Module - 3 Introduction to OOPS Programming

1. Introduction to C++

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

1. Approach

- **Procedural Programming:** Follows a **top-down approach** where the program is divided into procedures (functions).
- **OOP:** Follows a **bottom-up approach** where the program is built around objects.

---

2. Main Focus

- **Procedural Programming:** Focuses on **functions** (what actions to perform).
- **OOP:** Focuses on **objects** (data and methods together).

---

3. Data Handling

- **Procedural Programming:** Data is **separate** from functions. Functions operate on data.
- **OOP:** Data and functions are **bundled together** inside objects (encapsulation).

---

4. Security

- **Procedural Programming:** Data is less secure, as it can be accessed globally.
- **OOP:** Data is more secure due to **encapsulation** and **access modifiers** (private, protected, public).

---

5. Reusability

- **Procedural Programming:** Code reusability is low; mainly achieved through functions.
- **OOP:** Code reusability is high through **inheritance** and **polymorphism**.

---

6. Examples

- **Procedural Programming Languages:** C, Pascal, FORTRAN.
- **OOP Languages:** C++, Java, Python, C#.

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

1. Encapsulation (Data Security)

- **OOP:** Data and functions are bundled together inside objects, and access can be restricted using access modifiers (private, protected, public).
- **POP:** Data is global and can be accessed by any function, making it less secure.

Advantage: OOP provides better **data protection and security**.

---

2. Code Reusability

- **OOP:** Supports **inheritance**, allowing classes to reuse methods and attributes of other classes.
- **POP:** Functions can be reused, but not entire structures with data.

Advantage: OOP reduces code duplication and increases efficiency.

---

3. Maintainability

- **OOP:** Code is modular (divided into objects/classes). If a change is required, it can be made in one place without affecting the whole program.
- **POP:** Code is less modular, so modifications are harder.

Advantage: OOP makes programs **easier to update and maintain**.

---

4. Polymorphism (Flexibility)

- **OOP:** Allows the same function or operator to behave differently (method overloading, overriding).
- **POP:** Does not support polymorphism directly.

Advantage: OOP gives **flexible and dynamic behavior** to programs.

---

5. Abstraction

- **OOP:** Complex details can be hidden; only essential features are shown to the user.
- **POP:** Requires step-by-step functions; hiding implementation is harder.

Advantage: OOP reduces **complexity** for the user.

---

6. Real-world Modeling

- **OOP:** Objects represent real-world entities (e.g., Student, Car, BankAccount).
- **POP:** Works on procedures, making it harder to relate to real-life problems.

Advantage: OOP is **closer to real-world problem solving**

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

Steps Involved in Setting Up a C++ Development Environment

To write, compile, and run C++ programs, you need to set up a development environment. The main steps are:

---

1. Install a C++ Compiler

- A compiler is required to convert C++ code into machine code.
- Popular compilers:
    - **GCC/G++** (Linux, Windows via MinGW)
    - **Clang** (Mac, Linux)
    - **MSVC** (Microsoft Visual C++ for Windows)
- On Windows, **MinGW** or **TDM-GCC** is commonly used.
- On Linux/Mac, GCC is usually pre-installed.

---

2. Install an IDE or Text Editor

- An **IDE (Integrated Development Environment)** helps in writing, compiling, and debugging code easily.
- Popular choices:
    - **Code::Blocks**
    - **Dev C++**
    - **Visual Studio**
    - **Eclipse CDT**
    - **CLion**
- Or, you can use simple editors like **Notepad++**, **VS Code**, **Sublime Text**, etc.

---

3. Configure the Compiler in the IDE

- Link the installed compiler (like GCC) with your IDE.
- Example: In Code::Blocks, set the compiler path to **g++.exe**.
- This ensures that when you build a program, the IDE knows which compiler to use.

