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. Focus

- **Procedural Programming (POP):**
  Focus is on functions — step-by-step instructions to perform tasks.
  Like writing a recipe.
- **Object-Oriented Programming (OOP):**
  Focus is on objects — which combine both data and the functions that work on that data.
  Like making a machine with all parts working together.

2. Data Handling

- **POP:**
  Data is global and shared by all functions.
  Less secure — anyone can access/change it.
- **OOP:**
  Data is kept private inside objects.
  More secure — only object's own functions can access it (using encapsulation).

3. Code Structure

- **POP:**
  Uses **functions/procedures** only.
  Difficult to manage in large programs.
- **OOP:**
  Uses **classes and objects**.
  Easier to organize and manage large projects.

4. Reusability

- **POP:**
  Code **cannot be reused easily**. You need to rewrite the same logic again.
- **OOP:**
  You can **reuse code** using **inheritance** — one class can use the features of another.

5. Real-World Representation

- **POP:**
  Hard to relate to real-world things.
- **OOP:**
  Objects represent real-world items (like a car, student, bank account).

6. Examples of Languages

- **POP Languages:**
  C, BASIC, Pascal
- **OOP Languages:**
  C++, Java, Python, C#, etc**.**

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

List:

Here is a list and explanation of the main advantages of Object-Oriented Programming (OOP) over Procedural Oriented Programming (POP):

(1) Class
(2) Object
(3) Encapsulation
(4) Abstraction
(5) Inheritance
(6) Polymorphism

(1) Object

- ✓ Any Entity which has own state and behaviour that is called object.
  Ex: pen, paper, chair etc...

- ✓ A class is a blueprint or template that defines data and functions together.

(2) Class

- ✓ collection of objects or it's had a data members and method's collection.
  ex: human body

- ✓ An object is an instance of a class — it is the actual entity created using the class.

(3) Encapsulation

- ✓ Wrapping up of data or binding of data is called Encapsulation.
  Ex: capsule

(4) Abstraction

- ✓ Hiding internal details and showing functionalities is called Abstraction.
  Ex: login page

(5) Inheritance

- ✓ When One object acquires all the properties and behaviour of parent class is called Inheritance.
  ex: father-son

(6) Polymorphism

- ✓ Many ways to perform anything is called Polymorphism.
  Ex: road ways

  1) Method Overloading
  2) Method Overriding

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

- ✓ **Install a C++ Compiler**

  You need a **compiler** to convert your C++ code into executable programs.

- **For Windows:**
  - Install **MinGW (Minimalist GNU for Windows)**
    https://sourceforge.net/projects/mingw/
  - OR install **Code: Blocks** with built-in GCC compiler.

- ✓ **Install an IDE (Optional but Recommended)**

An IDE (Integrated Development Environment) makes coding easier by providing tools like syntax highlighting, autocomplete, etc.

Popular IDEs for C++:

IDE	Platform
Code::Blocks	Windows/Linux
Visual Studio	Windows
Dev C++	Windows
VS Code	All OS

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

1. cin (Standard Input)

- Used to take input from the keyboard.
- It is an object of the istream class.

Example:

```
#include <iostream>
using namespace std;
int main()
{
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "Your age is " << age << endl;
    return 0;
}
```

2. cout (Standard Output)

- Used to display output on the screen.
- It is an object of the ostream class.

  Example:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!" << endl;  // prints message to screen
    return 0;
}
```

2. Variables, Data Types, and Operators

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

1. Built-in Data Types

These are the basic data types provided by the language.

Type	Description	Example
int	Stores integers	int age = 21;
float	Stores decimal numbers (single precision)	float price = 99.99;
double	Stores decimal numbers (double precision)	double pi = 3.14159;
char	Stores a single character	char grade = 'A';
bool	Stores true or false	bool passed = true;
void	Represents no value/empty	Used in functions: void display();

2. Derived Data Types

These are based on the built-in types.

Type	Description	Example
array	Collection of elements of same type	int arr[5] = {1, 2, 3, 4, 5};
pointer	Stores the address of another variable	int* ptr = &x;
function	Group of statements	int add(int a, int b)
reference	Alternate name for a variable	int& ref = x;

3. User-Defined Data Types

These are created by the user to model real-world entities.

Type	Description	Example
struct	Group of related variables	struct Student { int id; string name; };
union	Like struct but shares memory	union Data { int i; float f; };
enum	Set of named integer constants	enum Color { RED, GREEN, BLUE };
class	Blueprint for objects (OOP)	class Car { public: void drive(); };

Example Program

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

int main() {
    int age = 18;              // integer
    float height = 5.9;        // floating point
    char grade = 'A';          // character
    bool passed = true;        // boolean
    double pi = 3.1415926535;  // double
    string name = "Dhiraj";    // string (in C++, from <string>)

    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Height: " << height << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << passed << endl;
```

