

**Module 3:**  
**Introduction to OOPS Programming**

**Name: Chandvaniya Abhay**

# 1. Introduction to C++

## Lab Exercise:

### 1. First C++ Program: Hello World:

- Write a simple C++ program to display "Hello, World!".
- Objective: Understand the basic structure of a C++ program, including **#include**, **main()**, and **cout**.

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

int main()
{
    cout<<"Hello World";

    return 0;
}
```

## 2. Basic Input/Output:

- Write a C++ program that accepts user input for their name and age and then displays a personalized greeting.
- Objective: Practice input/output operations using **cin** and **cout**.

```
#include <iostream> // For input and output
#include <string>    // To use the string data type
using namespace std;

int main()
{
    string name; // Variable to store the user's name
    int age;     // Variable to store the user's age

    // Ask for user's name
    cout << "Enter your name: ";
    cin >> name; // Use getline to read full name (with spaces)

    // Ask for user's age
    std::cout << "Enter your age: ";
    cin >> age;

    // Display a personalized greeting
    cout << "Hello, " << name << "! You are " << age << " years old.";

    return 0;
}
```

### 3. POP vs. OOP Comparison Program

- Write two small programs: one using Procedural Programming (POP) to calculate the area of a rectangle, and another using Object-Oriented Programming (OOP) with a class and object for the same task.
- Objective: Highlight the difference between POP and OOP approaches.
- POP:

```
// POP

#include <iostream>
using namespace std;

// Function to calculate area
float findArea(float length, float width)
{
    return length * width;
}

int main()
{
    float length, width, area;

    // Input
    cout << "Enter length: ";
    cin >> length;
    cout << "Enter width: ";
    cin >> width;
```

```

// Calculate area
area = findArea(length, width);

// Output
cout << "Area of rectangle = " << area << endl;

return 0;
}

```

- **OOP:**

```

// OOP

#include <iostream>
using namespace std;

// Class definition
class Rectangle
{
public:

// Function to calculate area
float getArea(float l, float w)
{
return l * w;
}
};

int main()
{
float l, w;
Rectangle rect;

// Input
cout << "Enter length: ";
cin >> l;

```

```
cout << "Enter width: ";  
cin >> w;  
  
cout << "Area of rectangle = " << rect.getArea(l , w) << endl;  
  
return 0;  
}
```

## 4. Setting Up Development Environment

- Write a program that asks for two numbers and displays their sum. Ensure this is done after setting up the IDE (like Dev C++ or CodeBlocks).
- Objective: Help students understand how to install, configure, and run programs in an IDE.

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

int main()
{
    // Declare variables to store the numbers
    double num1, num2, sum;

    // Ask the user for the first number
    cout << "Enter the first number: ";
    cin >> num1;

    // Ask the user for the second number
    cout << "Enter the second number: ";
    cin >> num2;

    // Calculate the sum
    sum = num1 + num2;

    // Display the result
    cout<<"The sum of "<<num1<<" and "<<num2<<" is "<<sum <<endl;

    return 0;
}
```

# Theory Exercise:

## 1. What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

### 1. Programming Paradigm

- **Procedural Programming:** Follows a **top-down** approach and focuses on **procedures or routines** (i.e., functions).
- **OOP:** Follows a **bottom-up** approach and is based on the concept of **objects and classes**.

### 2. Core Focus

- **Procedural:** Focuses on **functions** and the **sequence of actions** to be performed.
- **OOP:** Focuses on **objects** that contain both **data (attributes)** and **methods (behaviors)**.

### 3. Data Handling

- **Procedural:** Data is **separate** from functions and is often **global**, making it less secure.
- **OOP:** Data is **encapsulated** within objects, enhancing **data security and integrity**.

### 4. Code Reusability

- **Procedural:** Less emphasis on code reuse. Reusability is limited to function calls.
- **OOP:** Promotes code reuse through **inheritance** and **polymorphism**.

### 5. Modularity

- **Procedural:** Code is divided into **functions**, but often less modular as data is shared.



- **OOP:** Code is divided into **classes and objects**, making it highly modular.

## 6. Examples of Languages

- **Procedural:** C, Pascal, Fortran
- **OOP:** Java, C++, Python, C#, Ruby

## 7. Ease of Maintenance and Scalability

- **Procedural:** Can become difficult to maintain and scale as project size grows.
- **OOP:** Easier to maintain, update, and scale due to encapsulation and modularity.

## 8. Key Concepts

- **Procedural:**
  - Functions
  - Procedures
  - Sequential execution
- **OOP:**
  - Classes and Objects
  - Inheritance
  - Polymorphism
  - Encapsulation
  - Abstraction

## 2. List and explain the main advantages of OOP over POP.

### 1. Encapsulation (Data Hiding)

- **Explanation:** In OOP, data and the methods that operate on it are bundled together into **classes**, and internal details can be hidden from outside access using access modifiers (private, public, protected).
- **Advantage:** This prevents accidental or unauthorized modification of data, increasing **data security and integrity**.
- **In POP:** Data is usually global and accessible by any function, making it vulnerable to unintended changes.

### 2. Code Reusability through Inheritance

- **Explanation:** OOP supports **inheritance**, allowing a new class (child) to inherit properties and methods from an existing class (parent).
- **Advantage:** Promotes **code reuse**, reducing duplication and making maintenance easier.
- **In POP:** No concept of inheritance; code must be rewritten or manually copied.

### 3. Polymorphism

- **Explanation:** OOP allows methods or functions to behave differently based on the object that invokes them (e.g., method overloading or overriding).
- **Advantage:** Enables **flexibility and extensibility** in code, allowing the same interface to work with different data types or behaviors.
- **In POP:** Functionality must be implemented separately for each data type or condition, increasing complexity.

### 4. Modularity

- **Explanation:** Programs are divided into smaller, self-contained units called **objects**, each representing a real-world entity.
- **Advantage:** Makes it easier to **design, debug, test, and maintain** code, especially in large projects.

- **In POP:** Modularity is limited to functions, and managing large codebases becomes harder.

## 5. Scalability and Maintainability

- **Explanation:** OOP's structure makes it easier to add new features or update existing ones without breaking the whole system.
- **Advantage:** OOP is **ideal for large-scale software development** where teams may work on different parts of a system.
- **In POP:** Any change may require reworking many parts of the program due to tightly coupled code.

## 6. Real-World Modeling

- **Explanation:** OOP allows you to model real-world objects (like Car, BankAccount, User) with attributes and behaviors.
- **Advantage:** Leads to **more intuitive and natural program design**, especially useful in GUI, games, and simulations.
- **In POP:** Abstracts tasks as procedures, which can be harder to relate to real-world entities.

## 7. Improved Collaboration in Teams

- **Explanation:** In OOP, different developers can work on different classes/modules independently.
- **Advantage:** Encourages **team development** and simplifies version control and integration.
- **In POP:** Tight coupling makes collaboration more difficult and increases the chance of code conflicts.

### 3. Explain the steps involved in setting up a C++ development environment.

#### Step 1: Install a C++ Compiler

A compiler is required to translate C++ code into executable programs.

