# Module-3(Theory)
# Introduction to OOP Programming

## *1.* Introduction to C++

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

**Key Differences between Procedural Programming and Object-Oriented Programming**

1. **Approach**
   - Procedural Programming uses a **top-down approach** where problems are solved step by step using procedures or functions.
   - OOP uses a **bottom-up approach** where problems are solved by creating objects that combine data and functions.
2. **Focus**
   - Procedural Programming focuses on **procedures (functions)** that operate on data.
   - OOP focuses on **objects** that contain both data (attributes) and functions (methods).
3. **Data Security**
   - In Procedural Programming, data is **global** and can be accessed by any function, making it less secure.
   - In OOP, data is **encapsulated** inside objects, and access is controlled, making it more secure.
4. **Reusability**
   - Procedural Programming has less code reusability.
   - OOP supports high reusability through concepts like **classes** and **inheritance**.
5. **Examples**
   - Procedural: C, Fortran.
   - OOP: C++, Java

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

1. **Better Data Security**
   - In OOP, data is hidden inside objects and can only be accessed through defined functions (methods). This prevents accidental changes and protects

important data, unlike POP where global data can be modified by any function.

2. **Reusability of Code**
    - OOP uses **classes** and **inheritance**, allowing code to be reused in new programs without rewriting it. In POP, reusability is limited because functions are tied to specific tasks.
3. **Easy Maintenance and Modification**
    - In OOP, changes can be made in a single class without affecting the entire program, making it easier to update and maintain. In POP, changing global data or functions can affect many parts of the program.
4. **Real-World Modeling**
    - OOP represents real-world entities as **objects**, making programs easier to understand and design. POP focuses on steps and procedures, which can be harder to relate to real-world problems.
5. **Reduced Complexity through Abstraction**
    - OOP hides unnecessary details using **abstraction**, letting programmers work with simple, high-level concepts. POP usually exposes all details, which can make large programs complex.
6. **Flexibility through Polymorphism**
    - OOP allows one function or method to work in different ways (polymorphism), making the code more flexible. POP requires writing separate functions for each case.

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

**Steps to Set Up a C++ Development Environment**

1. **Install a C++ Compiler**
    - You need a compiler to convert C++ code into machine language.
    - For Windows, you can use **MinGW** or **MSYS2**.
    - For Linux, install using: `sudo apt install g++`.
    - For macOS, use: `brew install gcc`.
2. **Install an IDE or Code Editor**
    - IDEs (Integrated Development Environments) like **Code::Blocks**, **Dev-C++**, **Visual Studio**, or **Eclipse** provide tools for writing, compiling, and debugging code.
    - Lightweight editors like **VS Code** or **Sublime Text** can also be used with compiler integration.
3. **Write Your First Program**
    - Create a `.cpp` file and write a simple C++ program, for example:

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

```
}
```

4. **Compile the Program**
   - Open the terminal/command prompt, go to your file location, and run:

     ```
     g++ filename.cpp -o output
     ```

     This compiles the code into an executable file.

5. **Run the Program**
   - Execute the compiled program:
     - On Windows: `output.exe`
     - On Linux/macOS: `./output`
   - You should see the output, for example:

     ```
     Hello, World!
     ```

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

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

In C++, **input** means taking data from the user, and **output** means displaying data to the user. The standard library `<iostream>` provides two main objects for I/O:

---

1. **Output using `cout`**
   - Used to display data on the screen.
   - Syntax:

     ```
     cout << data;
     ```

   - Example:

     ```
     #include <iostream>
     using namespace std;
     int main() {
       cout << "Hello, World!" << endl;
       cout << "The number is: " << 10;
       return 0;
     }
     ```

   - **Explanation**:
     - `<<` is the insertion operator, which sends data to the output stream (`cout`).
     - `endl` moves to the next line.

2. **Input using cin**
    - Used to take input from the user.
    - Syntax:

```
cin >> variable;
```

    - Example:

```
#include <iostream>
using namespace std;
int main() {
   int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "You are " << age << " years old.";
   return 0;
}
```

    - **Explanation**:
        - >> is the extraction operator, which takes data from the input stream (cin) and stores it in a variable.