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


    return 0;
}
```

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

1. Implicit Type Conversion (Automatic Conversion)

- ✓ The compiler automatically converts one data type to another.
- ✓ When you assign a value of one type to a variable of another type without using a cast.

Example:

```
#include <iostream>

using namespace std;


int main() {
    int x = 10;
    float y = x;  // int automatically converted to float


    cout << "x: " << x << endl;
    cout << "y: " << y << endl;


    return 0;
}
```

Here, int → float conversion is done automatically (implicitly).

2. Explicit Type Conversion (Type Casting)

- ✓ The programmer manually tells the compiler to convert one type to another.
- ✓ When you use a cast like (type) before the variable/value.

Example:

```
#include <iostream>

using namespace std;


int main() {

   float pi = 3.14;

   int intPi = (int)pi;  // manually converting float to int


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

   cout << "intPi: " << intPi << endl;


   return 0;

}
```

Here, float → int is done manually using (int).

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

Types of Operators in C

(1) Arithmetic Operator

- These operators are used to perform **basic mathematical calculations** like addition, subtraction, multiplication, division, and finding the remainder (modulus).

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

(2) Relational Operators

- Relational operators **compare two values** and decide the relationship between them. The result of this comparison is either **true (1)** or **false (0).**
- These operators are commonly used in decision-making and loops.

Operator	Meaning	Example
==	Double Equal to (Comparison the value)	a == b
!=	Not equal to	a! = b
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

(3) Logical Operator

- Logical operators **combine multiple conditions** or expressions and return true or false based on their logical relation.

- && means logical AND — both conditions are true.
- || means logical OR — at least one condition is true.
- ! means logical NOT — These conditions are not true.

Operator	Meaning	Example
&&	Logical AND	(a > 0) && (b > 0)
||	Logical OR	(a > 0) || (b > 0)
!	Logical NOT	!(a > 0)

(4) Assignment Operator (shorthand operators)

- Assignment operators are used to **assign values** to variables.
- The basic assignment operator is =, which assigns the value on the right to the variable on the left.
- There are also called **shorthand operators** which perform an operation and assign the result in one step.
- shorthand assignment operators work as **value replace** the variables.

Operator	Meaning	Example
=	Assign	a = 5
+=	Add and assign	a += 3 (same as a = a + 3)
-=	Subtract and assign	a -= 2 (same as a = a - 2)
*=	Multiply and assign	a *= 4
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 3

(5) Increment and Decrement Operators

- These operators are special types of unary operators used to increase or decrease the value.
- Increment operator ++ increases the value by 1, and decrement operator -- decreases it by 1.
- They can be used before or after the variable (prefix or postfix).

Operator	Meaning	Example
++	Increment by 1	a++ or ++a
--	Decrement by 1	a-- or --a

(6) Bitwise Operators

- Bitwise operators work on the binary (bit-level) representation of integers.
- They perform operations like AND, OR, XOR, NOT, and shifting bits left or right.

Operator	Symbol	Description
AND	&	Bits that are 1 in both
OR	|	Bits that are 1 in either
XOR	^	Bits that are 1 in one, but not both
NOT	~	Inverts bits
Left Shift	<<	Shifts bits left
Right Shift	>>	Shifts bits right

(7) Conditional (Ternary) Operator

- ✓ The conditional operator, also called the ternary operator, is a shorthand way of writing an if-else statement in a single line.

Syntax:

condition ? expression1 : expression2;

- If the condition is true, expression1 is executed.
- If the condition is false, expression2 is executed.

Example:

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

int main()
{
        int a = 10, b = 20;
        int max;

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

        cout("Maximum is: %d\n", max);

        return 0;
}
```

Output:

Maximum is: 20

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

Constants in C++

- ✓ Constants are values that do not change during the execution of a program.

