MAA OMWATI DEGREE COLLEGE

V.P.O. Hassanpur, Teh. Hodal Distt. Palwal

(HR.)

BCA-3rd (NEP) (OBJECT ORIENTED

PROGRAMMING USING C++)

COURSE CODE - 24BCA403DS02

UNIT - 1

Introduction to OOPs Concepts (Object-Oriented Programming)

Object-Oriented Programming (OOP) is a programming paradigm centered around **objects** rather than actions. It helps organize software design by bundling data and the functions that operate on them into a single unit called an object. OOP makes it easier to maintain, modify, and scale applications.

Core Concepts of OOP

1. Class

- A **class** is a blueprint for creating objects.
- It defines attributes (data) and methods (functions) that the objects created from the class can use.
- **Example (Python):**

```
python
CopyEdit
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def drive(self):
        print(f"{self.color} {self.brand} is driving.")
```

2. Object

- An **object** is an instance of a class.
- It represents a real-world entity and can access the class's methods and properties.
- **Example:**

```
python
CopyEdit
my_car = Car("Toyota", "Red")
my_car.drive()
```

3. Encapsulation

- The practice of keeping the internal details of an object hidden from the outside world.
- Data is kept private and accessed through public methods.
- Promotes data security and code modularity.

- **Example:**

```
python
CopyEdit
class Person:
    def __init__(self, name):
        self.__name = name  # Private attribute

    def get_name(self):
        return self.__name
```

4. Inheritance

- Allows one class to inherit the attributes and methods of another class.
- Promotes code reusability and hierarchical classification.
- **Example:**

```
python
CopyEdit
class Vehicle:
    def start(self):
        print("Vehicle started.")

class Bike(Vehicle):
    def ride(self):
        print("Bike is being ridden.")
```

5. Polymorphism

- Means "many forms". Allows methods to do different things based on the object calling them.
- Achieved through method overriding or overloading (language-dependent).
- **Example:**

```
python
CopyEdit
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")
```

6. Abstraction

- Hides complex implementation details and shows only the necessary features of an object.
- Achieved using abstract classes or interfaces.
- **Example (Python):**

```
python
CopyEdit
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def area(self):
        return "πr²"
```

Benefits of OOP

- Improved code reusability and organization
- Easier troubleshooting and debugging
- Scalability and maintainability
- Real-world modeling

Procedural vs Object-Oriented Programming

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Definition	Follows a step-by-step approach with functions and procedures.	Organizes code using objects based on real-world entities.
Focus	Focus is on **functions** and **logic flow**.	Focus is on **objects** and **data encapsulation**.
Structure	Program is divided into **functions**.	Program is divided into **classes and objects**.
Data Handling	Data is **global** and can be modified by any function.	Data is **private/protected** and accessed via methods.
Code Reusability	Limited – mostly through functions.	High – through **inheritance and polymorphism**.
Security	Low – data is openly accessible.	High – **encapsulation** hides data.
Real-world Modeling	Less natural.	Closely mimics **real-world entities and relationships**.
Examples	C, Pascal, BASIC	Java, C++, Python (OOP style), C#

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Ease of Maintenance	Harder as codebase grows.	Easier due to modular and scalable structure.
Speed (Performance)	Often faster for small programs.	May be slower due to abstraction layers.

□ Example Comparison

Procedural (C-style)

```
c
CopyEdit
int sum(int a, int b) {
    return a + b;
}
```

Object-Oriented (Python-style)

```
python
CopyEdit
class Calculator:
    def sum(self, a, b):
        return a + b
```

Principles of Object-Oriented Programming (OOP)

OOP is built on four key principles, often abbreviated as **PIEA**:
Polymorphism, Inheritance, Encapsulation, Abstraction

□ 1. Encapsulation

Definition:
Encapsulation is the practice 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.

Benefits:

- Protects data from unauthorized access or modification
- Makes code modular and easier to maintain
- Enables data hiding and controlled access using getters/setters

---

□ 2. Inheritance

Definition:
Inheritance allows a new class (child/subclass) to **inherit properties and behaviors** from an existing class (parent/superclass).

Benefits:

- Promotes **code reusability**
- Supports **hierarchical classification**
- Makes it easy to **extend and modify** existing code

□ 3. Polymorphism

Definition:
Polymorphism means **"many forms"**. It allows the same method to behave **differently based on the object** calling it.

- **Compile-time polymorphism** (method overloading)
- **Run-time polymorphism** (method overriding)

Benefits:

- Simplifies code readability and maintenance
- Enables flexibility and scalability
- Supports dynamic method resolution during runtime

□ 4. Abstraction

Definition:
Abstraction means hiding **complex implementation details** and showing only the **essential features** to the user.

Benefits:

- Reduces complexity for the user
- Enhances code security and design clarity
- Improves focus on **what** an object does instead of **how**

□ Overall Benefits of OOP

Benefit	Explanation
Modularity	Code is organized into classes and objects
Reusability	Code can be reused through inheritance and composition
Maintainability	Easier to debug and update due to encapsulated and modular structure
Security	Encapsulation and abstraction protect sensitive data
Scalability	OOP systems are easier to expand and scale with growing requirements
Real-World Modeling	OOP reflects real-world entities, making design more intuitive

1. Object

An **object** is an instance of a class. It represents a **real-world entity** with attributes (data) and behaviors (methods).