---

3. **Multiple Inputs and Outputs**
    - You can take or display multiple values in one statement.
    - Example:

```
#include <iostream>
using namespace std;
int main() {
   int a, b;
   cout << "Enter two numbers: ";
   cin >> a >> b;
   cout << "Sum is: " << a + b;
   return 0;
}
```

## *2.* Variables, Data Types, and Operators

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

These are the most commonly used data types.

| Data Type | Description | Example |
|---|---|---|
| int | Stores integers (whole numbers) | int age = 25; |
| float | Stores decimal numbers (single precision) | float price = 99.50; |
| double | Stores decimal numbers (double precision) | double pi = 3.14159; |
| char | Stores a single character | char grade = 'A'; |
| bool | Stores true or false | bool isPass = true; |

---

## 2. Derived Data Types

These are based on basic types.

- **Arrays** – Collection of elements of the same type.

```
int marks[3] = {85, 90, 78};
```

- **Pointers** – Stores the memory address of another variable.

```
int x = 5;
int *ptr = &x;
```

- **Functions** – Block of code that performs a task.

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

---

## 3. User-defined Data Types

Created by the programmer to represent complex data.

- **Structures**

```
struct Student {
   string name;
   int age;
};
```

- **Classes** (OOP concept)

```
class Car {
   public:
   string model;
};
```

- **Enumerations**

```
enum Color { Red, Green, Blue };
```

---

## 4. Void Type

Represents "no value". Often used for functions that do not return anything.

```
void greet() {
   cout << "Hello!";
}
```

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

**Difference between Implicit and Explicit Type Conversion in C++**

In C++, **type conversion** means changing one data type into another. There are two main types:

---

## 1. Implicit Type Conversion (Type Casting done automatically)

- Also called **Type Promotion** or **Type Casting by Compiler**.
- The compiler automatically converts a smaller data type to a larger data type to avoid data loss.
- Done without the programmer writing any special code.
- **Example**:

```
#include <iostream>
using namespace std;
int main() {
   int num = 5;
   float result = num;  // int is automatically converted to float
   cout << result;     // Output: 5
   return 0;
}
```

- Here, int → float happens automatically.

---

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

- Also called **Type Casting by Programmer**.

- The programmer uses casting operators to convert data.
- Syntax:

```
(new_data_type) value
or
static_cast<new_data_type>(value)
```

- **Example**:

```
#include <iostream>
using namespace std;
int main() {
   double pi = 3.14159;
   int intPi = (int)pi;  // Old-style casting
   int intPi2 = static_cast<int>(pi); // Modern C++ casting
   cout << intPi << " " << intPi2;   // Output: 3 3
   return 0;
}
```

- Here, double → int happens because we explicitly told the compiler to do it.

---

**Main Difference Table**

| Feature | Implicit Conversion | Explicit Conversion |
|---|---|---|
| Who performs it? | Compiler | Programmer |
| Syntax | No special syntax needed | Casting operators required |
| Safety | Can cause unexpected changes | More control over conversion |
| Example | `float x = 5;` | `int y = (int)3.5;` |

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

**Different Types of Operators in C++**

In C++, **operators** are special symbols used to perform operations on variables and values.

---

### 1. Arithmetic Operators

Used for basic mathematical operations.

```
int a = 10, b = 3;
cout << a + b; // Addition → 13
cout << a - b; // Subtraction → 7
cout << a * b; // Multiplication → 30
cout << a / b; // Division → 3
cout << a % b; // Modulus → 1
```

---

## 2. Relational Operators

Used to compare two values and return true or false.

```
int x = 5, y = 8;
cout << (x == y); // Equal to → false
cout << (x != y); // Not equal → true
cout << (x > y);  // Greater than → false
cout << (x < y);  // Less than → true
cout << (x >= y); // Greater or equal → false
cout << (x <= y); // Less or equal → true
```

---

## 3. Logical Operators

Used to combine or reverse logical conditions.

```
bool a = true, b = false;
cout << (a && b); // Logical AND → false
cout << (a || b); // Logical OR  → true
cout << (!a);     // Logical NOT → false
```

