Module #3: Introduction to OOPS Programming

1. Introduction to C++:

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

Ans: Procedural Programming:

1. **Programming Approach**
   Procedural Programming follows a step-by-step approach where the program is divided into functions or procedures.
2. **Program Focus**
   It mainly focuses on procedures or functions rather than data.
3. **Data Security**
   Data security is low because data can be accessed and modified by any function.
4. **Code Reusability**
   Code reusability is limited as functions are not closely bound with data.
5. **Program Complexity**
   It becomes difficult to manage when programs grow large and complex.
6. **Program Maintenance**
   Maintenance and modification of programs are harder.
7. **Real-World Representation**
   It does not effectively represent real-world entities.

Object-Oriented Programming (OOP) :

1. **Programming Approach**
   Object-Oriented Programming is based on objects and classes that combine data and functions.

2. **Program Focus**
   It mainly focuses on data and the objects that operate on that data.
3. **Data Security**
   Data security is high due to encapsulation and access specifiers.
4. **Code Reusability**
   Code reusability is high through inheritance and polymorphism.
5. **Program Complexity**
   It handles large and complex programs efficiently.
6. **Program Maintenance**
   Programs are easier to maintain, modify, and extend.
7. **Real-World Representation**
   It closely represents real-world entities and their relationships.

Difference table:

Basis	**Procedural Programming**	**Object-Oriented Programming**
Approach	Function-oriented	Object-oriented
Main Focus	Procedures / Functions	Objects and Data
Data Security	Less secure	More secure
Code Reusability	Limited	High
Maintenance	Difficult	Easy
Suitability	Small programs	Large and complex programs
Real-World Modeling	Poor	Excellent

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

Ans: Object-Oriented Programming provides several benefits over Procedural-Oriented Programming, especially when developing large and complex software systems.

Object-Oriented Programming is more powerful than Procedural-Oriented Programming because it provides better security, reusability, maintainability, and scalability, making it ideal for modern software development.

1. Better Data Security:

OOP provides better data security through **encapsulation**. Data members are protected using access specifiers, which prevents unauthorized access.

2. Code Reusability:

OOP supports **code reusability** through inheritance. Existing classes can be reused to create new classes, reducing code duplication.

3. Easy Maintenance:

Programs written using OOP are easier to maintain and modify because changes in one part of the program do not affect other parts.

4. Real-World Modeling:

OOP allows programs to be designed using real-world entities such as objects, making the system easier to understand and design.

5. Reduced Complexity:

OOP divides a large program into smaller objects, which reduces complexity and improves readability.

6. Improved Scalability:

OOP makes it easier to extend programs by adding new classes or features without changing existing code.

7. Better Code Organization:

OOP organizes code into classes and objects, making the program structured, modular, and easier to manage.

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

Ans: A C++ development environment provides the necessary tools to write, compile, run, and debug C++ programs. The following steps explain how to set up a basic C++ development environment.

Step 1: Choose an Operating System

First, select an operating system such as **Windows, Linux, or macOS** on which the C++ programs will be developed.

Step 2: Install a C++ Compiler

A compiler is required to convert C++ source code into executable machine code. Commonly used C++ compilers include **GCC, Clang, and MSVC**.

Step 3: Install an Integrated Development Environment (IDE) or Code Editor

An IDE or code editor helps in writing and managing code efficiently.
Popular choices include **Visual Studio Code, Code::Blocks, Dev-C++, and Visual Studio**.

Step 4: Configure the Compiler Path

After installing the compiler, its path must be added to the system environment variables so that the compiler can be accessed from the command line.

Step 5: Install Required Extensions or Plugins

If using a code editor like Visual Studio Code, install C++ extensions for syntax highlighting, debugging, and code completion.

Step 6: Write a Sample C++ Program

Create a simple C++ program using a text editor or IDE to verify the setup.

Step 7: Compile and Run the Program

Use the compiler to compile the program and run the executable to check whether the environment is correctly set up.

