Module-3

Introduction To Oops Programming

1. Introduction to C++

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

Point	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Program is written using functions.	Program is written using objects and classes.
Focus	Focus is on actions (functions).	Focus is on data (objects).
Data Handling	Data is open and passed between functions.	Data is inside objects and protected.
Security	Less secure, anyone can access data.	More secure, access is controlled.
Reusability	Code reuse is difficult.	Code can be reused easily using inheritance.
Real-World Use	Hard to model real-world problems.	Easy to model real-world problems.
Examples	C, BASIC, Pascal	C++, Java, Python

2. List and Explain the Main Advantages of Object-Oriented Programming (OOP) Over Procedural-Oriented Programming (POP

(1) Object

- An **object** is something that has its own **state (data)** and **behavior (functions)**.

- Example: A **pen**, **chair**, or **phone** — all are real-world objects.
- Objects are created using **classes** in OOP.

(2) Class

- A **class** is like a **blueprint or template** used to create objects.
- It contains **data members** (variables) and **member functions** (methods).
- Example: A **human body** can be seen as a class, and hands, eyes, etc., are like objects.

(3) Encapsulation

- **Encapsulation** means wrapping or binding data and functions into a single unit (class).
- It hides the internal details and protects the data from direct access.
- Example: A **medicine capsule** — it holds everything tightly inside.

(4) Abstraction

- **Abstraction** means showing only the necessary details and hiding the background processes.
- This helps in reducing complexity for the user.
- Example: A **login page** only asks for username and password — it hides how the system checks them.

(5) Inheritance

- **Inheritance** allows a class to take (inherit) properties and behavior from another class.
- This helps with code reuse and building relationships between classes.
- Example: A **son inherits traits** from his **father**.

(6) Polymorphism

- **Polymorphism** means performing a task in **many different ways**.
- It allows the same function name to behave differently depending on the object.
- Example: There are many **ways to travel** like by car, bike, or bus.

Types:

1. **Method Overloading** – Same function name but different number/types of arguments.
2. **Method Overriding** – A child class changes the function behavior of its parent class.

3. Steps to Set Up a C++ Development Environme

1. Install a C++ Compiler

A **compiler** is required to convert your C++ code into a format the computer can understand and run.

For Windows:

You can use any of the following options:

- **MinGW (Minimalist GNU for Windows)**
  - Download from: https://sourceforge.net/projects/mingw/
  - It provides the `g++` compiler for C++.
- **Code::Blocks with built-in GCC Compiler**
  - This is easier to set up as it includes the compiler and editor in one package.

2. Install an IDE (Optional but Recommended)

An **IDE (Integrated Development Environment)** helps you write, edit, and run code easily. It provides features like:

- Syntax highlighting
- Auto-complete
- Error checking
- Easy compiling and running

 Popular IDEs for C++:

Code::Blocks	Windows / Linux
Visual Studio	Windows
Dev C++	Windows
VS Code	Windows / Linux / Mac (All OS)

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

1. `cin` (Standard Input)

- Used to take input from the user via 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 = 99.99f;`
`double`	Stores decimal numbers (double precision)	`double pi = 3.14159;`

Type	Description	Example
`char`	Stores a single character	`char grade = 'A';`
`bool`	Stores true or false	`bool passed = true;`
`void`	Represents no value (used in functions)	`void display();`

2. Derived Data Types

These are built from 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 programmer.

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 concept)	`class Car { public: void drive(); };`

Example Program:

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

int main() {
    int age = 18;                // Integer
    float height = 5.9f;         // Floating point
    char grade = 'A';            // Character
    bool passed = true;          // Boolean
    double pi = 3.1415926535;    // Double
    string name = "Dhiraj";      // String (from <string> library)

    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.
- Happens **without using any cast**, when assigning a value of one type to a variable of another type.