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

Main Input/Output Operations in C++

In C++, input/output (I/O) operations are mainly handled using the **iostream** library.
The two most common operations are:

---

1. Output Operation (Displaying Data)

- Done using **cout** (character output).
- Syntax:
- cout << data;
- Example:
- #include <iostream>
- using namespace std;
-
- int main() {
-     cout << "Hello, World!" << endl;   // Output text
-     int age = 20;
-     cout << "Age: " << age << endl;   // Output variable
-     return 0;
- }

---

2. Input Operation (Taking Data from User)

- Done using **cin** (character input).
- Syntax:
- cin >> variable;

- Example:
- #include <iostream>
- using namespace std;
-
- int main() {
-     int num;
-     cout << "Enter a number: ";
-     cin >> num;   // Take input from user
-     cout << "You entered: " << num;
-     return 0;

}

2. Variables, Data Types, and Operators

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

Different Data Types in C++

C++ provides several **data types** to store different kinds of values. They are mainly divided into **basic (primitive), derived, and user-defined** types.

---

1. Basic (Primitive) Data Types

These are the fundamental types used to declare variables.

- **int** → Stores integers (whole numbers).
- int age = 21;
- **float** → Stores single-precision decimal values.
- float price = 99.50;
- **double** → Stores double-precision decimal values.

- double pi = 3.14159;
- **char** → Stores a single character (enclosed in single quotes).
- char grade = 'A';
- **bool** → Stores true or false.
- bool isPassed = true;
- **void** → Represents no value (used in functions that don't return anything).
- void display() {
-     cout << "No return type!";
- }

---

2. Derived Data Types

These are built from basic data types.

- **Array** → Collection of elements of same type.
- int marks[5] = {90, 85, 76, 88, 95};
- **Pointer** → Stores memory address of another variable.
- int x = 10;
- int* ptr = &x;
- **Function** → Group of statements that performs a task.
- int add(int a, int b) { return a + b; }

---

3. User-Defined Data Types

Created by the programmer.

- **Structure (struct)**
- struct Student {

- int id;
- string name;
- };
- **Class**
- class Car {
- public:
- string brand;
- int year;
- };
- **Enumeration (enum)**

enum Color { Red, Green, Blue };

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 types**:

---

1. Implicit Type Conversion (Type Casting / Type Promotion)

- Also called **Type Casting by Compiler**.
- Happens **automatically** when a smaller data type is converted into a larger data type.
- No data loss usually occurs (e.g., int → float, char → int).
- **Example:**
- #include <iostream>
- using namespace std;
- int main() {

-    int x = 10;
-    double y = x;   // int (10) is automatically converted to double (10.0)
-    cout << y;   // Output: 10
-    return 0;
- }

---

2. Explicit Type Conversion (Type Casting by Programmer)

- Also called **Type Casting by User**.
- Done **manually** using **cast operators**.
- Syntax:
- (new_type) expression
- **Example:**
- #include <iostream>
- using namespace std;
-
- int main() {
-    double pi = 3.14159;
-    int num = (int) pi;   // Explicitly convert double → int
-    cout << num;   // Output: 3
-    return 0;

}

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

Different Types of Operators in C++

Operators are special symbols used to perform operations on variables and values.
C++ provides several categories of operators:

---

1. Arithmetic Operators

Used for mathematical calculations.

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

Example:

```
int a = 10, b = 3;

cout << a + b; // 13

cout << a % b; // 1
```

---

2. Relational Operators

Used to compare values (returns true or false).

Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b

Operator	Meaning	Example
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

Example:

```
int x = 5, y = 10;

cout << (x < y); // 1 (true)
```

---

3. Logical Operators

Used for logical decisions.

Operator	Meaning	Example
&&	Logical AND	(x > 0 && y > 0)
`		`
!	Logical NOT	!(x > y)

Example:

```
int x = 5, y = -3;

cout << (x > 0 && y > 0); // 0 (false)
```

---

4. Assignment Operators

Used to assign values to variables.

Operator	Meaning	Example
=	Assign	a = 10
+=	Add and assign	a += 5
-=	Subtract and assign	a -= 2

Operator	Meaning	Example
*=	Multiply and assign	a *= 3
/=	Divide and assign	a /= 2

---

5. Increment/Decrement Operators

Increase or decrease the value of a variable by 1.

Operator	Meaning	Example
++	Increment	++a or a++
--	Decrement	--a or a--

Example:

```
int a = 5;
cout << ++a; // 6 (pre-increment)
```

---

6. Bitwise Operators

Used to perform operations on bits.

