# Module 3 – Introduction to OOPS Programming

## ❖ THEORY EXERCISE:-

## Introduction to C++:-

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

**Ans:-** Here are the key differences between Procedural Programming and Object-Oriented Programming (OOP):

| Aspect | Procedural Programming | Object-Oriented Programming (OOP) |
|---|---|---|
| Approach | Follows a top-down approach | Follows a bottom-up approach |
| Focus | Focuses on functions and procedures (actions) | Focuses on objects (entities) that contain data and methods |
| Data Handling | Data is global or passed between functions | Data is encapsulated within objects and accessed via methods |
| Code Reusability | Limited; achieved using functions | High; achieved through inheritance and polymorphism |
| Security | Less secure, data can be accessed from any function | More secure; uses encapsulation and access specifiers (private, public, protected) |
| Examples of Languages | C, Pascal, Fortran | C++, Java, Python (supports both) |
| Modularity | Program is divided into functions | Program is divided into classes and objects |
| Real-world Modeling | Harder to model real-world problems directly | Easier to model real-world entities with objects |
| Function Overloading / Polymorphism | Not supported or limited | Supported (key feature in OOP) |
| Extensibility | Difficult to extend or modify | Easy to extend using concepts like inheritance |

## Que 2):- List and explain the main advantages of OOP over POP.

**Ans:-** Here are the main advantages of Object-Oriented Programming (OOP) over Procedural-Oriented Programming (POP):

### 1}Encapsulation:-

**OOP:** Data and functions are bundled together into **objects**. Data is hidden from outside access and can only be modified through **methods**.
**Advantage:** Increases **data security** and **integrity**.
**POP:** Data is often global or loosely passed between functions, making it more prone to accidental changes.

### 2}Reusability through Inheritance:-

**OOP:** Code can be reused using **inheritance**. A new class can inherit properties and methods from an existing class.
**Advantage:** Reduces code duplication and improves **code maintainability**.
**POP:** Functions can be reused, but there is **no concept of inheritance**.

### 3}Polymorphism:-

**OOP:** Supports **polymorphism**, allowing the same method to behave differently depending on the object calling it.
**Advantage:** Enhances **flexibility** and allows for **dynamic method behaviour**
**POP:** Does not natively support polymorphism.

### 4}Modularity:-

**OOP:** Code is organized into **classes and objects**, making it modular and easier to manage.
**Advantage:** Easier to debug, test, and scale large applications.
**POP:** Code is split into functions, but managing complexity becomes harder as the project grows.

### 5}Real-World Modeling:-

**OOP:** Objects represent real-world entities, making it easier to design systems based on **real-life scenarios**.
**Advantage:** Improves **conceptual clarity** and **design structure**.
**POP:** More abstract and less aligned with real-world concepts

### 6}Ease of Maintenance and Upgrades:-

**OOP:** Due to modular structure, making changes in one part of the program has **minimal impact** on others.
**Advantage:** Facilitates **scalability** and **long-term maintenance**.
**POP:** Changes can have widespread effects, leading to more **bugs** and complexity.

### 7}Extensibility:-

**OOP:** Easily extend existing functionality using **inheritance and method overriding**.
**Advantage:** Makes programs more **adaptable to change**.
**POP:** Requires rewriting or duplicating functions to extend features.

## Que 3):- Explain the steps involved in setting up a C++ development environment.

**Ans:-** Setting up a C++ development environment involves installing the necessary tools and configuring them so you can write, compile, and run C++ programs. Here are the step-by-step instructions for setting it up on different operating systems:

- **Basic Tools Needed:-**
  **1]Text Editor or IDE** (for writing code)
  **2]C++ Compiler** (for compiling code)
  **3]Debugger** (optional, often included in IDEs)

### 1. Choose Your Platform:-

### Windows:-

### Option 1: Using VS Code (advanced/flexible)

1]Install **VS Code** from code.visualstudio.com.
2] Install the **C/C++ extension** by Microsoft (from Extensions tab).
3] Install **MinGW** (or any GCC-based compiler):
>Download from https://www.mingw-w64.org
>Add bin folder to the system PATH
4] Verify in terminal:
>g++ --version
5] Create and compile code:
>Save file as program.cpp
>Compile with: g++ program.cpp -o program.exe
>Run with: ./program.exe

### macOS:-

1]Install Xcode Command Line Tools**:**
>xcode-select --install
2] (Optional) Install **VS Code** and the **C++ extension**.
3] Compile code using terminal:
>g++ program.cpp -o program
./program

### Linux (Ubuntu/Debian):-

1] Open terminal and install compiler:
>sudo apt update
>sudo apt install g++
2] Use any text editor (like VS Code**,** Vim**,** Geany**).**
3] Compile and run:
>g++ program.cpp -o program
./program

## 2. Write a Test Program:-

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

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

>Save as hello.cpp, compile and run using your chosen method.