Example:

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

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

    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    return 0;
}
```

Here, `int->` is **implicitly** converted to `float` by the compiler

2. Explicit Type Conversion (Type Casting)

- The **programmer manually** tells the compiler to convert one data type to another.
- Done using a **type cast**, like `(type)` before the value or variable.

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->` is **explicitly** converted to `int` using `(int)`.

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

1. Arithmetic Operators

These operators do basic maths like add, subtract, multiply, divide, etc.

Operator	Use	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Remainder	a % b

2. Relational Operators

These are used to **compare** two values. They give `true` or `false`.

Operator	Use	Example
==	Equal to	`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 Operators

These check **multiple conditions** and return `true` or `false`.

Operator	Meaning	Example
&&	AND	`a > 0 && b > 0`
`		`
!	NOT	`!(a > 0)`

4. Assignment Operators

These are used to **assign values** to variables.

Operator	Meaning	Example
=	Assign value	`a = 5`
+=	Add and assign	`a += 3` → `a = a + 3`
-=	Subtract and assign	`a -= 2`
*=	Multiply and assign	`a *= 4`
/=	Divide and assign	`a /= 2`

Operator	Meaning	Example
%=	Modulus and assign	a %= 3

5. Increment & Decrement Operators

These are used to **increase or decrease** the value by 1.

Operator	Meaning	Example
++	Add 1	a++ or ++a
--	Subtract 1	a-- or --a

6. Bitwise Operators

These work on numbers in **binary (0 and 1)** form.

Operator	Meaning
&	AND
`	`
^	XOR
~	NOT (invert)
<<	Left shift
>>	Right shift

7. Ternary (Conditional) Operator

It is a **short form of if-else**.

Syntax:

```
condition ? true_part : false_part;
```

Example:

```
int a = 10, b = 20;
```

```
int max = (a > b) ? a : b;
cout << "Maximum is: " << max;
```

Output: `Maximum is: 20`

Q4. What is the purpose and use of constants and literals in C++?

Constants in C++

Constants are values that do **not change** while the program is running.

- To **protect important values** from being changed by mistake.
- To make the program **easier to read** and understand.

Literals in C++

Literals are fixed values written **directly in the code**. They are the actual values assigned to variables.

Types of literals with examples:

```
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 Program using Constant and Literals:

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

const float PI = 3.14;  // constant value

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 `if-else` and `switch` with examples.

1. if Statement

The `if` statement runs code **only if a condition is true**.

Example:

```
int age = 18;

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

If the condition `age >= 18` is true, it prints the message.

2. if-else Statement

This checks a condition. If it's **true**, it runs one block of code, otherwise it runs another.

Example:

```
int age = 16;

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

Since age is 16, the output will be:

```
You are too young to vote.
```

3. if-else if Ladder

Used when you have **multiple conditions** to check one by one.

Example:

```
int marks = 75;

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

```
}
```

Since marks are 75, the output will be:

```
Grade B
```

4. switch Statement

`switch` is used when you want to compare one variable with **many fixed values**, like menu options or days of the week.

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++?

1. for loop

- Use this loop **when you know how many times** you want to repeat something.
- Initialization, condition, and update — **all are written in one line**.

Example:

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

Output:

```
0 1 2 3 4
```

Flow:

1. Start from `i = 0`
2. Check if `i < 5`
3. Print value
4. Increase `i`
5. Repeat until condition is false

2. while loop

- Use this loop when you **don't know exactly how many times** the loop will run.
- The condition is checked **before** the loop runs.

Example:

```
int i = 0;

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

Output:

```
0 1 2 3 4
```

Flow:

1. Check condition
2. Run code inside loop
3. Update value
4. Repeat if condition is true

3. do-while loop

- This loop is similar to `while`, but the **code runs at least once**, even if the condition is false.
- The condition is checked **after** the loop body.

Example:

```
int i = 0;

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

Output:

```
0 1 2 3 4
```

3. How are `break` and `continue` statements used in loops? Explain with examples.

1. break statement

- The `break` statement is used to **stop the loop completely**, even if the condition is still true.
- It is mostly used when you want to **exit the loop early**.

Example:

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

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

Output:

```
1 2 3 4
```

The loop stops when `i == 5`, so numbers after 4 are not printed.

2. continue statement

- The `continue` statement is used to **skip one loop step** and go to the next.
- It does **not stop** the whole loop — it just **jumps to the next turn**.

Example:

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

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

Output:

```
1 2 4 5
```

4. What are Nested Control Structures in C++? Explain with an example.

What can be nested?

- `if` inside another `if`
- Loops inside loops (`for`, `while`, etc.)
- `switch` inside `if`, or vice versa
- `if` inside loops, or loops inside `if`

Why do we use nested structures?

- To make **decisions inside other decisions**
- To do **repeated tasks inside other loops**
- To handle **multiple levels of conditions**
- For things like **pattern printing**, **tables**, or **filtering data**

1. Nested `if` Statement

Used when one condition depends on another.

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;
}
```

Output: `Eligible for admission.`

2. Nested Loops

A loop inside another loop is called a nested loop. Mostly used for **tables, patterns, or matrix operations**.