Step 8: Debug and Test

Use debugging tools provided by the IDE to find and fix errors in the program.

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

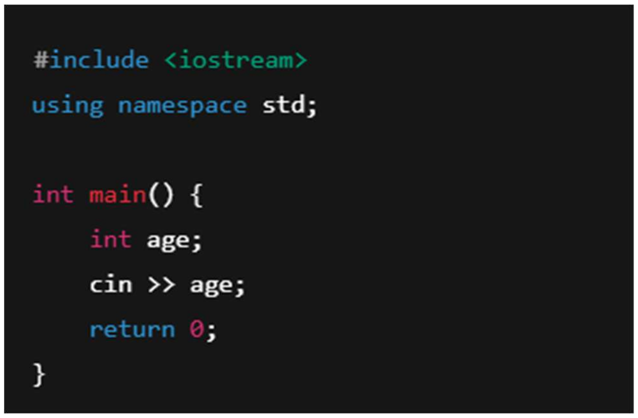
Ans: Main Input/Output Operations in C++

Input and output operations in C++ are used to take data from the user and display results on the screen. These operations are performed using the **iostream** library.

1. cin (Standard Input)

- cin is used to take input from the user through the keyboard.
- It uses the extraction operator >>.

Example:



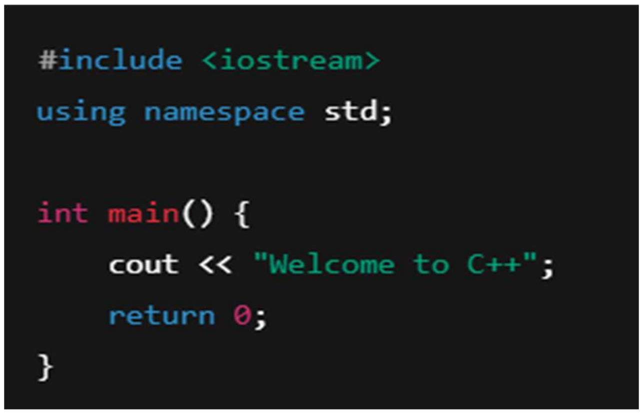
```
#include <iostream>
using namespace std;

int main() {
    int age;
    cin >> age;
    return 0;
}
```

2. cout (Standard Output)

- cout is used to display output on the screen.
- It uses the insertion operator <<.

Example:



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

int main() {
    cout << "Welcome to C++";
    return 0;
}
```

2. Variables, Data Types, and Operators:

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

Ans: 1. Main (Fundamental) Data Types

Data Type	One-Line Definition	Example
int	Stores whole numbers.	int a = 10;
float	Stores decimal numbers with single precision.	float price = 45.5;
double	Stores decimal numbers with double precision.	double salary = 50000.75;
char	Stores a single character.	char grade = 'A';
bool	Stores true or false values.	bool isValid = true;

2. User-Defined Data Types:

Data Type	One-Line Definition	Example
struct	Groups variables of different data types under one name.	struct Student { int roll; };

Data Type	One-Line Definition	Example
class	Blueprint for creating objects using OOP concepts.	class Car { public: int speed; };
union	Stores different data types in the same memory location.	union Data { int x; float y; };
enum	Defines named constant values.	enum Days { Mon, Tue, Wed };

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

Ans. 1. Implicit Type Conversion (Automatic):

Implicit type conversion is done automatically by the compiler without any instruction from the programmer.
It usually happens when a smaller data type is converted into a larger or compatible data type.

Characteristics

- Done by the compiler
- No data loss in most cases
- Also called *type promotion*
- Makes code simpler and cleaner

Example:

```
#include <iostream>

using namespace std;

int main() {

  int a = 10;

  double b = a;   // int → double (implicit)

  cout << b;
```

```
   return 0;
}
```

2. Explicit Type Conversion (Type Casting):

Explicit type conversion is **manually forced by the programmer**.
It is used when converting between incompatible types or when you want control over the conversion.

Characteristics

- Done by the programmer
- May cause data loss
- Also called *type casting*
- More control, but requires care

Example:

```
#include <iostream>
using namespace std;
int main() {
double x = 10.75;
 int y = (int)x;   // explicit conversion
cout << y;
return 0;
}
```

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

Ans: **1. Arithmetic Operators**

Used to perform mathematical calculations.

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

Example:

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

cout << a + b;   // 13

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

2. Relational (Comparison) Operators

Used to compare two values and return true or false.

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example:

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

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

3. Logical Operators

Used to combine multiple conditions.

Operator	Meaning
&&	Logical AND
`	
!	Logical NOT

Example:

```
int a = 5, b = 10;

cout << (a < b && b > 0);   // true
```

4. Assignment Operators

Used to assign values to variables.

Operator	Example	Meaning
=	a = 10	Assign
+=	a += 5	a = a + 5
-=	a -= 2	a = a - 2
*=	a *= 3	a = a * 3
/=	a /= 2	a = a / 2

Example:

int a = 10;

a += 5;   // 15

5. Increment and Decrement Operators

Used to increase or decrease a value by 1.

Operator	**Type**
++	Increment
--	Decrement

Example:

int a = 5;

cout << ++a;   // 6 (pre-increment)

cout << a++;   // 6 (post-increment, then becomes 7)

6. Bitwise Operators

Used to perform operations at the bit level.

Operator	Meaning
&	Bitwise AND
`	`
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift

Example:

int a = 5, b = 3;

cout << (a & b);   // 1

7. Conditional (Ternary) Operator

Used to make decisions in a single line.

Example:

int a = 10, b = 20;

int max = (a > b) ? a : b;

8. Special Operators

Used for specific purposes.

Operator	Purpose	Example
sizeof	Size of data type	sizeof(int)
,	Comma operator	a = (b = 3, b + 2)
.	Member access	obj.x
->	Pointer member access	ptr->x
[]	Array subscript	arr[0]
&	Address of	&a
*	Dereference	*ptr

Example:

```
int a = 10;

int *p = &a;

cout << *p;   // 10
```

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

Ans: 1. Constants in C++

Purpose of Constants

Constants are variables whose value **cannot be changed once assigned**.

They are used to:

- Prevent accidental modification of important values

- Make programs easier to understand
- Improve code safety and reliability

Key Points

- Constants are fixed throughout the program
- const is preferred over #define in modern C++
- Constants improve program safety

2. Literals in C++

Purpose of Literals

Literals are **fixed values written directly in the program code**.

They represent constant data and **do not use variable names**.

Types of Literals with Examples:

Literal Type	**Example**
Integer	10, -45
Floating-point	3.14, 2.5f
Character	'A', '9'
String	"Hello"
Boolean	true, false
Null pointer	nullptr

Difference Between Constants and Literals

Feature	Constants	Literals
Named value	Yes	No
Can change	No	No
Stored in variable	Yes	Used directly
Readability	High	Limited

3. Control Flow Statements:

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

Ans: Conditional statements control the flow of execution by checking conditions and running specific code accordingly.

Common conditional statements in C++ are:

- if
- if-else
- else if
- switch

1. if-else Statement

Purpose:

The if-else statement is used when a decision is based on **conditions or ranges**.

Syntax:

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

How it Works

- Condition is checked
- If true, if block runs
- If false, else block runs

2. switch Statement

Purpose:

The switch statement is used when a decision depends on **one variable with multiple fixed values**.

Syntax:

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
```

```
        break;

    default:

        // code

}
```

Key Points

- expression must be an integer or character
- break stops fall-through
- default runs if no case matches

Difference Between if-else and switch:

Feature	**if-else**	**switch**
Condition type	Logical or relational	Fixed values only
Best used for	Ranges and complex conditions	Menu-driven programs
Data type	Any type	int or char
Performance	Slower for many conditions	Faster for many cases

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

Ans: Comparison Table

Feature	**for**	**while**	**do-while**
Condition check	Before loop	Before loop	After loop
Minimum executions	0	0	1
Best use case	Known iterations	Unknown iterations	Must run once
Syntax complexity	Compact	Simple	Slightly longer

Simple Explanation

- **for**: When you know how many times to loop
- **while**: When you don't know how many times
- **do-while**: When the loop must run at least once

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

Ans: **1. break Statement**

Purpose

The break statement is used to immediately terminate the loop when a specific condition becomes true.