##### ► *Common Compilers:*

- **GCC (GNU Compiler Collection)** – For Linux/macOS/Windows (via MinGW)
- **MSVC (Microsoft Visual C++)** – For Windows (part of Visual Studio)
- **Clang** – Popular on macOS and some Linux systems

##### ► *How to Install:*

##### Windows:

- Option 1: **Install MinGW** (Minimalist GNU for Windows)
  1. Download MinGW from [mingw-w64.org](http://mingw-w64.org)
  2. Install it with C++ support.
  3. Add bin folder (e.g., C:\MinGW\bin) to your **System PATH**.
- Option 2: **Install Visual Studio** (Recommended for beginners)
  1. Download Visual Studio Community Edition from [visualstudio.microsoft.com](https://visualstudio.microsoft.com)
  2. During installation, select "**Desktop development with C++**" workload.

##### Linux:

```
sudo apt update
```

```
sudo apt install build-essential
```

##### macOS:

Install Xcode Command Line Tools:

```
xcode-select --install
```

## Step 2: Choose and Install a Text Editor or IDE

### ► Popular IDEs for C++:

- **Visual Studio** (Windows only)
- **Code::Blocks** (Cross-platform)
- **CLion** (JetBrains, paid with free student license)
- **Dev C++** (Lightweight)
- **Eclipse CDT** (for C/C++)

### ► Popular Text Editors with Extensions:

- **VS Code** (Lightweight and powerful)
  - Install VS Code from: [code.visualstudio.com](https://code.visualstudio.com)
  - Add the **C/C++ extension** by Microsoft

## Step 3: Set Up Environment Variables (if needed)

If using GCC or MinGW manually:

- Add the path to the compiler's bin directory to your system's **PATH** environment variable.
- This allows you to run g++ or gcc from the terminal or command prompt.

## Step 4: Write Your First C++ Program

Open your IDE or text editor and create a new file, e.g., main.cpp:

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

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

## Step 5: Compile and Run the Program

### ► *Using Command Line (GCC):*

1. Open terminal or command prompt
2. Navigate to the directory with main.cpp
3. Compile:
4. `g++ main.cpp -o myprogram`
5. Run:
  - On Windows:
    - `myprogram.exe`
  - On Linux/macOS:
    - `./myprogram`

### ► *Using an IDE:*

- Press **Run** or **Build and Run** — the IDE handles compilation and execution.

## Step 6: (Optional) Set Up Debugging and Build Tools

- Most modern IDEs include a debugger (like GDB).
- You can set breakpoints, inspect variables, and step through code.

## 4. What are the main input/output operations in C++? Provide examples.

### Main Input/Output Operations in C++

C++ uses the **iostream** library for basic I/O operations, which includes:

- cin → Standard **input** stream (from keyboard)
- cout → Standard **output** stream (to screen)
- cerr → Standard **error** stream (for error messages)
- clog → Standard **log** stream (for general logging info)

All of these are part of the <iostream> header.

#### 1. cout – Output Operation

Used to print output to the screen.

##### ► Syntax:

```
cout << data;
```

##### ► Example:

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

int main() {
    cout << "Hello, world!" << endl;
    cout << "The answer is: " << 42 << endl;
    return 0;
}
```

- << is the **insertion operator**.
- endl is used to move to a new line (can also use \n).

## 2. cin – Input Operation

Used to get input from the user (keyboard).

### ► Syntax:

cin >> variable;

### ► Example:

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

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You entered: " << age << endl;
    return 0;
}
```

- >> is the **extraction operator**.
- Reads input and stores it in the variable.

## 3. cerr – Error Output

Used to display error messages.

### ► Example:

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

int main() {
    cerr << "Error: Invalid input!" << endl;
    return 0;
}
```

- cerr is **unbuffered** — messages appear immediately.



## 4. clog – Logging Output

Used for general-purpose logging or debugging messages.

### ► Example:

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

int main() {
    clog << "Program started..." << endl;
    return 0;
}
```

- clog is **buffered** — messages may appear later than cerr.

## 2. Variables, Data Types, and Operators

### Lab Exercise:

#### 1. Variables and Constants

- Write a C++ program that demonstrates the use of variables and constants. Create variables of different data types and perform operations on them.
- Objective: Understand the difference between variables and constants.

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

int main()
{
    // Constant
    const float PI = 3.14;

    // Variables
    int age = 20;
    float height = 5.6;
    double radius = 3.0;
    char grade = 'A';
    bool passed = true;

    // Operations
    double area = PI * radius * radius;

    // Output
    cout << "Age: " << age << endl;
```

```
cout << "Height: " << height << " feet" << endl;
cout << "Grade: " << grade << endl;
cout << "Passed: " << (passed ? "Yes" : "No") << endl;
cout << "Radius: " << radius << endl;
cout << "Area of Circle: " << area << endl;
cout << "Value of PI: " << PI << endl;

return 0;
}
```

## 2. Type Conversion

- Write a C++ program that performs both implicit and explicit type conversions and prints the results.
- Objective: Practice type casting in C++.

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

int main()
{
    //IMPLICIT TYPE CONVERSION
    int intVal = 42;
    double doubleVal;

    //Implicit conversion from int to double
    doubleVal = intVal;

    cout << "Implicit Conversion:" << endl;
    cout << "intVal (int): " << intVal << endl;
    cout << "doubleVal (double, after assigning intVal): " << doubleVal << endl;

    //EXPLICIT TYPE CONVERSION
    double originalDouble = 3.14159;

    //Explicit conversion double to int
    int intFromDouble1 = (int)originalDouble;

    //Explicit conversion double to int
    int intFromDouble2 = static_cast<int>(originalDouble);

    cout << "\nExplicit Conversion:" << endl;
    cout << "originalDouble (double): " << originalDouble << endl;
    cout << "intFromDouble1 (C-style cast): " << intFromDouble1 << endl;
    cout << "intFromDouble2 (static_cast): " << intFromDouble2 << endl;
```

```
    return 0;  
}
```

### 3. Operator Demonstration

- Write a C++ program that demonstrates arithmetic, relational, logical, and bitwise operators. Perform operations using each type of operator and display the results.
- Objective: Reinforce understanding of different types of operators in C++.