---

## 4. Assignment Operators

Used to assign values to variables.

```
int num = 10;
num += 5; // num = num + 5 → 15
num -= 3; // num = num - 3 → 12
num *= 2; // num = num * 2 → 24
num /= 4; // num = num / 4 → 6
num %= 5; // num = num % 5 → 1
```

---

## 5. Bitwise Operators

Used to perform operations on bits (binary level).

```
int a = 5, b = 3;
cout << (a & b);  // AND → 1
```

```
cout << (a | b);  // OR  → 7
cout << (a ^ b);  // XOR → 6
cout << (~a);     // NOT → -6
cout << (a << 1); // Left shift → 10
cout << (a >> 1); // Right shift → 2
```

---

## 6. Miscellaneous Operators

- **sizeof()** – Returns size of a data type or variable.

  ```
  cout << sizeof(int); // Usually 4
  ```

- **? : (Ternary Operator)** – Short form of if-else.

  ```
  int age = 18;
  string result = (age >= 18) ? "Adult" : "Minor";
  cout << result; // Adult
  ```

- **&** – Address-of operator.

  ```
  int x = 10;
  cout << &x; // Prints memory address
  ```

- ***** – Pointer dereference.

  ```
  int y = 10;
  int *p = &y;
  cout << *p; // 10
  ```

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

**Constants and Literals in C++**

In C++, **constants** and **literals** are used to store fixed values that do not change during program execution. They help make code more readable, secure, and maintainable.

---

## 1. Constants

A **constant** is a variable whose value cannot be changed after it is assigned.

**Purpose:**

- To protect important values from accidental changes.

- To make programs easier to understand (e.g., using PI instead of 3.14159).

**Ways to Declare Constants:**

1. **Using const keyword**

   const int MAX = 100;

2. **Using #define preprocessor directive**

   #define PI 3.14

**Example:**

```
#include <iostream>
using namespace std;
int main() {
   const int MAX_USERS = 50;
   cout << "Max users allowed: " << MAX_USERS;
   return 0;
}
```

---

## 2. Literals

A **literal** is a fixed value written directly in the code.
They represent data exactly as it is.

**Types of Literals:**

- **Integer Literal**

  int age = 20; // 20 is an integer literal

- **Floating-point Literal**

  float price = 99.99; // 99.99 is a floating-point literal

- **Character Literal**

  char grade = 'A'; // 'A' is a character literal

- **String Literal**

  string name = "John"; // "John" is a string literal

- **Boolean Literal**

```
bool isPass = true; // true is a boolean literal
```

---

**Difference between Constant and Literal:**

| Constant | Literal |
|---|---|
| Named value stored in a variable that cannot change. | Actual fixed value written in the code. |
| Example: `const int MAX = 10;` | Example: `10` in the above code. |

## *3.*Control Flow Statements

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

**Conditional Statements in C++**

Conditional statements are used to **make decisions** in a program.
They allow the program to execute certain code **only if** a specific condition is true.

---

### 1. if-else Statement

- Used when you want to choose between two possible actions based on a condition.
- **Syntax:**

```
if (condition) {
   // code runs if condition is true
} else {
   // code runs if condition is false
}
```

- **Example:**

```
#include <iostream>
using namespace std;
int main() {
   int age;
   cout << "Enter your age: ";
   cin >> age;

   if (age >= 18) {
      cout << "You are eligible to vote.";
   } else {
      cout << "You are not eligible to vote.";
```

```
    }
    return 0;
}
```

- **How it works:**
    - If the condition is **true**, the first block runs.
    - If the condition is **false**, the else block runs.

---

## 2. switch Statement

- Used when you have **multiple possible cases** for a single variable or expression.
- Often easier to read than using many if-else statements.
- **Syntax:**

```
switch (expression) {
    case value1:
        // code for case 1
        break;
    case value2:
        // code for case 2
        break;
    default:
        // code if no case matches
}
```

- **Example:**

```
#include <iostream>
using namespace std;
int main() {
    int day;
    cout << "Enter day number (1-3): ";
    cin >> day;

    switch (day) {
        case 1:
            cout << "Monday";
            break;
        case 2:
            cout << "Tuesday";
            break;
        case 3:
            cout << "Wednesday";
            break;
        default:
            cout << "Invalid day";
    }
    return 0;
}
```

- **How it works:**
  - The switch checks the expression's value.
  - Matches it with a case and runs that code.
  - break stops the execution from falling into the next case.
  - default runs if no cases match.

