Module #3 Introduction to OOPS Programming

1: Introduction to C++

THEORY EXERCISE:

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

Ans: differences between Procedural Programming and Object Oriented Programming

Aspect	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Basic Approach	Follows a top-down approach (problem is broken into smaller functions).	Follows a bottom-up approach (problem is solved by creating objects).
Program Structure	Program is divided into functions.	Program is divided into objects and classes.
Data Handling	Data is often global and shared among functions; less secure.	Data is encapsulated within objects, making it more secure.
Reusability	Functions can be reused, but managing large code is harder.	Classes and objects provide high reusability through inheritance and polymorphism.
Security	Weak security – any function can access global data.	Strong security – data can be made private and accessible only through methods.
Abstraction	No built-in abstraction mechanism; done manually.	Provides abstraction using classes and interfaces.
Examples	C, Pascal, Fortran.	C++, Java, Python, C#.

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

Ans: Main Advantages of OOP over POP

Advantage	Explanation
1. Modularity (Code Organization)	OOP organizes code into classes and objects, making it easier to manage large projects, while POP uses functions which can get messy in big programs.
2. Reusability	Through inheritance, existing classes can be reused to create new classes, reducing code duplication. In POP, reusability is limited to copying functions.
3. Data Security (Encapsulation)	OOP uses encapsulation (data hiding) by making class data private and providing controlled access. POP often uses global data, which is less secure.
4. Flexibility (Polymorphism)	OOP supports polymorphism, allowing the same function name or operator to work in different ways (e.g., function overloading). POP does not support this.
5. Extensibility	OOP programs are easier to modify and extend because new classes/objects can be added without disturbing existing code. In POP, modifying code may affect many functions.
6. Abstraction	OOP supports abstraction, letting you focus on essential details while hiding complexity. POP has no direct support for abstraction.
7. Better Maintenance	Since OOP code is modular and organized, it is easier to debug, test, and maintain compared to large POP programs.
8. Real-world Modeling	OOP naturally maps to real-world entities (like Car, BankAccount, Student) as objects, making programs

Advantage	Explanation
	more intuitive. POP cannot represent real-world concepts as naturally.

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

Ans: Steps to Set Up a C++ Development Environment

Step 1: Install a C++ Compiler

- A compiler is required to convert your C++ code into machine code.
- Popular compilers:
    - GCC (GNU Compiler Collection) → used on Linux and Windows (via MinGW or Cygwin).
    - Clang → often used on macOS and Linux.
    - MSVC (Microsoft Visual C++) → comes with Visual Studio on Windows.

◆ Windows:

- Install MinGW or TDM-GCC for GCC.
- Alternatively, install Microsoft Visual Studio (includes MSVC compiler).

◆ macOS:

- Install Xcode Command Line Tools by running in terminal:

Step 2: Choose an IDE or Code Editor

- IDE = Integrated Development Environment (comes with editor, debugger, compiler integration).
- Popular choices:
    - Code::Blocks (lightweight IDE, beginner friendly).

- Dev-C++ (classic Windows IDE).
- Visual Studio (Windows, professional).
- CLion (cross-platform, JetBrains IDE).
- VS Code (lightweight editor, needs extensions).

Step 3: Configure Compiler in the IDE

- If you install an IDE like **Code::Blocks with MinGW**, compiler comes pre-configured.
- For **VS Code**:
  1. Install **C/C++ extension** by Microsoft.
  2. Configure "tasks.json" and "launch.json" for build and debug.
  3. Ensure your compiler (like g++) is in the system PATH.