Purpose:

- To protect important values from being changed.
- To make the code safe and easy to understand.

Literals in C++

- ✓ Literals are fixed values written directly in the code.

Example of literals:

```
int a = 10;        // 10 is an integer literal
float pi = 3.14;    // 3.14 is a float literal
char grade = 'A';   // 'A' is a character literal
bool passed = true; // true is a boolean literal
string name = "Ram"; // "Ram" is a string literal
```

Example :

```
#include <iostream>
using namespace std;
const float PI = 3.14;  // constant
int main()
 {
   int radius = 5;
   float area = PI * radius * radius;  // 3.14 is a literal
   cout << "Area: " << area << endl;
   return 0;
}
```

3. Control Flow Statements

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

Conditional Statements in C++

Conditional statements allow you to **make decisions** in your program. Based on a condition (true or false), different blocks of code can be executed.

1. if Statement

- ✓ Used to execute code **only if a condition is true**.

Example:

```
int age = 18;


if (age >= 18) {
   cout << "You are eligible to vote.";
}
```

2. if-else Statement

- ✓ Runs one block **if condition is true**, another **if false**.

Example:

```
int age = 16;
if (age >= 18) {
   cout << "You can vote.";
} else {
   cout << "You are too young to vote.";
}
```

3. if-else if Ladder

- ✓ Used when there are **multiple conditions**.

Example:

```
int marks = 75;
if (marks >= 90) {
   cout << "Grade A";
} else if (marks >= 70) {
   cout << "Grade B";
} else {
   cout << "Grade C";
}
```

4. switch Statement

- ✓ Best for **checking a variable against many fixed values** (like options).

Example:

```
int day = 3;
switch (day)
{
   case 1: cout << "Monday"; break;
   case 2: cout << "Tuesday"; break;
   case 3: cout << "Wednesday"; break;
   case 4: cout << "Thursday"; break;
   default: cout << "Invalid day";
}
```

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

In C++, for, while, and do-while are looping control structures used to repeat a block of code multiple times.

1. for Loop

- ✓ Used when the number of iterations is known.
- ✓ Initialization, condition, and update are all in one line.

Example:

```
for (int i = 0; i < 5; i++) {
   cout << i << " ";
}
```

Flow:

1. Initialization → 2. Condition check → 3. Loop body → 4. Increment → Repeat

Output: 0 1 2 3 4

2. while Loop

- ✓ Used when the number of iterations is unknown, and condition is checked before each iteration.

Example:

```
int i = 0;
while (i < 5) {
   cout << i << " ";
   i++;
}
```

Flow:

1. Condition check → 2. Loop body → 3. Increment → Repeat

Output: 0 1 2 3 4

3. do-while Loop

- ✓ Similar to while, but the loop body runs at least once, because the condition is checked after the loop body.

Example:

```
int i = 0;
do {
   cout << i << " ";
   i++;
} while (i < 5);
```

Flow:

1. Loop body → 2. Condition check → 3. Repeat if true

Output: 0 1 2 3 4

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

In C++, break and continue are loop control statements used to alter the flow of loops (for, while, and do-while).

1.Break Statement

Purpose:

Used to exit the loop immediately, even if the loop condition is still true.

Use Case:

When a certain condition is met and there's no need to continue further iterations.

Example:

```
#include <iostream>

using namespace std;


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

Output: 1 2 3 4

2. continue Statement

Purpose:

Skips the current iteration and moves to the next iteration of the loop.

Use Case:

When you want to skip certain values or situations but keep looping.

Example:

```
#include <iostream>

using namespace std;
```

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

Output: 1 2 4 5

4. Explain nested control structures with an example.

Nested Control Structures

A nested control structure is when one control structure is placed inside another. These can include:

- if statements inside other if, else, or loops
- for loops inside if statements or other loops
- switch statements inside if or loops, and vice versa

Why Use Nested Control Structures?

Nested structures are used when:

- You need to make decisions within decisions
- You need to perform repeated tasks inside other repeated tasks
- You need to filter data or process combinations of values

Types of Nested Structures

1. Nested if Statements

- ✓ Used when a decision depends on another condition.

Syntax:

```
if (condition1) {
   if (condition2) {
      // Code runs only if both condition1 and condition2 are true
   }
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
   int age = 20, marks = 75;
   if (age >= 18) {
      if (marks >= 70)
         cout << "Eligible for admission." << endl;
      else
         cout << "Need higher marks." << endl;
   } else {
      cout << "Underage." << endl;
   }

   return 0;
}
```

2. Nested Loops

- ✓ Used when an outer loop controls the number of inner loop executions.

- ✓ Very useful for:

- Multiplication tables
- Matrix operations
- Pattern printing

Example:

```
for (int i = 1; i <= 3; i++) {
   for (int j = 1; j <= 2; j++) {
      // Runs 3 * 2 = 6 times
   }
}
```

3. Loop inside if or if inside loop

- ✓ You can place a loop inside a condition, or check a condition during each loop iteration.

Example:

```
for (int i = 1; i <= 5; i++) {
   if (i % 2 == 0) {
      cout << i << " is even" << endl;
   }
}
```

4. Nested switch Statements

- ✓ Sometimes one switch statement appears inside another case of a switch.

Example:

```
#include <iostream>
using namespace std;
int main()
 {
```

```
    int userType = 1;
    char choice = 'b';

    if (userType == 1) { // Admin
        switch (choice) {
            case 'a':
                cout << "Add User" << endl;
                break;
            case 'b':
                cout << "Delete User" << endl;
                break;
            default:
                cout << "Invalid Option" << endl;
        }
    }

    return 0;
}
```

4. Functions and Scope

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

function in C++

- ✓ A function in C++ is a block of code that performs a specific task.
- ✓ Functions help in breaking a large program into smaller, reusable, and manageable pieces.

Why Use Functions?

- Avoid repetition of code
- Modularity: Break complex logic into smaller parts
- Reusability: Use the same function in different places
- Improves readability and debugging

Parts of a Function

1. Function Declaration (Prototype)

- ✓ It tells the compiler about the function's name, return type, and parameters, before it is used.

```
int add(int, int);  // Declaration
```

2. Function Definition

- ✓ It contains the actual code (body) that gets executed when the function is called.

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

3. Function Call

- ✓ This is how you use or execute the function in your main program.

```
int result = add(5, 3);   // Calling the function
```

Example:

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

// 1. Function Declaration
int add(int, int);

int main() {
   // 3. Function Call
   int sum = add(10, 20);
   cout << "Sum = " << sum << endl;
   return 0;
}

// 2. Function Definition
int add(int a, int b) {
   return a + b;
}
```

Output: Sum = 30

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

Scope of Variables in C++

Scope refers to the region of the program where a variable is accessible or visible.

In C++, variable scope determines:

- Where a variable can be used
- How long the variable exists in memory

Types of Variable Scope:

1. Local Scope

- Variable declared inside a function, loop, or block {}.
- Accessible only within that block.
- Memory is released after the block ends.

Example:

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

Variable x is local to the function show().

2. Global Scope

- Variable declared outside all functions.
- Accessible by any function in the same file (unless shadowed).
- Exists for the entire program life.

Example:

```
int x = 100;  // global variable

void display()

{

   cout << x;  // accessible here

}
```

Variable x is global and can be used anywhere.

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

- ✓ Recursion is a programming technique where a function calls itself to solve a problem.
- ✓ A recursive function breaks a large problem into smaller subproblems, solving each one by calling itself repeatedly.

Syntax of Recursive Function

```
returnType functionName(parameters) {

   if (base_condition)

      return value; // base case to stop recursion

   else

      return functionName(smaller_problem); // recursive call

}
```

Key Terms :

Term	Meaning
Base Case	The condition where recursion stops
Recursive Case	The condition where the function calls itself

Example: Factorial using Recursion

Factorial of n: n! = n × (n-1) × (n-2) × ... × 1
5! = 5 × 4 × 3 × 2 × 1 = 120

```
#include <iostream>
using namespace std;
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 << " is: " << factorial(num);
   return 0;
}
```

Output:

Factorial of 5 is: 120

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

- ✓ A function prototype in C++ is a declaration of a function that tells the compiler:
    - The function's name
    - Its return type
    - The number and types of parameters

Syntax of a Function Prototype:

return_type function_name(parameter_type1, parameter_type2, ...);

The parameter names are optional in the prototype.