- **Example (Python):**

```
python
CopyEdit
class Dog:
    def bark(self):
        print("Woof!")

my_dog = Dog()  # 'my_dog' is an object
my_dog.bark()
```

□ 2. Class

A **class** is a blueprint or template for creating objects. It defines the structure and behavior (i.e., attributes and methods) of objects.

- **Example:**

```
python
CopyEdit
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def drive(self):
        print(f"{self.color} {self.brand} is driving.")
```

---

3. Inheritance

Inheritance allows one class (child) to acquire the properties and methods of another class (parent).

- **Benefits:** Code reuse, extension of existing functionality.
- **Example:**

```
python
CopyEdit
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        print("Bark")
```

---

4. Abstraction

Abstraction hides unnecessary details and shows only the essential features of an object.

- **Implemented using:** Abstract classes or interfaces.
- **Example:**

```
python
CopyEdit
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def area(self):
        return 3.14 * 5 * 5
```

---

5. Encapsulation

Encapsulation is the process of wrapping data and methods into a single unit (class) and restricting direct access to it.

- **Use of:** Private variables and public methods.
- **Example:**

```
python
CopyEdit
```

```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance
```

---

6. Polymorphism

Polymorphism means the same function name behaves differently depending on the object.

- **Types:**
  - Method Overloading (same name, different parameters)
  - Method Overriding (child class changes behavior of parent method)
- **Example:**

```
python
CopyEdit
class Bird:
    def speak(self):
        print("Bird sounds")

class Parrot(Bird):
    def speak(self):
        print("Parrot talks")
```

---

7. Dynamic Binding (Late Binding)

Dynamic Binding means that method calls are resolved **at runtime** rather than compile time.

- **Useful in:** Polymorphism and method overriding
- **Example:**

```
python
CopyEdit
def make_sound(animal):
    animal.sound()  # Actual method called is decided at runtime
```

---

8. Message Passing

Message Passing refers to the process of **objects communicating** with each other by **sending messages** (method calls).

- **Example:**

```
python
CopyEdit
car = Car("Toyota", "Red")
car.drive()  # drive() is a message sent to the object
```

□ Summary Table

Concept	Description
Object	Instance of a class
Class	Blueprint for creating objects
Inheritance	Acquiring features from another class
Abstraction	Hiding complex details and showing essential features
Encapsulation	Protecting data by restricting access
Polymorphism	One interface, many implementations
Dynamic Binding	Method call resolved at runtime
Message Passing	Objects communicate via method calls

C++ PROGRAMMING BAISCS:-

Basic Structure of a C++ Program

```
cpp
CopyEdit
#include <iostream>        // Preprocessor directive

using namespace std;       // Standard namespace

// main function - execution starts here
int main() {
    cout << "Hello, World!";  // Output statement
    return 0;                 // Return value to OS
}
```

□ Detailed Breakdown

Part	Description
`#include <iostream>`	Preprocessor directive to include input/output stream

Part	Description
`using namespace std;`	Allows you to use standard C++ library names without `std::` prefix
`int main()`	Main function – starting point of the program
`{}`	Curly braces define the body of functions or classes
`cout <<`	Used to print output to the console
`return 0;`	Exits the main function; `0` typically means successful execution

---

✅ Basic Syntax Elements

✅ 1. Variables and Data Types

```
cpp
CopyEdit
int age = 25;
float weight = 60.5;
char grade = 'A';
```

✅ 2. Operators

```
cpp
CopyEdit
int a = 10 + 5;      // Arithmetic
a += 2;              // Assignment
if (a > 10) {}       // Comparison
```

✅ 3. Control Statements

```
cpp
CopyEdit
if (a > b) {
    cout << "a is greater";
} else {
    cout << "b is greater";
}

for (int i = 0; i < 5; i++) {
    cout << i;
}
```

✅ Structure of an Object-Oriented C++ Program

```
cpp
```

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

class Car {                    // Class definition
public:
    void drive() {             // Method
        cout << "Driving...";
    }
};

int main() {
    Car myCar;                 // Object creation
    myCar.drive();             // Method call
    return 0;
}
```

✅ C++ Program Structure Summary

Section	Description
Header Files	Included using `#include`, e.g., `<iostream>`
Namespace	Usually `using namespace std;` to simplify code
Main Function	Required entry point for every C++ program (`int main()`)
Functions	Blocks of reusable code (`void greet() {}`)
Classes/Objects	Used in OOP to define and use objects
Comments	`// Single-line`, `/* Multi-line */`

1. Data Types in C++

Data types tell the compiler what kind of data a variable will hold.

✅ Basic Data Types:

Type	Keyword	Description	Example
Integer	`int`	Whole numbers	`int age = 25;`
Floating point	`float`	Numbers with decimals (less precision)	`float temp = 36.5;`
Double	`double`	Double-precision floating point	`double pi = 3.14159;`

Type	Keyword	Description	Example
Character	`char`	A single character	`char grade = 'A';`
Boolean	`bool`	True or False values	`bool passed = true;`
Void	`void`	No value (used in functions)	`void greet();`

---

2. Variables in C++

A **variable** is a named location in memory used to store a value.

Syntax:

```
cpp
CopyEdit
data_type variable_name = value;
```

Examples:

```
cpp
CopyEdit
int age = 20;
float height = 5.9;
char gender = 'M';
```

Rules for Naming Variables:

- Must begin with a letter or underscore _
- No spaces or special characters (except _)
- Case-sensitive (`Age` and `age` are different)

---

3. Constants in C++

A **constant** is a value that **does not change** during program execution.

Ways to Define Constants:

Using `const` keyword:

```
cpp
CopyEdit
const float PI = 3.14159;
```

✅ *Using `#define` preprocessor:*

```
cpp
CopyEdit
#define PI 3.14159
```

✅ Example:

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

int main() {
    const int MAX_USERS = 100;
    cout << "Maximum allowed users: " << MAX_USERS;
    return 0;
}
```

✅ Summary Table

Concept	Description	Example
Data Type	Defines type of data a variable holds	`int`, `float`, `char`
Variable	Named memory location to store data	`int score = 90;`
Constant	Fixed value that cannot be changed after assigned	`const int x = 10;`

1. Control Structures in C++

Control structures control the **flow of execution** in a program. They are divided into two main categories:

✅ A. Decision Making (Conditional Statements)

Used to make decisions based on conditions.

1. if statement

```
cpp
CopyEdit
if (condition) {
    // code to execute if condition is true
}
```

✅ Example:

```
cpp
CopyEdit
int age = 20;
if (age >= 18) {
    cout << "You are an adult.";
}
```

---

2. if-else statement

```
cpp
CopyEdit
if (condition) {
    // true block
} else {
    // false block
}
```

---

3. else-if ladder

Used to check **multiple conditions**.

```
cpp
CopyEdit
if (score >= 90) {
    cout << "Grade A";
} else if (score >= 80) {
    cout << "Grade B";
} else {
    cout << "Grade C";
}
```

---

4. switch-case statement

Used when you have **multiple fixed options**.

```
cpp
CopyEdit
int day = 3;
switch(day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    case 3: cout << "Wednesday"; break;
    default: cout << "Invalid day";
}
```

🔁 B. Looping Constructs (Iteration Statements)

Loops allow you to execute a block of code **multiple times**.

---

1. for loop

Used when the number of iterations is known.

```
cpp
CopyEdit
for (int i = 0; i < 5; i++) {
    cout << i << " ";
}
```

---

2. while loop

Used when the number of iterations is **not known** in advance.

```
cpp
CopyEdit
int i = 0;
while (i < 5) {
    cout << i << " ";
    i++;
}
```

---

3. do-while loop

Executes the code **at least once**, then checks the condition.

```
cpp
CopyEdit
int i = 0;
do {
    cout << i << " ";
    i++;
} while (i < 5);
```

🔄 Loop Control Statements

Statement	Use Case

Statement	Use Case
`break;`	Exit the loop immediately
`continue;`	Skip current iteration and continue

□ Summary Table

Type	Statement	Purpose
Decision Making	`if`, `if-else`	Execute code based on a condition
	`else-if`, `switch`	Multiple condition checking
Looping	`for`	Fixed number of repetitions
	`while`	Repeats while condition is true
	`do-while`	Executes at least once
Loop Control	`break`	Exit the loop
	`continue`	Skip to next iteration

CONTROL STRUCTURES IN C++

Control structures manage the **flow of execution** in a program. They are mainly divided into:

- □ **Decision Making (Conditional Statements)**
- □ **Looping Constructs (Iteration Statements)**

□ 1. DECISION MAKING STATEMENTS

These allow the program to make choices and execute code based on conditions.

➤ if Statement

Executes a block **only if** the condition is true.

```
cpp
CopyEdit
int age = 18;
```

```
if (age >= 18) {
    cout << "You are eligible to vote.";
}
```

➤ if-else Statement

Chooses between two blocks of code.

```
cpp
CopyEdit
int marks = 45;
if (marks >= 50) {
    cout << "Pass";
} else {
    cout << "Fail";
}
```

➤ else-if Ladder

Checks multiple conditions.

```
cpp
CopyEdit
int grade = 85;
if (grade >= 90) {
    cout << "A";
} else if (grade >= 75) {
    cout << "B";
} else if (grade >= 60) {
    cout << "C";
} else {
    cout << "F";
}
```

➤ switch Statement

Used for **multiple fixed choices**.

```
cpp
CopyEdit
int day = 3;
switch(day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    case 3: cout << "Wednesday"; break;
    default: cout << "Invalid day";
}
```

🔁 2. LOOPING CONSTRUCTS

Loops are used to **repeat a block of code** as long as a condition is true.

➤ for Loop

Used when the number of repetitions is **known**.

```
cpp
CopyEdit
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

➤ while Loop

Used when the number of repetitions is **not known in advance**.

