(); // allowed
// s.rollNo = 101;     // not allowed
```

### Benefits

- Protects data from unauthorized access.
- Ensures data validity and consistency.
- Easy to maintain and modify code.
- Promotes reusability of code.

## 4 KEY POINTS

- Data members are declared private.
- Public methods (getters and setters) are used to access and modify data.
- Access modifiers (private, protected, public) help in encapsulation.
- Encapsulation is a pillar of OOP.

## 5 EXAMPLE (REAL LIFE ANALOGY)

Bank Account (Class) → You can use only allowed services →

- ✓ Deposit()
- ✓ Withdraw()
- ✓ CheckBalance()

(Public Methods)

You cannot access directly
- ✗ Balance
- ✗ Account Number
- ✗ Internal Records

(Private Data)

**SUMMARY:** Encapsulation binds data and methods together in a class and hides internal details from the outside world. It provides data hiding, protects data, and helps in building secure and reliable applications.

OOP CONCEPTS

# INTRODUCTION AND DEFINING A CLASS

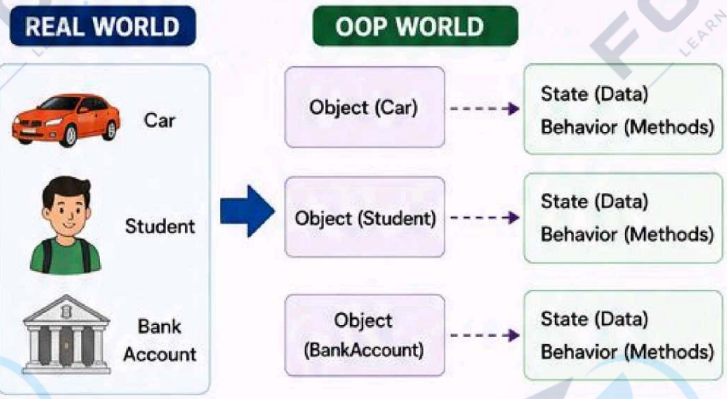
Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**. It helps in organizing software design around data, or objects, rather than functions and logic. OOP makes programs **modular**, **reusable**, **easy to maintain** and **extend**.

## 1 INTRODUCTION

OOPs is based on real-world entities called **objects**.

An object has:

- **State** (data/attributes)
- **Behavior** (methods/functions)
- **Identity** (unique name or reference)




### BENEFITS OF OOP

- ✓ Code reusability
- ✓ Modularity
- ✓ Data hiding & security
- ✓ Easy maintenance
- ✓ Scalability
- ✓ Real-world modeling

## 2 DEFINING A CLASS

A class is a **blueprint** or **template** for creating objects. It defines the **data (attributes)** and the **methods (functions)** that will be common to all objects of that class.

### SYNTAX (in Java)

```
class ClassName {
   // Data members (attributes)
   type variable1;
   type variable2;

   // Methods (functions)
   returnType methodName() {
      // body of method
   }
   returnType methodName2() {
      // body of method
   }
}
```

### EXPLANATION

- **class** – keyword used to define a class.
- **ClassName** – name of the class (should start with a capital letter).
- **Attributes** – variables that hold the data.
- **Methods** – functions that define behavior.

### EXAMPLE: Defining a Class

```
class Student {
   // Attributes
   int rollNo;
   String  name;
   double  marks;

   // Method to display student details
   void display() {
      System.out.println("Roll No: " + rollNo);
      System.out.println("Name: " + name);
      System.out.println("Marks: " + marks);
   }
}
```

### KEY POINTS ABOUT A CLASS

- ✓ A class does not occupy memory.
- ✓ It is a logical entity.
- ✓ It acts as a blueprint for objects.
- ✓ Objects are created from a class.

### CLASS vs OBJECT

| Class | Object |
|---|---|
| Blueprint or template | Instance of a class |
| Logical entity | Physical entity |
| Does not occupy memory | Occupies memory |
| Declared once | Created many times |
| Example: Student | Example: s1, s2 (objects of Student) |

### FLOW: CLASS TO OBJECT

Define a Class
↓
Create Objects
↓
Access Data via Methods
↓
Perform Operations

**SUMMARY** A class is the foundation of OOP. It defines data and behavior. Objects are created from classes to represent real-world entities and perform operations.

# Unit - II


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

OOP CONCEPTS

# FUNCTIONS: FUNCTION PROTOTYPE, INLINE FUNCTION

Functions are block of code that perform a specific task. They help in **modularity**, **code reusability** and better program organization.

## 1 FUNCTION PROTOTYPE

**A function prototype (or function declaration)** tells the compiler about the function's **name**, **return type** and **parameters** before it is used in the program.

### SYNTAX

```
returnType functionName(param1_type,
        param2_type, ...);

// Example
int  add(int, int);
```

No body of the function is written in prototype (ends with ; )

### WHY USE PROTOTYPES?

- Allows calling a function before it is defined.
- Improves program readability and modularity.
- Necessary when function definition is placed after `main()`.
- Helps in separate compilation (large programs).

### EXAMPLE

```
#include <iostream>
using namespace std;
// Function Prototype
int add(int a, int b);

int main() {
    int x = 10, y = 20;
    int sum = add(x, y);  // calling
    cout << "Sum = " << sum;
    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;
}
```

Prototype ≠ Definition → Prototype ends with ';' and has no body. Definition has body of the function.

## 2 INLINE FUNCTION

An **inline function** is a function whose code is **expanded (copied)** at the point of call instead of a normal function call.

### SYNTAX

```
inline returnType functionName(parameters) {
    // body of the function
    return value;  // if return type is not void
}
```

**Example**

```
inline int square(int n) {
    return n * n;
}
```

### HOW INLINE FUNCTION WORKS

Function Call → Compiler → Copy of function body is inserted at call location → Execution

### EXAMPLE

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

// Inline Function
inline int cube(int n) {
    return n * n * n;
}
int main() {
    int x = 3;
    int result = cube(x);   // inline call
    cout << "Cube = " << result;
    return 0;
}
```

### ADVANTAGES

- Reduces function call overhead (no call/return instructions).
- Increases program speed for small functions.
- Useful for short, frequently used functions.
- Improves performance in time-critical applications.

### DISADVANTAGES

- Increases code size (function code is copied every time).
- Not suitable for large functions (leads to memory waste).
- May decrease performance if code size grows too much.

### FUNCTION PROTOTYPE vs INLINE FUNCTION

| Feature | Function Prototype | Inline Function |
|---|---|---|
| Purpose | Declares a function before use | Suggests compiler to expand function at call |
| Contains Body? | No | Yes |
| Ends With | ; (semicolon) | { } (function body) |
| When Used | When function is defined later | For small, frequently used functions |
| Effect | No performance change | Can improve execution speed (reduces call overhead) |

### WHEN TO USE INLINE FUNCTION?

- Function is small and simple.
- Function is called many times.
- Performance is critical.
- The function is in header files (commonly in C++).

**Summary:** Function prototype helps the compiler understand a function before it is defined. Inline function helps improve performance by expanding the code at the point of call.

OOP CONCEPTS

# FUNCTIONS: DEFAULT ARGUMENT, FUNCTION OVERLOADING

Functions make code modular and reusable. **Default Arguments** and **Function Overloading** provide flexibility in defining and using functions.

## 1 DEFAULT ARGUMENT

A **default argument** is a value assigned to a function parameter in function declaration. If argument is not passed during function call, the **default value** is used.

### SYNTAX

```
returnType functionName(
    dataType param1,
    dataType param2 = defaultValue,
    dataType param3 = defaultValue
    ...
);
```

- Default arguments are specified from right to left.
- Once a parameter has a default value, all parameters to its right must also have default values.