Example:

```
#include <iostream>
using namespace std;
//  Function Prototype
int add(int, int);
int main() {
   int result = add(10, 5);  // Function call
   cout << "Sum = " << result;
   return 0;
}

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

5. Arrays and Strings

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

- ✓ An array in C++ is a collection of variables of the same data type, stored in a contiguous block of memory, and accessed using an index.
- ✓ It allows you to store multiple values (like a list) using one variable name.

Syntax of Array Declaration:

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

Example:

```
int numbers[5];  // an array of 5 integers
```

Types of Arrays:

1. Single-Dimensional Array (1D Array)

- ✓ It is a linear array—a simple list of elements.

Declaration:

```
int arr[5];
```

Example:

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

cout << marks[1];  // Output: 90
```

2. Multi-Dimensional Array

- ✓ Used for storing tables or grids, where elements are organized in rows and columns.

Declaration:

```
int matrix[2][3];  // 2 rows, 3 columns
```

Example:

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

```
cout << matrix[1][2];  // Output: 6
```

Difference Between Single-Dimensional and Multi-Dimensional Arrays

Feature	1D Array	2D / Multi-Dimensional Array
Structure	Linear (like a list)	Table (rows and columns)
Declaration	int arr[5];	int arr[3][4];
Access Elements	arr[index]	arr[row][col]
Use Case	Marks of 1 student, list of IDs	Matrix, timetable, image pixels

Real-Life Analogy:

- 1D Array → A single row of lockers
- 2D Array → A table with rows and columns

Example Program Using Both:

```
#include <iostream>
using namespace std;
int main()
 {
   // 1D Array
```

```
    int scores[3] = {10, 20, 30};
    cout << "1D Array: " << scores[1] << endl;

    // 2D Array
    int table[2][2] = {{1, 2}, {3, 4}};
    cout << "2D Array: " << table[1][0] << endl;

    return 0;
}
```

🔹 Output:

```
1D Array: 20
2D Array: 3
```

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

C++ string Class

- Easier and safer to use.
- Part of C++ Standard Library.
- Header file needed: <string>

Example:

```
#include <iostream>
#include <string>     // Required for string
using namespace std;

int main() {
    string firstName = "John";
    string lastName = "Doe";
    string fullName = firstName + " " + lastName;
```

```
    cout << "Full Name: " << fullName << endl;
    cout << "Length: " << fullName.length() << endl;

    return 0;
}
```

Common Methods in string:

Method	Description
length() or size()	Returns number of characters
append()	Appends another string
substr(pos, len)	Returns substring
find("str")	Returns position of substring
compare("str")	Compares two strings
empty()	Checks if string is empty
erase(pos, len)	Erases characters from the string

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

Array

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

C++ supports arrays of any data type — int, float, char, etc.

1. One-Dimensional (1D) Arrays

Declaration:

```
type arrayName[size];
```

Initialization Methods:

Method 1: With size

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

Method 2: Without size (size auto-calculated)

```
int arr[] = {10, 20, 30};  // Size = 3
```

Method 3: Partial initialization

```
int arr[5] = {1, 2};  // Remaining will be 0
```

Example: 1D Array Program

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

int main() {
   int marks[5] = {78, 85, 90, 67, 88};

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

2. Two-Dimensional (2D) Arrays

Declaration**:**

```
type arrayName[rows][columns];
```

Initialization:

Method 1: Full initialization

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

Method 2: Single-line

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

Example: 2D Array Program

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

int main() {
   int matrix[2][3] = {
      {1, 2, 3},
      {4, 5, 6}
   };

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

```
    return 0;
}
```

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

- ✓ A string is a sequence of characters.
- ✓ C++ provides the string class in the Standard Template Library (STL) to simplify string handling.

To use strings:

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

Declaration and Initialization

Syntax:

```
string str;                 // Empty string
string str = "Hello";       // Initialization
string str("World");        // Constructor-style
```

Basic String Operations

Operation	Description	Example
Assignment	Assign one string to another	str2 = str1;
Concatenation	Combine two strings	str3 = str1 + str2;
Access Characters	Access individual characters using index	str[0], str.at(2)
Input	Read string from user	getline(cin, str);
Output	Display string to screen	cout << str;

Example:

```
string name = "John";
string greet = "Hello, " + name;
cout << greet;
```

Output: Hello, John

Commonly Used String Functions

1. length() / size()

- Returns the number of characters in the string.

```
string str = "Programming";
cout << str.length();  // Output: 11
```

2. append()

- Adds another string to the end of the current string.

```
string a = "Hello";
a.append(" World");
cout << a;  // Output: Hello World
```

3. compare()

- Compares two strings.