## 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 used to **repeat** a block of code multiple times until a condition becomes false. The main difference between them is **when and how the condition is checked**.

---

### 1. for Loop

- **When used:** When you know **exactly how many times** you want to repeat the code.
- **Condition check:** Before the loop starts each time (entry-controlled).
- **Syntax:**

```
for (initialization; condition; update) {
   // code to repeat
}
```

- **Example:**

```
for (int i = 1; i <= 5; i++) {
   cout << i << " ";
}
// Output: 1 2 3 4 5
```

---

### 2. while Loop

- **When used:** When the number of repetitions is **not known in advance**; runs while the condition is true.
- **Condition check:** Before each iteration (entry-controlled).
- **Syntax:**

```
while (condition) {
   // code to repeat
}
```

- **Example:**

```
int i = 1;
while (i <= 5) {
   cout << i << " ";
   i++;
}
// Output: 1 2 3 4 5
```

---

### 3. do-while Loop

- **When used:** When you **must run the code at least once**, even if the condition is false.
- **Condition check:** After each iteration (exit-controlled).
- **Syntax:**

```
do {
   // code to repeat
} while (condition);
```

- **Example:**

```
int i = 1;
do {
   cout << i << " ";
   i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

---

### Main Differences Table

| Feature | for Loop | while Loop | do-while Loop |
|---|---|---|---|
| Condition Check | Before first run | Before first run | After first run |
| Guaranteed Execution | No | No | Yes (at least once) |
| Use Case | Fixed repetitions | Unknown repetitions | Must run at least once |

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

**Break and Continue Statements in Loops (C++)**

In C++, **break** and **continue** are control statements used to change the normal flow of a loop.

## 1. break Statement

- **Purpose:** Immediately **exits** the loop, even if the loop condition is still true.
- **Use Case:** When you want to stop looping once a certain condition is met.

**Example:**

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         break; // loop ends when i is 5
      }
      cout << i << " ";
   }
   return 0;
}
// Output: 1 2 3 4
```

---

## 2. continue Statement

- **Purpose:** Skips the current loop iteration and goes to the **next** one.
- **Use Case:** When you want to ignore some values but still continue the loop.

**Example:**

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3) {
         continue; // skips when i is 3
      }
      cout << i << " ";
   }
   return 0;
}
// Output: 1 2 4 5
```

---

## Key Difference

| Statement | Effect |
|---|---|
| **break** | Ends the loop completely. |

| Statement | Effect |
|---|---|
| **continue** | Skips only the current iteration and continues with the next one. |

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

**Nested Control Structures in C++**

A **nested control structure** means placing one control structure **inside another**. Control structures include loops (for, while, do-while) and decision-making statements (if, switch).

---

## How it works:

- The **outer** control structure starts first.
- Inside it, another control structure runs completely before the outer one continues.
- You can nest **if inside if**, **loop inside loop**, or even **loops inside if statements**.

---

## Example 1: Nested if-else

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;

   if (num > 0) {
      if (num % 2 == 0) {
         cout << "Positive Even Number";
      } else {
         cout << "Positive Odd Number";
      }
   } else {
      cout << "Number is zero or negative";
   }
   return 0;
}
```

**Explanation:**

- The outer if checks if the number is positive.
- The inner if checks if it is even or odd.