Where it Works

- for loop
- while loop
- do-while loop
- switch statement

Example (break in a loop)

```
#include <iostream>
```

```
using namespace std;

int main() {

 for (int i = 1; i <= 10; i++) {

 if (i == 5) {

   break;   // loop stops here

  }

 cout << i << " ";

}

return 0;

}
```

Output

```
1 2 3 4
```

Explanation

- Loop starts normally
- When i == 5, break stops the loop completely
- Control moves outside the loop

2. continue Statement

Purpose

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

Example (continue in a loop)

```
#include <iostream>

using namespace std;
```

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

Output

```
1 2 4 5
```

Explanation

- When i == 3, code after continue is skipped
- Loop continues with the next value

Feature	**break**	**continue**
Effect	Terminates the loop	Skips current iteration
Loop execution	Stops completely	Continues to next cycle
Control flow	Goes outside loop	Goes to loop condition

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

Ans: A nested control structure occurs when:

- An if is inside another if
- A loop is inside another loop
- An if is inside a loop (or vice versa)

They are used when **one condition or operation depends on another**.

Example: Nested if Statement

```
#include <iostream>

using namespace std;

int main() {

  int age = 20;

  bool hasID = true;

  if (age >= 18) {

    if (hasID) {

      cout << "Allowed to enter";

    } else {

      cout << "ID required";

    }

  } else {

    cout << "Not allowed due to age";

  }

  return 0;

}
```

Explanation

- First if checks age
- Inside it, another if checks ID
- Decision depends on both conditions

Example: Nested Loop

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

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

Explanation

- Outer loop controls rows
- Inner loop runs completely for each outer loop iteration
- Commonly used in patterns and matrices

Where Nested Control Structures Are Used

- Menu-driven programs
- Pattern printing
- Matrix operations
- Searching and sorting algorithms

Key Points

- Nesting increases program control
- Inner structure executes fully for each outer execution
- Over-nesting should be avoided for readability

4. Functions and Scope :

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

Ans: A function:

- Has a **name**
- May take **parameters (inputs)**
- May return a **value**
- Executes only when it is **called**

Example (simple function)

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

Concept of Function Declaration, Definition, and Calling

In C++, a function works in **three main steps**:

1. Function Declaration (Function Prototype)

Purpose

A function declaration tells the **compiler in advance**:

- Function name
- Return type

- Parameters

This allows the function to be used before it is defined.

Syntax

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

Example

```
int add(int, int);
```

2. Function Definition:

Purpose

The function definition contains the **actual code** that performs the task.

Syntax

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

Example

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

3. Function Calling:

Purpose

A function is executed only when it is **called** from main() or another function.

Syntax

```
function_name(arguments);
```

Example

```
int result = add(10, 20);
```

Complete Program Example

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

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

int main() {
   int sum = add(10, 20);   // Function call
   cout << "Sum = " << sum;
   return 0;
}

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

Flow of Function Execution

1. Compiler reads the **declaration**
2. main() calls the function
3. Control transfers to the **function definition**
4. Function executes and returns value
5. Control returns to main()

Advantages of Functions

- Code reusability
- Better readability
- Easier debugging
- Modular programming

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

Ans: **Scope** is the region of a program in which a variable can be **used or accessed**.

In C++, variables mainly have two common scopes:

- **Local scope**
- **Global scope**

1. Local Scope

Definition

A **local variable** is declared **inside a function or a block** and can be accessed **only within that block**.