Step 4: Write Your First C++ Program

Example: hello.cpp

```
#include <iostream>

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

Step 5: Compile and Run the Program

Command line (Linux/macOS/Windows with MinGW)

```
g++ hello.cpp -o hello
./hello     # (Linux/macOS)
hello.exe   # (Windows)
```

Step 6: Debugging Tools (Optional but Important)

- Use the **debugger** in your IDE (like GDB with Code::Blocks or VS Code).

- Helps find logic errors and runtime bugs step by step.

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

Ans: Main Input/Output Operations in C++

- C++ uses the iostream library for input and output operations. The two most common objects are:
- cin → used for input.
- cout → used for output.
- Example:

```
#include <iostream.h>
int main(){
  int age;
  string name;
  cout << "Enter your name: ";
  cin>>name;
  cout << "Enter your age: ";
  cin >> age;
  cout << "Name:" << name << "Age" << age;
  return 0;
}
```

2. Variables, Data Types, and Operators

THEORY EXERCISE:

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

Ans: There are several types of data types in c++;

- **Basic (Primitive) Data Types**

  These are the fundamental types built into C++.

  - **int →** stores integers (whole numbers, positive or negative).
  - **float →** stores decimal numbers (single precision).

- **double →** stores decimal numbers (double precision, more accurate than float).
- **char →** stores a single character (enclosed in ' '), actually stored as ASCII value.
- **bool →** stores Boolean values true (1) or false (0).
- **void →** represents "nothing" or "no value" (often used in functions that don't return a value).
- **Example: #include <iostream.h>**

```
int main() {

  int age = 20;          // integer

  float price = 49.99;    // single precision decimal

  double pi = 3.14159265;  // double precision decimal

 char grade = 'A';       // single character

  bool isPass = true;     // boolean (true/false)

  cout << "Age: " << age << endl;

  cout << "Price: " << price << endl;

  cout << "PI: " << pi << endl;

  cout << "Grade: " << grade << endl;

  cout << "Pass Status: " << isPass << endl;

  return 0;

}
```

➢ Derived Data Types

Built using basic data types.

- **Array → collection of same type of elements.**
- **Pointer → stores memory address of another variable.**
- **Example:** #include <iostream.h>
  int main() {

```
int numbers[3] = {10, 20, 30};  // array
 int x = 5;
 int* ptr = &x;              // pointer
 int& ref = x;               // reference
cout << "Array[1]: " << numbers[1] << endl;
 cout << "Pointer value: " << *ptr << endl;
 cout << "Reference value: " << ref << endl;
 return 0;
}
```

- **User-Defined Data Types**
  - **struct →** groups variables of different types under one name.
  - **class →** similar to struct, but with data + functions (OOP).
  - **enum →** collection of named integral constants.
- **Example:**

```
#include <iostream>
// structure
struct Student {
   int id;
   char grade;
};

// enum
enum Color { RED, GREEN, BLUE };

int main() {
   Student s1 = {101, 'A'};
   Color favColor = GREEN;

   cout << "Student ID: " << s1.id << ", Grade: " << s1.grade << endl;
   cout << "Favorite Color (enum): " << favColor << endl;
   return 0;
}
```

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

Ans: difference between implicit and explicit type conversion in C++

Aspect	Implicit Type Conversion (Type Casting)	Explicit Type Conversion (Type Casting)
Definition	Conversion of one data type to another automatically by the compiler.	Conversion of one data type to another done manually by the programmer.
Also Known As	Type promotion / Type casting (done by compiler).	Type casting (done explicitly by programmer).
Control	No control by programmer; compiler decides.	Full control by programmer.
When It Happens	When assigning a smaller data type value to a larger data type variable.	When the programmer specifies the desired type using cast operators.
Syntax	No special syntax needed; happens automatically.	Requires cast operator like (type) or static_cast<type>().
Example	cpp\nint a = 5;\ndouble b = a; // int → double (implicit)\n	cpp\nfloat a = 9.87;\nint b = (int)a; // explicit casting\n
Risk of Data Loss	May cause unintentional data loss (e.g., float → int).	Programmer is aware and responsible for possible data loss.
Use Case	Used for safety and convenience by compiler.	Used when precision control is required.

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

Ans: In C++, operators are special symbols that perform operations on variables and values. They are grouped into categories based on their functionality.

1. Arithmetic Operators

- **Used for mathematical calculations.**

Operator	Description	Example (a=10, b=3)	Result
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3
%	Modulus (remainder)	a % b	1

- **Example:**
```
#include <iostream.h>
int main() {
   int a = 10, b = 3;
   cout << "a + b = " << (a + b) << endl;
   cout << "a - b = " << (a - b) << endl;
   cout << "a * b = " << (a * b) << endl;
   cout << "a / b = " << (a / b) << endl;
   cout << "a % b = " << (a % b) << endl;
   return 0;
}
```

2. Relational (Comparison) Operators

- **Used to compare two values (result is true or false).**

Operator	Description	Example (a=10, b=3)	Result
==	Equal to	a == b	false
!=	Not equal to	a != b	true
>	Greater than	a > b	true
<	Less than	a < b	false
>=	Greater or equal	a >= b	true
<=	Less or equal	a <= b	false

- **Example:** #include <iostream.h>