## 3. Verify Setup:-

1]Check compiler: g++ --version
2] Run a simple "Hello World" program
3] Ensure no errors on compile and output is shown

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

**Ans:-** In C++, input/output (I/O) operations are performed using streams provided by the iostream library.

- **Main I/O Operations in C++:-**

| Operation Type | Object Used | Purpose |
|---|---|---|
| Output | Cout | Display data to user |
| Input | Cin | Read data from user |

Both are part of the iostream header.

## 1. Input with cin:-

>cin is used to take input from the standard input device (usually keyboard).
> It uses the extraction operator >>.

### Example:-

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

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

## 2. Output with cout:-

>cout is used to print output to the standard output device (usually screen).
> It uses the insertion operator <<.

**Example:-**

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

    main() {
   string name = "Alice";
   cout << "Hello, " << name << "!" << endl;
}
```

### 3. Taking Multiple Inputs:-

**Example:-**

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

   main() {
   int a, b;
   cout << "Enter two numbers: ";
   cin >> a >> b;
   cout << "Sum is: " << (a + b) << endl;
}
```

### 4. Reading Full Lines with getline():-

>cin stops reading at the first whitespace.
>Use getline() for full-line input (like names with spaces).

**Example:-**

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

   main() {
   string fullName;
   cout << "Enter your full name: ";
   getline(cin, fullName);
   cout << "Hello, " << fullName << "!" << endl;
}
```

## Variables, Data Types, and Operators:-

### Que 1):- What are the different data types available in C++? Explain with examples.

**Ans:-** C++ provides a rich set of **data types** that are used to declare variables and define the type of data they can hold. These data types are broadly categorized as:-

## 1. Primary (Built-in) Data Types:-

| Data Type | Description | Example |
|---|---|---|
| Int | Integer values | int age = 25; |
| Float | Single-precision decimal | float pi = 3.14f; |
| Double | Double-precision decimal | double g = 9.81; |
| Char | Single character | char grade = 'A'; |
| Bool | Boolean (true/false) | bool passed = true; |
| void | No value (used for functions) | void display(); |

## 2. Derived Data Types:-

| Type | Description | Example |
|---|---|---|
| Array | Collection of elements of same type | int numbers[5]; |
| Pointer | Stores memory address of another variable | int* ptr = &age; |
| Reference | Alias to another variable | int& ref = age; |
| Function | Represents a block of code | int add(int a, int b); |

## 3. User-Defined Data Types:-

| Type | Description | Example |
|---|---|---|
| Struct | Groups different data types | struct Person {string name; int age;}; |
| Class | Blueprint for objects (OOP) | class Car { public: string model; }; |
| Union | Shares memory for all members | union Data {int i; float f;}; |
| enum | Defines a set of named constants | enum Color {Red, Green, Blue}; |

## 4. Modifiers for Built-in Types:-

| Modifier | Description | Example |
|---|---|---|
| Signed | Default for integers (can be + or -) | signed int x = -10; |
| Unsigned | Only positive numbers | unsigned int y = 100; |
| Short | Uses less memory | short int s = 32767; |

| | | |
|---|---|---|
| Long | Uses more memory | long int l = 123456789; |
| long long | Very large numbers | long long big = 1e18; |

**Example:-**

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

    main() {
   int age = 20;
   float height = 5.9f;
   char grade = 'A';
   bool isPassed = true;
   double pi = 3.1415926535;

   struct Student {
      string name;
      int id;
   };

   Student s = {"Dhruvin", 101};
   cout << "Age: " << age << endl;
   cout << "Height: " << height << endl;
   cout << "Grade: " << grade << endl;
   cout << "Passed: " << isPassed << endl;
   cout << "Pi: " << pi << endl;
   cout << "Student: " << s.name << ", ID: " << s.id << endl;
}
```

## Que 2):- Explain the difference between implicit and explicit type conversion in C++.

**Ans:-** In C++, type conversion refers to converting a variable from one data type to another. There are two types of type conversions:

## 1. Implicit Type Conversion (Type Promotion):-

- **Definition:-**

>Automatically performed by the compiler.
> Happens when you assign a value of one type to a variable of another compatible type.