### EXAMPLE

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

// Function with default arguments
int power(int base, int exp = 2)
{
    int result = 1;
    for(int i = 0; i < exp; i++)
        result *= base;
    return result;
}

int main()
{
    cout << power(3) << endl;     // 3^2 = 9
    cout << power(3, 3) << endl; // 3^3 = 27
    return 0;
}
```

### OUTPUT

```
9
27
```

### KEY POINTS

- Simplifies function calls.
- Improves code readability.
- Useful when a parameter has a common value.
- Default value is used only when argument is omitted.

## 2 FUNCTION OVERLOADING

Function overloading allows multiple functions to have the **same name** but different parameter lists (number, type or order of parameters).

### SYNTAX

```
returnType functionName(type1 a, ...);
returnType functionName(type1 a, type2 b, ...);
returnType functionName(type2 a, type1 b, ...);
...
```

### EXAMPLE

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

// Overloaded functions
int add(int a, int b)
{
    return a + b;     // two integers
}
double add(double a, double b)
{
    return a + b;     // two doubles
}
int add(int a, int b, int c)
{
    return a + b + c;  // three integers
}

int main()
{
    cout << add(10, 20) << endl;       // calls add(int, int)
    cout << add(5.5, 2.5) << endl;     // calls add(double, double)
    cout << add(1, 2, 3) << endl;      // calls add(int, int, int)
    return 0;
}
```

### OUTPUT

```
30
8
6
```

### HOW IT WORKS?

Function Call
e.g. add(10, 20)

↓

Compiler checks the arguments (number, type, order)

↓

Matches with the best suitable function

↓

Executes the matched function

### RULES FOR OVERLOADING

- Functions must have the same name.
- Parameter list must be different in number, type or order.
- Return type alone cannot be used to differentiate functions.

### DEFAULT ARGUMENT vs OVERLOADING

| Aspect | Default Argument | Function Overloading |
|---|---|---|
| Purpose | Provides a default value for a parameter. | Allows multiple functions with same name. |
| Based on | Missing arguments. | Different parameter lists. |
| When Used | When some arguments are optional. | When functions perform similar tasks on different inputs. |
| Example | `int fun(int a, int b = 10)` | `int fun(int a)`<br>`int fun(int a, int b)` |
| Benefit | Simplifies function call. | Increases flexibility. |

### IMPORTANT NOTE

- Default arguments are resolved at compile time.
- Overloaded functions are resolved at compile time (static polymorphism).
- Both features improve code reusability and reduce redundancy.

**SUMMARY** **Default arguments** allow optional parameters with preset values.
**Function overloading** allows multiple functions with the same name but different parameter lists.
Both enhance flexibility, readability and reusability of programs.

OOP CONCEPTS

# CONSTRUCTORS: TYPES OF CONSTRUCTORS

A **constructor** is a special member function of a class that is used to **initialize objects**.

- ✓ It has the same name as the class.
- ✓ It has no return type (not even void).
- ✓ It is called automatically when an object is created.
- ✓ It is used to assign initial values to data members.

## TYPES OF CONSTRUCTORS

### 1 Default (No-Argument) Constructor

A constructor that takes no parameters.
If we don't create any constructor, the compiler provides a default constructor automatically.

```
class Student {
    int rollNo;
    String name;

    // Default constructor
    Student() {
        rollNo = 0;
        name = "Unknown";
    }
}

class Test {
   public static void main(String[] args) {
      Student s = new Student();  // calls default constructor
      System.out.println(s.rollNo + " " + s.name);
   }
}
```

**Output**

```
0 Unknown
```

**Note:**
Compiler provides default constructor only if no other constructor is defined.

### 2 Parameterized Constructor

A constructor that takes one or more parameters.
Used to initialize objects with specific values.

```
class Student {
    int rollNo;
    String name;
    int age;

    // Parameterized constructor
    Student(int r, String n, int a) {
        rollNo = r;
        name = n;
        age = a;
    }
}

class Test {
   public static void main(String[] args) {
      Student s = new Student(101, "Aman", 19);
      System.out.println(s.rollNo + " " + s.name + " " + s.age);
   }
}
```

**Output**

```
101 Aman 19
```

**Note:**
We must create parameterized constructor if we want to initialize objects with values.

### 3 Default Constructor (Compiler Provided)

If a class has no constructor, the compiler automatically provides a no-argument default constructor.

```
class Car {
    String color;
    int price;
}
class Test {
   public static void main(String[] args) {
      Car c = new Car();  // Compiler provides Car()
      System.out.println(c.color); // null
      System.out.println(c.price); // 0
   }
}
```

**Output**

```
null
0
```

This constructor is provided by the compiler only when we do not define any constructor.

### 4 Copy Constructor (User Defined)

A constructor that creates an object by copying the values from another object of the same class.

```
class Student {
    int rollNo;
    String name;

    Student(int r, String n) {
        rollNo = r;
        name = n;
    }

    // Copy constructor
    Student(Student s) {
        rollNo = s.rollNo;
        name = s.name;
    }
}
class Test {
   public static void main(String[] args) {
      Student s1 = new Student(101, "Aman");
      Student s2 = new Student(s1);  // copy constructor
      System.out.println(s2.rollNo + " " + s2.name);
   }
}
```

**Output**

```
101 Aman
```

**Note:**
Java does not provide copy constructor by default.
It must be defined explicitly.

### 5 Private Constructor

A private constructor prevents the creation of objects from outside the class. Useful for classes that contain only static members (e.g., Utility classes).

```
class MathUtil {
    private MathUtil() {
        // private constructor
    }
    public static int add(int a, int b) {
        return a + b;
    }
}
class Test {
    public static void main(String[] args) {
        // MathUtil m = new MathUtil();  // Error
        System.out.println(MathUtil.add(5, 3));
    }
}
```

**Output**

```
8
```

**Note:**
Objects cannot be created. Only static methods can be used.

### 6 Dynamic (Runtime) Constructor – via this()

A constructor can call another constructor of the same class using this(). This helps in constructor chaining.

```
class Student {
    int rollNo;
    String name;

    Student() {
        this(0, "Unknown");  // calling parameterized constructor
    }

    Student(int r, String n) {
        rollNo = r;
        name = n;
    }
}

class Test {
   public static void main(String[] args) {
      Student s = new Student();  // calls Student(), which calls Student(int,String)
      System.out.println(s.rollNo + " " + s.name);
   }
}
```

**Output**

```
0 Unknown
```

**Note:**
this() must be the first statement in a constructor.

## QUICK COMPARISON

| Constructor Type | Parameters | Provided By | Purpose | Example Call |
|---|---|---|---|---|
| Default (No-Arg) | None | User / Compiler (if none defined) | Initializes with default values | `new Student();` |
| Parameterized | One or more | User | Initializes with given values | `new Student(101, "Aman", 19);` |
| Compiler Provided Default | None | Compiler | Provided automatically if no constructor is defined | `new Car();` |
| Copy Constructor | One object of same class | User | Creates object by copying another object | `new Student(s1);` |
| Private Constructor | None | User | Prevents object creation | (Cannot call from outside) |
| Dynamic (via this()) | Varies | User | Calls another constructor in same class | `this(...);` |

**SUMMARY** Constructors are used to initialize objects. Different types of constructors provide flexibility and control over object creation in OOP.

OOP CONCEPTS

# CONSTRUCTORS.

## DEFAULT, PARAMETERIZED AND COPY CONSTRUCTOR

**A constructor** is a special member function of a class that is used to initialize objects.

- ✔ It has the same name as the class.
- ✔ It is called automatically when an object is created.
- ✔ It has no return type (not even void).
- ✔ It is used to assign initial values to data members.

## 1 DEFAULT (NO-ARGUMENT) CONSTRUCTOR

A constructor that takes **no parameters**.
If we don't create any constructor, the compiler provides a **default constructor** automatically.

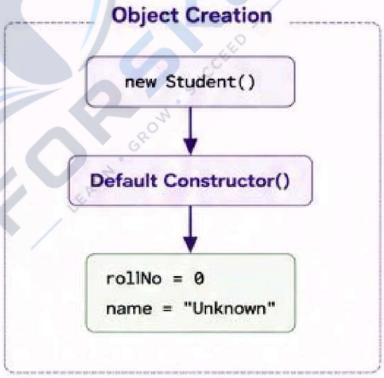
```
class Student {
   int rollNo;
   String name;

   // Default constructor
   Student() {
      rollNo = 0;
      name = "Unknown";
   }
}
class Test {
   public static void main(String[] args) {
      Student s = new Student();    // calls default constructor
      System.out.println(s.rollNo + " " + s.name);
   }
}
```

**Output**

```
0 Unknown
```

**Key Points**

- Takes no parameters.
- Provided automatically by compiler if no other constructor is defined.




## 2 PARAMETERIZED CONSTRUCTOR

A constructor that takes **one or more parameters**.
Used to initialize objects with **specific values**.