```
int main() {
   int a = 10, b = 3;
  cout << "(a == b) = " << (a == b) << endl;
   cout << "(a != b) = " << (a != b) << endl;
   cout << "(a > b)  = " << (a > b) << endl;
   cout << "(a < b)  = " << (a < b) << endl;
   cout << "(a >= b) = " << (a >= b) << endl;
   cout << "(a <= b) = " << (a <= b) << endl;
 return 0;
}
```

3. Logical Operators

- **Used to combine conditions.**

Operator	Description	Example (a=10, b=3)	Result
&&	Logical AND	(a > 5 && b < 5)	true
||	Logical OR	(a > 5 || b > 5)	true
!	Logical NOT	!(a > b)	false

- **Example:** #include <iostream.h>

```
int main() {
    int a = 10, b = 3;
    cout << "(a > 5 && b < 5) = " << (a > 5 && b < 5) << endl;
    cout << "(a > 5 || b > 5) = " << (a > 5 || b > 5) << endl;
    cout << "!(a > b) = " << !(a > b) << endl;
    return 0;
}
```

4. Assignment Operators

- **Used to assign values to variables.**

Operator	Example (a=10)	Equivalent To
=	a = 5	a = 5
+=	a += 3	a = a + 3 → 13
-=	a -= 2	a = a - 2 → 8
*=	a *= 2	a = a * 2 → 20
/=	a /= 2	a = a / 2 → 5
%=	a %= 3	a = a % 3 → 1

- **Example:**

```
#include <iostream.h>
int main() {
    int a = 10;
    cout << "Initial a = " << a << endl;
   a += 5; cout << "a += 5 -> " << a << endl;
    a -= 2; cout << "a -= 2 -> " << a << endl;
    a *= 3; cout << "a *= 3 -> " << a << endl;
    a /= 4; cout << "a /= 4 -> " << a << endl;
    a %= 5; cout << "a %= 5 -> " << a << endl;
    return 0;
```

}

5. Increment & Decrement Operators

- **Increase or decrease variable value by 1.**

Operator	Example (a=5)	Result
Work at bit level (used in low-level programming).	Pre-increment	a=6 (increments first, then uses)
a++	Post-increment	a=5 (uses first, then increments → next time 6)
--a	Pre-decrement	a=4
a--	Post-decrement	a=5 (then decrements to 4)

- **Example:**
```
#include <iostream.h>
int main() {
   int a = 5;
   cout << "a++ = " << a++ << " (then a = " << a << ")" << endl;
   cout << "++a = " << ++a << endl;
   cout << "a-- = " << a-- << " (then a = " << a << ")" << endl;
   cout << "--a = " << --a << endl;
   return 0;
}
```

6. Bitwise Operators

- **Work at bit level (used in low-level programming).**

Operator	Description	Example (a=5(0101), b=3(0011))
&	AND	a & b = 0001 = 1

Operator	Description	Example (a=5(0101), b=3(0011))
|	OR	a | b = 0111 = 7
^	XOR	a ^ b = 0110 = 6
~	NOT (1's complement)	~a = ...1010 (depends on system, result -6 in 2's complement)
<<	Left shift	a << 1 = 1010 = 10
>>	Right shift	a >> 1 = 0010 = 2

- **Example:** #include <iostream.h>

```
int main() {
   int a = 5, b = 3;  // binary: a=0101, b=0011
   cout << "a & b = " << (a & b) << endl;
   cout << "a | b = " << (a | b) << endl;
   cout << "a ^ b = " << (a ^ b) << endl;
   cout << "~a = " << (~a) << endl;
   cout << "a << 1 = " << (a << 1) << endl;
   cout << "a >> 1 = " << (a >> 1) << endl;
   return 0;
}
```

7. Conditional (Ternary) Operator

- **Shorthand for if-else.**
- **Example:** #include <iostream.h>