- **Key Points:-**

> Also known as automatic type conversion.
> Usually occurs in expressions where operands are of different types.
> Follows type hierarchy (e.g., int → float → double).

**Example:-**

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

   main() {
   int a = 5;
   float b = 2.5;
   float result = a + b; // 'a' is implicitly converted to float
   cout << "Result: " << result << endl;
   return 0;
}
```

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

- **Definition:-**

> Manually performed by the programmer using type casting syntax.
> Converts a variable to a specific type on purpose.

- **Key Points:-**

> Helps control the way values are converted.
> Useful when data might be truncated or interpreted differently
> Safer in terms of logic clarity.

**Example:-**

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

   main() {
   int a = 5, b = 2;
   float result = (float)a / b; // explicit conversion
   cout << "Result: " << result << endl;
   return 0;
}
```

**Key Differences:-**

| Feature | Implicit Conversion | Explicit Conversion |
|---|---|---|
| Performed by | Compiler | Programmer |
| Control | Automatic, no programmer control | Fully under programmer's control |
| Risk | May lead to unexpected results | Safer if used correctly |
| Syntax | No special syntax | Uses casting: (type) value |

| | | |
|---|---|---|
| Use Case | ype promotion in expressions | Precise conversion, preventing truncation |

**Que 3):- What are the different types of operators in C++? Provide examples of each.**

**Ans:-** In C++, operators are special symbols used to perform operations on variables and values. They are categorized based on their functionality.

## 1. Arithmetic Operators:-

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

## 2. Logical Operators:-

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

## 3. Assignment Operators:-

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment | a = 10 |
| += | Add and assign | a += 5 (a = a + 5) |
| -= | Subtract and assign | a -= 3 |
| *= | Multiply and assign | a *= 2 |

| | | |
|---|---|---|
| /= | Divide and assign | a /= 4 |
| %= | Modulus and assign | a %= 3 |

## 4. Relational Operators:-

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

## 5. Bitwise Operators:-

| 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 >> 2 |

**Example:-**

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

   main() {
   int a = 10, b = 3;
   cout << "Add: " << a + b << endl;          // Arithmetic
   cout << "Equal: " << (a == b) << endl;     // Relational
   cout << "Logic: " << (a > 5 && b < 5) << endl; // Logical
```

```
    a += 5;                          // Assignment
    cout << "New a: " << a << endl;
    cout << "Size of int: " << sizeof(int) << " bytes" << endl; // Special
    return 0;
}
```

## Que 4):- Explain the purpose and use of constants and literals in C++.

**Ans:-** In C++, constants and literals are used to represent fixed values that do not change during program execution. They are essential for improving code readability, reliability, and maintainability.

## 1. Constants:-

- **Definition:-**

> A constant is a variable whose value cannot be changed once it is assigned.

- **Purpose:-**

> Prevent accidental modification of values
> Make code easier to understand
> Improve maintainability

- **Ways to Declare Constants:-**

1]Using const Keyword:-
const float PI = 3.14159;

2]Using #define Preprocessor Directive (Old Style)
#define MAX_SIZE 100

3]Using constexpr (C++11 onwards)
constexpr int square = 25;

### Example:-

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

  main() {
 const int DAYS_IN_WEEK = 7;
    cout << "There are " << DAYS_IN_WEEK << " days in a week." << endl;
    return 0;
}
```

## 2. Literals:-

- **Definition:-**

> A literal is a fixed value used directly in the code. It represents constant values of various types.

- **Types of Literals in C++:-**

| Type | Example | Description |
|---|---|---|
| Integer | 10, -25 | Whole numbers |
| Floating Point | 3.14, -0.001 | Numbers with decimal points |
| Character | 'A', '9' | Enclosed in single quotes |
| String | "Hello" | Enclosed in double quotes |
| Boolean | true, false | Represent logical values |
| Null Pointer | nullptr (C++11) | Represents a null pointer |

**Example:-**

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

   main() {
   int age = 20;          // 20 is an integer literal
   float pi = 3.14;       // 3.14 is a float literal
   char grade = 'A';      // 'A' is a character literal
   string name = "Alice";  // "Alice" is a string literal

   cout << name << " is " << age << " years old." << endl;
}
```

# Control Flow Statements:-

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

**Ans:-**

Conditional statements in C++ are used to make decisions in a program based on certain conditions. They allow the program to execute specific blocks of code only when certain criteria are met.

Conditional statements in C++ are programming constructs that enable a program to make decisions and execute different blocks of code based on whether specific conditions are true or false. They are fundamental for controlling the flow of a program and implementing decision-making logic.

- **Purpose of Conditional Statements:-**

> To control the flow of execution in a program.
> To perform different actions based on different inputs or states.
> To implement decision-making logic (e.g., if the user enters 10, do X; otherwise, do Y).

### 1}if, else if, and else Statements:-

- **Purpose:-**

> To execute different code blocks based on Boolean conditions**.**

## Syntax:-