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

int main()
{
    int a = 10, b = 3;

    // Arithmetic Operators
    cout << "Arithmetic Operators:" << endl;
    cout << "a + b = " << (a + b) << endl;
    cout << "a - b = " << (a - b) << endl;
    cout << "a * b = " << (a * b) << endl;
    cout << "a / b = " << (a / b) << endl;
    cout << "a % b = " << (a % b) << endl;

    cout << "\nRelational Operators:" << endl;
    cout << "a == b : " << (a == b) << endl;
    cout << "a != b : " << (a != b) << endl;
    cout << "a > b : " << (a > b) << endl;
    cout << "a < b : " << (a < b) << endl;
    cout << "a >= b : " << (a >= b) << endl;
    cout << "a <= b : " << (a <= b) << endl;

    // Logical Operators
    bool x = true, y = false;
```

```
cout << "\nLogical Operators:" << endl;
cout << "x && y : " << (x && y) << endl;
cout << "x || y : " << (x || y) << endl;
cout << "!x : " << (!x) << endl;
```

```
// Bitwise Operators
```

```
cout << "\nBitwise Operators:" << endl;
cout << "a & b = " << (a & b) << endl;
cout << "a | b = " << (a | b) << endl;
cout << "a ^ b = " << (a ^ b) << endl;
cout << "~a = " << (~a) << endl;
cout << "a << 1 = " << (a << 1) << endl;
cout << "a >> 1 = " << (a >> 1) << endl;
```

```
return 0;
```

```
}
```

# Theory Exercise:

1. What are the different data types available in C++?  
Explain with examples.

## 1. Basic (Primitive) Data Types

These are the fundamental data types provided by the language.

| Type   | Description                       | Example             |
|--------|-----------------------------------|---------------------|
| Int    | Integer values (whole numbers)    | int age = 25;       |
| Float  | Floating point (single precision) | float price = 5.75; |
| Double | Double precision floating point   | double pi = 3.1415; |
| Char   | Single character                  | char grade = 'A';   |
| bool   | Boolean (true or false)           | bool isOpen = true; |



## 2. Derived Data Types

These are based on fundamental types.

| Type      | Description                         | Example  |
|-----------|-------------------------------------|--|
| Array     | Collection of elements of same type | <code>int nums[5] = {1, 2, 3, 4, 5};</code>                |
| Pointer   | Stores memory address               | <code>int* ptr = &amp;age;</code>                          |
| Function  | Block of code that performs a task  | <code>int sum(int a, int b) {<br/>    return a+b; }</code> |
| Reference | An alias for another variable       | <code>int&amp; ref = age;</code>                           |

### 3. User-defined Data Types

Created by the programmer for specific needs.

| Type            | Description                           | Example   |
|-----------------|---------------------------------------|---|
| Struct          | Group of variables of different types | <pre>struct Person { string name; int age; };</pre> |
| class           | Blueprint for objects (OOP)           | <pre>class Car { public: string brand; };</pre>     |
| union           | Like struct but shares memory         | <pre>union Data { int i; float f; };</pre>          |
| enum            | Named set of integer constants        | <pre>enum Color { RED, GREEN, BLUE };</pre>         |
| typedef / using | Aliases for data types                | <pre>typedef int Marks; or using Marks = int;</pre> |

## 4. Void Type

Represents the absence of any value or type.

| Type | Description                            | Example  |
|------|--|--|
| Void | Used for functions that return nothing | <code>void greet() { cout &lt;&lt; "Hi!"; }</code> |

## 5. Modifiers with Data Types

Used to alter the meaning/size of basic data types.

| Type      | Description                              | Example                              |
|-----------|--|--------------------------------------|
| Signed    | Allows both positive and negative values | <code>signed int x = -100;</code>    |
| Unsigned  | Only allows positive values              | <code>unsigned int y = 200;</code>   |
| short     | Reduces storage size                     | <code>short int a = 10;</code>       |
| Long      | Increases storage size                   | <code>long int b = 1000000;</code>   |
| long long | Even bigger than long                    | <code>long long int c = 1e12;</code> |

## 2. Explain the difference between implicit and explicit type conversion in C++.

### 1. Implicit Type Conversion (Type Coercion)

- Performed automatically by the compiler.
- Happens when you assign or operate on variables of different types.
- It follows the standard type promotion rules (e.g., int to float, char to int).
- No data loss *if* the conversion is to a "larger" or compatible type.

### 2. Explicit Type Conversion (Type Casting)

- Performed manually by the programmer.
- Used when implicit conversion doesn't work or may cause data loss.
- Syntax involves *casting operators* or *C-style casting*.

| Feature      | Implicit Conversion      | Explicit Conversion         |
|--------------|--------------------------|-----------------------------|
| Performed by | Compiler                 | Programmer                  |
| Syntax       | Automatic                | (type) or static_cast<>     |
| Safety       | Generally safe           | Riskier (can lose data)     |
| Use Case     | Convenience, mixed types | Precision, overriding rules |

### 3. What are the different types of operators in C++? Provide examples of each.

#### 1. Arithmetic Operators

Used for basic mathematical operations.

| Operator | Description         | Example  |
|----------|---------------------|----------|
| +        | Addition            | $a + b$  |
| -        | Subtraction         | $a - b$  |
| *        | Multiplication      | $a * b$  |
| /        | Division            | $a / b$  |
| %        | Modulus (remainder) | $a \% b$ |

## 2. Relational (Comparison) Operators

Used to compare two values.

| Operator | Description           | Example |
|----------|-----------------------|---------|
| ==       | Equal to              | a == b  |
| !=       | Not equal to          | a != b  |
| >        | Greater than          | a > b   |
| <        | Less than             | a < b   |
| >=       | Greater than or equal | a >= b  |
| <=       | Less than or equal    | a <= b  |

### 3. Logical Operators

Used to combine multiple conditions.

| Operator | Description | Example          |
|----------|-------------|------------------|
| &&       | Logical AND | (a > 0 && b > 0) |
| !        | Logical NOT | !(a > b)         |

## 4. Assignment Operators

Used to assign values to variables.

| Operator | Description         | Example            |
|----------|---------------------|--------------------|
| =        | Assign              | a = 10             |
| +=       | Add and assign      | a +=5 // a = a + 5 |
| -=       | Subtract and assign | a -= 2             |
| *=       | Multiply and assign | a *= 3             |
| /=       | Divide and assign   | a /= 4             |
| %=       | Modulus and assign  | a %= 2             |



## 5. Unary Operators

Operate on a single operand.

| Operator | Description | Example    |
|----------|-------------|------------|
| +        | Unary plus  | +a         |
| -        | Unary minus | -a         |
| ++       | Increment   | ++a or a++ |
| --       | Decrement   | --a or a-- |
| !        | Logical NOT | ! true     |

## 6. Bitwise Operators

Operate at the binary level.

| Operator | Description | Example |
|----------|-------------|---------|
| &        | Bitwise AND | a & b   |
|          | Bitwise OR  | a   b   |
| ^        | Bitwise XOR | a ^ b   |
| ~        | Bitwise NOT | ~a      |
| <<       | Left Shift  | a << 2  |
| >>       | Right shift | a >> 1  |

## 4. Explain the purpose and use of constants and literals in C++.

### 1. Constants in C++

- **Purpose:**

Constants are named identifiers used to store values that must **not change** during program execution.

- **Why Use Constants?**

Prevent accidental modification of important values

Improve code clarity (PI is more meaningful than 3.14159)

Easier maintenance (change the value in one place)

#### **Example:**