```
cpp
CopyEdit
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

➤ do-while Loop

Executes **at least once**, then checks the condition.

```
cpp
CopyEdit
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

🔁 3. LOOP CONTROL STATEMENTS

Statement	Description
`break;`	Exits the loop immediately

Statement	Description
`continue;`	Skips the current iteration and goes to next

➤ Example of `break`:

```
cpp
CopyEdit
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;
    cout << i << " ";
}
```

➤ Example of `continue`:

```
cpp
CopyEdit
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    cout << i << " ";
}
```

Summary Chart

Type	Statements	Purpose
Decision Making	`if`, `if-else`, `switch`	Execute code based on conditions
Looping	`for`, `while`, `do-while`	Repeat code blocks
Loop Control	`break`, `continue`	Control loop execution

UNIT - 2

1. What is a Class?

A **class** is a **blueprint** or **template** for creating objects.
It groups **data members (variables)** and **member functions (methods)** into a single unit.

□ Syntax of a Class:

```
cpp
CopyEdit
class ClassName {
   public:
        // data members
        // member functions
};
```

□ 2. What is an Object?

An **object** is an **instance** of a class.
It uses the class's structure and can access its data and functions.

□ Example: Class and Object in C++

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

// Define a class
class Car {
   public:
        string brand;
        int speed;

        void drive() {
            cout << "The " << brand << " is driving at " << speed << " km/h."
<< endl;
        }
};

int main() {
    Car myCar;                  // Object creation
    myCar.brand = "Toyota";     // Assigning values
    myCar.speed = 100;

    myCar.drive();              // Calling a method
    return 0;
}
```

Key Components of a Class

Component	Description	Example
Data Members	Variables that store data	`string brand; int speed;`
Member Functions	Functions that operate on data members	`void drive()`
Access Specifiers	Control access to class members	`public`, `private`, `protected`

Access Specifiers

Specifier	Description
`public`	Accessible from anywhere
`private`	Accessible only within the class
`protected`	Accessible within the class and derived classes

Real-Life Analogy

- **Class**: Blueprint of a car
- **Object**: A real Toyota or Honda created from that blueprint

Each object has its own **values**, but they share the same **structure and behavior**.

1. What is a Class?

A **class** is a **user-defined data type** that contains:

- **Data Members** → variables (attributes)
- **Member Functions** → functions (methods) that operate on the data

---

2. Defining a Class in C++

Syntax:

```
cpp
```

```
CopyEdit
class ClassName {
   // Access specifier: public, private, protected
   public:
       // Data members
       // Member functions
};
```

□ Example:

```
cpp
CopyEdit
class Student {
   public:
       string name;
       int age;

       void displayInfo() {
           cout << "Name: " << name << endl;
           cout << "Age: " << age << endl;
       }
};
```

□ 3. Creating and Using Objects

□ Object Creation Syntax:

```
cpp
CopyEdit
ClassName objectName;
```

□ Example:

```
cpp
CopyEdit
int main() {
    Student s1;              // Creating object
    s1.name = "Alice";       // Accessing data members
    s1.age = 20;

    s1.displayInfo();        // Calling member function
    return 0;
}
```

□ 4. Data Members

These are **variables defined inside the class** that represent the properties of an object.

□ Example:

```
cpp
CopyEdit
string name;    // Data member
int age;        // Data member
```

✅ 5. Member Functions

These are **functions defined inside the class** to perform operations on the object's data.

🔹 Types:

- **Inside the class definition**
- **Outside the class definition** using `scope resolution operator (::)`

🔹 Defining Inside the Class:

```
cpp
CopyEdit
class Student {
   public:
        void greet() {
            cout << "Hello, Student!";
        }
};
```

🔹 Defining Outside the Class:

```
cpp
CopyEdit
class Student {
   public:
        void greet();  // Declaration only
};

// Definition outside
void Student::greet() {
    cout << "Hello, Student!";
}
```

✅ 6. Access Specifiers

Used to set the accessibility of members.

Specifier	Accessibility
`public`	Accessible from outside the class
`private`	Accessible only within the class
`protected`	Accessible within class and derived classes

Full Working Example

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

class Car {
   public:
       string brand;
       int speed;

       void drive() {
           cout << brand << " is driving at " << speed << " km/h." << endl;
       }
};

int main() {
    Car myCar;
    myCar.brand = "Honda";
    myCar.speed = 120;
    myCar.drive();
    return 0;
}
```

1. ACCESS SPECIFIERS in C++

Access specifiers define the **visibility** or **access level** of **data members** and **member functions** inside a class.

a. public

- Accessible from **anywhere** (outside and inside the class).
- Most used for member functions.

```
cpp
CopyEdit
class Student {
public:
    string name;  // Accessible outside
    void display() {
        cout << "Name: " << name;
```

```
    }
};
```

□ b. private

- Accessible **only within the class**.
- Used for **data hiding** (encapsulation).
- Can't be accessed directly from outside the class.

```
cpp
CopyEdit
class Student {
private:
    int rollNumber;

public:
    void setRoll(int r) {
        rollNumber = r;
    }

    void getRoll() {
        cout << "Roll No: " << rollNumber;
    }
};
```

□ c. protected

- Accessible **within the class** and by **derived (child) classes**, but **not from outside**.