```
if (condition) {
   // Executes if condition is true
} else if (another_condition) {
   // Executes if another condition is true
} else {
   // Executes if none of the above are true
}
```

### Example:-

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

  main() {
   int age = 18;

   if (age < 13) {
      cout << "Child" << endl;
   } else if (age < 20) {
      cout << "Teenager" << endl;
   } else {
      cout << "Adult" << endl;
   }
}
```

## 2}Switch Statement:-

- **Purpose:-**

> To select one of many code blocks to be executed, based on the value of a variable (usually int, char, or enum).

## Syntax:-

```
switch(expression){
   case 1:
      // Code block
      break;
   case
      // Code block
      break;
   default:
      // Code block
}
```

**Example:-**

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

   main() {
   int day = 3;

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

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

**Ans:-** In C++, for, while, and do-while are looping statements used to execute a block of code repeatedly based on a condition. The key difference lies in how and when the condition is evaluated**.**

### 1. for Loop:-

### Syntax:-

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

**Example:-**

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

### 2. while Loop:-

### Syntax:-

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

**Example:-**

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

### 3. do-while Loop:-

### Syntax:-

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

**Example:-**

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

### Key Differences:-

| Feature | for Loop | while Loop | do-while Loop |
|---|---|---|---|
| Condition Check | Before each iteration | Before each iteration | **After** each iteration |
| Guaranteed Execution | No | No | Yes, at least once |
| Use Case | Known number of iterations | Unknown, condition-controlled | Run once, then check condition |
| Structure | Compact (all in one line) | Split: init outside, update inside | Split: init outside, update inside |

## Que 3):- How are break and continue statements used in loops? Provide examples.

**Ans:-** In C++, break and continue statements are used to control the flow of loops. They help manage how and when a loop should stop or skip an iteration.

### 1. break Statement:-

- **Purpose:-**

> To exit the loop immediately, regardless of the loop condition.

- **Use Case:-**

> Exiting early when a condition is met
> Common in switch, for, while, and do-while loops

**Example:-**

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

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

## 2. continue Statement:-

- **Purpose:-**

> To skip the current iteration and continue with the next one.

- **Use Case:-**

> Skip unwanted steps
> Skip unwanted steps

**Example:-**

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

    main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3)
         continue; // skips the rest of loop body when i is 3
      cout << i << " ";
   }
}
```

**Break and continue statement both Example:-**

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

   main() {
   int i = 0;
   while (i < 10) {
      i++;
      if (i == 6)
         break;
      if (i % 2 == 0)
```

```
        continue;
      cout << i << " ";
   }
}
```

## Que 4):- Explain nested control structures with an example.

**Ans:-** Nested control structures are control statements (like if, for, while, or switch) placed inside other control structures**.** They allow you to write more complex decision-making and looping logic by combining multiple layers of control.

- **Types of Nested Control Structures:-**

**1]** Nested if statements
2] if inside a for or while loop
3] for loop inside another for loop
4] switch inside if, or vice versa

- **Why Use Nested Structures?**

> To handle multi-level decision making
> To perform actions within loops based on conditions
> To build complex logic in a structured way.

### Example 1: Nested if Statements:-

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

   main() {
   int age = 20;
   char gender = 'M';

   if (age >= 18) {
      if (gender == 'M') {
         cout << "You are an adult male." << endl;
      } else {
         cout << "You are an adult female." << endl;
      }
   } else {
      cout << "You are a minor." << endl;
   }
}
```

### Example 2: Nested for Loops:-

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

   main() {
   for (int i = 1; i <= 3; i++) {
      for (int j = 1; j <= 5; j++) {
```

```
            cout << i * j << "\t";
        }
        cout << endl;
    }
}
```

**Example 3: if Inside a Loop:-**

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

    main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            cout << i << " is even" << endl;
        } else {
            cout << i << " is odd" << endl;
        }
    }
}
```

## Functions and Scope:-

### Que 1):- What is a function in C++? Explain the concept of function declaration, definition, and calling.

**Ans:-** A function in C++ is a block of code designed to perform a specific task. Instead of writing the same code again and again, you can define a function once and reuse it whenever needed.

A function in C++ is a self-contained block of code designed to perform a specific task. It can optionally accept input data (parameters), process that data, and potentially return a result. Functions are fundamental to structured programming and offer several key advantages

- **Purpose of Functions:-**

**1]** To organize code into logical blocks
2] To avoid repetition (code reusability)
3] To make programs easier to read, maintain, and test

**Basic Structure of a Function:-**