```
int main() {
   int a = 10, b = 20;
   int max = (a > b) ? a : b;
```

```
    cout << "Max value = " << max << endl;
    return 0;
}
```

8. Special Operator

- **Used with classes, structures, and pointers.**

Operator	Description	Example
.	Dot operator (access member)	object.data
->	Arrow operator (access via pointer)	ptr->data
::	Scope resolution (access global or class member)	std::cout

Example:

```
#include <iostream.h>
int value = 100; // global variable

class Student {
public:
   string name;
   void show() {
      cout << "Name: " << name << endl;
   }
};

int main() {
   Student s;
   s.name = "Alice";   // dot operator
   s.show();

   Student *p = &s;
   p->name = "Bob";    // arrow operator
   p->show();
   int value = 50;     // local variable
   cout << "Local value = " << value << endl;
```

```
    cout << "Global value = " << ::value << endl;  // scope resolution
    return 0;
}
```

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

Ans: 1. Constants in C++

A constant is a variable whose value cannot be changed once it is defined. They are useful when you want to protect values from being modified accidentally. Using a const keyword.

- **Example:** #include <iostream.h>

```
int main() {
   const int maxStudents = 50;   // constant integer
   cout << "Maximum students allowed: " << maxStudents << endl;
   // maxStudents = 100;   Error: cannot modify constant
   return 0;
}
```

Purpose of Constants:

- Improves readability (meaningful names instead of numbers).
- Prevents accidental modification.
- Makes program easier to maintain.

2. Literals in C++

A literal is a fixed value that appears directly in the source code.
They are the actual values you assign to variables or use directly.

3. Control Flow Statements

THEORY EXERCISE:

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

Ans: There are several conditional statements in c++;

- **if Statement**
  - Executes a block of code only if the condition is true.
  - **Syntax:** if (condition) { // code if condition is true }
- **if-else statement**
  - Executes one block if the condition is true, another if false.
  - **Syntax:** if (condition) { // code if condition is true } else { // code if condition is false }
- **Nested if Statement**
  - Used when there are multiple conditions to check.
- **if-else if Ladder**
  - Used when there are multiple conditions to check.
  - **Syntax:** if (condition1) { // code if condition1 is true
    } else if (condition2) {
    // code if condition2 is true
    } else if (condition3) {
    // code if condition3 is true
    } else {
    // code if all are false}
- **switch Statement**
  - Used when you need to choose between multiple constant values of an expression.
  - **Syntax:** switch(expression) {
    case value1:
    // code
    break;
    case value2:
    // code
    break;
    ...
    default:
    // code if no match

}

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

Ans: Difference between For, While and Do-While Loop in Programming:

Feature	For Loop	While Loop	do- While Loop
Syntax	for (initialization; condition; increment/decrement) {}	while (condition) { }	do { } while (condition);
Initialization	Declared within the loop structure and executed once at the beginning.	Declared outside the loop; should be done explicitly before the loop.	Declared outside the loop structure
Condition	Checked before each iteration.	Checked before each iteration.	Checked after each iteration.
Update	Executed after each iteration.	Executed inside the loop; needs to be handled explicitly.	Executed inside the loop; needs to be handled explicitly.
Use Cases	Suitable for a known number of iterations or when looping over ranges.	Useful when the number of iterations is not known in advance or based on a condition.	Useful when the loop block must be executed at least once, regardless of the initial condition.
Initialization and Update Scope	Limited to the loop body.	Scope extends beyond the loop; needs to be handled explicitly.	Scope extends beyond the loop; needs to be handled explicitly.

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

Ans: 1. break Statement

- **Used to immediately terminate a loop.**
- **Control jumps out of the loop completely.**

- **Example: #include <iostream.h>**

```
int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         break; // loop ends when i = 5
      }
      cout << i << " ";
   }
   return 0;
} // Output:- 1234
```

2. continue Statement

- **Used to skip the current iteration of the loop.**
- **Control goes back to the loop condition for the next iteration.**
- **Example: #include <iostream.h>**