Example:

```
for (int i = 1; i <= 3; i++) {

    for (int j = 1; j <= 2; j++) {
        cout << "i = " << i << ", j = " << j << endl;
    }
}
```

Output:

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
... (total 6 times)
```

3. Loop inside `if` OR `if` inside Loop

You can run a condition in every loop turn.

Example:

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

Output:

```
2 is even
4 is even
```

4. Nested `switch` Statement

Sometimes, you use a `switch` inside another condition like `if`.

Example:

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

int main() {
    int userType = 1;
    char choice = 'b';

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

```
                cout << "Invalid Option" << endl;
            }
        }

        return 0;
}
```

Output: `Delete User`

4. Functions and Scope

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

What is a Function in C++?

A **function** in C++ is a block of code that does a specific job.

Why Use Functions?

- To **avoid repeating code**
- To break big programs into **smaller and easier parts**
- To **reuse** the same code in different places
- To make the code **easy to read and fix**

Parts of a Function

1. Function Declaration (also called Prototype)

- It tells the computer the name of the function, what it returns, and what inputs it takes.

```
int add(int, int);  // This is a declaration
```

2. Function Definition

- This is where we write what the function actually does.

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

3. Function Call

- This is when we **use** or **run** the function.

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

Example Program:

```
#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++?

In C++, **scope** means the part of the program where a variable can be used or seen.

The scope decides:

- **Where** the variable can be accessed
- **How long** it stays in memory

Types of Variable Scope

1. Local Scope

- A variable made **inside a function**, loop, or `{}` block.
- It can only be used **inside** that block.
- The memory is cleared once the block finishes.

Example:

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

Here, `x` is a **local variable** and can only be used inside the `show()` function.

2. Global Scope

- A variable made **outside** all functions.
- It can be used in **any function** in the same file.
- It stays in memory for the whole time the program is running.

Example:

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

void display() {
  cout << x;  // x is accessible here
}
```

Here, `x` is a **global variable** and can be used in any function.

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

Recursion is a method in programming where a function **calls itself** to solve a problem.

It helps break a big problem into **smaller, easier** problems, and solves each one by calling the same function again.

How a Recursive Function Works:

```
returnType functionName(parameters) {
  if (base condition)
    return value; // this stops the recursion
  else
    return functionName(smaller input); // function calls itself
}
```

Important Terms:

Term	Meaning
Base Case	Condition that **ends** the recursion
Recursive Case	The part where the function **calls itself**

Example: Find Factorial Using Recursion

The factorial of a number `n` (written as `n!`) means:
`n! = n × (n-1) × (n-2) × ... × 1`

Example:
`5! = 5 × 4 × 3 × 2 × 1 = 120`

C++ Code:

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

// Recursive function to find factorial
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** is a simple way to tell the compiler about a function **before** it is used in the program. It gives the compiler three things:

- The **name** of the function

- The **type of value** it will return
- The **number and types of inputs** (parameters)

Why Do We Use Function Prototypes?

- So we can **use a function before writing its full code**
- To help the compiler **check for errors** in function calls
- To make the code **more organized and easy to understand**

Function Prototype Syntax:

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

Note: You can skip the parameter names in the prototype.

Example Program:

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

// Function Prototype
int add(int, int);  // Just telling the compiler about the function

int main() {
    int result = add(10, 5);  // Calling the function
    cout << "Sum = " << result;
    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;  // Actual code of the function
}
```

Output:

```
Sum = 15
```

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 special type of variable that can hold **multiple values of the same type**. Instead of creating separate variables for each value, you can use one array and access its elements using an **index**.

Basic Syntax:

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

Example:

```
int numbers[5];  // This creates space for 5 integers
```

Types of Arrays in C++

1. Single-Dimensional Array (1D Array)

- Stores elements in a **straight line** (like a list).
- Accessed with **one index**.