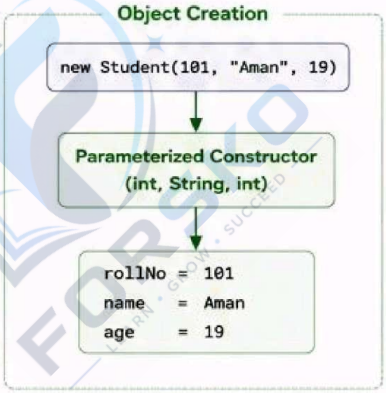
```
class Student {
   int rollNo;
   String name;
   int age;

   // Parameterized constructor
   Student(int r, String n, int a) {
      rollNo = r;
      name = n;
      age = a;
   }
}
class Test {
   public static void main(String[] args) {
      Student s = new Student(101, "Aman", 19);
      System.out.println(s.rollNo + " " + s.name + " " + s.age);
   }
}
```

**Output**

```
101 Aman 19
```

**Key Points**

- Takes parameters.
- Values are supplied at the time of object creation.
- Allows initialization with given values.




## 3 COPY CONSTRUCTOR

A constructor that creates an object by **copying values** from another object of the same class.

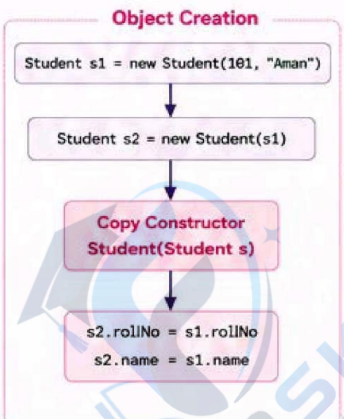
```
class Student {
   int rollNo;
   String name;
   // Parameterized constructor
   Student(int r, String n) {
      rollNo = r;
      name = n;
   }

   // Copy constructor
   Student(Student s) {
      rollNo = s.rollNo;
      name = s.name;
   }
}
class Test {
   public static void main(String[] args) {
      Student s1 = new Student(101, "Aman");
      Student s2 = new Student(s1);   // copy constructor
      System.out.println(s2.rollNo + " " + s2.name);
   }
}
```

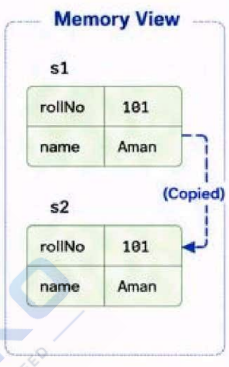
**Output**

```
101 Aman
```

**Key Points**

- Takes an object of same class as parameter.
- Creates a new object as a copy of another object.
- Useful in cloning objects.



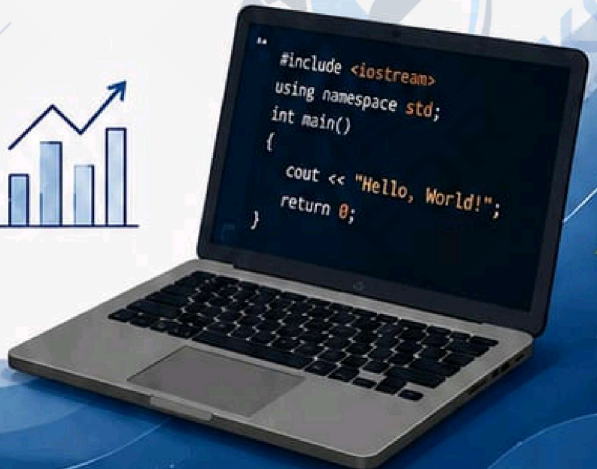

**COMPARISON**

| Constructor Type | Parameters | Purpose | When Called | Example Call |
|---|---|---|---|---|
| Default (No-Argument) | None | Initializes object with default values | When object is created (using no arguments) | `new Student()` |
| Parameterized | One or more | Initializes object with specific values | When object is created with arguments | `new Student(101, "Aman", 19)` |
| Copy Constructor | One object of same class | Creates a new object as a copy of another object | When object is created by passing another object | `new Student(s1)` |

**SUMMARY:** Constructors are used to initialize objects. Default constructor initializes with default values, parameterized constructor initializes with specific values, and copy constructor creates a copy of another object.

# Unit - III


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

OOP CONCEPTS

# FUNCTIONS AND OBJECTS: ACCESS SPECIFIERS

Access specifiers are used to control the **visibility (accessibility)** of class members from outside the class. They help in **data hiding** and **security.**

## WHAT ARE ACCESS SPECIFIERS?

They are keywords used in a class to specify the accessibility of:

- Data members (variables)
- Member functions (methods)

## THREE TYPES OF ACCESS SPECIFIERS

**public** — Members are accessible from anywhere.

**private** — Members are accessible only within the same class.

**protected** — Members are accessible within the same class and its derived (child) classes.

## SYNTAX

```
class ClassName {
private:
    int a;          // private member
public:
    void show();    // public member
protected:
    int b;          // protected member
};
```

## EXAMPLE

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

class Demo {
 private:
    int a;
 public:
    int b;
 protected:
    int c;

public:
    void setValues(int x, int y, int z) {
        a = x; b = y; c = z;
    }
    void show() {
        cout << "a = " << a
             << "\nb = " << b
             << "\nc = " << c << endl;
    }
};

int main() {
    Demo d;
    d.setValues(10, 20, 30);
    d.show();
    cout << "\nCan access b (public): " << d.b;  // OK
    // cout << d.a;  // Error: private
    // cout << d.c;  // Error: protected
    return 0;
}
```

**Output**

```
a = 10
b = 20
c = 30

Can access b (public): 20
```

## ACCESSIBILITY TABLE

| Access Specifier | Where Accessible? | | |
|---|---|---|---|
| | Within Class | Outside Class | In Derived Class |
| public | Yes | Yes | Yes |
| private | Yes | No | No |
| protected | Yes | No | Yes |

## EXPLANATION

- **public** members can be accessed from anywhere.
- **private** members can be accessed only inside the class.
- **protected** members can be accessed inside the class and in derived classes, but not from outside.

## ACCESS IN INHERITANCE

```
class Base {
 private:
    int a;
 protected:
    int b;
 public:
    int c;
};
class Derived : public Base {
 public:
   void show() {
     // cout << a; // Error: private not accessible
     cout << b;    // OK: protected
     cout << c;    // OK: public
   }
};
```

When a class is inherited (public inheritance) the access level of base class members in derived class:

| Base Class Specifier | In Derived Class | | |
|---|---|---|---|
| | Private Inheritance | Protected Inheritance | Public Inheritance |
| public | private | protected | public |
| protected | private | protected | protected |
| private | private | private | private |

## KEY POINTS

- Access specifiers implement the principle of data hiding.
- Use private for data members by default.
- Use protected when members should be accessible in derived classes.
- Use public for interfaces (functions) that should be accessible from outside.

## SUMMARY

Access specifiers (public, private, protected) control the visibility of class members. They ensure data hiding, security, and better control over how objects and functions are accessed in a program.

OOP CONCEPTS

# MEMORY ALLOCATION FOR OBJECTS

When an object is created, **memory is allocated** to it to store its data members.
In C++, memory for objects can be allocated in **three different ways.**

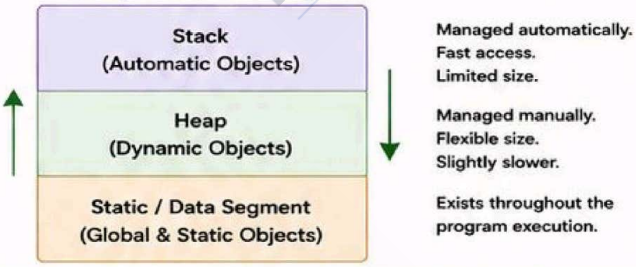
## 1 WHERE IS MEMORY ALLOCATED?

| Memory Area | Description |
|---|---|
| Stack | Automatic objects are stored here. Memory is allocated when the block is entered and deallocated when the block exits. |
| Heap | Dynamic objects are stored here. Memory is allocated using `new` and deallocated using `delete`. |
| Static / Data Segment | Global objects and static objects are stored here. Memory exists for the entire duration of the program. |

## 2 TYPES OF OBJECT MEMORY ALLOCATION

1. Automatic (Stack Allocation) – using object directly
2. Dynamic (Heap Allocation) – using new keyword
3. Static Allocation – for global and static objects

### Memory Model of a Program



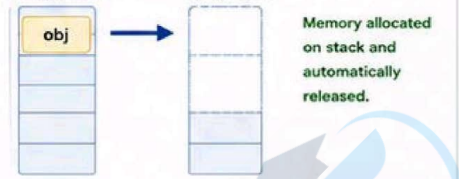

## 1 AUTOMATIC (STACK) ALLOCATION

- Object is created in a block/function.
- Memory is allocated on stack automatically and released when block/function ends.

Example:

```
class Demo {
public:
    int a;
    void show() { cout << "a = " << a; }
};

void func() {
    Demo obj;       // automatic object
    obj.a = 10;
    obj.show();
}    // obj is destroyed here
```

### Memory Representation



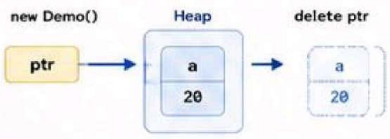

## 2 DYNAMIC (HEAP) ALLOCATION

- Object is created using **new** operator.
- Memory is allocated on **heap** and remains until **delete** is used.

Example:

```
class Demo {
public:
    int a;
    void show() { cout << "a = " << a; }
};

int main() {
    Demo *ptr = new Demo();   // dynamic object
    ptr->a = 20;
    ptr->show();
    delete ptr;          // memory released
    return 0;
}
```

### Memory Representation




Memory exists in heap until delete is called.

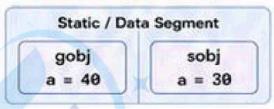
## 3 STATIC ALLOCATION (GLOBAL / STATIC OBJECTS)

- Global objects and static objects are stored in static/data segment.
- Memory exists for the entire program.

Example:

```
class Demo {
public:
    int a;
    void show() { cout << "a = " << a; }
};
Demo gobj;          // global object

void func() {
    static Demo sobj;     // static local object
    sobj.a = 30;
    sobj.show();
}

int main() {
    gobj.a = 40;
    gobj.show();
    func();
}
```

### Memory Representation




Memory is allocated once and exists till the end of program.

## COMPARISON SUMMARY

| Aspect | Automatic (Stack) | Dynamic (Heap) | Static (Global/Static) |
|---|---|---|---|
| Created using | Object name | new operator | Global or static keyword |
| Memory area | Stack | Heap | Static/Data Segment |
| Lifetime | Till block ends | Till delete is called | Entire program |
| Deallocation | Automatic | Manual (delete) | Automatic |
| Size | Fixed & limited | Flexible & large | Fixed |
| Access speed | Fast | Slower than stack | Fast |

## KEY POINTS

- Objects can be allocated in stack, heap or static memory.
- Stack allocation is automatic and fast but size is limited.
- Heap allocation is manual and provides flexibility.
- Static allocation exists for the entire program execution.
- Proper memory management (especially in heap) is important to avoid memory leaks.

**SUMMARY:** Memory allocation for objects determines where and how long an object exists in program memory. Understanding stack, heap and static allocation helps in writing efficient and error-free C++ programs.

OOP CONCEPTS

# OBJECTS AS FUNCTION ARGUMENTS

An object can be passed as an argument to a function.
This allows functions to access and operate on the data members of the object.

## 1 INTRODUCTION

- Objects can be passed to functions in the same way as basic data types.
- When an object is passed, a copy of the object is created in the function (call by value) unless references are used.

## 2 SYNTAX

```
return_type function_name( ClassName obj )
{
   // function body
}
```

→ **ClassName** – name of the class
→ **obj** – object passed as argument

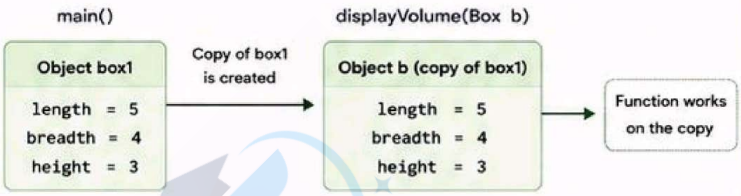
## 3 EXAMPLE PROGRAM

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

class Box {
public:
    int length, breadth, height;

    Box(int l=0, int b=0, int h=0) {
        length = l; breadth = b; height = h;
    }

    int volume() {
        return length * breadth * height;
    }
};

// Function that takes object as argument
void displayVolume(Box b) {     // object passed by value
    cout << "Volume = " << b.volume() << endl;
}

int main() {
    Box box1(5, 4, 3);          // object created
    displayVolume(box1);        // object passed to function
    return 0;
}
```

**Output**

```
Volume = 60
```

## 4 HOW IT WORKS (CALL BY VALUE)




- A copy of the object is created in the function.
- Changes made to the object inside the function do not affect the original object.

**Example:**

```
void change(Box b) {
    b.length = 10;    // changes only in the copy
}

Box box1(5,4,3);
change(box1);         // box1 remains unchanged
```

**After function call:**

```
box1.length  = 5
box1.breadth = 4
box1.height  = 3
```

## 5 PASSING BY REFERENCE (RECOMMENDED)

- To avoid copying and to allow changes to affect the original object, pass object by reference.

```
void displayVolume(const Box &b) {      // pass by reference
    cout << "Volume = " << b.volume() << endl;
}
void change(Box &b) {                   // pass by reference
    b.length = 10;                      // affects original object
}

int main() {
    Box box1(5,4,3);
    change(box1);
    cout << "After change, box1.length = " << box1.length << endl;
    return 0;
}
```

**Output**

```
After change, box1.length = 10
```

## 6 COMPARISON

| Aspect | Pass by Value | Pass by Reference |
|---|---|---|
| Declaration | void func(ClassName obj) | void func(ClassName &obj) |
| What is passed? | Copy of object | Reference to original object |
| Memory | Extra memory for copy | No extra memory |
| Changes affect original? | No | Yes |
| Efficiency | Slower (copying overhead) | Faster |
| Use when | Object is small and no changes needed | Object is large or changes are needed |

## KEY POINTS

- Objects can be passed to functions as arguments.
- By default, objects are passed by value (copy).
- Use pass by reference to improve performance and to modify the original object.
- This concept improves code modularity and reusability.

## SUMMARY

Passing objects as function arguments allows functions to work with object data. Understanding pass by value and pass by reference is important for efficient and effective C++ programming.

OOP CONCEPTS

# RETURNING OBJECTS FROM FUNCTIONS

A function can return an object of a class.
This allows functions to create an object and return it to the calling function.

## 1 INTRODUCTION

- A function can return a complete object of a class using **return** statement.
- The returned object is received in the calling function.
- This is useful when we want a function to compute and return result as an object.

## 2 SYNTAX

```
ClassName function_name(parameters)
{
    // create object
    ClassName obj;
    // process
    return obj;        // return object
}
```

- Return type is the class name
- obj is returned to the calling function
- Object is returned by value (copy)

## 3 EXAMPLE PROGRAM

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

class Complex {
public:
  int real, imag;
  Complex(int r=0, int i=0) { real = r; imag = i; }

  // Function to display complex number
  void display() {
      cout << real;
      if (imag >= 0) cout << " + " << imag << "i" << endl;
      else           cout << " - " << -imag << "i" << endl;
  }
};

// Function that returns object
Complex addComplex(Complex c1, Complex c2) {
  Complex temp(c1.real + c2.real, c1.imag + c2.imag);
  return temp;      // returning object
}

int main() {
  Complex c1(3, 4), c2(1, -2);
  Complex result;     // object to receive return value

  result = addComplex(c1, c2);   // function call

  cout << "Result: ";
  result.display();
  return 0;
}
```

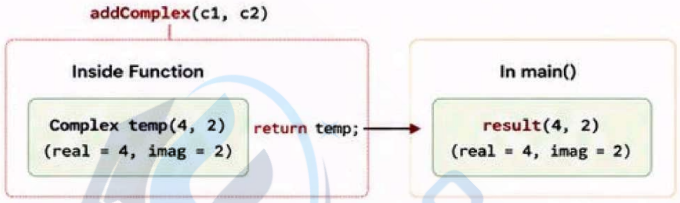
**Output**

```
Result: 4 + 2i
```

**Flow of Execution**

main() calls addComplex(c1, c2)
↓
addComplex() creates object temp
↓
temp is returned
↓
Returned object is assigned to result in main()

## 4 HOW IT WORKS




- An object temp is created in the function.
- Its value is returned using return statement.
- A copy of temp is made and assigned to result in main().
- (RVO/Copy elision in modern C++ may optimize the copy.)

**Note:** The returned object is a copy (by value). Original object inside the function is destroyed when the function ends.

## 5 RETURNING TEMPORARY OBJECT DIRECTLY

```
Complex addComplex(Complex c1, Complex c2) {
   // returning temporary object directly
   return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

A temporary object is created and returned.
This is more efficient and preferred.