Operator	Meaning	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

---

7. Conditional (Ternary) Operator

A shorthand for if-else.

int age = 20;

string result = (age >= 18) ? "Adult" : "Minor";

cout << result; // Adult

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

1. Constants

- **Definition:** A constant is a variable whose value **cannot be changed** after initialization.
- They are used to store fixed values that remain the same throughout the program.

Types of Constants in C++:

1. **Integer Constant** → Whole numbers
2. const int maxStudents = 50;
3. **Floating-point Constant** → Decimal numbers
4. const float pi = 3.14;
5. **Character Constant** → Single characters in single quotes
6. const char grade = 'A';
7. **String Constant** → Sequence of characters in double quotes
8. const string course = "C++ Programming";

Purpose of Constants:

- Improves **readability** of code (meaningful names instead of raw values).
- Provides **data security** (accidental modification prevented).
- Makes programs **easy to maintain**.

---

2. Literals

- **Definition:** A literal is a **fixed value written directly in the code** (not stored in a variable).
- They represent **constant values** of different types.

Examples of Literals:

1. **Integer Literal**
2. int x = 100;   // 100 is an integer literal
3. **Floating-point Literal**
4. float y = 3.14; // 3.14 is a float literal
5. **Character Literal**
6. char ch = 'A'; // 'A' is a character literal
7. **String Literal**
8. cout << "Hello World"; // "Hello World" is a string literal
9. **Boolean Literal**

bool flag = true; // true is a boolean literal

3. Control Flow Statements

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

Conditional Statements in C++

Definition

Conditional statements are used to **make decisions in a program**.
They allow the program to **execute certain blocks of code only when specific conditions are true**.

C++ mainly uses:

1. **if-else statements**
2. **switch statements**

1. if-else Statement

- The **if** statement checks a condition.
- If the condition is **true**, the code inside the if block executes.
- If the condition is **false**, the else block executes (if provided).

Syntax:

```
if (condition) {
   // Code if condition is true
} else {
   // Code 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 an adult.";
   } else {
      cout << "You are a minor.";
   }
```

```
    return 0;
}
```

---

2. switch Statement

- The **switch** statement is used when we need to compare a single variable with multiple possible values.
- It is an alternative to using multiple if-else statements.

Syntax:

```
switch (expression) {
    case value1:
        // Code if expression == value1
        break;
    case value2:
        // Code if expression == value2
        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;
}
```

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 are used to **repeat a block of code** until a condition is satisfied.
C++ provides three main types of loops: **for, while, and do-while**.

---

1. for Loop

- Used when the **number of iterations is known**.

- Initialization, condition, and increment/decrement are written in one line.

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

- Used when the **number of iterations is not known** in advance.
- Condition is checked **before** entering the loop.

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

- Similar to while, but condition is checked **after** executing the loop body.
- The loop runs **at least once**, even if the condition is false.

Syntax:

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

Example:

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

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

1. break Statement

- The **break** statement is used to **exit a loop immediately**, even if the loop condition is still true.
- Control is transferred to the **first statement after the loop**.

Example (using break):

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

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

Output:

```
1 2 3 4
```

---

2. continue Statement

- The **continue** statement is used to **skip the current iteration** of the loop and move to the next iteration.
- The loop itself continues running.

Example (using continue):

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

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

Output:

```
1 2 4 5
```

4. Explain nested control structures with an example

Nested Control Structures in C++

Definition

- **Nested control structures** mean placing one control structure **inside another**.
- Control structures include decision-making (if, switch), loops (for, while, do-while), and others.
- They allow more **complex decision-making and looping** in programs.

Types of Nesting

1. **Nested if-else** → One if inside another.
2. **Loop inside loop** (nested loops).
3. **Mix of loops and decision-making**.