Returns:

- 0 → strings are equal
- < 0 → first < second
- > 0 → first > second

```
string   a = "apple", b = "banana";
```

```
int result = a.compare(b);
```

4. substr(start, length)

- Extracts a substring from the string.

```
string s = "Artificial";
cout << s.substr(0, 3);  // Output: Art
```

5. replace(pos, len, str)

- Replaces part of the string starting at pos with new string str.

```
string s = "Hello World";
s.replace(6, 5, "C++");
cout << s;  // Output: Hello C++
```

6. insert(pos, str)

- Inserts string str at index pos.

```
string s = "Good";
s.insert(4, " Morning");
cout << s;  // Output: Good Morning
```

7. getline()

- Used to read a full line (including spaces) from user input.

```
string sentence;
getline(cin, sentence);
```

Example Program: All Key Functions

```
#include <iostream>
#include <string>
```

```
using namespace std;

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

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

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

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

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

    s3.insert(5, ",");           // Insert comma
    cout << "After insert: " << s3 << endl;

    s3.erase(5, 1);              // Erase comma
    cout << "After erase: " << s3 << endl;

    cout << "Find C++: " << s3.find("C++") << endl;

    return 0;
}
```

6. Introduction to Object-Oriented Programming

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

- the key concepts of OOP
  (1) Class
  (2) Object
  (3) Encapsulation
  (4) Abstraction
  (5) Inheritance
  (6) Polymorphism

Object

- ✓ Any Entity which has own state and behaviour that is called object.
  Ex: pen, paper, chair etc...

- ✓ A class is a blueprint or template that defines data and functions together.

Class

- ✓ collection of objects or it's had a data members and method's collection.
  ex: human body

- ✓ An object is an instance of a class — it is the actual entity created using the class.

Encapsulation

- ✓ Wrapping up of data or binding of data is called Encapsulation.
  Ex: capsule

Abstraction

- ✓ Hiding internal details and showing functionalities is called Abstraction.
  Ex: login page

Inheritance

- ✓ When One object acquires all the properties and behaviour of parent class is called Inheritance.
  ex: father-son

Polymorphism

- ✓ Many ways to perform anything is called Polymorphism.
  Ex: road ways

  1) Method Overloading
  2) Method Overriding

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

Classes and Objects in C++

CLASS

A class is a user-defined data type that acts as a blueprint to create objects.
It can contain:

- Data Members (variables)
- Member Functions (methods)

Syntax:

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

OBJECT

An object is an instance of a class.
It holds actual data and is used to access class members.

Syntax:

```
ClassName objectName;
```

Example:

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

class Car {
public:
    string brand;
    int speed;

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

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

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

Inheritance:

Inheritance allows a new class (derived) to use the features (properties and methods) of an existing class (base).

- Promotes code reusability.
- Reduces redundancy.

Types of Inheritance in C++:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Example: Single Inheritance

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

class Animal {
public:
   void eat() {
      cout << "This animal eats food\n";
   }
};

class Dog : public Animal {  // Dog inherits Animal
public:
   void bark() {
      cout << "Dog barks\n";
   }
};
```

```
int main() {
   Dog d;
   d.eat();   // Inherited from Animal
   d.bark();  // Dog's own function
}
```

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

Encapsulation:

Encapsulation means binding data and functions together into a single unit (class) and restricting access to the internal data.

- Achieved using access specifiers: private, public, and protected.

Access Specifiers:

Specifier	Accessible In
private	Only within the class
public	Everywhere
protected	Class + Derived classes only

Example:

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

class Account {
private:
   int balance;

public:
```

```
    void setBalance(int b) {
        if (b >= 0)
            balance = b;
    }

    int getBalance() {
        return balance;
    }
};

int main() {
    Account a;
    a.setBalance(5000);          // Securely set value
    cout << a.getBalance();      // Output: 5000
}
```