## 6 IMPORTANT POINTS

- Return type of function must be the class name.
- Function returns object by value (a copy is made).
- The returned object is used in the calling function.
- Returning objects makes code modular and expressive.
- Modern C++ compilers optimize and may eliminate unnecessary copies (Return Value Optimization - RVO).

**COMPARISON: RETURN BY VALUE vs RETURN BY REFERENCE**

| Aspect | Return by Value | Return by Reference |
|---|---|---|
| Syntax | ClassName func(...) | ClassName& func(...) |
| What is returned? | Copy of object | Reference to existing object |
| Extra memory | Yes (for copy) | No |
| Safe? | Yes (original not affected) | Less safe (modifications possible) |
| Use when | Need a separate object | Need to modify original object |

## SUMMARY

Returning objects from functions allows functions to create and return meaningful results.
It improves code readability, modularity, and reusability in C++ programs.

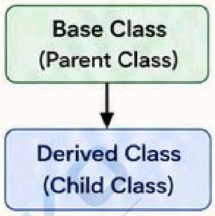
OOP CONCEPTS

# INHERITANCE

## DEFINITION & TYPES OF INHERITANCE

## 1. DEFINITION

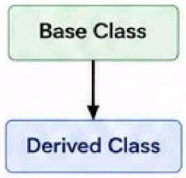
**Inheritance** is a mechanism in OOP that allows a new class (**derived class** or **child class**) to acquire the properties and characteristics (data members and member functions) of an existing class (**base class** or **parent class**).




## 2 TYPES OF INHERITANCE

### 1. Single Inheritance

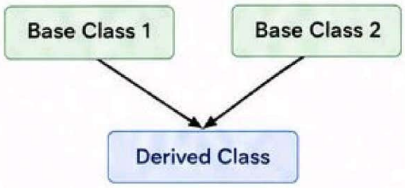
A derived class inherits from only one base class.




```
class Base {
    // members
};
class Derived : public Base {
    // members
};
```

### 2. Multiple Inheritance

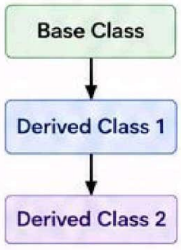
A derived class inherits from more than one base class.




```
class Base1 { };
class Base2 { };
class Derived : public Base1,
                public Base2 {
    // members
};
```

### 3. Multilevel Inheritance

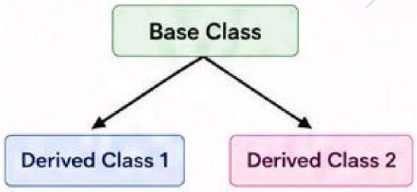
A derived class is created from another derived class.




```
class Base { };
class Derived1 : public Base { };
class Derived2 : public Derived1 {
    // members
};
```

### 4. Hierarchical Inheritance

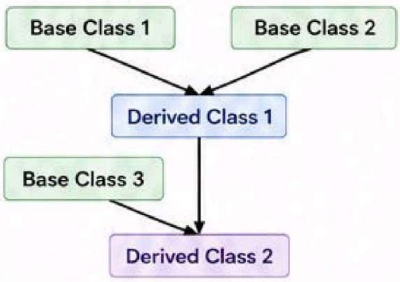
Multiple derived classes are created from a single base class.




```
class Base { };
class Derived1 : public Base { };
class Derived2 : public Base { };
```

### 5. Hybrid Inheritance

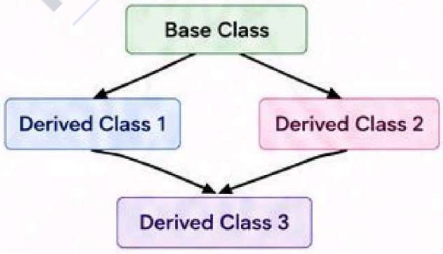
Combination of two or more types of inheritance.




```
class Base1 { }; class Base2 { };
class Base3 { };
class Derived1 : public Base1, public Base2 { };
class Derived2 : public Derived1, public Base3 {
    // members
};
```

### 6. Multiple (Multipath) Inheritance

A class is derived from two classes which are derived from the same base class.




Note: Not supported in C++
(To avoid ambiguity problem).

### KEY POINTS

- ✓ Inheritance promotes code reusability.
- ✓ It establishes an **"is-a"** relationship.
- ✓ The base class is also called **parent or superclass.**
- ✓ The derived class is also called **child or subclass.**
- ✓ Access specifiers (public, protected, private) control the visibility of inherited members.

### TERMS USED

- **Base Class (Parent Class)** : The class whose members are inherited.
- **Derived Class (Child Class)** : The class that inherits from another class.
- **Reusability** : Using existing code in a new class.
- **Overriding** : Redefining a base class function in derived class.
- **Constructor in Inheritance** : Derived class constructor calls base class constructor.

Inheritance allows a new class to inherit the data and functions of an existing class.
It is a powerful OOP feature that helps in building modular, extensible and maintainable programs.

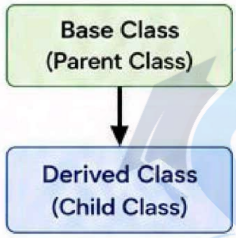
OOP CONCEPTS

# INHERITANCE

## DEFINITION & TYPES OF INHERITANCE

### DEFINITION

Inheritance is a mechanism in OOP (Object Oriented Programming) that allows a new class (**derived class** or **child class**) to inherit the data members and member functions of an existing class (**base class** or **parent class**).



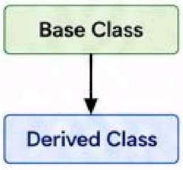

### BENEFITS

- Code reusability
- Extensibility
- Easier maintenance
- Establishes "is-a" relationship

## TYPES OF INHERITANCE IN C++

### 1 SINGLE INHERITANCE

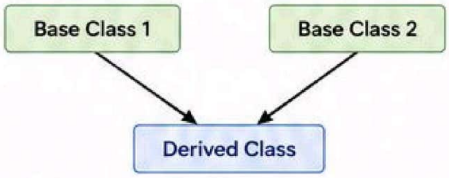
A derived class inherits from only one base class.




```
class Base {
public:
   int x;
};
class Derived : public Base {
public:
   int y;
};
```

### 2 MULTIPLE INHERITANCE

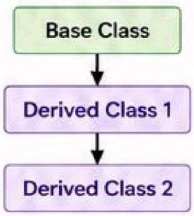
A derived class inherits from more than one base class.




```
class Base1 {
public:
   int x;
};
class Base2 {
public:
   int y;
};
class Derived : public Base1, public Base2 {
public:
   int z;
};
```

### 3 MULTILEVEL INHERITANCE

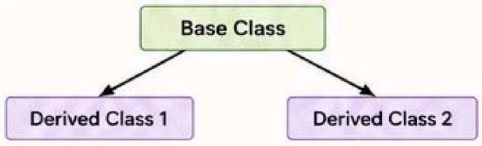
A derived class is created from another derived class.




```
class Base {
public:
   int x;
};
class Derived1 : public Base {
public:
   int y;
};
class Derived2 : public Derived1 {
public:
   int z;
};
```

### 4 HIERARCHICAL INHERITANCE

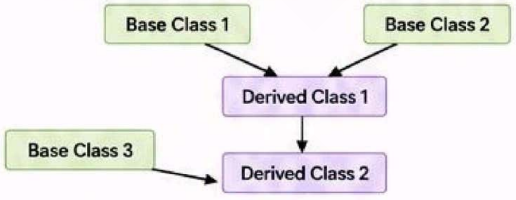
Multiple derived classes are created from a single base class.




```
class Base {
public:
   int x;
};
class Derived1 : public Base {
public:
   int y;
};
class Derived2 : public Base {
public:
   int z;
};
```

### 5 HYBRID INHERITANCE

Combination of two or more types of inheritance.




```
class Base1 { };  class Base2 { };
class Base3 { };
class Derived1 : public Base1, public Base2 { };
class Derived2 : public Derived1, public Base3 { };
```

### SUMMARY TABLE

| Type | Description | Diagram |
|---|---|---|
| Single | Inherits from one base class. | A → B |
| Multiple | Inherits from multiple base classes. | A, B → C |
| Hierarchical | Multiple derived classes from one base class. | A → B, C |
| Multilevel | Inheritance in a chain. | A → B → C |
| Hybrid | Combination of two or more types. | (Combination structure) |

### IMPORTANT NOTES

- Inheritance helps in reusing the existing code.
- The derived class (child class) is a specialized version of the base class (parent class).
- Derived class can access public and protected members of base class.
- Private members of base class are not directly accessible in derived class.
- Multiple inheritance and hybrid inheritance may lead to ambiguity (diamond problem), which is solved using virtual base class (not supported for now in basic level).