Characteristics:

- Exists only inside its function or block
- Created when the block starts
- Destroyed when the block ends
- Cannot be accessed outside the function

Example:

```
#include <iostream>

using namespace std;
```

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

int main() {
   show();
   // cout << x;  // error: x not accessible here
   return 0;
}
```

2. Global Scope

Definition

A **global variable** is declared **outside all functions** and can be accessed by **any function** in the program.

Characteristics

- Declared outside functions
- Accessible throughout the program
- Exists for the entire program execution
- May cause naming conflicts if overused

Example

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

int g = 50;   // global variable
```

```
void display() {
  cout << g;
}

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

Difference Between Local and Global Scope:

Feature	**Local Variable**	**Global Variable**
Declaration	Inside function/block	Outside all functions
Accessibility	Within block only	Entire program
Lifetime	During block execution	Entire program run
Memory	Stack	Data segment

Feature	Local Variable	Global Variable
Safety	More secure	Less secure

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

Ans: A function is called **recursive** if it calls itself.

Every recursive function must have:

1. **Base case** – a condition to stop recursion
2. **Recursive case** – the function calling itself

Without a base case, the program will go into **infinite recursion**.

Example: Factorial Using Recursion

Factorial Definition

factorial(n) = n × factorial(n−1)
factorial(0) = 1 (base case)

```
#include <iostream>

using namespace std;

int factorial(int n) {

  if (n == 0)        // base case

    return 1;

  else

    return n * factorial(n - 1);   // recursive call

}


int main() {
    int num = 5;

    cout << "Factorial = " << factorial(num);

    return 0;

}
```

Output

```
Factorial = 120
```

How Recursion Works :

For factorial(5):

factorial(5) → 5 * factorial(4)

factorial(4) → 4 * factorial(3)

factorial(3) → 3 * factorial(2)

factorial(2) → 2 * factorial(1)

factorial(1) → 1 * factorial(0)

factorial(0) → 1

Then results return back step by step.

Advantages of Recursion

- Code becomes shorter and cleaner
- Ideal for problems like factorial, Fibonacci, tree traversal
- Easy to understand for mathematical problems

Disadvantages of Recursion

- Uses more memory (stack)
- Slower than loops in many cases
- Risk of stack overflow if base case is wrong

Simple Explanation

- **Recursion**: A function solving a problem by calling itself
- **Base case**: Stop condition
- **Recursive case**: Function calling itself

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

Ans: a **function prototype** is a **declaration of a function** that tells the compiler **about the function before it is actually used** in the program.

It specifies:

- Function name
- Return type
- Number and types of parameters

but **does not contain the function body**.

What is a Function Prototype?

A function prototype informs the compiler:

"This function exists, and this is how it will be called."

Syntax

```
return_type function_name(parameter_types);
```

Example

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

Why Function Prototypes Are Used

Function prototypes are used to:

1. **Inform the compiler in advance**
   The compiler knows the function's return type and parameters before the function call.
2. **Allow function calls before definition**
   You can define functions **after main()**.
3. **Enable type checking**
   The compiler checks whether correct arguments are passed.
4. **Improve program structure**
   Helps in organizing large programs logically.

Example Without Prototype (Error)

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


int main() {
   cout << add(5, 3);   // error: add not declared
   return 0;
}


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

Example With Prototype (Correct)

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

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

int main() {
   cout << add(5, 3);   // valid
   return 0;
}

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

Key Points

- Prototype ends with a semicolon ;
- Parameter names are optional
- Required when function is defined after main()

5. Arrays and Strings:

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

Ans: **arrays** are data structures used to **store multiple values of the same data type** under a **single variable name**.
Each value in an array is stored in **contiguous memory locations** and accessed using an **index**.

An array allows you to store a collection of similar data items such as numbers, marks, or prices.

Syntax (1D Array)

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

Example

```
int marks[5] = {60, 70, 80, 90, 85};
```

- marks is the array name
- 5 is the size
- Index starts from 0

1. Single-Dimensional Array (1D Array)

Definition

A **single-dimensional array** stores data in a **linear form**, like a list.

Example