```
int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         continue; // skip printing 5
      }
      cout << i << " ";
   }
   return 0;
}
```

4. Explain nested control structures with an example.

Ans: <u>Nested Control Structures in C++</u>

- nested control structure means placing one control structure (like if, for, while, or switch) inside another.
  This allows you to build more complex logic by combining multiple conditions or loops.
- **Example:** #include <iostream.h>

```
int main() {
   int num;
```

```
    cout << "Enter a number: ";
    cin >> num;
    if (num > 0) {
        if (num % 2 == 0) {
            cout << "The number is positive and even.";
        } else {
            cout << "The number is positive and odd.";
        }
    } else {
        cout << "The number is not positive.";
    }
    return 0;
}
```

4. Functions and Scope

THEORY EXERCISE:

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. Functions help in modular programming, making the code organized, reusable, and easier to debug.

- **Syntax:** return_type function_name(parameters) { // body of the function }
- **Function Declaration (Prototype)**
  - A function declaration tells the compiler about the function's name, return type, and parameters without defining its body. It is usually placed before main() so that the compiler knows about the function before it is called.
  - **Syntax: return_type function_name(parameter_list);**
  - **Example: int add(int a, int b); // function declaration**
- **Function Definition**
  - A function definition provides the actual body of the function — the code that will run when the function is called.

- **Syntax:** return_type function_name(parameter_list) { // body of the function}
- **Example:** int add(int a, int b) { // function definition
  return a + b;}

➢ **Function Calling**

- A function call executes the function by passing arguments (if any). The program jumps to the function definition, executes it, and returns (if it has a return value).
- **Syntax: function_name(arguments);**
- **Example: sum = add(5, 3); // function call**

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

Ans: <u>Difference between Local and Global Variables</u>

Feature	Local Variable	Global Variable
Declared	Inside a function or block	Outside all functions
Scope	Only within the function/block	Throughout the program
Lifetime	Exists only during function execution	Exists throughout program execution
Memory Allocation	Automatic (stack)	Static (data segment)
Default Value	Garbage (if not initialized)	0 (for basic types)
Access	Not accessible outside its function	Accessible by all functions

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

Ans: Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem.
It is useful for problems that can be broken down into smaller, similar sub-problems.

- **Key points:**

- A recursive function must have a base condition to stop recursion, otherwise it will run indefinitely and cause a stack overflow.
- It consists of:
    1. Base case: Condition where recursion stops.
    2. Recursive case: Function calls itself with a smaller problem.

- **Syntax: return_type function_name(parameters) {**
  if (base_condition) {
  // stop recursion
  return value;
  } else {
  // recursive call
  return function_name(smaller_problem);
  }
  }
- **Example:** #include <iostream.h>

// Recursive function to calculate factorial

```
int factorial(int n) {
   if (n == 0)   // base case
      return 1;
   else         // recursive case
      return n * factorial(n - 1);
}

int main() {
```

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

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

Ans: A function prototype is a declaration of a function that informs the compiler about:

- The function's name
- Its return type
- Its parameters (number and type)

- **Syntax: return_type function_name(parameter_list);**
- **Example: int add(int a, int b); // function prototype**
- **Why Are Function Prototypes Used?**
    - **Inform the compiler beforehand:**
      The compiler knows about the function before its actual definition. This is important when the function is **called before it is defined** in the code.
    - **Enable modular programming:**
      Function prototypes allow splitting programs into **multiple files**, where definitions can be in another file.
    - **Type checking:**
      The compiler checks whether the **arguments in the function call** match the **parameters in the prototype**, reducing errors.

5. Arrays and Strings

THEORY EXERCISE:

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

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

Arrays allow you to store multiple values under a single variable name and access them using an index.

- **Syntax: data_type array_name[size];**
- **Example: int numbers[5] = {1, 2, 3, 4, 5}; // single-dimensional array**

1. **Single-Dimensional Array**
    - A single-dimensional array is a linear array with one row of elements.
    - **Example:**
```
#include <iostream.h>
int main() {
   int arr[5] = {10, 20, 30, 40, 50};

   for (int i = 0; i < 5; i++) {
      cout << "arr[" << i << "] = " << arr[i] << endl;
   }
   return 0;
}
```
2. **Multi-Dimensional Array**
    - A multi-dimensional array has more than one index, commonly two-dimensional (like a matrix) but can have more dimensions.
    - **Syntax: data_type array_name[rows][columns];**
    - **Example:**
```
#include <iostream.h>
int main() {
   int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };

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

- **<u>Difference Between Single-Dimensional and Multi-Dimensional Arrays</u>**

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	Linear (1D)	Table-like, matrix or higher dimensions
Indexing	One index [i]	Multiple indices [i][j], [i][j][k]
Memory Representation	Contiguous block	Contiguous block row-wise (2D) or more
Example	int arr[5];	int mat[2][3];
Use Case	Simple list of elements	Storing tabular data, matrices, grids

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

Ans: A string is a sequence of characters. In C++, strings can be handled in two ways:

1. **C-style strings –** Arrays of characters ending with a null character \0.
    - Declared as a **character array**.
    - End with a **null character \0** to mark the end.
    - **Example:**
```
#include <iostream.h>
int main() {
   char str[20] = "Hello C++"; // C-style string
   cout << "String: " << str << endl;
   return 0;
}
```

Common C-style string functions (#include <cstring>):

Function	Description	Example
strlen(str)	**Returns length of string**	**strlen(str)**
strcpy(dest, src)	**Copies string from src to dest**	**strcpy(str2, str1)**
strcat(dest, src)	**Concatenates src to dest**	**strcat(str1, str2)**
strcmp(str1, str2)	**Compares two strings**	**Returns 0 if equal**

- **Example:** #include <iostream.h>

```
#include <cstring.h>
int main() {
   char str1[20] = "Hello";
   char str2[20] = "World";
   cout << "Length of str1: " << strlen(str1) << endl;
   strcpy(str1, str2); // Copy str2 into str1
   cout << "After copy, str1: " << str1 << endl;
   strcat(str1, " C++"); // Concatenate
   cout << "After concatenation: " << str1 << endl;
   return 0;
}
```

2. **C++ string class –** Part of the Standard Template Library (STL), provides convenient string operations.
   - Part of <string> library.
   - Easier and safer than C-style strings.
   - Supports operators and functions for string handling.

Example: #include <iostream.h>

```
#include <string.h>
int main() {
   string str1 = "Hello";
   string str2 = "C++";
```

```
    string str3 = str1 + " " + str2; // Concatenation
    cout << "Concatenated String: " << str3 << endl;
    cout << "Length: " << str3.length() << endl;
    str3.append(" Programming"); // Append another string
    cout << "After append: " << str3 << endl;
    cout << "Substring: " << str3.substr(6, 3) << endl; // Substring
    return 0;
}
```

Common string class functions:

Function	Description
length()	Returns length of string
append()	Adds another string at the end
substr(pos, len)	Returns substring starting at pos with length len
find()	Finds position of substring
replace()	Replaces part of string with another
c_str()	Converts to C-style string (char*)

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

Ans: 1. Initializing 1D Arrays (Single-Dimensional Arrays)

- A 1D array can be initialized at the time of declaration in several ways.
- **Syntax: data_type array_name[size] = {value1, value2, ..., valueN};**
  Example: #include <iostream.h>

```
int main() {
    // Method 1: Initialize all elements explicitly
    int arr1[5] = {1, 2, 3, 4, 5};
    // Method 2: Partial initialization (rest elements will be 0)
    int arr2[5] = {10, 20};
    // Method 3: Automatic size deduction
    int arr3[] = {5, 10, 15, 20};
    // Print arr1
    cout << "arr1: ";
    for (int i = 0; i < 5; i++)
        cout << arr1[i] << " ";
    cout << endl;
    // Print arr2
    cout << "arr2: ";
    for (int i = 0; i < 5; i++)
        cout << arr2[i] << " ";
    cout << endl;
    // Print arr3
    cout << "arr3: ";
    for (int i = 0; i < 4; i++)
        cout << arr3[i] << " ";
    cout << endl;
    return 0;
}
```