```
cpp
CopyEdit
class Person {
protected:
    string name;
};

class Student : public Person {
public:
    void setName(string n) {
        name = n;  // Accessible here
    }
    void getName() {
        cout << "Name: " << name;
    }
};
```

□ 2. FUNCTIONS AND PARAMETER PASSING

□ Function Declaration & Definition:

```
cpp
CopyEdit
class Calculator {
public:
    int add(int a, int b);  // Declaration
};

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

? Parameter Passing Methods in C++

? a.Pass by Value (Default)

- A **copy** of the variable is passed.
- Changes made inside the function do **not affect** the original.

```
cpp
CopyEdit
void change(int x) {
    x = x + 10;
}
```

? b.Pass by Reference

- The **actual variable** is passed using `&`.
- Changes **do affect** the original variable.

```
cpp
CopyEdit
void change(int &x) {
    x = x + 10;
}
```

? c.Pass by Pointer

- A **pointer to the variable** is passed using `*`.
- Must be dereferenced to access the value.

```
cpp
CopyEdit
void change(int *x) {
    *x = *x + 10;
}
```

✅ Example: Function with Parameter Passing

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

class Number {
public:
    void add(int a, int b) {
        cout << "Sum: " << a + b << endl;
    }

    void square(int &n) {
        n = n * n;
    }
};

int main() {
    Number obj;
    int x = 5;
    obj.add(10, 20);       // Pass by value
    obj.square(x);         // Pass by reference
    cout << "Square: " << x;
    return 0;
}
```

1. ARRAYS IN C++

An **array** is a collection of **elements of the same data type**, stored in **contiguous memory locations**.

✅ Syntax:

```
cpp
CopyEdit
data_type array_name[size];
```

✅ Example:

```
cpp
CopyEdit
int marks[5] = {90, 85, 70, 95, 88};
```

✅ Accessing Elements:

```
cpp
CopyEdit
cout << marks[2];  // Output: 70
```

---

2. STRINGS IN C++

C++ offers **two ways** to work with strings:

a. Character Array (C-style string):

```
cpp
CopyEdit
char name[10] = "John";
```

b. `string` Class (Modern C++):

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

int main() {
    string name = "Alice";
    cout << "Name: " << name;
    return 0;
}
```

---

3. POINTERS IN C++

A **pointer** is a variable that **stores the address** of another variable.

Syntax:

```
cpp
CopyEdit
data_type* pointer_name;
```

Example:

```
cpp
CopyEdit
int x = 10;
int* p = &x;          // p stores the address of x
cout << *p;           // Output: 10 (value at the address)
```

Pointer Operations:

Operator	Meaning	Example

Operator	Meaning	Example
`&`	Address of	`p = &x;`
`*`	Dereferencing	`cout << *p;`

---

4. CONSTRUCTORS IN C++

A **constructor** is a **special function** that automatically gets called when an object is created.

Features:

- Same name as the class
- No return type
- Can be overloaded (multiple constructors)

Example:

```
cpp
CopyEdit
class Student {
public:
    string name;

    // Constructor
    Student(string n) {
        name = n;
    }

    void display() {
        cout << "Name: " << name;
    }
};

int main() {
    Student s1("Alice");
    s1.display();
    return 0;
}
```

---

5. DESTRUCTORS IN C++

A **destructor** is a special function that is called **automatically when an object is destroyed** (e.g., goes out of scope or program ends).

Syntax:

```
cpp
CopyEdit
~ClassName() {
    // clean-up code
}
```

□ Example:

```
cpp
CopyEdit
class Test {
public:
    Test() {
        cout << "Constructor called" << endl;
    }

    ~Test() {
        cout << "Destructor called" << endl;
    }
};

int main() {
    Test t1;
    return 0;
}
```

Inheritance:-

1. What is a Base Class?

A **base class** (also called **parent** or **superclass**) is the class **whose properties and behaviors are inherited** by another class.

```
cpp
CopyEdit
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
```

□ 2. What is a Derived Class?

A **derived class** (also called **child** or **subclass**) is a class that **inherits** data and functions from another (base) class.

```
cpp
CopyEdit
class Dog : public Animal {
```

```
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};
```

❓ Basic Syntax of Inheritance:

```
cpp
CopyEdit
class DerivedClass : access-specifier BaseClass {
    // additional members
};
```

- **access-specifier** determines how the base class members are inherited:
  - `public` → public and protected members of base class stay public/protected
  - `private` → everything becomes private
  - `protected` → public becomes protected

❓ Full Example: Base and Derived Class in C++

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

// Base Class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

// Derived Class
class Dog : public Animal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();   // Inherited from Animal
    myDog.bark();  // From Dog class
    return 0;
}
```

✅ Output:

```
nginx
```

```
CopyEdit
This animal eats food.
The dog barks.
```

❓ Types of Inheritance in C++

1. **Single Inheritance** – One base, one derived
2. **Multiple Inheritance** – Multiple base classes
3. **Multilevel Inheritance** – Base → Derived → Another Derived
4. **Hierarchical Inheritance** – One base, many derived
5. **Hybrid Inheritance** – Combination of types

Types of Inheritance in C++

Inheritance allows a **derived class** to acquire properties and behaviors (data members and functions) from a **base class**.

🔹 1. Single Inheritance

One **base class** → One **derived class**.