## Example 2: Nested Loops (Multiplication Table)

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 3; i++) {          // Outer loop
      for (int j = 1; j <= 3; j++) {       // Inner loop
         cout << i << " x " << j << " = " << i * j << endl;
      }
      cout << "---" << endl; // separator after each row
   }
   return 0;
}
```

**Explanation:**

- Outer loop controls rows.
- Inner loop controls columns.
- For every single outer loop iteration, the inner loop runs completely.

## 4.Functions and Scope

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

**Function in C++**

A **function** in C++ is a block of code that performs a specific task.
Instead of writing the same code multiple times, we can write it once inside a function and call it whenever needed.

---

### Advantages of Functions

- Avoids code repetition.
- Makes programs easier to read and maintain.
- Allows dividing a program into smaller, manageable parts.

---

## Parts of a Function

### 1. Function Declaration (Prototype)

- Tells the compiler about the function name, return type, and parameters **before** it is used.
- Syntax:

```
return_type function_name(parameter_list);
```

- Example:

```
int add(int a, int b);
```

---

### 2. Function Definition

- Contains the actual code (body) of the function.
- Syntax:

```
return_type function_name(parameter_list) {
   // function body
}
```

- Example:

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

---

### 3. Function Calling

- Used to execute the function's code.
- Syntax:

```
function_name(arguments);
```

- Example:

```
cpp
Copy code
int sum = add(5, 3); // Calls the add() function
```

---

## Complete Example

```
#include <iostream>
```

```
using namespace std;

// Function Declaration
int add(int a, int b);

// Main Function
int main() {
   int result = add(5, 3); // Function Call
   cout << "Sum: " << result;
   return 0;
}

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

**Output:**

```
Sum: 8
```

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

**Scope of Variables in C++**

The **scope** of a variable means the part of the program where the variable can be **accessed or used**.
In C++, variables can have **local scope** or **global scope** depending on where they are declared.

---

### 1. Local Scope

- A variable declared **inside** a function, block, or loop.
- Can **only** be accessed within that specific function or block.
- Created when the block starts, and destroyed when the block ends.

**Example:**

```
#include <iostream>
using namespace std;
int main() {
   int x = 10; // Local variable
   cout << x;  // Works here
   return 0;
}
// cout << x; // Error: x is not accessible here
```

## 2. Global Scope

- A variable declared **outside** all functions.
- Can be accessed **from any function** in the program.
- Created when the program starts, and destroyed when it ends.

**Example:**

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

int x = 50; // Global variable

void display() {
   cout << x; // Can access global variable
}

int main() {
   cout << x << endl;
   display();
   return 0;
}
```

---

## Difference Between Local and Global Variables

| Feature | Local Variable | Global Variable |
|---|---|---|
| **Declared in** | Inside a function or block | Outside all functions |
| **Accessible in** | Only inside the function/block declared | Any function in the program |
| **Lifetime** | Exists only while the function/block runs | Exists for the entire program |
| **Default Value** | Garbage value (if uninitialized) | Zero (if uninitialized) |

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

**Recursion in C++**

**Definition:**
Recursion is a process where a function **calls itself** directly or indirectly to solve a problem.
It is used for problems that can be broken into smaller, similar subproblems.

---

## How It Works

A recursive function has two main parts:

1. **Base Case** – Condition where the recursion stops (prevents infinite calls).
2. **Recursive Case** – The part where the function calls itself with a smaller/simpler problem.

---

## Example: Factorial Calculation

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

// Recursive function
int factorial(int n) {
   if (n == 0 || n == 1) { // Base case
      return 1;
   } else { // Recursive case
      return n * factorial(n - 1);
   }
}

int main() {
   int num = 5;
   cout << "Factorial of " << num << " is " << factorial(num);
   return 0;
}
```

**Output:**

```
Factorial of 5 is 120
```

**Explanation:**

- factorial(5) → 5 * factorial(4)
- factorial(4) → 4 * factorial(3)
- … continues until factorial(1) which returns 1 (base case).

---

## Advantages of Recursion

- Makes code shorter and easier to read for certain problems (like factorial, Fibonacci, tree traversal).

## Disadvantages of Recursion

- Uses more memory due to multiple function calls.
- Can be slower than loops for large inputs.

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

**Function Prototypes in C++**

**Definition:**
A **function prototype** is a declaration of a function that tells the compiler:

- The **function name**
- The **return type**
- The **type and number of parameters**

It does **not** contain the function's actual body (code).