2. Initializing 2D Arrays (Multi-Dimensional Arrays)

- **A 2D array is like a matrix with rows and columns. It can also be initialized at declaration.**
- **Syntax: data_type array_name[rows][columns] = { {row1_elements}, {row2_elements}, ... };**

Example:

```
#include <iostream.h>
int main() {
    // Method 1: Full initialization
    int matrix1[2][3] = { {1, 2, 3}, {4, 5, 6} };
    // Method 2: Partial initialization (remaining elements set to 0)
```

```
    int matrix2[2][3] = { {10, 20}, {30} };
    // Print matrix1
    cout << "matrix1:" << endl;
    for (int i = 0; i < 2; i++) {
       for (int j = 0; j < 3; j++)
          cout << matrix1[i][j] << " ";
       cout << endl;   }
    // Print matrix2
    cout << "matrix2:" << endl;
    for (int i = 0; i < 2; i++) {
       for (int j = 0; j < 3; j++)
          cout << matrix2[i][j] << " ";
       cout << endl;
    }
    return 0;
}
```

6. Introduction to Object-Oriented Programming

THEORY EXERCISE:

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

Ans: Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects rather than procedures or functions.

An **object** represents a real-world entity with **attributes (data)** and **behaviors (methods/functions)**.

C++ is an **object-oriented language**, allowing the use of OOP principles for better code organization, reusability, and maintainability.

2. Key Concepts of OOP

Concept	Description	Example in C++
Class	**A blueprint or template for creating objects. It defines data members and member functions.**	**class Car { int speed; void drive(); };**

Concept	Description	Example in C++
Object	**An instance of a class. Each object has its own state and behavior.**	**Car myCar;**
Encapsulation	**Wrapping data and methods together and restricting access using access specifiers (private, public, protected).**	**class Car { private: int speed; public: void setSpeed(int s); };**
Abstraction	**Hiding internal details and showing only essential features to the user.**	**Using getter/setter methods instead of directly accessing data.**
Inheritance	**A mechanism to derive a new class from an existing class, reusing code.**	**class SportsCar : public Car {};**
Polymorphism	**Ability of a function, object, or operator to take multiple forms. Two types: 1. Compile-time (Overloading) 2. Run-time (Virtual functions)**	**void print(int x); void print(double y); virtual void display();**
Message Passing	**Objects communicate by calling each other's methods.**	**myCar.drive();**

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

Ans: A class is a blueprint or template for creating objects.
It defines attributes (data members) and behaviors (member functions/methods) that the objects of the class will have.

- **Syntax: class ClassName {**
  // Access specifier
  access_specifier:
  data_members; // variables
  member_functions; // functions };
- **Example:** class Car {

```
public:
  int speed;           // Data member (attribute)

  void drive() {       // Member function (behavior)
    cout << "Car is driving at " << speed << " km/h" << std::endl;
  }
};
```

- **Object?**

An **object** is an **instance of a class**.

- Objects represent **real-world entities**.
- Each object has its **own copy of data members** and can call member functions defined in the class.
- **Syntax: ClassName object_name;**
- **Example:**

```
int main() {
  Car myCar;       // Object creation
  myCar.speed = 100; // Assign value to data member
  myCar.drive();   // Call member function
  Car yourCar;
  yourCar.speed = 150;
  yourCar.drive();
  return 0;
}
```

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

Ans: Inheritance is an OOP concept where a new class (derived class/child class) is created from an existing class (base class/parent class).