```
cpp
CopyEdit
class A {
public:
    void showA() { cout << "Class A"; }
};

class B : public A {
public:
    void showB() { cout << "Class B"; }
};
```

🔹 **Use Case:** Simple reuse of one class's code.

🔹 2. Multiple Inheritance

One **derived class** inherits from **two or more base classes**.

```
cpp
CopyEdit
class A {
public:
    void showA() { cout << "A"; }
```

```
};

class B {
public:
    void showB() { cout << "B"; }
};

class C : public A, public B {
public:
    void showC() { cout << "C"; }
};
```

 Use Case: Combine features from different classes.

□ 3. Multilevel Inheritance

Inheritance occurs in **multiple levels**: A → B → C.

```
cpp
CopyEdit
class A {
public:
    void showA() { cout << "A"; }
};

class B : public A {
public:
    void showB() { cout << "B"; }
};

class C : public B {
public:
    void showC() { cout << "C"; }
};
```

 Use Case: Extend behavior step-by-step.

□ 4. Hierarchical Inheritance

One **base class** is inherited by **multiple derived classes**.

```
cpp
CopyEdit
class A {
public:
    void showA() { cout << "A"; }
};
```

```
class B : public A {
public:
    void showB() { cout << "B"; }
};

class C : public A {
public:
    void showC() { cout << "C"; }
};
```

 Use Case: Shared base class functionality for multiple classes.

---

5. Hybrid Inheritance

Combination of **two or more types** of inheritance (can cause ambiguity if not managed).

```
cpp
CopyEdit
class A {
public:
    void showA() { cout << "A"; }
};

class B : public A {};
class C : public A {};
class D : public B, public C {};  // Ambiguity arises
```

 Solution: Use **virtual inheritance** to avoid ambiguity.

Access Control in Inheritance

When inheriting, **access specifiers** affect how the base class members are inherited:

Access Modifier Behavior Table

Base Class Member	Public Inheritance	Protected Inheritance	Private Inheritance
`public`	public	protected	private
`protected`	protected	protected	private
`private`	not inherited	not inherited	not inherited

□ Example: Public Inheritance

```
cpp
CopyEdit
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A {
    // x is public, y is protected, z is not accessible
};
```

UNIT - 3

Polymorphism :-

1. What is Polymorphism?

Polymorphism means **"many forms"**.
It allows functions or objects to **behave differently** based on context — a key feature of OOP in C++.

□ Types of Polymorphism in C++:

Type	Also Known As	Resolved At
Compile-time Polymorphism	Static Polymorphism	Compile Time
Run-time Polymorphism	Dynamic Polymorphism	Run Time

□ 2. Function Overloading (Compile-time Polymorphism)

Allows multiple functions with **same name** but **different parameters**.

□ Example:

```
cpp
CopyEdit
class Print {
public:
    void show(int x) {
        cout << "Integer: " << x << endl;
    }

    void show(double y) {
        cout << "Double: " << y << endl;
    }

    void show(string s) {
        cout << "String: " << s << endl;
    }
};
```

---

3. Operator Overloading (Compile-time Polymorphism)

Allows you to redefine the behavior of **operators** for user-defined types (classes).

Example:

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

    Complex operator + (const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2, 3), c2(1, 4);
    Complex c3 = c1 + c2;   // Uses overloaded +
    c3.display();           // Output: 3 + 7i
}
```

---

4. Virtual Functions (Run-time Polymorphism)

A **virtual function** allows **method overriding** and enables dynamic binding (late binding).

Example:

```
cpp
CopyEdit
class Animal {
public:
    virtual void sound() {
        cout << "Some animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Bark" << endl;
    }
};
```

```
int main() {
    Animal* a;       // Base class pointer
    Dog d;
    a = &d;
    a->sound();      // Output: Bark (not Animal's sound)
}
```

---

5. Dynamic Polymorphism

Achieved using **virtual functions and base class pointers**.
The function call is resolved **at runtime** based on the object type being pointed to.

Key Requirement:

- Function must be declared as `virtual` in base class.
- Call must be made through **pointer/reference**.

---

6. Abstract Classes

A class with at least **one pure virtual function** is called an **abstract class**.

- Cannot create objects from abstract class
- Used as a **base class** for inheritance

---

7. Pure Virtual Function

A function declared with `= 0` is a **pure virtual function**.

Syntax:

```
cpp
CopyEdit
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};
```

Example:

```
cpp
CopyEdit
class Shape {
public:
```

```
    virtual void draw() = 0; // Abstract
};

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

int main() {
    Shape* s;
    Circle c;
    s = &c;
    s->draw();  // Output: Drawing Circle
}
```

Memory Management:-

1. What is Dynamic Memory Allocation?

Dynamic memory allocation is when memory is **allocated at runtime** (not during compilation).
This is useful when the size of data or number of objects is not known in advance.

---

2. The `new` Operator

- Allocates memory **dynamically** on the heap.
- Returns the **address** of the allocated memory.

Syntax:

```
cpp
CopyEdit
pointer = new data_type;
pointer = new data_type[size];  // For arrays
```

Example:

```
cpp
CopyEdit
int* p = new int;      // Allocates memory for one int
*p = 10;
cout << *p;            // Output: 10
```

🗑️ 3. The `delete` Operator

- Used to **deallocate memory** created with `new`.
- Prevents **memory leaks**.