```
const int maxScore = 100;
```

## 2. Literals in C++

- **Purpose:**

Literals are **fixed values** written directly in the source code — not stored in a variable.

- **Why Use Literals?**

Represent constant values like numbers, characters, and strings

Used in expressions, assignments, and function calls

- **Types of Literals:**

| Literal Type   | Example     | Description                   |
|----------------|-------------|-------------------------------|
| Integer        | 42, 0xFF    | Decimal, hex, octal, binary   |
| Floating-point | 3.14, 2.5e3 | Real numbers (float/double)   |
| Character      | 'A', '9'    | Enclosed in single quotes     |
| String         | "Hello"     | Enclosed in double quotes     |
| Boolean        | true, false | Boolean values                |
| Null pointer   | nullptr     | Null pointer literal (C++11+) |

## 3. Control Flow Statements

### Lab Exercise:

#### 1. Grade Calculator

- Write a C++ program that takes a student's marks as input and calculates the grade based on if-else conditions.
- Objective: Practice conditional statements (if-else).

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

int main()
{
    int marks;

    cout << "Enter the student marks: ";
    cin >> marks;

    if (marks < 0 || marks > 100)
    {
        cout << "Invalid input." << endl;
    }
    else if (marks >= 90)
    {
        cout << "Grade: A+" << endl;
    }
    else if (marks >= 80)
    {
        cout << "Grade: A" << endl;
    }
}
```

```
else if (marks >= 70)
{
    cout << "Grade: B" << endl;
}
else if (marks >= 60)
{
    cout << "Grade: C" << endl;
}
else if (marks >= 50)
{
    cout << "Grade: D" << endl;
}
else
{
    cout << "Grade: F (Fail)" << endl;
}

return 0;
}
```

## 2. Number Guessing Game

- Write a C++ program that asks the user to guess a number between 1 and 100. The program should provide hints if the guess is too high or too low. Use loops to allow the user multiple attempts.
- Objective: Understand while loops and conditional logic.

```
#include <iostream>
#include <cstdlib> // For rand() and srand()
#include <ctime>   // For time()
using namespace std;

int main()
{
    int randomNumber, guess;

    // Seed the random number generator
    srand(time(0));

    // Generate random number between 1 and 100
    randomNumber = rand() % 100 + 1;

    cout << "Guess the number (between 1 and 100): ";

    // Loop until the user guesses correctly
    while (true)
    {
        cin >> guess;

        if (guess < 1 || guess > 100)
        {
            cout << "Please enter a number between 1 and 100." << endl;
```

```
}  
else if (guess < randomNumber)  
{  
    cout << "Too low. Try again: ";  
}  
else if (guess > randomNumber)  
{  
    cout << "Too high. Try again: ";  
}  
else  
{  
    cout << "Congratulations! You guessed the correct number: " <<  
randomNumber << endl;  
    break; // loop exit  
}  
}  
  
return 0;  
}
```



### 3. Multiplication Table

- Write a C++ program to display the multiplication table of a given number using a for loop.
- Objective: Practice using loops.

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

int main()
{
    int number;

    cout << "Enter a number: ";
    cin >> number;

    for (int i = 1; i <= 10; i++)
    {
        cout << number << " x " << i << " = " << number * i << endl;
    }

    return 0;
}
```

## 4. Nested Control Structures

- Write a program that prints a right-angled triangle using stars (\*) with a nested loop.
- Objective: Learn nested control structures.

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

int main()
{
    int i, j;

    for (i = 1; i <= 5; ++i)
    {
        for (j = 1; j <= i; ++j)
        {
            cout << "* ";
        }
        cout << endl;
    }

    return 0;
}
```

# Theory Exercise:

## 1. What are conditional statements in C++? Explain the if-else and switch statements.

- **Conditional Statements in C++**

Conditional statements in C++ are **decision-making constructs** that allow a program to choose between different paths of execution based on whether a given condition is **true** or **false**.

### 1. **if-else** Statement

The **if-else** statement is the most basic form of conditional control structure.

#### **Purpose:**

It allows the program to evaluate a condition and:

- Execute one block of code if the condition is **true**
- Execute a different block if the condition is **false**

#### **Key Concepts:**

- The condition is a **Boolean expression** (i.e., it evaluates to true or false)
- Can be extended using else if for multiple conditions

#### **Types:**

- **Simple if:** Executes code only if the condition is true
- **if-else:** Executes one block if the condition is true, another if false
- **if-else if-else:** Handles multiple conditions in sequence

## 2. switch Statement

The `switch` statement is used for **multi-way decision-making** when a variable or expression can take on a limited set of constant values.

### Purpose:

It simplifies the code when multiple if-else conditions are based on the **same variable** or **expression**.

### Key Concepts:

- The expression inside the switch must be of an **integral type** (e.g., int, char, enum)
- Each case represents a possible value of the expression
- The break statement prevents fall-through to the next case
- A default case handles any unmatched values

### Summary:

| Feature     | if-else Statement                        | switch Statement                               |
|-------------|--|--|
| Type        | General-purpose condition checking       | Multi-way branching based on constant value    |
| Conditions  | Boolean expressions (any logical test)   | Constant integral expressions only             |
| Flexibility | More flexible (can handle complex logic) | Less flexible (limited to discrete values)     |
| Use Case    | When conditions are complex or varied    | When checking one variable for multiple values |

## 2. What is the difference between for, while, and do-while loops in C++?

- Difference Between for, while, and do-while Loops in C++

Loops in C++ are **control structures** that allow you to **repeat a block of code** multiple times based on a condition.

### 1. for Loop

#### Definition:

A for loop is used when the **number of iterations is known** beforehand.

#### Syntax:

```
for (initialization; condition; update)
{
    // loop body
}
```

#### Characteristics:

- Initialization, condition, and update are all part of the loop declaration.
- Best suited for **count-controlled** loops (e.g., loops that run a fixed number of times).
- Compact and readable for simple iteration

## 2. while Loop

### Definition:

A while loop is used when the **number of iterations is not known** and depends on a condition being true.

### Syntax:

```
while (condition)
{
    // loop body
}
```

### Characteristics:

- The condition is checked **before** the loop body is executed.
- If the condition is false initially, the loop **may not execute at all**.
- Commonly used when the loop depends on **user input** or **external events**.

### 3. do-while Loop

#### Definition:

A do-while loop is similar to a while loop, but the **condition is checked after** executing the loop body.

#### Syntax:

```
do
{
    // loop body
} while (condition);
```

#### Characteristics:

- The loop body is **executed at least once**, regardless of the condition.
- Useful when the loop must run at least once (e.g., menus, retry prompts).

### 3. How are break and continue statements used in loops? Provide examples.

#### Break and Continue Statements in C++ Loops

In C++, `break` and `continue` are **loop control statements** used to **alter the normal flow** of loop execution.

##### 1. `break` Statement

###### Purpose:

- Used to **immediately exit** the loop, regardless of the loop condition.
- Control moves to the **first statement after the loop**.

###### Use Cases:

- Exiting a loop when a certain condition is met.
- Terminating early during search operations or menu-driven programs.