- The derived class inherits attributes and behaviors of the base class.
- It allows code reusability and extensibility.
- **Types of Inheritance in C++:**
  1. **Single inheritance – One base class → One derived class**
     - **Syntax: class Base { // Base class members};**

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

- **Example:** #include <iostream.h>

```
class Vehicle {
public:
   void honk() { cout << "Beep beep!" << endl; }
};
class Car : public Vehicle {
public:
   void show() { cout << "Car class" << endl; }
};
int main() {
   Car myCar;
   myCar.honk();  // Inherited
   myCar.show();  // Own method
   return 0;
}
```

2. **Multiple inheritance – Multiple base classes → One derived class**

- **Syntax: class Base1 { /* ... */ };**

class Base2 { /* ... */ };

class Derived : public Base1, public Base2 { /* ... */ };

- **Example:** #include <iostream.h>

```
class Engine {
public:
   void start() { cout << "Engine started" << endl; }
};
class Wheels {
public:
   void rotate() { cout << "Wheels rotating" << endl; }
};
class Car : public Engine, public Wheels {
public:
   void show() { cout << "Car is ready" << endl; }
};
int main() {
```

```
    Car myCar;
    myCar.start();  // From Engine
    myCar.rotate(); // From Wheels
    myCar.show();   // Own method
    return 0;
}
```

3. **Multilevel inheritance – Base → Derived → Another derived**
   - **Syntax: class Base { /* ... */ };**

   class Derived1 : public Base { /* ... */ };
   class Derived2 : public Derived1 { /* ... */ };
   - **Example:**

```
#include <iostream.h>
class Vehicle {
public:
   void honk() { cout << "Vehicle honks" << endl; }
};
class Car : public Vehicle {
public:
   void show() { cout << "Car class" << endl; }
};
class SportsCar : public Car {
public:
   void turbo() { cout << "Turbo mode activated" << endl; }
};
int main() {
   SportsCar myCar;
   myCar.honk();  // Inherited from Vehicle
   myCar.show();  // Inherited from Car
   myCar.turbo(); // Own method
   return 0;
}
```

4. **Hierarchical inheritance – One base → Multiple derived classes**
   - **Syntax: class Base { /* ... */ };**

   class Derived1 : public Base { /* ... */ };
   class Derived2 : public Base { /* ... */ };
   - **Example:**

```
#include <iostream.h>
```

```
class Vehicle {
public:
   void honk() { cout << "Vehicle honks" << endl; }
};
class Car : public Vehicle {
public:
   void show() { cout << "Car class" << endl; }
};
class Bike : public Vehicle {
public:
   void display() { cout << "Bike class" << endl; }
};
int main() {
   Car c;
   Bike b;
   c.honk();  // From Vehicle
   c.show();
   b.honk();  // From Vehicle
   b.display();
   return 0;
}
```

5. **Hybrid inheritance – : Combination of two or more inheritance types (usually multiple + multilevel).**
   - **Syntax: class Base { /* ... */ };**

   class Derived1 : public Base { /* ... */ };
   class Derived2 : public Base { /* ... */ };
   class Final : public Derived1, public Derived2 { /* ... */ };
   - **Example:** #include <iostream.h>

```
class Vehicle {
public:
   void honk() { cout << "Vehicle honks" << endl; }
};
class Car : public Vehicle {
```

```
public:
   void show() { cout << "Car class" << endl; }
};
class Engine {
public:
   void start() { cout << "Engine starts" << endl; }
};
class SportsCar : public Car, public Engine {
public:
void turbo() { cout << "Turbo mode activated" << endl; }
};
int main() {
   SportsCar myCar;
   myCar.honk();   // From Vehicle
   myCar.start();  // From Engine
   myCar.show();   // From Car
   myCar.turbo();  // Own method
   return 0;
}
```

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

Ans: Definition of Encapsulation

Encapsulation is an OOP concept that bundles data (attributes) and methods (functions) together in a single unit (class).

- **It restricts direct access to some of an object's components to protect the internal state.**

- **Data is made private, and access is provided through public member functions (getters and setters).**

Benefits

1. **Data Hiding:** Prevents unauthorized access to sensitive data.
2. **Controlled Access:** Allows validation before modifying data.
3. **Maintainability:** Changes in internal implementation don't affect external code.

❖ How Encapsulation is Achieved in Classes

Encapsulation is achieved in C++ classes using:

1. **Access specifiers:**
    - **private:** Members are accessible only within the class.
    - **public:** Members are accessible from outside the class.
    - **protected:** Members are accessible within the class and derived classes.
2. **Getter and Setter methods:**
    - **Setter:** Function to set or update the value of a private member.
    - **Getter:** Function to read or access the value of a private member.