Example 1: Nested if-else

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

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

    if(age >= 18) {
        if(gender == 'M') {
            cout << "You are an adult male.";
        } else {
            cout << "You are an adult female.";
        }
    } else {
        cout << "You are a minor.";
    }
    return 0;
}
```

Output:

```
You are an adult male.
```

Example 2: Nested Loop

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

int main() {
    for(int i = 1; i <= 3; i++) {        // Outer loop
        for(int j = 1; j <= 2; j++) {    // Inner loop
```

```
            cout << "i=" << i << ", j=" << j << endl;
        }
    }
    return 0;
}
```

Output:

```
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
i=3, j=1
i=3, j=2
```

4. Functions and Scope

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

Definition

A **function** in C++ is a **block of code** that performs a specific task.

- It allows **code reusability** (write once, use many times).
- It makes programs **modular, readable, and easy to maintain**.

Parts of a Function

1. Function Declaration (Prototype)

- Tells the compiler about the **function name, return type, and parameters** before its actual definition.
- Written **before main()** or in header files.

Syntax:

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

Example:

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

---

2. Function Definition

- Contains the **actual body** of the function.
- Describes what the function will do.

Syntax:

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

Example:

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

---

3. Function Calling

- When you **use** a function in the program.
- The control jumps to the function, executes it, and then returns the result.

Syntax:

```
function_name(arguments);
```

Example:

```
#include <iostream>
```

```
using namespace std;


// Declaration
int add(int, int);


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


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

Output:

```
Sum = 8
```

In short:

- **Declaration** → Introduces the function to compiler.
- **Definition** → Contains actual code (logic).
- **Calling** → Executes the function.

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

Definition

The **scope of a variable** in C++ means the **region of the program** where the variable can be **accessed or used**.
It defines the **lifetime and visibility** of a variable.

Types of Scope

1. Local Scope

- A variable declared **inside a function or block** (like { }) has **local scope**.
- It is **created when the block is entered** and **destroyed when the block ends**.
- Can only be accessed **within that function/block**.

Example:

```
#include <iostream>

using namespace std;


int main() {

    int x = 10;  // Local variable to main()

    if(true) {

        int y = 20;  // Local variable to this block

        cout << x << " " << y; // Accessible here

    }

    // cout << y;  // ERROR: y is not accessible outside block

    return 0;

}
```

---

2. Global Scope

- A variable declared **outside all functions** has **global scope**.
- It is created when the program starts and destroyed when it ends.
- Can be accessed **from any function** in the program.

Example:

```
#include <iostream>
using namespace std;
int g = 100;  // Global variable
void show() {
   cout << "Global variable g = " << g << endl;
}
int main() {
   cout << "Accessing global g in main: " << g << endl;
   show(); // Accessible in other functions
   return 0;
}
```

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

Definition

Recursion in C++ is a process where a **function calls itself** directly or indirectly until a base condition is met.
It is useful for problems that can be broken down into **smaller sub-problems of the same type** (e.g., factorial, Fibonacci, searching, sorting).

Key Points about Recursion

1. Every recursive function must have a **base case** (stopping condition).
2. Without a base case, recursion leads to **infinite calls** and **stack overflow error**.

3. Recursion is an alternative to **loops** for repetitive tasks.

General Syntax

```
return_type function_name(parameters) {
    if (base_condition) {
        // stop recursion
        return value;
    } else {
        // recursive call
        return function_name(modified_parameters);
    }
}
```

Example 1: Factorial using Recursion

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

// Recursive function
int factorial(int n) {
    if(n == 0 || n == 1)  // Base case
        return 1;
    else
        return n * factorial(n - 1);  // Recursive call
}
int main() {
    int num = 5;
    cout << "Factorial of " << num << " = " << factorial(num);
    return 0;
}
```

Output:

```
Factorial of 5 = 120
```

Example 2: Fibonacci Series using Recursion

```
#include <iostream>
using namespace std;
int fibonacci(int n) {
    if(n == 0) return 0;   // Base case
    if(n == 1) return 1;   // Base case
    return fibonacci(n-1) + fibonacci(n-2);  // Recursive call
}
int main() {
    int terms = 6;
    cout << "Fibonacci series: ";
    for(int i = 0; i < terms; i++) {
        cout << fibonacci(i) << " ";
    }
    return 0;
}
```

Output:

```
Fibonacci series: 0 1 1 2 3 5
```

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

Definition

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

- The **function's name**
- The **return type**
- The **parameter types (and order)**

It does **not contain the body** of the function.
It ends with a **semicolon (;)**.