```
return_type function_name(parameters) {
    // body of the function
    return value;
}
```

**Example:-**

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

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

int main() {
   int result = add(5, 3); // Function call
   cout << "Sum is: " << result;
}
```

- **Types of Functions in C++:-**

| Type | Example / Use Case |
|---|---|
| User-defined | Created by the programmer |
| Built-in (library) | Provided by C++ (e.g., main(), sqrt()) |
| Inline functions | Suggested to be expanded in-line for speed |
| Recursive functions | Call themselves to solve a problem recursively |

- **Function Components:-**

| **Component** | Description | Example |
|---|---|---|
| Return type | Type of value returned | int, void, float |
| Function name | Identifier for the function | add, display, main |
| Parameters | Inputs to the function (optional) | int a, int b |
| Function body | Code that defines the task | { return a + b; } |

- **Function Calling:-**
  greet(); // calls the function

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

**Ans:-** In C++, the scope of a variable refers to the region of the program where the variable is declared, accessible, and valid.
In C++, the scope of a variable defines the region of the program where that variable is visible and accessible. It determines where the variable can be referenced and used within the code. Understanding variable scope is crucial for managing memory, preventing naming conflicts, and writing organized and efficient C++ programs.

- Understanding scope is important to:-

> Avoid naming conflicts
> Manage memory efficiently

- **Types of Variable Scope:-**

**1] Local Scope**: Variable declared within a function or block.
**2] Global Scope**: Variable declared outside all functions.

- **Difference Between Local and Global Scope:-**

| Feature | Local Scope | Global Scope |
|---|---|---|
| Declaration | Inside a function or block {} | Outside all functions |
| Access | Only within the function or block it is declared | Accessible throughout the program |
| Lifetime | Created when the block is entered, destroyed when exited | Exists from program start to end |
| Memory Usage | Stack memory | Static/Global memory |
| Name Conflict | Can override global variables inside its block | Can be accessed using :: if shadowed |

## Que 3):- Explain recursion in C++ with an example.

## Ans:-

Recursion is a programming technique where a function calls itself to solve a problem. In C++, recursion allows problems to be broken down into smaller sub-problems.
Recursion in C++ is a programming technique where a function calls itself, either directly or indirectly, to solve a problem. This technique is typically used for problems that can be broken down into smaller, self-similar subproblems.

- **Key Concepts of Recursion:-**

**1] Base Case** – A condition that stops the recursion.
2] **Recursive Case** – The function calls itself with a smaller/simpler input.

- **Advantages of Recursion:-**

> Simplifies code for problems that have **natural recursive structure** (e.g., trees, backtracking).
> Improves readability and elegance.

- **Disadvantages:-**

> Uses more memory (call stack)**.**
> Can lead to stack overflow if base case is missing or unreachable.

- **Example: Factorial Using Recursion:-**

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

// Recursive function to calculate factorial
  factorial(int n) {
```

```
   if (n <= 1)         // base case
      return 1;
   else
      return n * factorial(n - 1);  // recursive case
}

main() {
   int number = 5;
   cout << "Factorial of " << number << " is " << factorial(number);
}
```

## Que 4):- What are function prototypes in C++? Why are they used?

## Ans:-

A function prototype in C++ is a declaration of a function that tells the compiler:
> The **function name**
> The **return type**
>The **parameter types** (and optionally their names)
It gives the compiler early information about the function—before it is defined or used in the code.
In C++, a function prototype is a declaration that provides the compiler with essential information about a function before its actual definition appears in the code. It essentially acts as a "blueprint" or "signature" for the function.

## Syntax:-

return_type function_name(parameter_list);

### Example:-

```
// Function prototype
int add(int, int);

   main() {
   cout << add(5, 3);  // Function call
   return 0;
}

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

- **Why Are Function Prototypes Used?**

| Purpose | Explanation |
|---|---|
| Tell the compiler about a function | So it can recognize and correctly compile function calls made **before** the function is defined. |
| Enable top-down programming | You can write main() first, then define the functions later. |
| Support separate compilation (header files) | Prototypes go in header files to share between multiple source files. |
| Ensure correct type checking | Compiler checks that function calls use the correct number and types of arguments. |

- **Without a Prototype – What Can Go Wrong?**

> Produce an error
> Or make incorrect assumptions about return type or parameters (in older compilers)

# Arrays and Strings:-

## Que 1):- What are arrays in C++? Explain the difference between single-dimensional and multidimensional arrays

**Ans:-**

An array in C++ is a collection of elements of the same data type, stored in contiguous memory locations. It allows you to store and access multiple values using a single variable name, with the help of an index

In C++, an array is a data structure that stores a fixed-size sequential collection of elements of the same data type in contiguous memory locations. This means all elements of an array are stored next to each other in memory, allowing for efficient access.