**Note:**
C++ supports all these types of inheritance except multiple inheritance of constructors.

### CONCLUSION

Inheritance is a powerful OOP feature that promotes code reusability and creates a natural hierarchy between classes using **"is-a"** relationship.

OOP CONCEPTS

# VISIBILITY MODES

Visibility modes (Access Specifiers) in C++ decide the accessibility of class members (data members and member functions) from different parts of the program.

## 1 VISIBILITY MODES (ACCESS SPECIFIERS)

| Access Specifier | Keyword | Within the Class | Within Derived Class | Outside the Class (through object) | Default |
|---|---|---|---|---|---|
| Public | public | ✔ Accessible | ✔ Accessible | ✔ Accessible | Default in class |
| Protected | protected | ✔ Accessible | ✔ Accessible | ✖ Not Accessible | |
| Private | private | ✔ Accessible | ✖ Not Accessible | ✖ Not Accessible | Default in struct |

## 2 WHERE THEY CAN BE ACCESSED?

Example Class

```
class Demo {
public:
    int a;          // public member
protected:
    int b;          // protected member
private:
    int c;          // private member
};
```

| Access Area | Members | | |
|---|---|---|---|
| | a (public) | b (protected) | c (private) |
| Inside the class (Demo) | ✔ | ✔ | ✔ |
| Inside derived class | ✔ | ✔ | ✖ |
| Outside the class (through object) | ✔ | ✖ | ✖ |

## 3 EXAMPLES

**Accessing Public Member**

```
Demo d;
d.a = 10;       // ✔ Allowed
// d.b = 20;    // ✖ Error
// d.c = 30;    // ✖ Error
```

**Inside Derived Class**

```
class Derived : public Demo {
public:
    void display() {
        a = 10;      // ✔ OK (public)
        b = 20;      // ✔ OK (protected)
        // c = 30;   // ✖ Error (private)
    }
};
```

**Outside the Class**

```
Demo d;
d.a = 10;       // ✔ OK
// d.b = 20;    // ✖ Error
// d.c = 30;    // ✖ Error
```

## 4 EFFECT OF INHERITANCE ON VISIBILITY

| Base Class Member | Mode of Inheritance | | |
|---|---|---|---|
| | Public Inheritance | Protected Inheritance | Private Inheritance |
| public | Remains public | Becomes protected | Becomes private |
| protected | Remains protected | Remains protected | Becomes private |
| private | Not accessible | Not accessible | Not accessible |

## 5 KEY POINTS

- Public members can be accessed from anywhere.
- Protected members can be accessed within the class and in derived classes.
- Private members can be accessed only within the class.
- Use access specifiers to implement data hiding and security.
- Default access in class is private, in struct is public.

**Note**

Use private for data hiding, protected for inheritance, and public for interface.

# Unit - IV


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

OOP CONCEPTS

# POLYMORPHISM

## COMPILE TIME & RUN TIME

Polymorphism means **"many forms"**. It allows objects of different classes to be treated as objects of a common base class. A function or operator can exhibit **different behaviors** in different situations.

## TYPES OF POLYMORPHISM

### 1 COMPILE TIME POLYMORPHISM
(Static / Early Binding)

- Resolved by the compiler at compile time.
- Achieved through function overloading and operator overloading.
- Based on the number and/or type of parameters.

**Example: Function Overloading**

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

class Add {
 public:
   int sum(int a, int b) { return a + b; }
   int sum(int a, int b, int c) { return a + b + c; }
   double sum(double a, double b) { return a + b; }
};

int main() {
   Add obj;
   cout << obj.sum(10, 20) << endl;        // calls sum(int,int)
   cout << obj.sum(10, 20, 30) << endl;    // calls sum(int,int,int)
   cout << obj.sum(10.5, 20.5) << endl;    // calls sum(double,double)
   return 0;
}
```

Which function to call is decided by the compiler based on arguments.

### 2 RUN TIME POLYMORPHISM
(Dynamic / Late Binding)

- Resolved at run time.
- Achieved through function overriding using virtual functions.
- Based on the actual object referred at run time.

**Example: Function Overriding**

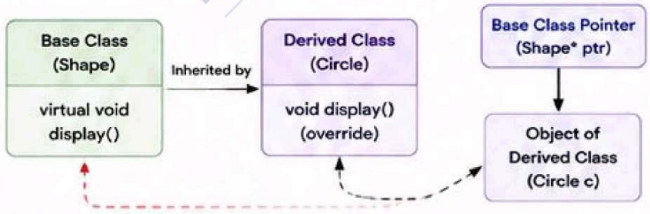
```cpp
#include <iostream>
using namespace std;

class Shape {
 public:
    virtual void display() {            // virtual function
        cout << "This is Shape" << endl;
    }
};

class Circle : public Shape {
 public:
    void display() override {           // overridden function
        cout << "This is Circle" << endl;
    }
};

int main() {
   Shape* ptr;          // base class pointer
   Circle c;
   ptr = &c;            // base pointer pointing to derived object
   ptr->display();      // calls Circle's display() at run time
   return 0;
}
```

Which function to call is decided at run time based on the actual object.

## KEY DIFFERENCES

| Aspect | Compile Time Polymorphism (Static Binding) | Run Time Polymorphism (Dynamic Binding) |
|---|---|---|
| Binding Time | At compile time | At run time |
| Also Called | Static polymorphism / Early binding | Dynamic polymorphism / Late binding |
| Achieved Through | Function overloading, Operator overloading | Function overriding (Virtual functions) |
| Parameter List | Must be different (number or type) | Must be same (in base and derived class) |
| Inheritance Required | No | Yes |
| Performance | Faster | Slightly slower (due to run time lookup) |
| Example | sum(int, int) and sum(int, int, int) | virtual void display() |

## HOW RUN TIME POLYMORPHISM WORKS




When ptr->display() is called, the actual function is decided at run time based on the object (Circle).

## EXAMPLE COMPARISON

**Compile Time (Function Overloading)**

```cpp
obj.sum(10, 20);        // sum(int,int)
obj.sum(10, 20, 30);    // sum(int,int,int)
obj.sum(10.5, 20.5);    // sum(double,double)
// Decided at compile time
```

**Run Time (Function Overriding)**

```cpp
Shape* ptr = &c;
ptr->display();      // Circle's display()
// Decided at run time
```

## KEY POINTS

- Polymorphism increases flexibility and extensibility of programs.
- Compile time polymorphism improves code readability.
- Run time polymorphism allows dynamic behavior.
- Use virtual functions for run time polymorphism.
- Overloading and overriding are the two main mechanisms.

## SUMMARY

Compile time polymorphism is achieved through overloading and is resolved by the compiler.
Run time polymorphism is achieved through overriding with virtual functions and is resolved at run time.

OOP CONCEPTS

# VIRTUAL BASE CLASSES, VIRTUAL FUNCTIONS & PURE VIRTUAL FUNCTIONS

These concepts are used to achieve flexibility, avoid ambiguity and implement runtime polymorphism and abstract classes in C++.

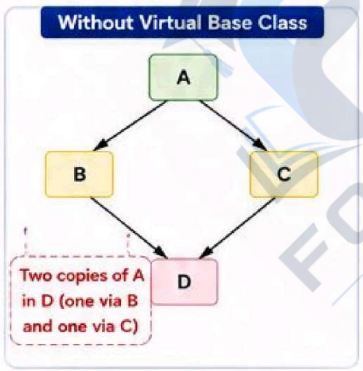
## 1 VIRTUAL BASE CLASSES

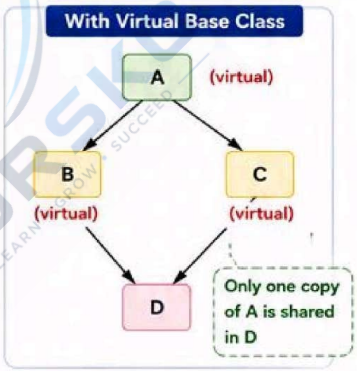
A virtual base class is a base class that is inherited using the keyword **virtual**.

It is used in multiple inheritance to ensure that only **one copy** of the base class is shared by all derived classes.

### Why?

In diamond problem, without virtual base class, multiple copies of the base class are created, leading to ambiguity and extra memory.




### Example