Syntax

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

Example:

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

Purpose / Why Function Prototypes Are Used

1. **Tells the compiler about the function before its use**
   - Functions are often defined **after main()**, but prototypes allow us to call them earlier.
2. **Helps in type checking**
   - Ensures that arguments passed match the declared parameter types.
3. **Improves program structure**
   - We can place all prototypes at the top, making the program more readable.

Example Program

```
#include <iostream>

using namespace std;

// Function prototype (declaration)

int add(int, int);

int main() {

   int result = add(10, 20);  // Function call

   cout << "Sum = " << result;

   return 0;

}

// Function definition

int add(int a, int b) {

   return a + b;

}
```

Output:

```
Sum = 30
```

5. Arrays and Strings

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

Definition

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

The index of an array starts from **0**.
Arrays allow storing multiple values in a single variable.

Syntax

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

Example (Single-Dimensional Array):

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

Types of Arrays

1. Single-Dimensional Array

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

Example:

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

int main() {
    int marks[5] = {90, 85, 88, 92, 75};

    cout << "Marks: ";
    for(int i = 0; i < 5; i++) {
        cout << marks[i] << " ";
    }
    return 0;
}
```

Output:

```
Marks: 90 85 88 92 75
```

---

2. Multi-Dimensional Array

- Stores data in **rows and columns (table-like structure)**.
- Accessed using **two or more indices**.
- Most common form: **2D array**.

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 << "Matrix:" << endl;
   for(int i = 0; i < 2; i++) {
      for(int j = 0; j < 3; j++) {
         cout << matrix[i][j] << " ";
      }
      cout << endl;
   }
   return 0;
}
```

Output:

```
Matrix:
1 2 3
4 5 6
```

2. Explain string handling in C++ with examples

String Handling in C++

What is a String?

A **string** in C++ is a **sequence of characters** used to represent text.
C++ supports strings in **two main ways**:

1. **C-style strings** → Character arrays (char str[])
2. **C++ string class (std::string)** → Part of the **Standard Template Library (STL)**

---

1. C-Style Strings (Character Arrays)

- Declared using char array.
- Must end with a **null character (\0)**.

Example:

```
#include <iostream>

#include <cstring>   // for string functions

using namespace std;

int main() {

   char name[20] = "Hardik"

   cout << "Name: " << name << endl;

   cout << "Length: " << strlen(name) << endl;   // string length

   cout << "Copy: " << strcpy(name, "ChatGPT") << endl; // copy string

   return 0;

}
```

Output:

```
Name: Hardik
```

```
Length: 6

Copy: ChatGPT
```

Common Functions (from <cstring>):

- strlen(str) → finds length
- strcpy(dest, src) → copies string
- strcat(str1, str2) → concatenates
- strcmp(str1, str2) → compares

---

2. C++ String Class (std::string)

- Easier and safer than C-style strings.
- Requires **#include <string>**.
- Supports operators like +, ==, [].

Example:

```
#include <iostream>

#include <string>

using namespace std;


int main() {

    string first = "Hello";

    string second = "World"

    cout << "Concatenation: " << first + " " + second << endl;

    cout << "Length of first: " << first.length() << endl;

    cout << "First character: " << first[0] << endl

    if(first == "Hello")

        cout << "Strings are equal" << endl;
    return 0;
}
```

Output:

```
Concatenation: Hello World
Length of first: 5
First character: H
Strings are equal
```

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

Array Initialization in C++

In C++, arrays can be initialized at the time of declaration or later by assigning values to each element.