## Syntax:-

data_type array_name[size];

- **Types of Arrays:-**

**1] One-dimensional:-** A single row of elements
2] **Multi-dimensional:-** Arrays with more than one index (e.g., 2D arrays like matrices)

## 1. Single-Dimensional Array (1D Array):-

- **Definition:-**

> A single row or column of elements accessed using one index**.**

## Syntax:-

data_type array_name[size];

**Example:-**
```
int numbers[5] = {10, 20, 30, 40, 50};
cout << numbers[2];
```

## 2. Multidimensional Array (e.g., 2D Array):-

- **Definition:-**

> An array of arrays. A grid or table-like structure, accessed using multiple indices**.**

## Syntax:-
```
data_type array_name[rows][columns];
```

**Example:-**
```
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
cout << matrix[1][2];
```

- **Comparison Table:-**

| Feature | 1D Array | Multidimensional Array |
|---|---|---|
| Structure | Linear (single row/column) | Table-like (rows and columns) |
| Indices used | One index (array[i]) | Two or more indices (array[i][j]) |
| Common use cases | List of values, marks, prices | Matrix, tables, grids |
| Memory layout | Contiguous in 1 line | Still stored linearly in memory |
| Complexity | Simple | More complex to access and manage |

## Que 2):- Explain string handling in C++ with examples.

**Ans:-** In C++, strings are used to store and manipulate sequences of characters (text). There are two main ways to handle strings:

### 1. C-Style Strings (char arrays):-

- **Definition:-**

> C-style strings are **arrays of characters** that end with a **null character (\0)**.

- **Declaration & Initialization:-**

```
char name[10] = "Dhruvin";
```

**Common C-String Functions (in <cstring>):**
> strlen() – Find string length
> strcpy() – Copy one string to another
> strcat() – Concatenate strings
> strcmp() – Compare two strings

**Example:-**
```
#include <iostream>
#include <cstring>
using namespace std;

   main() {
   char str1[20] = "Hello";
   char str2[] = "World";

   strcat(str1, str2); // str1 becomes "HelloWorld"
   cout << "Combined: " << str1 << endl;

   cout << "Length: " << strlen(str1); // Outputs 10
}
```

## 2. C++ Strings (std::string class):-

- **Definition:-**

> C++ provides the string class in the <string> header for more powerful and easier string handling.

- **Declaration & Initialization:-**

```
#include <string>
string name = "Dhruvin";
```

- **Common String Operations:-**

| Operation | Syntax |
|---|---|
| Concatenation | s1 + s2 |
| Length | s.length() |
| Access characters | s[i] |
| Substring | s.substr(pos, len) |
| Compare | s1 == s2, s1 < s2 etc. |
| Input with spaces | getline(cin, s) |

**Example:-**
```
#include <iostream>
#include <string>
using namespace std;
```

```
 main() {
   string name = "John";
   string greeting = "Hello, " + name;

   cout << greeting << endl;
   cout << "Length: " << greeting.length() << endl;
   cout << "First letter: " << greeting[0] << endl;
}
```

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

**Ans:-** In C++, arrays can be initialized in several ways at the time of declaration. Below are examples of initializing 1D (one-dimensional) and 2D (two-dimensional) arrays.

## 1}One-Dimensional Array (1D Array):-

## Syntax:-

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

**Example:-**

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

## 2}Two-Dimensional Array (2D Array):-

## Syntax:-

```
data_type array_name[rows][columns] = {
   {row1_col1, row1_col2, ...},
   {row2_col1, row2_col2, ...},
   ...
};
```

**Example:-**

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

- **Example Program Using 1D and 2D Arrays:-**

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

   main() {
   // 1D Array
   int marks[4] = {75, 80, 85, 90};
   cout << "1D Array:\n";
```