```
class A {
public:
   int x;
};
class B : virtual public A { };
class C : virtual public A { };
class D : public B, public C { };
int main() {
   D obj;
   obj.x = 10;  // Only one x in D
   return 0;
}
```

### Memory Layout (Conceptual)

| Without Virtual | With Virtual |
|---|---|
| D object | D object |
| B part (A subobject) | B part (shares A) |
| C part (A subobject) | C part (shares A) |
| | Single A subobject |

### Key Points

- Use **virtual** keyword while inheriting the base class.
- Solves the diamond problem.
- Saves memory by maintaining single copy of the base class.

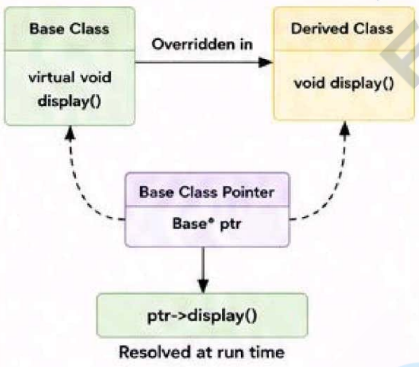
## 2 VIRTUAL FUNCTIONS

A virtual function is a member function that is declared using the keyword **virtual** in the base class.
It is redefined in derived classes and resolved at run time.
(Runtime Polymorphism)

### Example

```
class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};
class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};
int main() {
    Base* ptr;
    Derived d;
    ptr = &d;
    ptr->display();   // Calls Derived::display()
    return 0;
}
```

### How it Works




### Key Points

- Achieved through function overriding.
- Decision about which function to call is taken at run time.
- Requires pointer or reference of base class type.
- Improves flexibility and extensibility.

### Example Output

```
Resived display()
```

## 3 PURE VIRTUAL FUNCTIONS

A pure virtual function is a virtual function that has no implementation in the base class and is declared using `= 0`.

It makes the base class **abstract**.

A class containing at least one pure virtual function **cannot be instantiated**.

The derived class must provide implementation.

### Example

```
class Base {
public:
    virtual void display() = 0; // Pure virtual function
};
class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};
int main() {
    // Base b;     // Error: cannot create object of
                      abstract class
    Base* ptr = new Derived();
    ptr->display();   // Calls Derived::display()
    delete ptr;
    return 0;
}
```

### Key Points

- Declared using `virtual return_type function_name() = 0;`
- Acts as an interface.
- Provides a blueprint for derived classes.
- Achieves 100% abstraction.

### Use

Used to implement **Abstract Classes** and **Interfaces** in C++.

## SUMMARY

- ✓ **Virtual Base Classes** → Solve diamond problem in multiple inheritance.
- ✓ **Virtual Functions** → Achieve runtime polymorphism using function overriding.
- ✓ **Pure Virtual Functions** → Define interface, achieve abstraction and create abstract classes.

OOP CONCEPTS

# EARLY BINDING AND LATE BINDING

Binding (or linking) is the process of associating a function call with the function definition.
In C++, binding can occur either at compile time (early binding) or at run time (late binding).

## 1. EARLY BINDING (STATIC BINDING)

- Also called Static Binding.
- The function call is resolved at compile time by the compiler.
- Achieved through function overloading, operator overloading, and non-virtual functions.
- Faster execution (no runtime overhead).

**Example** Function Overloading

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

class Demo {
public:
   void show(int x) { cout << "Integer: " << x << endl; }
   void show(double x) { cout << "Double: " << x << endl; }
};

int main() {
  Demo d;
  d.show(10);        // Resolved at compile time
  d.show(10.5);      // Resolved at compile time
  return 0;
}
```

**Key Points**

- Decision is taken by the compiler.
- Depends on the type of data (number or types of arguments).
- Example: function overloading, operator overloading.
- Also called Static Polymorphism.

## 2. LATE BINDING (DYNAMIC BINDING)

- Also called Dynamic Binding.
- The function call is resolved at run time.
- Achieved through virtual functions and function overriding.
- Slightly slower (runtime lookup using vtable).

**Example** Function Overriding (Virtual Function)

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

class Base {
public:
   virtual void show() { cout << "Base class" << endl; }
};

class Derived : public Base {
public:
   void show() override { cout << "Derived class" << endl; }
};

int main() {
  Base* ptr;
  Derived d;
  ptr = &d;
  ptr->show();      // Resolved at run time
  return 0;
}
```

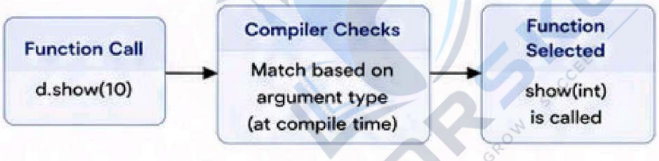
**Key Points**

- Decision is taken at run time.
- Depends on the actual object referred by the base class pointer/reference.
- Example: virtual functions (function overriding).
- Also called Dynamic Polymorphism.

## EARLY BINDING vs LATE BINDING

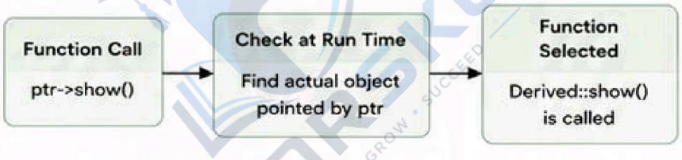
| Aspect | Early Binding (Static Binding) | Late Binding (Dynamic Binding) |
|---|---|---|
| When resolved | At compile time | At run time |
| How achieved | Function overloading, Operator overloading, Non-virtual functions | Virtual functions, Function overriding |
| Based on | Type of reference/variable | Actual object referred |
| Speed | Faster | Slower (due to runtime lookup) |
| Polymorphism | Compile time polymorphism (Static polymorphism) | Run time polymorphism (Dynamic polymorphism) |
| Example | `d.show(10); d.show(10.5);` | `ptr->show();` where `ptr` is `Base*` pointing to `Derived` object |

## EARLY BINDING – HOW IT WORKS




Binding happens at compile time.
The compiler decides which function to call.

## LATE BINDING – HOW IT WORKS




Binding happens at run time.
The function is decided during program execution.

**SUMMARY**

- Early binding improves performance and is used when the function to be called is known at compile time.
- Late binding provides flexibility and is used when the function to be called depends on the actual object at run time.

OOP CONCEPTS

# FUNCTION OVERRIDING & OPERATOR OVERLOADING

Both are forms of polymorphism in C++ that allow the same name (function or operator) to have different behavior in different situations.

## 1. FUNCTION OVERRIDING

Function overriding occurs when a derived class provides a specific implementation of a function that is already provided by its base class.

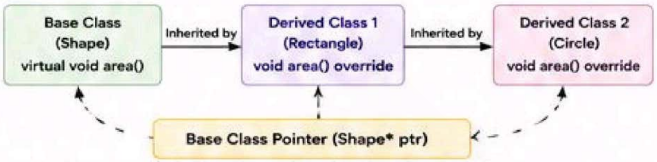
### Key Points

- Achieved through inheritance.
- The function in the base class must be declared virtual.
- The function in the derived class must have the same signature (name, parameters and return type).
- Resolved at run time (Late Binding).

### Example

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

class Shape {
public:
    virtual void area() {          // virtual function
        cout << "Area of Shape" << endl;
    }
};

class Rectangle : public Shape {
public:
    void area() override {         // overriding function
        cout << "Area of Rectangle" << endl;
    }
};

class Circle : public Shape {
public:
    void area() override {         // overriding function
        cout << "Area of Circle" << endl;
    }
};

int main() {
    Shape* ptr;                    // base class pointer
    Rectanple r;
    Circle c;
    ptr = &r;                      // points to Rectangle object
    ptr->area();                   // calls Rectangle::area()

    ptr = &c;                      // points to Circle object
    ptr->area();                   // calls Circle::area()
    return 0;
}
```

### How it Works




Function call is decided at run time based on the actual object pointed by the base class pointer.

### Important Notes

- Use the keyword `virtual` in base class.
- Use the keyword `override` in derived class (recommended).
- Return type must be same (or covariant).
- Access specifier cannot be more restrictive in derived class.

## 2. OPERATOR OVERLOADING

Operator overloading allows a user to redefine the meaning of operators for user-defined data types (class objects).