🔹 Syntax:

```
cpp
CopyEdit
delete pointer;          // For single variable
delete[] pointer;        // For arrays
```

🔹 Example:

```
cpp
CopyEdit
delete p;  // Frees the memory allocated earlier
```

📦 4. Dynamic Allocation for Arrays

```
cpp
CopyEdit
int* arr = new int[5];   // Dynamic array of 5 ints

for (int i = 0; i < 5; i++) {
    arr[i] = i + 1;
}

for (int i = 0; i < 5; i++) {
    cout << arr[i] << " ";
}

delete[] arr;   // Important!
```

🏗️ 5. Creating Objects Dynamically

You can also create class objects **at runtime** using `new`.

🔹 Example:

```
cpp
CopyEdit
class Student {
public:
    void display() {
        cout << "Student object created!" << endl;
    }
};

int main() {
```

```
    Student* s = new Student();  // Dynamic object
    s->display();                // Access using arrow (->)

    delete s;                    // Free memory
    return 0;
}
```

UNIT - 4

Exception handling:-

- Throwing and Catching exceptions
- Re-throwing exceptions
- Specifying exception types
- Unexpected exceptions
- `try-catch` blocks

---

1. What is Exception Handling?

Exception handling in C++ lets you **detect and handle errors** at runtime gracefully, rather than letting the program crash.

It uses three main keywords:

Keyword	Purpose
`try`	Code that might cause an exception
`catch`	Handle the exception
`throw`	Raise (throw) an exception

---

2. Basic Syntax: try–catch Block

```
cpp
CopyEdit
try {
    // Code that may cause an exception
    throw value;
}
catch (type var) {
    // Code to handle the exception
}
```

---

3. Throwing an Exception

You use `throw` to signal an error:

```
cpp
CopyEdit
```

```
throw 404;                 // Throwing an integer
throw "Error occurred!";   // Throwing a string
```

---

4. Catching an Exception

You use `catch` to handle specific exception types:

```
cpp
CopyEdit
try {
    throw 10;
}
catch (int e) {
    cout << "Caught integer exception: " << e << endl;
}
```

---

5. Re-throwing an Exception

You can re-throw an exception to let another part of the program handle it:

```
cpp
CopyEdit
void test() {
    try {
        throw "Error!";
    }
    catch (const char* msg) {
        cout << "Handled inside test(), rethrowing..." << endl;
        throw;  // Rethrow the same exception
    }
}

int main() {
    try {
        test();
    }
    catch (const char* msg) {
        cout << "Caught again in main(): " << msg << endl;
    }
}
```

---

6. Catching All Exceptions

Use `catch(...)` to catch **any type of exception** (generic handler):

```
cpp
CopyEdit
try {
    throw 3.14;
```

```
}
catch (...) {
    cout << "Caught unknown exception." << endl;
}
```

---

7. Specifying Exception Types in Functions

You can specify which exceptions a function might throw using a dynamic exception specification (old-style — deprecated in C++11 and later):

```
cpp
CopyEdit
void myFunction() throw(int, double); // Deprecated
```

Instead, use **`noexcept`** in modern C++ to specify that a function **won't throw**:

```
cpp
CopyEdit
void myFunction() noexcept {
    // Code that doesn't throw
}
```

---

8. Unexpected Exceptions

If a function throws an exception not listed in its specification (older C++), `unexpected()` is called.

Modern C++ uses `noexcept`, and throwing an exception from a `noexcept` function **terminates the program**.

Real-Life Analogy

- `try` = Try something risky (e.g., open a file)
- `throw` = If it fails, throw an error
- `catch` = Catch and handle that error gracefully

1. Exception Propagation in C++

Exception Propagation means an exception is passed (propagated) up the call stack until it is caught.

Example:

```
cpp
```

```
CopyEdit
void funcB() {
    throw "Exception in B";
}

void funcA() {
    funcB();  // Exception not caught here
}

int main() {
    try {
        funcA();  // Propagated to here
    }
    catch (const char* msg) {
        cout << "Caught: " << msg << endl;
    }
}
```

 The exception thrown in `funcB()` is **not caught in `funcA()`**, so it propagates to `main()`.

---

2. Templates in C++

Templates enable **generic programming**—writing code that works with **any data type**.

Function Templates

Allows writing a single function that works for different types.

```
cpp
CopyEdit
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(3, 4) << endl;       // int
    cout << add(2.5, 3.5) << endl;   // double
}
```

Class Templates

Allows defining a class for any data type.

```
cpp
CopyEdit
```

```
template <class T>
class Box {
    T value;
public:
    Box(T val) : value(val) {}
    void show() { cout << value << endl; }
};

int main() {
    Box<int> b1(10);
    Box<string> b2("Hello");
    b1.show();
    b2.show();
}
```

🔹 3. Standard Template Library (STL)

The **STL** is a set of ready-to-use **generic classes and functions**.