###### Example:

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

int main()
{
    for (int i = 1; i <= 10; i++)
    {
        if (i == 5)
        {
            break; // Exit the loop when i is 5
        }
        cout << i << " ";
    }
    return 0;
}
```



## 2. `continue` Statement

### Purpose:

- Used to **skip the current iteration** of the loop and move to the **next iteration**.
- The rest of the loop body **after `continue` is ignored** for the current iteration.

### Use Cases:

- Skipping unwanted or invalid data in a loop.
- Skipping execution based on a condition.

### Example:

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

int main()
{
    for (int i = 1; i <= 5; i++)
    {
        if (i == 3)
        {
            continue; // Skip the iteration when i is 3
        }
        cout << i << " ";
    }
    return 0;
}
```

## 4. Explain nested control structures with an example.

### Nested Control Structures in C++

#### Definition:

Nested control structures are **control statements placed inside other control statements**. This means you can put a loop inside another loop, an `if` inside a loop, a `switch` inside an `if`, etc.

They allow more complex decision-making and repetition in your program by combining multiple control structures.

#### Types of Nested Structures

1. Nested `if` statements
2. `if` inside loops (or vice versa)
3. Nested loops (`for`, `while`, `do-while`)
4. Nested `switch` statements (less common)

#### Example: Nested `if` Statement

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

int main()
{
    int number;
    cout << "Enter a number: ";
    cin >> number;

    if (number > 0)
    {
        if (number % 2 == 0)
        {
            cout << "The number is positive and even." << endl;
        }
        else
```

```

    {
        cout << "The number is positive and odd." << endl;
    }
}
else
{
    cout << "The number is not positive." << endl;
}

return 0;
}

```

### Example: Nested for Loops (Printing a pattern)

```

#include <iostream>
using namespace std;

int main()
{
    for (int i = 1; i <= 3; i++)
    {
        for (int j = 1; j <= 5; j++)
        {
            cout << "* ";
        }
        cout << endl;
    }
    return 0;
}

```

## 4. Functions and Scope

### Lab Exercise:

#### 1. Simple Calculator Using Functions

- Write a C++ program that defines functions for basic arithmetic operations (add, subtract, multiply, divide). The main function should call these based on user input.
- Objective: Practice defining and using functions in C++.

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

// Function declare
float add(float a, float b)
{
    return a + b;
}

float subtract(float a, float b)
{
    return a - b;
}

float multiply(float a, float b)
{
    return a * b;
}

float divide(float a, float b)
```

```
{  
    if (b != 0)  
        return a / b;  
    else  
    {  
        cout << "Zero Division is not allowed." << endl;  
        return 0;  
    }  
}
```

```
int main()  
{  
    float num1, num2;  
    char op;  
  
    cout << "Enter number: ";  
    cin >> num1;  
  
    cout << "Enter operator (+, -, *, /): ";  
    cin >> op;  
  
    cout << "Enter number: ";  
    cin >> num2;  
  
    float result;  
  
    switch (op)  
    {  
        case '+':  
            result = add(num1, num2);  
            cout << "Result: " << result << endl;  
            break;  
        case '-':  
            result = subtract(num1, num2);  
            cout << "Result: " << result << endl;  
            break;  
        case '*':
```

```
    result = multiply(num1, num2);  
    cout << "Result: " << result << endl;  
    break;  
case '/':  
    result = divide(num1, num2);  
    cout << "Result: " << result << endl;  
    break;  
default:  
    cout << "Invalid operator!" << endl;  
}  
return 0;  
}
```

## 2. Factorial Calculation Using Recursion

- Write a C++ program that calculates the factorial of a number using recursion.
- Objective: Understand recursion in functions.

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

int factorial(int n) // Recursive function
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1); // recursive call
}

int main()
{
    int num;

    cout << "Enter a positive number: ";
    cin >> num;

    if (num < 0)
    {
        cout << "Factorial is not negative numbers." << endl;
    }
    else
    {
        int result = factorial(num);
        cout << "Factorial of " << num << " is: " << result << endl;
    }
    return 0;
}
```

### 3. Variable Scope

- Write a program that demonstrates the difference between local and global variables in C++. Use functions to show scope.
- Objective: Reinforce the concept of variable scope.

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

// Global variable
int number = 100;

void showGlobal()
{
    //global variable
    cout << "showGlobal, global number = " << number << endl;
}

void showLocal()
{
    // Local variable
    int number = 50;
    cout << "showLocal, local number = " << number << endl;
}

int main()
{
    cout << "main, global number = " << number << endl;

    showGlobal(); //global variable
    showLocal();  //local variable

    cout << "Back in main, global number = " << number << endl;
```



```
    return 0;  
}
```

# Theory Exercise:

1. What is a function in C++? Explain the concept of function declaration, definition, and calling.

## What is a Function in C++?

A **function** in C++ is a block of code that performs a specific task. Functions help **organize code**, **avoid repetition**, and **make programs easier to read and maintain**.

⇒ **3 Key Parts of a Function**

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

### 1. Function Declaration (Prototype)

- Tells the compiler **what the function looks like**.
- Usually placed **above main()**, or in a header file.
- Includes the return type, function name, and parameters (if any).

### Syntax:

```
returnType functionName(parameterType1, parameterType2, ...);
```

### Example:

```
int add(int a, int b); // Function declaration
```

## 2. Function Definition

- This is where you **write the actual code** that the function performs.
- Must match the declaration.

### Syntax:

```
returnType functionName(parameters)
{
    // function body (code to run)
}
```

### Example:

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

### 3. Function Call

- This is how you **use** the function in your program.
- You "call" the function by its name and pass required arguments.

#### Example:

```
int result = add(5, 3); // Function call
```

## 2. What is the scope of variables in C++? Differentiate between local and global scope.

### What is *Scope* in C++?

In C++, the **scope of a variable** defines **where in the program the variable can be accessed or used**.

### Types of Variable Scope in C++

There are mainly two types:

1. **Local Scope**
2. **Global Scope**

#### 1. Local Scope

- A variable declared **inside a function or a block** ({ }) is called a **local variable**.
- It can **only be accessed within that function or block**.
- It is **created when the function/block runs** and **destroyed when it ends**.

#### Example:

```
void show()
{
    int x = 10; // Local variable
    cout << x << endl;
}
```

## 2. Global Scope

- A variable declared **outside all functions**, typically at the top of the program.
- It can be accessed **from any function in the same file** (after its declaration).
- It exists **throughout the program's lifetime**.

### Example:

```
int x = 100; // Global variable

void show()
{
    cout << x << endl; // Can access x here
}
```

### 3. Explain recursion in C++ with an example.

#### What is Recursion?

**Recursion** is a programming technique where a function **calls itself** to solve a smaller part of the problem.

In C++, a recursive function must have:

1. A **base case** – to stop the recursion.
2. A **recursive case** – where the function calls itself.

#### Simple Real-Life Analogy

Imagine you have a stack of plates. To count them:

- You take one plate.
- Ask someone to count the rest.
- When no plates are left, you stop.

That's recursion! Breaking a big problem into smaller versions of itself.