---

## Syntax:

```
return_type function_name(parameter_list);
```

Example:

```
int add(int a, int b);
```

---

## Purpose of Function Prototypes

1. **Inform the Compiler Before Use**
   - If a function is called **before** it is defined, the compiler needs its prototype to understand its details.
2. **Type Checking**
   - Ensures that the correct number and types of arguments are passed when calling the function.
3. **Organizing Code**
   - Allows writing function definitions **after** main() without errors.

---

## Example:

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

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

int main() {
   int sum = add(5, 3); // Function call
   cout << "Sum: " << sum;
   return 0;
}

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

**Output:**

```
Sum: 8
```

## 5.Arrays and Strings

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

An **array** is a collection of elements of the **same data type** stored in **continuous memory locations**.
Each element can be accessed using an **index**, starting from 0.

---

### Why Use Arrays?

- Store multiple values in a single variable name.
- Access and modify data easily using indexes.
- Useful for working with lists, tables, and matrices.

---

### 1. Single-Dimensional Array

- Stores data in **one row** (linear form).
- Accessed using **one index**.

**Syntax:**

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

**Example:**

```
#include <iostream>
using namespace std;
int main() {
   int marks[5] = {90, 85, 88, 92, 80};
   cout << "First mark: " << marks[0]; // Output: 90
   return 0;
}
```

---

## 2. Multi-Dimensional Array

- Stores data in **rows and columns** (like a table).
- Accessed using **two or more indexes**.

**Syntax:**

```
data_type array_name[rows][columns];
```

**Example (2D array):**

```
#include <iostream>
using namespace std;
int main() {
   int matrix[2][3] = {
      {1, 2, 3},
      {4, 5, 6}
   };
   cout << "Element at [1][2]: " << matrix[1][2]; // Output: 6
   return 0;
}
```

---

## Difference Table

| Feature | Single-Dimensional Array | Multi-Dimensional Array |
|---|---|---|
| **Structure** | Linear (one row) | Table-like (rows & columns) |
| **Indexing** | One index | Two or more indexes |
| **Example** | marks[3] | matrix[2][1] |
| **Use Case** | Lists, scores, names | Matrices, tables, gr |

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

In C++, **strings** are used to store and work with sequences of characters (text).
We can handle strings in **two ways**:

1. Using **C-style strings** (character arrays)
2. Using the **string class** from the C++ Standard Library

---

## 1. C-Style Strings (Character Arrays)

- Stored as an **array of characters** ending with a **null character** '\0'.
- We use functions from <cstring> to work with them.

**Example:**

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

int main() {
   char name[20] = "John"; // C-style string
   cout << "Length: " << strlen(name) << endl; // Finds length
   strcat(name, " Doe"); // Concatenate
   cout << "Full Name: " << name;
   return 0;
}
```

**Output:**

```
Length: 4
Full Name: John Doe
```

---

## 2. Strings Using string Class

- Easier and safer than C-style strings.
- We use the <string> header.

**Example:**

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

int main() {
   string first = "Hello";
   string second = "World";

   string full = first + " " + second; // Concatenate
   cout << "Full String: " << full << endl;
   cout << "Length: " << full.length(); // String length
   return 0;
}
```

**Output:**

Full String: Hello World
Length: 11

---

## Difference Table

| Feature | C-Style Strings | string Class |
|---|---|---|
| **Storage** | Character array | String object |
| **Functions** | From <cstring> | Built-in member functions |
| **Ease of Use** | More manual handling | Easier and safer |
| **Example** | char name[10] | string name |

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

In C++, **initialization** means assigning values to an array at the time of declaration. Arrays can be **1D (single-dimensional)** or **2D (multi-dimensional)**.

---

### 1. Initializing a 1D Array

**Syntax:**

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

**Examples:**

1. **Full Initialization**

```
int marks[5] = {90, 85, 88, 92, 80};
```

2. **Partial Initialization** (remaining values become 0)

```
int marks[5] = {90, 85}; // → {90, 85, 0, 0, 0}
```

3. **Omitting Size** (size is determined automatically)

```
int marks[] = {90, 85, 88, 92, 80}; // Size is 5
```

---

## 2. Initializing a 2D Array

**Syntax:**

```
data_type array_name[rows][columns] = {
   {row1_values},
   {row2_values},
   ...
};
```

**Examples:**

1. **Full Initialization**

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

2. **Partial Initialization** (missing values become 0)

```
int matrix[2][3] = {
   {1, 2},    // → {1, 2, 0}
   {4}        // → {4, 0, 0}
};
```

3. **Omitting Inner Braces** (row-wise filling)

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

---

## Key Points

- If you **don't initialize** an array, it contains **garbage values** (for local variables).
- Global/static arrays are automatically initialized to 0.
- The number of initializers **cannot exceed** the array size.