📦 STL Components

Category	Examples
Containers	`vector`, `list`, `deque`, `set`, `map`
Algorithms	`sort()`, `find()`, `binary_search()`
Iterators	Used to point and traverse containers

🧪 Example with Vector and Algorithm

```
cpp
CopyEdit
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> nums = {5, 2, 8, 1};

    sort(nums.begin(), nums.end());  // Sorting using STL

    for (int n : nums) {
        cout << n << " ";
    }
}
```

---

4. Benefits of STL and Generic Programming

Benefit	Description
Reusability	Write once, use for any type
Efficiency	STL containers and algorithms are optimized
Simplicity	Reduces code length and complexity
Consistency	Unified way of working with data
Maintainability	Easy to update and debug reusable code

Working with files :-

1. What Are Streams in C++?

In C++, **streams** are used to perform **input and output (I/O)** operations. A stream is simply a flow of data.

- **Input stream**: data flows **into the program**
- **Output stream**: data flows **out of the program**

---

2. Common Stream Classes

Class	Description	Header File
`istream`	Handles input (e.g., `cin`)	`<iostream>`
`ostream`	Handles output (e.g., `cout`)	`<iostream>`
`ifstream`	Input file stream	`<fstream>`
`ofstream`	Output file stream	`<fstream>`
`fstream`	Input and output file stream	`<fstream>`

---

3. File I/O in C++

To perform file I/O, include:

```
cpp
CopyEdit
#include <fstream>
```

📝 4. Writing to a File (`ofstream`)

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

int main() {
    ofstream file("example.txt");  // Create & open file

    if (file.is_open()) {
        file << "Hello, file!" << endl;
        file << "Writing data to file." << endl;
        file.close();  // Always close the file
    } else {
        cout << "Unable to open file";
    }

    return 0;
}
```

📖 5. Reading from a File (`ifstream`)

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

int main() {
    ifstream file("example.txt");
    string line;

    if (file.is_open()) {
        while (getline(file, line)) {
            cout << line << endl;
        }
        file.close();
    } else {
        cout << "Unable to open file";
    }

    return 0;
}
```

🔄 6. Using `fstream` for Both Input and Output

```
cpp
CopyEdit
```

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

int main() {
    fstream file("data.txt", ios::in | ios::out | ios::app);

    if (file.is_open()) {
        file << "Appending text...\n";

        file.seekg(0);  // Move read pointer to beginning
        string line;
        while (getline(file, line)) {
            cout << line << endl;
        }

        file.close();
    } else {
        cout << "Error opening file";
    }

    return 0;
}
```

---

7. File Opening Modes

Mode	Description
`ios::in`	Open for reading
`ios::out`	Open for writing
`ios::app`	Append to the end of file
`ios::ate`	Go to end of file on open
`ios::trunc`	Truncate (delete) file content
`ios::binary`	Open in binary mode

---

8. Checking File Status

```
cpp
CopyEdit
if (!file) {
    cout << "File error!" << endl;
}
```

---

9. Closing a File

Always close the file using:

```
cpp
CopyEdit
file.close();
```

Why Handle Errors in File Operations?

File operations can fail due to reasons like:

- File not found
- File permission issues
- File not opened properly
- Read/write errors
- Disk space issues

C++ provides ways to detect and handle such errors gracefully using **stream state flags** and functions.

---

1. Basic File Open Error Check

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

int main() {
    ifstream file("nonexistent.txt");

    if (!file) {
        cout << "Error: Could not open file." << endl;
    } else {
        cout << "File opened successfully." << endl;
    }

    return 0;
}
```

---

2. Stream State Functions

These functions check the status of file streams:

Function	Description
`.good()`	No error, stream is OK
`.eof()`	End-of-file reached
`.fail()`	Logical error (e.g., wrong type read)
`.bad()`	Read/write error
`.clear()`	Clears error flags

□ Example: Detecting Read Error

```
cpp
CopyEdit
ifstream file("data.txt");

int number;
file >> number;

if (file.fail()) {
    cout << "Read failed (wrong format?)" << endl;
}
```

□ 3. Handling End-of-File (EOF)

```
cpp
CopyEdit
string line;
while (getline(file, line)) {
    cout << line << endl;
}

if (file.eof()) {
    cout << "End of file reached." << endl;
}
```

□ 4. Using `.clear()` to Reset Stream State

```
cpp
CopyEdit
ifstream file("data.txt");

int value;
file >> value;

if (file.fail()) {
    file.clear();  // Clear the fail flag
```

```
    file.ignore(); // Skip the bad input
    cout << "Recovered from read error." << endl;
}
```

✅ 5. Combining All for Robust File I/O

```
cpp
CopyEdit
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    ifstream file("info.txt");

    if (!file.is_open()) {
        cerr << "Error: Could not open file!" << endl;
        return 1;
    }

    string data;
    while (getline(file, data)) {
        cout << data << endl;
    }

    if (file.bad()) {
        cerr << "I/O error while reading!" << endl;
    } else if (!file.eof()) {
        cerr << "File read stopped unexpectedly!" << endl;
    }

    file.close();
    return 0;
}
```

✅ 6. Summary Table of File Error Functions

Function	Triggered When
`file.fail()`	Wrong input type or failed read/write
`file.bad()`	Irrecoverable stream error
`file.eof()`	End-of-file reached
`file.good()`	No errors so far
`file.clear()`	Clears all error flags