#### Example: Factorial Using Recursion

The **factorial** of a number  $n$  (written as  $n!$ ) is:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

#### With recursion:

$$n! = n \times (n-1)!$$

Base Case:  $0! = 1$

## Example:

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

int factorial(int n) // Recursive function
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1); // recursive call
}

int main()
{
    int num;

    cout << "Enter a positive number: ";
    cin >> num;

    if (num < 0)
    {
        cout << "Factorial is not negative numbers." << endl;
    }
    else
    {
        int result = factorial(num);
        cout << "Factorial of " << num << " is: " << result << endl;
    }
    return 0;
}
```



## 4. What are function prototypes in C++? Why are they used?

### What Are Function Prototypes in C++?

A **function prototype** in C++ is a **declaration of a function** that tells the compiler:

- The **function name**
- The **return type**
- The **parameter types** (but not necessarily the names)
- **Without providing the function body**

It lets the compiler know **how the function will be used later in the code**, even if the actual function definition comes after the call.

### Syntax of a Function Prototype:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

### Example:

```
int add(int, int); // This is a function prototype
```

## Why Are Function Prototypes Used?

| Purpose                       | Explanation   |
|-------------------------------|---|
| Inform the compiler early     | So it knows the function's signature before its actual definition appears.    |
| Enable type checking          | Ensures the function is called with the correct number and type of arguments. |
| Allow flexible code structure | Lets you place main() at the top and actual function code later.              |
| Useful in multiple files      | Prototypes can be placed in header files (.h) for sharing between files.      |

## 5. Arrays and Strings

### Lab Exercise:

#### 1. Array Sum and Average

- Write a C++ program that accepts an array of integers, calculates the sum and average, and displays the results.
- Objective: Understand basic array manipulation.

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

int main()
{
    int n, sum = 0;
    float average;
    int numbers[100];

    cout << "How many numbers? ";
    cin >> n;

    cout << "Enter the numbers:\n";
    for (int i = 0; i < n; i++)
    {
        cin >> numbers[i];
        sum += numbers[i];
    }

    average = (float)sum / n;

    cout << "Sum = " << sum << endl;
```

```
cout << "Average = " << average << endl;  
  
return 0;  
}
```

## 2. Matrix Addition

- Write a C++ program to perform matrix addition on two 2x2 matrices.
- Objective: Practice multi-dimensional arrays.

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

int main()
{
    int A[2][2], B[2][2], sum[2][2];

    // Input first matrix
    cout << "Enter elements of first 2x2 matrix:\n";
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> A[i][j];
        }
    }

    // Input second matrix
    cout << "Enter elements of second 2x2 matrix:\n";
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> B[i][j];
        }
    }

    // Add both matrices and store result in sum matrix
    for (int i = 0; i < 2; i++)
```

```
{
    for (int j = 0; j < 2; j++)
    {
        sum[i][j] = A[i][j] + B[i][j];
    }
}

// Output result
cout << "Sum of the two matrix is:\n";
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 2; j++)
    {
        cout << sum[i][j] << " ";
    }
    cout << endl;
}

return 0;
}
```

### 3. String Palindrome Check

- Write a C++ program to check if a given string is a palindrome (reads the same forwards and backwards).
- Objective: Practice string operations.

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

int main()
{
    string str, reversedStr;

    cout << "Enter a string: ";
    cin >> str;

    // Reverse the string
    for (int i = str.length() - 1; i >= 0; i--)
    {
        reversedStr += str[i];
    }

    if (str == reversedStr)
    {
        cout << str << " is a palindrome." << endl;
    }
    else
    {
        cout << str << " is not a palindrome." << endl;
    }

    return 0;
}
```

# Theory Exercise:

1. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

## What Are Arrays in C++?

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

Each element is accessed using an **index**, which starts from 0.

## Basic Syntax of an Array:

```
data_type array_name[size];
```

## Example:

```
int numbers[5]; // Declares an array of 5 integers
```

## Types of Arrays in C++

There are mainly two types of arrays:

| Type               | Meaning   |
|--------------------|---|
| Single-dimensional | A linear list of elements                                 |
| Multi-dimensional  | Arrays with 2 or more dimensions (like a table or matrix) |



## 1. Single-Dimensional Array

A **single-dimensional array** stores data in a **linear form (1D)** — like a list.

**Example:**

```
int arr[4] = {10, 20, 30, 40};
```

| Index | Value |
|-------|-------|
| 0     | 10    |
| 1     | 20    |
| 2     | 30    |
| 3     | 40    |

## 2. Multi-Dimensional Array

A **multi-dimensional array** stores data in **rows and columns** (like a table or matrix).

The most common type is the **two-dimensional array**.

### Declaration:

```
int matrix[2][3]; // 2 rows, 3 columns
```

### Example Initialization:

```
int matrix[2][3] =  
{  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

|       | Col 0 | Col 1 | Col 2 |
|-------|-------|-------|-------|
| Row 0 | 1     | 2     | 3     |
| Row 1 | 4     | 5     | 6     |

## Key Differences: 1D vs 2D Arrays

| Feature             | 1D Array                  | 2D Array                      |
|---------------------|---------------------------|-------------------------------|
| Structure           | Linear (like a list)      | Tabular (like a grid/matrix)  |
| Declaration example | <code>int a[5];</code>    | <code>int a[2][3];</code>     |
| Access element      | <code>a[i]</code>         | <code>a[i][j]</code>          |
| Use case examples   | Marks, prices, names list | Matrices, tables, game boards |
| Memory layout       | Continuous in 1 direction | Continuous in row-major order |

## 2. Explain string handling in C++ with examples.

### String Handling in C++

In C++, **strings** are sequences of characters. There are **two main ways** to handle strings:

#### 1. C-style Strings (Old way)

- Based on character arrays.
- End with a **null character** '\0'.
- Uses functions from <cstring> like strcpy(), strlen(), etc.

#### Example:

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

int main()
{
    char name[20];

    cout << "Enter your name: ";
    cin >> name;

    cout << "Length = " << strlen(name) << endl; // Counts characters

    return 0;
}
```

## 2. C++ String Class (Modern way)

- Provided by the `<string>` header.
- Safer and easier to use.
- Supports many built-in operations like concatenation, comparison, length, etc.

Common String Operations (C++ String Class)

### 1. Declare and Initialize

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

int main()
{
    string str1 = "Hello";
    string str2("World");

    cout << str1 << " " << str2 << endl; // Output: Hello World
    return 0;
}
```

### 2. Input and Output

```
string name;
cout << "Enter your name: ";
cin >> name; // Only reads a single word
cout << "Hello, " << name << "!" << endl;
For full lines (including spaces), use getline():
string fullName;
getline(cin, fullName);
```

### 3. Concatenation

```
string a = "Good";  
string b = "Morning";  
string result = a + " " + b;  
cout << result; // Output: Good Morning
```

### 4. Length of String

```
string text = "example";  
cout << "Length = " << text.length(); // Output: 7
```

### 5. Access Characters

```
string word = "Apple";  
cout << word[0]; // Output: A  
word[1] = 'u'; // Changes 'p' to 'u'  
cout << word; // Output: Auple
```

### 6. Compare Strings