```
#include <iostream>

using namespace std;


int main() {

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


  for (int i = 0; i < 4; i++) {

    cout << a[i] << " ";

  }

  return 0;

}
```

Characteristics

- Uses one index
- Easy to declare and access
- Suitable for lists and sequences

2. Multi-Dimensional Array

Definition

A **multi-dimensional array** stores data in **rows and columns** (matrix form). The most common type is a **two-dimensional array**.

Syntax (2D Array)

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

Example

```
#include <iostream>

using namespace std;


int main() {

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


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

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

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

    }

    cout << endl;

  }

  return 0;

}
```

Characteristics

- Uses multiple indices
- Represents tables, matrices, grids
- Requires nested loops for traversal

Difference Between Single-Dimensional and Multi-Dimensional Arrays

Feature	**Single-Dimensional Array**	**Multi-Dimensional Array**
Structure	**Linear (list)**	**Tabular (rows & columns)**
Indices	**One**	**Two or more**
Syntax	**a[5]**	**a[3][4]**
Complexity	**Simple**	**More complex**
Usage	**Lists, scores**	**Matrices, tables**

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

Ans: string handling means storing, manipulating, and working with text data.
C++ supports string handling in two ways:

1. C-style strings (character arrays)
2. C++ string class (recommended and modern)

1. C-Style Strings (Character Arrays)

C-style strings are arrays of characters that end with a null character '\0'.

Declaration

```
char name[20] = "Mithlesh";
```

Common String Handling Functions

(Available in <cstring>)

Function	**Purpose**
strlen()	Finds length
strcpy()	Copies string
strcat()	Concatenates strings
strcmp()	Compares strings

Example

```
#include <iostream>

#include <cstring>

using namespace std;


int main() {

  char s1[20] = "Hello";

  char s2[20] = "World";


  cout << strlen(s1) << endl;   // Length

  strcat(s1, s2);            // Concatenate
```

```
    cout << s1 << endl;


    return 0;
}
```

Limitations

- Fixed size
- Risk of buffer overflow
- Manual handling required

2. C++ string Class (Modern Method)

The string class is part of the **Standard Template Library (STL)** and is much safer and easier to use.

Declaration

```
string name = "Mithlesh";
```

Common String Operations:

Function	**Purpose**
length() / size()	**Length of string**
append()	**Add string**
+	**Concatenation**
compare()	**Compare strings**
substr()	**Extract substring**
find()	**Find character or word**

Example

```
#include <iostream>
```

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

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

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

   cout << s3.length() << endl;   // Length
   return 0;
}
```

Input and Output with Strings

Using cin

```
string name;
cin >> name;   // reads single word
```

Using getline

```
string fullName;
getline(cin, fullName);   // reads full line with spaces
```

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

Ans: array initialization means assigning values to array elements at the time of declaration.
Arrays can be initialized in different ways depending on whether they are one-dimensional (1D) or two-dimensional (2D).

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

(a) Full Initialization

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

(b) Partial Initialization

Remaining elements are automatically set to 0.

```
int b[5] = {1, 2};
```

(c) Without Specifying Size

Compiler determines the size automatically.

```
int c[] = {5, 10, 15, 20};
```

(d) Character Array Initialization

```
char name[] = "Mithlesh";
```

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

(a) Row-wise Initialization

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

(b) Single-Line Initialization

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

(c) Partial Initialization

Uninitialized elements become 0.

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

Example Program: 1D Array Initialization

```
#include <iostream>

using namespace std;


int main() {

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


   for (int i = 0; i < 4; i++) {

      cout << arr[i] << " ";

   }

   return 0;

}
```

Example Program: 2D Array Initialization

```
#include <iostream>

using namespace std;


int main() {

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


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

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

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

      }

      cout << endl;

   }

   return 0;
}
```

Key Points

- Indexing starts from 0
- Partial initialization fills remaining elements with 0
- 2D arrays require row and column sizes (except first dimension sometimes

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

Ans: string operations and functions are used to store, manipulate, compare, and process text data.
C++ supports string handling using the string class, which is part of the Standard Library and is the preferred approach in modern C++.