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

**String Operations and Functions in C++**

In C++, we can work with strings in **two main ways**:

1. **C-style strings** (character arrays)
2. **string class** from the Standard Library

## 1. String Operations with C-Style Strings

- Require the <cstring> header.
- Common functions:

| Function | Purpose | Example |
|---|---|---|
| strlen(str) | Finds length of string | strlen("Hello") → 5 |
| strcpy(dest, src) | Copies one string to another | strcpy(a, b) |
| strcat(str1, str2) | Concatenates two strings | "Hello" + "World" |
| strcmp(str1, str2) | Compares two strings | Returns 0 if equal |

**Example:**

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
   char str1[20] = "Hello";
   char str2[] = "World";

   strcat(str1, str2); // Concatenate
   cout << "Concatenated: " << str1 << endl;
   cout << "Length: " << strlen(str1);
   return 0;
}
```

---

## 2. String Operations with string Class

- Require the <string> header.
- Easier and safer than C-style strings.

**Common Functions:**

| Function | Purpose | Example |
|---|---|---|
| .length() or .size() | Finds length | name.length() |
| .append(str) | Adds another string | s1.append(s2) |

| Function | Purpose | Example |
|---|---|---|
| .insert(pos, str) | Inserts at position | s.insert(3, "Hi") |
| .erase(pos, len) | Removes part of string | s.erase(2, 3) |
| .substr(pos, len) | Extracts substring | s.substr(0, 5) |
| .find(str) | Finds position of substring | s.find("Hi") |
| .replace(pos, len, str) | Replaces part of string | s.replace(0, 5, "Hey") |

**Example:**

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string str = "Hello World";

   cout << "Length: " << str.length() << endl;
   str.append("!!!");
   cout << "After append: " << str << endl;
   cout << "Substring: " << str.substr(0, 5) << endl;
   str.replace(0, 5, "Hi");
   cout << "After replace: " << str;
   return 0;
}
```

---

## Key Differences

- **C-style strings**: Faster but require manual memory handling.
- **string class**: Safer, more convenient, and comes with many built-in functions.

## 6.introduction to Object-Oriented Programming

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

### Key Concepts of Object-Oriented Programming (OOP)

OOP is a programming style that organizes code into **objects**, which are based on **classes**. It helps make programs **modular, reusable, and easier to maintain**.

The main concepts of OOP are:

## 1. Class

- A **blueprint** or template for creating objects.
- Defines **data members** (variables) and **member functions** (methods).
- Example:

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

---

## 2. Object

- An **instance** of a class.
- Has its own copy of data members and can use the class's functions.
- Example:

```
Car myCar; // Object creation
myCar.brand = "Toyota";
myCar.start();
```

---

## 3. Encapsulation

- **Wrapping** data and functions into a single unit (class).
- Access to data is controlled using **access specifiers** (private, public, protected).
- Helps protect data from unauthorized access.

---

## 4. Abstraction

- Showing only **essential details** and hiding the complex background.
- Example: You use start() to start a car, but you don't see the engine's internal code.

---

## 5. Inheritance

- One class can **inherit** properties and methods from another class.
- Promotes **code reuse**.

- Example:

```
class Vehicle { public: void move() { cout << "Moving"; } };
class Car : public Vehicle {};
```

---

### 6. Polymorphism

- **One name, many forms**.
- A function or method can behave differently based on context.
- Types:
  - **Compile-time (Function Overloading)**
  - **Run-time (Function Overriding using virtual functions)**

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

---

### 1. Class

- A **class** is a blueprint or template for creating objects.
- It defines **data members** (variables) and **member functions** (methods) that describe an object's properties and behavior.
- **Syntax:**

```
class ClassName {
   // Access specifier
   public:
      // data members
      // member functions
};
```

---

### 2. Object

- An **object** is an **instance** of a class.
- Each object has its own copy of the data members but shares the class's functions.