```
string a = "hello";  
string b = "hello";  
  
if (a == b)  
{  
    cout << "Strings are equal";  
}  
else  
{  
    cout << "Strings are not equal";  
}
```

## 7. Other Useful Functions

| Function           | Description                 |
|--------------------|-----------------------------|
| s.length()         | Returns length of string    |
| s.empty()          | Checks if string is empty   |
| s.substr(pos, n)   | Returns substring           |
| s.find("text")     | Finds position of substring |
| s.erase(pos, n)    | Removes part of the string  |
| s.insert(pos, str) | Inserts string at position  |

### 3. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

#### How Are Arrays Initialized in C++?

In C++, **arrays can be initialized** at the time of declaration or later in the program. Let's look at examples of **both 1D and 2D arrays**.

#### 1. Initialization of One-Dimensional (1D) Arrays

##### Syntax:

```
data_type array_name[size] = {values};
```

##### Example 1: Full Initialization

```
int numbers[5] = {10, 20, 30, 40, 50};
```

- This creates an array of size 5 and fills it with the values.

##### Example 2: Partial Initialization

```
int numbers[5] = {10, 20};
```

- Only the first two elements are set.
- Remaining values are automatically initialized to **0**.

##### Example 3: Size Inferred from Values

```
int numbers[] = {1, 2, 3};
```

- Compiler automatically sets the size to 3.

##### Example 4: Default Initialization

```
int numbers[3] = {}; // All values will be 0
```



## 2. Initialization of Two-Dimensional (2D) Arrays

### Syntax:

```
data_type array_name[rows][columns] = { {row1}, {row2}, ... };
```

### Example 1: Full Initialization

```
int matrix[2][3] =  
{  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

|       | Col 0 | Col 1 | Col 2 |
|-------|-------|-------|-------|
| Row 0 | 1     | 2     | 3     |
| Row 1 | 4     | 5     | 6     |

### Example 2: Flat Initialization

```
int matrix[2][3] = {1, 2, 3, 4, 5, 6};
```

- Elements are filled row by row.

### Example 3: Partial Initialization

```
int matrix[2][3] =  
{  
    {1, 2},  
    {4}  
};
```

- Missing values are initialized to 0.

|       | Col 0 | Col 1 | Col 2 |
|-------|-------|-------|-------|
| Row 0 | 1     | 2     | 0     |
| Row 1 | 4     | 0     | 0     |

## 4. Explain string operations and functions in C++.

### String Operations and Functions in C++

In C++, strings are commonly handled using the **std::string** class, which provides a wide range of **operations** and **built-in functions** to manipulate and analyze strings easily.

You need to include:

```
#include <string>
```

### ⇒ Basic String Operations

#### 1. Declaration and Initialization

```
string s1 = "Hello";  
string s2("World");
```

#### 2. Input/Output

```
string name;  
cin >> name;    // Reads single word  
getline(cin, name); // Reads full line (with spaces)  
  
cout << "Hello, " << name;
```

#### 3. Concatenation

```
string a = "Good";  
string b = "Morning";  
string result = a + " " + b;  
  
cout << result; // Output: Good Morning
```

## 4. String Length

```
string word = "example";  
cout << word.length(); // Output: 7
```

## 5. Access Characters

```
string text = "Hello";  
cout << text[0];    // Output: H  
text[1] = 'a';      // Changes 'e' to 'a'  
cout << text;       // Output: Hallo
```

### ⇒ Useful String Functions

#### 1. length() / size()

Returns the number of characters in the string.

```
string s = "apple";  
cout << s.length(); // Output: 5
```

#### 2. empty()

Checks if the string is empty.

```
string s = "";  
if (s.empty())  
{  
    cout << "String is empty";  
}
```

#### 3. append()

Adds another string at the end.

```
string s = "Hello";  
s.append(" World");  
cout << s; // Output: Hello World
```

#### 4. substr(start, length)

Extracts a substring from the string.

```
string s = "Programming";  
string sub = s.substr(0, 4); // Output: "Prog"
```

#### 5. find(substring)

Finds the position of the first occurrence of a substring.

```
string s = "banana";  
int pos = s.find("na"); // Output: 2
```

#### 6. replace(pos, len, new\_str)

Replaces part of the string.

```
string s = "I like apples";  
s.replace(7, 6, "oranges");  
cout << s; // Output: I like oranges
```

#### 7. erase(pos, len)

Removes characters from the string.

```
string s = "Hello World";  
s.erase(5, 6);  
cout << s; // Output: Hello
```

#### 8. insert(pos, str)

Inserts characters at a given position.

```
string s = "Hello";  
s.insert(5, " World");  
cout << s; // Output: Hello World
```

## 9. compare()

Compares two strings (returns 0 if equal).

```
string a = "apple";  
string b = "banana";  
  
if (a.compare(b) == 0)  
    cout << "Equal";  
else  
    cout << "Not Equal";
```

## 6. Introduction to Object-Oriented Programming

### Lab Exercise:

#### 1. Class for a Simple Calculator

- Write a C++ program that defines a class Calculator with functions for addition, subtraction, multiplication, and division. Create objects to use these functions.
- Objective: Introduce basic class structure.

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

// Define the Calculator class
class Calculator
{
public:
    double add(double a, double b)
    {
        return a + b;
    }

    double subtract(double a, double b)
    {
        return a - b;
    }

    double multiply(double a, double b)
    {
```

```

        return a * b;
    }

    double divide(double a, double b)
    {
        if (b == 0)
        {
            cout << "Error: Division by zero!" << endl;
            return 0;
        }
        return a / b;
    }
};

int main()
{
    Calculator calc;
    double num1, num2;

    cout << "Enter First number: ";
    cin >> num1;

    cout << "Enter Second number: ";
    cin >> num2;

    cout << "Addition: " << calc.add(num1, num2) << endl;
    cout << "Subtraction: " << calc.subtract(num1, num2) << endl;
    cout << "Multiplication: " << calc.multiply(num1, num2) << endl;
    cout << "Division: " << calc.divide(num1, num2) << endl;

    return 0;
}

```



## 2. Class for Bank Account

- Create a class BankAccount with data members like balance and member functions like deposit and withdraw. Implement encapsulation by keeping the data members private.
- Objective: Understand encapsulation in classes.

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

class BankAccount
{
private:
    double balance; // private data member

public:
    // Constructor
    BankAccount(double initialBalance)
    {
        if (initialBalance >= 0)
            balance = initialBalance;
        else
            balance = 0;
    }

    // Deposit function
    void deposit(double amount)
    {
        if (amount > 0)
        {
            balance += amount;
            cout << "Deposited: " << amount << endl;
        }
    }
}
```

```

    else
    {
        cout << "Invalid deposit amount" << endl;
    }
}

// Withdraw function
void withdraw(double amount)
{
    if (amount > 0 && amount <= balance)
    {
        balance -= amount;
        cout << "Withdrew: " << amount << endl;
    }
    else
    {
        cout << "Invalid or insufficient balance" << endl;
    }
}

// Function to get current balance
double getBalance()
{
    return balance;
}
};

int main()
{
    BankAccount account(1000); // Create account with initial balance

    account.deposit(500);
    account.withdraw(200);

    cout << "Current Balance: " << account.getBalance() << endl;

    return 0;
}

```

### 3. Inheritance Example

- Write a program that implements inheritance using a base class Person and derived classes Student and Teacher. Demonstrate reusability through inheritance.
- Objective: Learn the concept of inheritance.