String Class in C++

To use strings, include:

```
#include <string>
```

Declaration

```
string s = "Hello";
```

Common String Operations and Functions

1. Length of String

Returns the number of characters in a string.

```
string s = "Hello";

cout << s.length();   // 5
```

2. String Concatenation

Joins two strings.

```
string s1 = "Hello";
string s2 = " World";
string s3 = s1 + s2;
```

3. Append String

Adds one string to the end of another.

```
s1.append(s2);
```

4. String Comparison

Compares two strings.

```
if (s1 == s2) {
   cout << "Strings are equal";
}
Using compare():
s1.compare(s2);   // returns 0 if equal
```

5. Substring

Extracts part of a string.

```
string s = "Programming";
cout << s.substr(0, 7);   // Program
```

6. Find Character or Word

Searches for a character or substring.

```
string s = "Hello World";
cout << s.find("World");   // position
```

7. Access Characters

Access individual characters using index.

```
cout << s[0];   // H
```

8. Input Strings

Using cin

```
string name;
cin >> name;   // single word
```

Using getline

```
string fullName;
getline(cin, fullName);   // full sentence
```

C-Style String Functions (Optional / Old Method)

Using <cstring>:

Function	**Purpose**
strlen()	Find length
strcpy()	Copy
strcat()	Concatenate
strcmp()	Compare

Example

```
char a[20] = "Hello";
char b[20] = "World";
strcat(a, b);
```

Difference: string vs C-Style Strings

Feature	**C++ string**	**C-Style String**
Memory	Dynamic	Fixed
Safety	High	Low
Ease	Easy	Complex
Recommended	Yes	No

6. Introduction to Object-Oriented Programming:

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

Ans: Object-Oriented Programming (OOP) is a programming paradigm that organizes a program around objects rather than functions and logic.
It helps in building modular, reusable, secure, and easy-to-maintain software.

The key concepts of OOP in C++ are explained below.

1. Class

Explanation

A class is a blueprint or template used to create objects. It defines data members (variables) and member functions (methods) together.

Example

```
class Student {
public:
  int roll;
  void show() {
    cout << roll;
  }
};
```

2. Object

Explanation

An object is an instance of a class. It represents a real-world entity and can access class members.

Example

```
Student s1;
s1.roll = 10;
s1.show();
```

3. Encapsulation

Explanation

Encapsulation means wrapping data and functions into a single unit (class) and controlling access using access specifiers.

Purpose

- Data protection
- Better control over data

Example

```
class Account {
private:
  int balance;
public:
  void setBalance(int b) {
    balance = b;
  }
};
```

4. Abstraction

Explanation

Abstraction means hiding internal implementation details and showing only what is necessary.

Purpose

- Reduce complexity
- Improve usability

Example

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

(User doesn’t need to know how the engine works.)

5. Inheritance

Explanation

Inheritance allows one class to acquire properties and behaviors of another class.

```
}
```

Purpose

- Code reusability
- Hierarchical classification

Example

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

6. Polymorphism

Explanation

Polymorphism means one function name, multiple forms.It allows the same function to behave differently in different situations.

Types

- Compile-time (function overloading, operator overloading)
- Runtime (function overriding)

Example

```
class A {
public:
  void show() {
    cout << "Class A";
  }
};

class B : public A {
public:
  void show() {
    cout << "Class B";
  }
};
```

7. Dynamic Binding

Explanation

Dynamic binding means the function call is resolved at runtime instead of compile time.

Example

```
A* obj;
B b;
obj = &b;
obj->show();   // calls B's show()
```

Summary:

Concept	**Meaning**
Class	**Blueprint of objects**
Object	**Instance of a class**
Encapsulation	**Data hiding**
Abstraction	**Hide implementation**

Concept	**Meaning**
Inheritance	**Reuse existing class**
Polymorphism	**One name, many forms**
Dynamic Binding	**Runtime function call**

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

Ans: classes and objects are the core building blocks of Object-Oriented Programming (OOP).
They help represent real-world entities in program form by combining data and behavior.