---

### Example:

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

// Class definition
class Car {
```

```
public:
   string brand;
   int year;

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

int main() {
   Car car1;  // Creating first object
   car1.brand = "Toyota";
   car1.year = 2020;

   Car car2;  // Creating second object
   car2.brand = "Honda";
   car2.year = 2018;

   car1.displayInfo();
   car2.displayInfo();

   return 0;
}
```

**Output:**

```
Brand: Toyota, Year: 2020
Brand: Honda, Year: 2018
```

---

**Key Points:**

- A **class** is just a definition — it does not take memory until an **object** is created.
- You can create multiple objects from the same class, each with its own data.

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

**Definition:**
Inheritance is an **OOP feature** in C++ that allows one class (**child/derived class**) to use the **properties and methods** of another class (**parent/base class**).
It helps in **code reuse** and makes programs easier to maintain.

---

## Types of Inheritance in C++

1. **Single Inheritance** – One base class → One derived class
2. **Multiple Inheritance** – Multiple base classes → One derived class
3. **Multilevel Inheritance** – A class derived from another derived class
4. **Hierarchical Inheritance** – One base class → Multiple derived classes
5. **Hybrid Inheritance** – Combination of two or more types

---

## Example: Single Inheritance

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

// Base class
class Vehicle {
public:
   void move() {
      cout << "This vehicle moves on the road" << endl;
   }
};

// Derived class
class Car : public Vehicle {
public:
   void honk() {
      cout << "Car horn: Beep Beep!" << endl;
   }
};

int main() {
   Car myCar;
   myCar.move(); // Inherited from Vehicle
   myCar.honk(); // Defined in Car
   return 0;
}
```

### Output:

```
This vehicle moves on the road
Car horn: Beep Beep!
```

---

### Key Points:

- **public inheritance** keeps base class's public members public in the derived class.
- Inheritance promotes **code reuse** — we didn't rewrite `move()` for `Car`.
- Access specifiers (`public`, `protected`, `private`) control how members are inherited.

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

**Encapsulation in C++**

Encapsulation is one of the fundamental concepts of Object-Oriented Programming (OOP). It refers to the bundling of data (variables) and methods (functions) that operate on the data into a single unit called a **class**. Encapsulation also restricts direct access to some of an object's components, which is a way of preventing accidental or unauthorized modifications of data.

### Key points of Encapsulation:

- It hides the internal state of an object from the outside world.
- Only exposes a controlled interface (public methods) to interact with the object's data.
- Protects the integrity of the data by preventing external code from directly accessing or modifying it.
- Helps in modularity, maintainability, and security of code.

---

### How is Encapsulation achieved in C++ classes?

Encapsulation is achieved in C++ using **access specifiers** within a class:

1. **private**: Members declared as private can only be accessed within the class itself. They cannot be accessed directly from outside the class.
2. **public**: Members declared as public can be accessed from outside the class.
3. **protected**: Members are accessible within the class and by derived (inherited) classes.

By declaring data members as **private** and providing **public** getter and setter functions (or other public methods), we control how the data is accessed and modified.

---

### Example of Encapsulation in C++

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

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

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

    // Public method to deposit money
    void deposit(double amount) {
        if (amount > 0)
            balance += amount;
    }

    // Public method to withdraw money
    void withdraw(double amount) {
        if (amount > 0 && amount <= balance)
            balance -= amount;
        else
            cout << "Insufficient balance or invalid amount." << endl;
    }

    // Public method to get the current balance
    double getBalance() {
        return balance;
    }
};

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

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

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

    // Direct access to balance is not allowed:
    // myAccount.balance = 5000;  // Error!

    return 0;
}
```

---

## Summary:

- **Encapsulation** groups data and functions into a class.
- Data members are made **private** to restrict direct access.
- Public methods (getters/setters) provide controlled access to private data.
- This protects the object's data integrity and hides implementation details.