```
    for (int i = 0; i < 4; i++) {
        cout << marks[i] << " ";
    }

    // 2D Array
    int table[2][3] = {{1, 2, 3}, {4, 5, 6}};
    cout << "\n\n2D Array:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << table[i][j] << " ";
        }
        cout << endl;
    }
}
```

## Que 4):- Explain string operations and functions in C++.

**Ans:-** In C++, string operations and functions are crucial for manipulating and managing text data. C++ supports two types of strings:

**1]** C-style strings (char arrays)
**2]** C++ string class (from the Standard Template Library - STL)
Let's look at string operations and functions using both types:

1. C-Style Strings **(char[]):-**
C-style strings are arrays of characters ending with a null character '\0'.
Common Operations with <cstring> Functions:

| **Function** | **Description** |
|---|---|
| strlen(str) | Returns length of str (excluding \0) |
| strcpy(dest, src) | Copies src to dest |
| strcat(dest, src) | Appends src to dest |
| strcmp(str1, str2) | Compares two strings |
| strchr(str, ch) | Finds first occurrence of character ch in str |
| strstr(str, substr) | Finds first occurrence of substring in str |

**Example:-**

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

    main() {
    char str1[20] = "Hello";
    char str2[] = "World";

    strcat(str1, str2);       // str1 = "HelloWorld"
```

```
    cout << "Concatenated: " << str1 << endl;

    cout << "Length: " << strlen(str1) << endl;

    if (strcmp(str1, "HelloWorld") == 0)
        cout << "Strings are equal" << endl;
}
```

## 2. C++ string Class (std::string):-

More convenient and safer than C-style strings. Requires #include <string>.

- **Common String Operations:-**

| Operation | Example | Description |
|---|---|---|
| Concatenation | s1 + s2 | Join two strings |
| Append | s1.append(s2) | Adds s2 to s1 |
| Length | s.length() | Returns number of characters |
| Access | s[i] | Access individual character |
| Compare | s1 == s2 | Checks equality |
| Substring | s.substr(pos, len) | Gets substring |
| Find | s.find("text") | Finds index of first match |
| Replace | s.replace(pos, len, "new") | Replaces part of string |
| Erase | s.erase(pos, len) | Deletes characters |

**Example:-**

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

    main() {
    string s1 = "Hello";
    string s2 = "World";

    string s3 = s1 + " " + s2;    // Concatenation
    cout << "Combined: " << s3 << endl;

    cout << "Length: " << s3.length() << endl;

    s3.replace(6, 5, "C++");      // Replace "World" with "C++"
    cout << "After replace: " << s3 << endl;

    cout << "Substring: " << s3.substr(0, 5) << endl;
}
```

# Introduction to Object-Oriented Programming:-

**Que 1):- Explain the key concepts of Object-Oriented Programming (OOP).**

**Ans:-** Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects"**,** which bundle data (attributes) and functions (methods) together. It helps in designing software that is modular, reusable, and easier to maintain.

- **Key Concepts of OOP:-**

**1. Class:-**

**>** A class is a blueprint for creating objects.
> It defines properties (data members) and behaviors (member functions).

**Example:-**

```
class Car {
public:
   string brand;
   void start() {
      cout << "Car started";
   }
};
```

**2. Object:-**

**>** An object is an instance of a class.
> It represents a real-world entity like a car, person, etc.

**Example:-**

```
Car myCar;   // myCar is an object of class Car
myCar.brand = "Kia";
myCar.start();
```

**3. Encapsulation:-**

**>** Wrapping of data and functions into a single unit (class).
> Keeps data private and provides access through public methods**.**
> Improves security and prevents unauthorized access.

**Example:-**

```
class BankAccount {
private:
   int balance;

public:
   void deposit(int amount) {
```

```
        balance += amount;
    }
    int getBalance() {
        return balance;
    }
};
```

### 4. Abstraction:-

> Hides complex implementation details and shows only essential features**.**
> Achieved using access specifiers (private, public, protected).

### Example:-

```
class Printer {
public:
    void printDocument() {
        // Hides low-level details
        loadPaper();
        sendToPrinter();
    }

private:
    void loadPaper() {}
    void sendToPrinter() {}
};
```

### 5. Inheritance:-

> Enables one class to inherit properties and methods from another class.
> Promotes code reusability and a logical hierarchy.

### Example:-

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

class Dog : public Animal {
public:
    void bark() {
        cout << "Barking...";
    }
};
```

### 6. Polymorphism:-

> Allows a function or method to behave differently based on context.
> Types:
1] **Compile-time Polymorphism** (Function Overloading, Operator Overloading)
2] **Run-time Polymorphism** (Function Overriding using Virtual Functions)