### Key Points

- Achieved through a special member function named operator followed by the operator symbol.
- At least one operand must be a user-defined type.
- We cannot change the meaning of operators for built-in types.
- Resolved at compile time (Early Binding).

### Example

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

class Complex {
  int real, imag;
public:
    Complex(int r = 0, int i = 0) { real = r; imag = i; }

    // Overloading + operator to add two Complex objects
    Complex operator + (const Complex& c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }

    void display() {
        cout << real;
        if (imag >= 0) cout << " + " << imag << "i";
        else  cout << " - " << -imag << "i";
        cout << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, -2), c3;
    c3 = c1 + c2;              // Calls operator+() function
    c3.display();
    return 0;
}
```

### Commonly Overloaded Operators

| Category | Operators |
|---|---|
| Arithmetic | + - * / % |
| Relational | == != < > <= >= |
| Logical | && \|\| ! |
| Increment / Decrement | ++ -- |
| Assignment | = += -= *= /= %= |
| Other | << >> [] () -> new delete |

### Rules / Restrictions

- At least one operand must be a user-defined type.
- We cannot overload the following operators:
  `.` `.*` `::` `?:` `sizeof`
- We cannot create new operators.
- Precedence and associativity of operators cannot be changed.

## FUNCTION OVERRIDING vs OPERATOR OVERLOADING

| Aspect | Function Overriding | Operator Overloading |
|---|---|---|
| Purpose | To provide specific implementation in derived class | To give special meaning to operators for user-defined types |
| Achieved Through | Inheritance | Special member function (operator function) |
| Binding Time | Run time (Late Binding) | Compile time (Early Binding) |
| Function Keyword | `virtual` in base class | `operator` keyword followed by operator symbol |
| Operands | Objects (through pointer/reference) | At least one operand must be user-defined type |
| Polymorphism Type | Run time polymorphism | Compile time polymorphism |
| Example | `virtual void area()` | `Complex operator + (const Complex& c)` |

### SUMMARY

- Function Overriding enables Runtime Polymorphism using inheritance and virtual functions.
- Operator Overloading enables Compile-time Polymorphism by redefining operators for user-defined types.
- Together, they make C++ more powerful and expressive.

OOP CONCEPTS

# OPERATOR OVERLOADING
# UNARY AND BINARY OPERATOR

Operator Overloading allows a user to redefine the meaning of operators for user-defined types (class objects).
Operators are of two types: **Unary** (operate on single operand) and **Binary** (operate on two operands).

## 1. OVERLOADING UNARY OPERATOR

Unary operators work on a single operand.
Examples: `++`, `--`, `-`, `+`, `!`, `~`

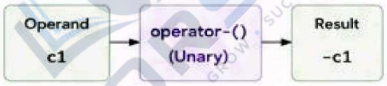
**Example:** Overloading Unary – (negation) Operator

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

class Complex {
   int real, imag;
public:
   Complex(int r = 0, int i = 0) { real = r; imag = i; }

   // Overloading unary - operator
   Complex operator-() {
      return Complex(-real, -imag);
   }

   void display() const {
      cout << real;
      if (imag >= 0) cout << " + " << imag << "i";
      else   cout << " - " << -imag << "i";
   }
};

int main() {
   Complex c1(3, 4);
   Complex c2 = -c1;     // Calls unary - operator
   cout << "c1 = "; c1.display(); cout << endl;
   cout << "c2 = -c1 = "; c2.display(); cout << endl;
   return 0;
}
```

**Output**

```
c1 = 3 + 4i
c2 = -c1 = -3 - 4i
```

**How it Works**




**Other Unary Operators that can be Overloaded**

| Operator | Meaning | Example Usage |
|---|---|---|
| + | Unary plus | +obj |
| - | Unary minus | -obj |
| ++ | Increment | ++obj (prefix) , obj++ (postfix) |
| -- | Decrement | --obj (prefix) , obj-- (postfix) |
| ! | Logical NOT | !obj |
| ~ | Bitwise NOT | ~obj |

## 2. OVERLOADING BINARY OPERATOR

Binary operators work on two operands.
Examples: `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `=`

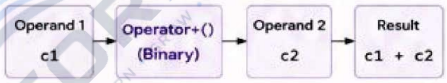
**Example:** Overloading Binary + (addition) Operator

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

class Complex {
   int real, imag;
public:
   Complex(int r = 0, int i = 0) { real = r; imag = i; }

   // Overloading binary + operator
   Complex operator+(const Complex &c) {
      return Complex(real + c.real, imag + c.imag);
   }

   void display() const {
      cout << real;
      if (imag >= 0) cout << " + " << imag << "i";
      else   cout << " - " << -imag << "i";
   }
};

int main() {
   Complex c1(2, 3), c2(4, 5);
   Complex c3 = c1 + c2;   // Calls binary + operator
   cout << "c1 = "; c1.display(); cout << endl;
   cout << "c2 = "; c2.display(); cout << endl;
   cout << "c3 = c1 + c2 = "; c3.display(); cout << endl;
   return 0;
}
```

**Output**

```
c1 = 2 + 3i
c2 = 4 + 5i
c3 = c1 + c2 = 6 + 8i
```

**How it Works**




**Commonly Overloaded Binary Operators**

| Category | Operators |
|---|---|
| Arithmetic | + - * / % |
| Relational | == != < > <= >= |
| Logical | && \|\| ! |
| Assignment | = += -= *= /= %= |
| Others | << >> [] () -> |

## UNARY vs BINARY OPERATOR OVERLOADING

| Aspect | Unary Operator Overloading | Binary Operator Overloading |
|---|---|---|
| Operands | Works on single operand | Works on two operands |
| Function Definition | member Class operatorop() or friend Class operatorop(Class) | member Class operatorop(Class) or friend Class operatorop(Class, Class) |
| Example | -obj, ++obj, !obj | obj1 + obj2, obj1 == obj2 |
| Number of Parameters | No explicit parameter (only this) or one parameter for friend function | One parameter (member) or two parameters (friend function) |
| Operator Symbol | No change in syntax | No change in syntax |

## RULES / RESTRICTIONS

- At least one operand must be a user-defined type.
- We cannot overload the following operators:
  `.` `.*` `::` `?:` `sizeof`
- We cannot change the precedence and associativity of operators.
- We cannot change the number of operands.
- Overloading does not mean adding a new operator.
- Binary assignment operator (=) must be overloaded as a member function.

## KEY POINTS

- Unary operators operate on single operand objects.
- Binary operators operate on two operand objects.
- Operator overloading improves code readability and makes programs easier to use.
- It is a form of compile-time polymorphism (early binding).

**Note:** Use operator overloading carefully to maintain clarity and avoid confusion.

C++ OOP CONCEPTS

# RULES FOR OPERATOR OVERLOADING

Operator overloading allows us to give special meaning to operators for user-defined types (class objects) without changing their original meaning for built-in types.

## RULES

**1 At least one operand must be a user-defined type.**

We cannot overload operators for built-in types only.


**2 We cannot change the meaning of operators for built-in types.**

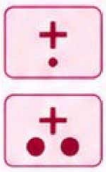
For example, we cannot change the meaning of '+' for int, float etc.

**3 We cannot change the precedence and associativity of operators.**

The precedence and associativity of an operator remain the same as defined by C++.

**4 We cannot change the number of operands.**

Unary operators will continue to have one operand and binary operators two operands.



**5 We cannot create new operators.**

We can only overload existing C++ operators.


**6 Some operators cannot be overloaded.**

The following operators cannot be overloaded: :: (scope resolution), . (member access), .* (member pointer access), ?: (conditional), sizeof.

**7 Overloaded operators are implemented using either member functions or non-member (friend) functions.**

Most operators can be overloaded using either member or friend function, but some must be member functions.

**8 Assignment operator (=) must be overloaded as a member function.**

It cannot be overloaded as a non-member function.

**9 The return type of operator overloading function cannot be a built-in type when overloading new/delete.**

new and delete cannot be overloaded to return any type other than void*.

`void*`

**10 Overloading does not mean adding a new operator.**

It only gives a special meaning to an existing operator for user-defined types.

## IMPORTANT NOTES

- Operator overloading improves code readability and makes programs easier to understand.
- It is a form of compile-time polymorphism (early binding).
- Use operator overloading carefully to maintain clarity and avoid confusion.

Example Valid Overloading

`+ , - , * , / , == , [] , () , << , >>`

Example Invalid Overloading

`. , :: , .* , ?: , sizeof`

**SUMMARY**

Follow these rules to overload operators correctly and effectively in C++ programs.