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

// Base class
class Person
{
public:
    string name;
    int age;

    void getDetails()
    {
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter age: ";
        cin >> age;
    }

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

// Derived class
```

```

class Student : public Person
{
public:
    string studentID;

    void getStudentDetails()
    {
        getDetails(); // reuse class
        cout << "Enter student ID: ";
        cin >> studentID;
    }

    void showStudentDetails()
    {
        cout << "\n--- Student Details ---\n";
        showDetails(); // reuse class
        cout << "Student ID: " << studentID << endl<<endl;
    }
};

// Derived class
class Teacher : public Person
{
public:
    string subject;

    void getTeacherDetails()
    {
        getDetails(); // reuse class
        cout << "Enter subject: ";
        cin >> subject;
    }

    void showTeacherDetails()
    {
        cout << "\n--- Teacher Details ---\n";
        showDetails(); // reuse class
    }
};

```

```
        cout << "Subject: " << subject << endl;
    }
};

// Main function
int main()
{
    Student s;
    Teacher t;

    s.getStudentDetails();
    s.showStudentDetails();

    t.getTeacherDetails();
    t.showTeacherDetails();

    return 0;
}
```

# Theory Exercise:

## 1. Explain the key concepts of Object-Oriented Programming (OOP).

### 1. Encapsulation

- **Definition:** Wrapping data (variables) and functions (methods) into a single unit — the **class**.
- **Goal:** Hide the internal details of an object and only expose what is necessary (e.g., through public methods).

### Example:

```
class BankAccount
{
private:
    double balance; // Hidden from outside

public:
    void deposit(double amount)
    {
        balance += amount;
    }

    double getBalance()
    {
        return balance;
    }
};
```

## 2. Abstraction

- **Definition:** Hiding complex implementation details and showing only the essential features.
- **Goal:** Make code easier to use and maintain.

### Example:

- You **use cout** to print, but don't need to know how it works internally.
- In your own class, you provide simple methods like `startCar()` instead of showing all engine processes.

### 3. Inheritance

- **Definition:** One class (child or derived class) inherits properties and behaviors from another (base class).
- **Goal:** Reuse code and build relationships between classes.

#### Example:

```
class Person
{
public:
    string name;
};

class Student : public Person
{
public:
    int studentID;
};
```

### 4. Polymorphism

- **Definition:** "Many forms" – the ability to use the same function name with different behaviors.
- **Types:**
  - **Compile-time** (Function Overloading)
  - **Run-time** (Function Overriding with Virtual Functions)

#### Example:

```
class Shape
{
public:
```



```
virtual void draw()
{
    cout << "Drawing shape" << endl;
}
};

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

## 2. What are classes and objects in C++? Provide an example.

### What Are Classes and Objects in C++?

In C++, **classes** and **objects** are fundamental concepts of **Object-Oriented Programming (OOP)**.

#### Class:

A **class** is a **blueprint or template** for creating objects. It defines **data members** (variables) and **member functions** (methods) that operate on the data.

Think of a class like a "recipe" or "design".

#### Object:

An **object** is a **real-world instance** of a class. It has its own values for the variables defined in the class.

□ Think of an object like a "cake" made from the "recipe" (class).

### Syntax of a Class in C++

```
class ClassName
{
    // Access specifier
public:
    // Data members (variables)
    int value;
```

```
// Member functions
void display()
{
    cout << "Value is: " << value << endl;
}
};
```

## Example: Class and Object in C++

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

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

    void displayInfo()
    {
        cout << "Brand: " << brand << endl;
        cout << "Year: " << year << endl;
    }
};

int main()
{
    // Create an object of the class Car
    Car myCar;

    // Assign values to the object's data members
    myCar.brand = "Toyota";
    myCar.year = 2022;

    // Call a member function
    myCar.displayInfo();
}
```

```
return 0;  
}
```

### 3. What is inheritance in C++? Explain with an example.

#### What is Inheritance in C++?

**Inheritance** is a core concept of Object-Oriented Programming (OOP) that allows a class (**derived class**) to **inherit** properties (data members) and behaviors (member functions) from another class (**base class**).

#### Why Use Inheritance?

- **Code reusability:** You don't have to rewrite code for common functionality.
- **Extensibility:** Easily extend or customize behavior in derived classes.
- **Logical hierarchy:** Models real-world relationships (e.g., Car is a type of Vehicle).

#### Syntax

```
class Base
{
    // base class members
};

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

## Example: Inheritance in C++

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

// Base class
class Person
{
public:
    string name;
    int age;

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

// Derived class
class Student : public Person
{
public:
    string studentID;

    void displayStudent()
    {
        displayInfo(); // call base class function
        cout << "Student ID: " << studentID << endl;
    }
};

int main()
{
    Student s1;

    // Accessing base class members
```

```
s1.name = "Alice";  
s1.age = 20;  
  
// Accessing derived class member  
s1.studentID = "S123";  
  
// Display all info  
s1.displayStudent();  
  
return 0;  
}
```

### Types of Inheritance in C++:

| Type         | Description   |
|--------------|---|
| Single       | One base → one derived class                            |
| Multiple     | One derived class → inherits from multiple base classes |
| Multilevel   | Derived from a derived class                            |
| Hierarchical | Multiple derived classes from one base                  |
| Hybrid       | Combination of two or more types                        |

## 4. What is encapsulation in C++? How is it achieved in classes?

### What is Encapsulation in C++?

**Encapsulation** is one of the fundamental concepts of **Object-Oriented Programming (OOP)**.

It refers to the **bundling of data (variables) and functions (methods)** that operate on that data into a **single unit (class)**. It also restricts **direct access** to some of the object's components — which is known as **data hiding**.

### Goal of Encapsulation:

- **Protect data** from unauthorized access or modification.
- Control how data is accessed or changed using methods.
- Improve code **security, maintainability, and modularity**.

### How is Encapsulation Achieved in C++?

Encapsulation is achieved by:

1. **Declaring data members as private** (cannot be accessed directly outside the class).
2. **Providing public getter/setter functions** to access or update the private data safely.



## Example: Encapsulation in C++

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

class Employee
{
private:
    int salary; //private data member

public:
    //Setter: sets value of salary
    void setSalary(int s)
    {
        if (s > 0)
            salary = s;
        else
            cout << "Invalid salary!" << endl;
    }

    //Getter: returns value of salary
    int getSalary()
    {
        return salary;
    }
};

int main()
{
    Employee emp;

    emp.setSalary(50000); //Safe access via setter
    cout << "Salary: " << emp.getSalary() << endl; //Access via getter

    return 0;
}
```

## Why Use Encapsulation?

| Benefit              | Explanation  |
|----------------------|--|
| Data Protection      | Prevents accidental or unauthorized access                         |
| Controlled Access    | Use logic in setters/getters to validate data                      |
| Code Maintainability | Internal implementation can change without affecting external code |
| Modularity           | Keeps code organized and easier to manage                          |