---

1. Initializing a Single-Dimensional (1D) Array

Methods:

1. **Explicit initialization with all elements**

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

2. **Partial initialization (rest will be 0 by default)**

```
int arr[5] = {10, 20};   // arr = {10, 20, 0, 0, 0}
```

3. **Compiler counts size automatically**

```
int arr[] = {1, 2, 3, 4};   // size = 4
```

4. **Assigning values one by one**

```
int arr[3];
arr[0] = 100;
```

```
arr[1] = 200;
arr[2] = 300;
```

Example Program (1D Array):

```
#include <iostream>
using namespace std;
int main() {
   int marks[5] = {90, 80, 70, 60, 50};
   cout << "Student Marks: ";
   for(int i = 0; i < 5; i++) {
      cout << marks[i] << " ";
   }
   return 0;
}
```

Output:

```
Student Marks: 90 80 70 60 50
```

---

2. Initializing a Multi-Dimensional (2D) Array

Methods:

1. **Row-wise initialization**

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

2. **Single line initialization**

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

3. **Partial initialization (remaining set to 0)**

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

Example Program (2D Array):

```
#include <iostream>

using namespace std;

int main() {

   int matrix[2][3] = { {10, 20, 30}, {40, 50, 60} };

   cout << "Matrix:" << endl;

   for(int i = 0; i < 2; i++) {

      for(int j = 0; j < 3; j++) {

         cout << matrix[i][j] << " ";

      }

      cout << endl;

   }

   return 0;

}
```

Output:

```
Matrix:

10 20 30

40 50 60
```

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

String Operations and Functions in C++

In C++, strings can be handled in **two ways**:

1. **C-style strings** (char[]) → use <cstring> functions.
2. **C++ string class (std::string)** → part of the Standard Library <string>.

---

1. String Operations with C-Style Strings

C-style strings are **character arrays** ending with a '\0' (null character).
We use functions from the <cstring> header.

Common Functions:

- strlen(str) → returns length of the string
- strcpy(dest, src) → copies string
- strcat(str1, str2) → concatenates two strings
- 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[20] = "World";


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

   cout << "Copy str2 into str1: " << strcpy(str1, str2) << endl;

   cout << "Concatenate: " << strcat(str1, " C++") << endl;

   cout << "Compare: " << strcmp("ABC", "ABD") << endl;


   return 0;

}
```

Output:

```
Length of str1: 5
```

```
Copy str2 into str1: World

Concatenate: World C++

Compare: -1   (means "ABC" < "ABD")
```

---

2. String Operations with C++ std::string

The std::string class is more powerful and safer. It supports operators (+, ==, []) and has many built-in functions.

Common Operations and Functions:

- **Concatenation:** str1 + str2
- **Comparison:** ==, !=, <, >
- **Access character:** str[i]
- **Length of string:** str.length() or str.size()
- **Substring:** str.substr(pos, len)
- **Find substring:** str.find("word")
- **Insert:** str.insert(pos, "text")
- **Erase:** str.erase(pos, len)
- **Append:** str.append("text")

Example:

```
#include <iostream>

#include <string>

using namespace std;


int main() {

   string s1 = "Hello";

   string s2 = "World";
```

```
    cout << "Concatenation: " << s1 + " " + s2 << endl;

    cout << "Length of s1: " << s1.length() << endl;

    cout << "First char of s1: " << s1[0] << endl;

    s1.append(" C++");

    cout << "After append: " << s1 << endl;

    cout << "Substring (2, 3): " << s2.substr(2, 3) << endl;

    cout << "Find 'World': " << s1.find("World") << endl;

    s1.erase(0, 6);

    cout << "After erase: " << s1 << endl;

    return 0;

}
```

Output:

```
Concatenation: Hello World

Length of s1: 5

First char of s1: H

After append: Hello C++

Substring (2, 3): rld

Find 'World': npos (not found)

After erase: C++
```

6. Introduction to Object-Oriented Programming

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

Key Concepts of OOP

OOP (Object-Oriented Programming) is a programming paradigm based on the idea of **objects** that represent real-world entities.

It provides a way to structure code for **reusability, flexibility, and maintainability**.

---

1. Class and Object

- **Class** → A blueprint/template for creating objects. It defines data (attributes) and functions (methods).
- **Object** → An instance of a class.

Example:

```
class Car {
public:
   string brand;
   int speed;

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

int main() {
   Car c1;          // Object
   c1.brand = "BMW";
   c1.speed = 120;
   c1.drive();
}
```

---

2. Encapsulation

- Wrapping **data (variables)** and **functions (methods)** together inside a class.
- Restricts direct access to data → provides **data hiding**.
- Achieved using **access specifiers** (private, protected, public).

Example:

```
class BankAccount {
private:
   int balance;

public:
   void deposit(int amount) { balance += amount; }
   int getBalance() { return balance; }
};
```

---

3. Abstraction

- Showing **essential features** of an object while hiding unnecessary details.
- Achieved using **abstract classes** or **interfaces** (in C++ → pure virtual functions).

Example:

```
class Shape {
public:
   virtual void draw() = 0;   // Pure virtual function
};
class Circle : public Shape {
public:
   void draw() { cout << "Drawing Circle" << endl; }
};
```

---

4. Inheritance

- Mechanism of creating a new class (**child/derived**) from an existing class (**parent/base**).
- Promotes **code reusability**.

Types of Inheritance in C++:

- Single
- Multiple
- Multilevel
- Hierarchical
- Hybrid

Example:

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

---

5. Polymorphism

- **Polymorphism = Many forms**
- Allows functions or operators to behave differently based on input or object type.

Types:

1. **Compile-time (Static)**
    - Function Overloading
    - Operator Overloading
2. **Run-time (Dynamic)**
    - Function Overriding using **virtual functions**

Example (Function Overloading):

```
class Math {
public:
   int add(int a, int b) { return a + b; }
   double add(double a, double b) { return a + b; }
};
```

Example (Function Overriding):

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

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

1. Class

- A **class** is a user-defined data type.
- It acts like a **blueprint** or **template** for creating objects.

- A class defines **data members (variables)** and **member functions (methods)**.

Think of a **class as a design** (like the blueprint of a car).

---

2. Object

- An **object** is an instance of a class.
- It represents a real-world entity created from the class.
- Each object has its **own copy of data members**, but **shares the methods** defined in the class.

Think of an **object as the actual car** built from the blueprint.

Example Program

```
#include <iostream>
using namespace std;
// Class definition
class Car {
public:
    string brand;
    int speed;
    // Member function
    void drive() {
        cout << brand << " is driving at " << speed << " km/h." << endl;
    }
};

int main() {
    // Creating objects of Car
    Car car1;
    car1.brand = "BMW";
    car1.speed = 120;
    car1.drive();

    Car car2;
    car2.brand = "Audi";
    car2.speed = 150;
    car2.drive();

    return 0;
}
```

Output

```
BMW is driving at 120 km/h.
Audi is driving at 150 km/h.
```

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

Definition

Inheritance is an OOP concept in C++ where one class (**derived/child class**) acquires the properties and functions of another class (**base/parent class**).
It supports **code reusability** and models real-world relationships (*is-a* relationship).

Types of Inheritance

1. **Single** → One base, one derived.

2. **Multiple** → One derived from many bases.
3. **Multilevel** → Derived from another derived class.
4. **Hierarchical** → Many derived from one base.
5. **Hybrid** → Combination.

Syntax

```
class Derived : access Base {
  // extra members
};
```

- Access: public, protected, private.

Example (Single Inheritance)

```
#include <iostream>
using namespace std;
class Animal {      // Base Class
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};
class Dog : public Animal {   // Derived Class
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};
int main() {
   Dog d;
   d.eat();   // Inherited from Animal
   d.bark();  // Defined in Dog
   return 0;
}
```

Output

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

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

Definition

Encapsulation is one of the **key concepts of OOP**.
It means **binding data (variables) and methods (functions) together in a single unit (class)** and **restricting direct access** to the data.

Simply put: **“Wrapping data and code into one unit and protecting it from outside interference.”**

How It Is Achieved

1. **Using classes** → Data members and methods are grouped together.
2. **Access specifiers** (private, public, protected) control accessibility:
   - private → data hidden from outside
   - public → accessible through functions
   - protected → accessible by child classes

Example

```
#include <iostream>
using namespace std;
class Student {
private:   // Data hidden from outside
    string name;
    int age;
public:
    // Setter function
    void setData(string n, int a) {
        name = n;
        age = a;
    }
    // Getter function
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
int main() {
    Student s1;
    s1.setData("Hardik", 20);  // Access through method
    s1.display();
    return 0;
}
```

Output

```
Name: Hardik, Age: 20
```