**Example (Function Overloading):-**

```
class Print {
public:
   void show(int x) {
      cout << "Integer: " << x;
   }
   void show(string s) {
      cout << "String: " << s;
   }
};
```

**Example (Function Overriding):-**

```
class Base {
public:
   virtual void display() {
      cout << "Base display";
   }
};

class Derived : public Base {
public:
   void display() override {
      cout << "Derived display";
   }
};
```

## Que 2):- What are classes and objects in C++? Provide an example.

**Ans:-** In **C++**, a class is a user-defined data type that serves as a blueprint for creating objects**.** It encapsulates data members (variables) and member functions (methods) that operate on the data.
An object is an instanc**e** of a class — it represents a real-world entity with its own data and behavior.

- **Class:-**

> Defines what an object will contain and what it can do.
> Declared using the keyword class.

- **Object:-**

**>** Created using the class.
> Can access public data members and member functions using the dot operator (**.**)

## Syntax:-

```
class ClassName {
public:
   // data members
   // member function};
```

**Example:-**

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

// Class definition
class Car {
public:
   string brand;
   int year;

   void start() {
      cout << brand << " is starting..." << endl;
   }

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

// Main function
int main() {
   // Creating an object of class Car
   Car myCar;

   // Assigning values to the object
   myCar.brand = "Toyota";
   myCar.year = 2020;

   // Calling member functions using object
   myCar.start();
   myCar.displayInfo();

}
```

## Que 3):- What is inheritance in C++? Explain with an example

**Ans:-** Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) in C++.
It allows a class (called the derived or child class) to inherit properties and behaviors (data members and member functions) from another class (called the base or parent class).
This mechanism promotes code reusability as the derived class can leverage the functionality already defined in the base class, rather than having to redefine it. It also establishes an "is-a" relationship between classes, meaning the derived class is a specialized type of the base class (e.g., a "Dog is an Animal").

- **Key Features:-**

> Promotes code reusability.
> Models real-world relationships (e.g., Dog is a Animal).
> Allows for extending existing functionality.

- **Types of Inheritance in C++:-**

**1] Single Inheritance** – One derived class from one base class.
2] **Multiple Inheritance** – One derived class inherits from multiple base classes.
3] **Multilevel Inheritance** – A class inherits from a class which itself inherits from another class.
4] **Hierarchical Inheritance** – Multiple classes inherit from one base class.
5] **Hybrid Inheritance** – Combination of more than one type.

**Example of Single Inheritance:-**

```
#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;
   }
};

// Main function
int main() {
   Dog myDog;

   myDog.eat();   // Inherited from Animal
   myDog.bark();  // Defined in Dog
}
```

## Que 4):- What is encapsulation in C++? How is it achieved in classes?

**Ans:-** Encapsulation is an object-oriented programming (OOP) principle that refers to wrapping data (variables) and functions (methods) that operate on the data into a single unit — usually a class.
It is used to protect data from direct access and ensure controlled access through public methods.
Encapsulation in C++ is a fundamental concept of Object-Oriented Programming (OOP) that involves bundling data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class. It is often described as "data hiding" because it aims to restrict direct access to an object's internal state from outside the class

- **Key aspects of Encapsulation in C++:**

**1]Bundling Data and Methods:-**
Encapsulation combines related data members and member functions within a class definition. This creates a self-contained unit where the data is managed and manipulated by its own dedicated methods.

2] **Data Hiding (Information Hiding):-**
A core principle of encapsulation is to protect the internal state of an object from unauthorized or accidental modification. This is achieved through access specifiers like private and protected. Data members are typically declared as private to prevent direct external access.

**3]Controlled Access:-**
While direct access to private data is restricted, encapsulation provides controlled access through public member functions, often referred to as "getter" and "setter" methods. These methods allow external code to retrieve or modify the private data in a controlled and validated manner.

- **How is Encapsulation Achieved in C++?**

By using **access specifiers**:
> private: Members are **not accessible** outside the class.
> public: Members are **accessible** from outside.
> protected: Accessible only by the class and derived classes.

## Example:-

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

class BankAccount {
private:
   int balance;  // private data (hidden from outside)

public:
   // Constructor
   BankAccount() {
      balance = 0;
   }

   // Setter method
   void deposit(int amount) {
      if (amount > 0) {
         balance += amount;
      }
   }

   // Getter method
   int getBalance() {
      return balance;
   }
};
```

```
int main() {
    BankAccount myAccount;

    // myAccount.balance = 1000;
    myAccount.deposit(500);
    cout << "Balance: " << myAccount.getBalance() << endl;

}
```