Class:

A class is a blueprint or template used to create objects.
It defines:

- Data members (variables)
- Member functions (functions that operate on data)

Syntax

```
class ClassName {
   // data members
   // member functions
};
```

Object:

An object is an instance of a class.It represents a real entity and can access the class members.

Syntax

```
ClassName objectName;
```

Example: Class and Object in C++

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

class Student {
public:
   int roll;
   string name;

   void display() {
      cout << "Roll: " << roll << endl;
      cout << "Name: " << name << endl;
   }
};
int main() {
   Student s1;      // object creation
   s1.roll = 101;
   s1.name = "Amit";

   s1.display();    // object calling member function
   return 0;
}
```

Explanation of the Example

- Student is a class
- roll and name are data members
- display() is a member function
- s1 is an object of the class
- Object accesses class members using the dot (.) operator

Difference Between Class and Object

Feature	**Class**	**Object**
Meaning	Blueprint	Instance
Memory	No memory allocated	Memory allocated
Creation	Logical	Physical
Example	Student	s1

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

Ans: inheritance is an Object-Oriented Programming (OOP) concept that allows one class to acquire the properties and behaviors of another class.
It helps in code reusability, hierarchical classification, and easy maintenance of programs.

Inheritance enables a derived (child) class to use the members of a base (parent) class.

- Base class → Class being inherited from
- Derived class → Class that inherits

Syntax of Inheritance

```
class DerivedClass : access_specifier BaseClass {
  // additional members
};
```

Example: Inheritance in C++

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

// Base class
class Animal {
public:
  void eat() {
    cout << "Animal is eating" << endl;
  }
};

// Derived class
class Dog : public Animal {
public:
  void bark() {
    cout << "Dog is barking" << endl;
  }
};

int main() {
  Dog d;
  d.eat();   // inherited from Animal
```

```
    d.bark();  // own function
    return 0;
}
```

Explanation

- Animal is the base class
- Dog is the derived class
- Dog inherits the eat() function from Animal
- The derived class can use both base class members and its own members

Advantages of Inheritance

- Code reusability
- Reduces duplication
- Supports hierarchical structure
- Easier maintenance

Simple Real-Life Example

- Base class: Vehicle
- Derived class: Car, Bike

Car and Bike inherit common properties like speed and fuel type from Vehicle.

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

Ans: encapsulation is a core Object-Oriented Programming (OOP) concept that means binding data and the functions that operate on that data into a single unit, called a class, and restricting direct access to some of the object's data.

In simple terms:
data is hidden and accessed in a controlled way.

Encapsulation:

- Wraps data members and member functions together in a class
- Protects data from unauthorized access
- Allows controlled access using methods

Encapsulation is achieved mainly by:

1. Using classes
2. Using access specifiers (private, public, protected)
3. Providing public methods (getters/setters) to access private data

Access Specifiers and Their Role

Access Specifier	**Purpose**
private	Data accessible only within the class
public	Data accessible from anywhere
protected	Accessible in derived classes

Example: Encapsulation in a Class

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

class BankAccount {
private:
   int balance;   // hidden data
public:
  void setBalance(int b) {   // setter
      balance = b;
  }

  int getBalance() {       // getter
      return balance;
  }
};

int main() {
  BankAccount acc;
  acc.setBalance(5000);     // controlled access
  cout << acc.getBalance();  // read data safely
  return 0;
}
```

Explanation of Example

- balance is private, so it cannot be accessed directly
- Public methods setBalance() and getBalance() control access
- Data is protected from misuse

Advantages of Encapsulation

- Improves data security
- Makes code modular
- Easier maintenance

- Prevents accidental data modification

Simple Real-Life Example

- Capsule: medicine inside is protected
- ATM: you can withdraw money, but can’t directly access bank data