Example:

```
int marks[3] = {85, 90, 78};
cout << marks[1];  // Output: 90
```

2. Multi-Dimensional Array (2D Array or More)

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

Example:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
cout << matrix[1][2];  // Output: 6
```

Difference Between 1D and 2D Arrays

Feature	1D Array	2D / Multi-Dimensional Array
Structure	Linear (one line)	Table form (rows and columns)
Declaration	`int arr[5];`	`int arr[2][3];`
Access Elements	`arr[index]`	`arr[row][column]`
Use Case	List of scores	Matrices, tables, images
Real-Life Example	Row of lockers	Excel sheet or timetable

Example Program with Both 1D and 2D Arrays

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

```
int main() {
    // Single-Dimensional Array
    int scores[3] = {10, 20, 30};
    cout << "1D Array: " << scores[1] << endl;

    // Multi-Dimensional 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

In C++, strings can be handled using the **`string` class**, which is much easier and safer than using character arrays (`char[]`). The `string` class is a part of the **Standard Library**.

To use strings, include this header:

```
#include <string>
```

Example Program:

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

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

    // Combine (concatenate) two strings
    string fullName = firstName + " " + lastName;

    // Output the full name
    cout << "Full Name: " << fullName << endl;

    // Print the length of the full name
    cout << "Length: " << fullName.length() << endl;

    return 0;
}
```

Output:

```
Full Name: John Doe
Length: 8
```

Commonly Used String Functions

Function	What It Does
`length()` / `size()`	Returns the number of characters
`append("str")`	Adds another string to the end
`substr(pos, len)`	Gets a part (substring) of the string
`find("str")`	Finds the position of a substring
`compare("str")`	Compares two strings (returns 0 if same)
`empty()`	Checks if the string is empty
`erase(pos, len)`	Removes part of the string

3. How Are Arrays Initialized in C++?

An **array** is a group of variables of the **same data type**, stored together in memory. You can use **index numbers** to access each element.

C++ allows arrays of any type: `int`, `float`, `char`, etc.

1. One-Dimensional (1D) Arrays

Declaration:

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

Ways to Initialize 1D Arrays:

- **Method 1: With size and values**

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

- **Method 2: Without size (size is auto-detected)**

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

- **Method 3: Partial Initialization (rest becomes 0)**

```
int arr[5] = {1, 2};  // → {1, 2, 0, 0, 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:

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

Ways to Initialize 2D Arrays:

- **Method 1: Row-by-row initialization**

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

- **Method 2: In a 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 like `"Hello"` or `"C++"`.
C++ makes working with strings easier using the **`string` class**, which is part of the **Standard Library (STL)**.

To use strings in C++:

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

Declaring and Initializing Strings

```
string str1;                // Empty string
string str2 = "Hello";      // Direct assignment
string str3("World");       // Constructor-style
```

Basic String Operations

Operation	Description	Example
Assignment	Assign one string to another	`str2 = str1;`
Concatenation	Join two strings	`str3 = str1 + str2;`
Access	Get a character using index	`str[0]`, `str.at(1)`
Input	Read full line from user	`getline(cin, str);`
Output	Print the string	`cout << str;`

Example: Basic Use

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

Output:

```
Hello, John
```

Commonly Used String Functions

1. **`length()` or `size()`** – Gives the number of characters

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

2. **`append()`** – Adds more text to the end

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

3. **`compare()`** – Compares two strings

```
string a = "apple", b = "banana";
int result = a.compare(b); // result < 0 → "apple" < "banana"
```

4. **`substr(start, length)`** – Gets part of the string

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

5. **`replace(pos, len, newStr)`** – Replaces part of string

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

6. **`insert(pos, str)`** – Inserts text at a position

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

7. **`erase(pos, len)`** – Deletes part of the string

```
string s = "Good Morning";
s.erase(4, 8);
cout << s; // Output: Good
```

8. **`find(str)`** – Finds position of a substring

```
string s = "Hello C++";
cout << s.find("C++"); // Output: 6
```

9. **`getline()`** – Reads a full line with spaces

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

Example Program Using Multiple Functions

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

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

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

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

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

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

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

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

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

    return 0;
}
```

Output:

```
s3 = Hello World
Length: 11
After replace: Hello C++
Substring: Hello
After insert: Hello, C++
After erase: Hello C++
Find C++: 6
```

6. Introduction to Object-Oriented Programming

1. Key Concepts of Object-Oriented Programming (OOP)

OOP stands for **Object-Oriented Programming**, a method of writing programs using **objects** and **classes**. It makes code more organized, reusable, and easier to manage.

Here are the **main concepts** of OOP:

1. Object

- An **object** is anything that has a **state** (data) and **behavior** (functions).
- Real-life examples: a pen, a chair, a book, etc.
- In programming, an object is created using a class.

2. Class

- A **class** is like a **blueprint** or **template** used to create objects.
- It defines the data members (variables) and member functions (methods).
- Example: If "Car" is a class, then "Honda" and "BMW" are objects of that class.

3. Encapsulation

- Encapsulation means **wrapping data and functions** into a single unit (class).
- It helps protect the data from outside interference.
- Think of a **capsule** — it wraps all the medicine inside.

4. Abstraction

- Abstraction means **hiding the complex details** and showing only the useful parts.
- Example: When you use a mobile phone, you press buttons but don't see how the internal circuits work.
- Login page is a good example — you see only username and password fields, not the backend logic.

5. Inheritance

- Inheritance means a class can **use the properties and methods** of another class.
- It helps in **code reusability**.
- Example: A **child** inherits qualities from the **parent**.

6. Polymorphism

- Polymorphism means "**one name, many forms**".
- It allows the same function or operator to **work in different ways**.
- Example: A person can be a **student**, a **player**, and a **friend** — all at the same time.

Two types of Polymorphism:

1. **Method Overloading** – Same function name, different parameters (in same class)
2. **Method Overriding** – Same function name in child class that replaces parent's version

2. What Are Classes and Objects in C++? Provide an Example

In C++, **classes and objects** are the basic building blocks of Object-Oriented Programming.

What is a Class?

- A **class** is a **user-defined data type**.
- It works like a **template** or **blueprint** to create objects.
- A class groups together **data** (variables) and **functions** (methods) in one unit.

Syntax of a Class:

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

What is an Object?

- An **object** is a **real instance** of a class.
- It holds actual values and allows you to access class members.

Syntax to Create an Object:

```
ClassName objectName;
```

Example Program: Using Class and Object

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

// Class definition
class Car {
public:
    string brand;
    int speed;

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

int main() {
    Car c1;  // Creating object of class Car

    // Assign values to the object's members
    c1.brand = "BMW";
    c1.speed = 120;

    // Call the method
    c1.drive();
```

```
    return 0;
}
```

Output:

```
Driving BMW at 120 km/h
```

3. What is Inheritance in C++? Explain with an Example

Inheritance is a feature in C++ that allows one class to **get (inherit)** the properties and behaviors (functions) of another class.

- The existing class is called the **base class** (or parent class).
- The new class is called the **derived class** (or child class).

Why Use Inheritance?

- To **reuse existing code**
- To **avoid writing the same code again**
- To create a **relationship between classes**

Types of Inheritance in C++

1. **Single Inheritance** – One base class, one derived class
2. **Multiple Inheritance** – One derived class inherits from multiple base classes
3. **Multilevel Inheritance** – A class inherits from another derived class
4. **Hierarchical Inheritance** – Multiple classes inherit from one base class
5. **Hybrid Inheritance** – A mix of two or more types of inheritance

Example: Single Inheritance

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

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

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

```
int main() {
    Dog d;

    d.eat();   // Function inherited from Animal class
    d.bark();  // Dog's own function

    return 0;
}
```

Output:

```
This animal eats food
Dog barks
```

4. What is Encapsulation in C++? How is It Achieved in Classes?

Encapsulation means **hiding the internal details** of an object and **only exposing what is necessary**.

In simple words, it means **putting data (variables) and functions (methods)** together inside a **class**, and **controlling who can access what**.

Why Use Encapsulation?

- To **protect** data from direct access
- To make the code **secure and organized**
- To follow the **OOP principle** of "data hiding"

How Is Encapsulation Achieved in C++?

Encapsulation is done using **access specifiers**:

Access Specifier	Accessible Where?
`private`	Only inside the same class
`public`	Anywhere in the program
`protected`	In the class and its derived (child) classes

By declaring **data members as private** and **functions as public**, we ensure that the internal data cannot be accessed directly.

Example: Encapsulation in a Class

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

class Account {
```

```
private:
    int balance;  // private data

public:
    void setBalance(int b) {
        if (b >= 0)
            balance = b;  // setting value securely
    }

    int getBalance() {
        return balance;  // getting value
    }
};

int main() {
    Account a;

    a.setBalance(5000);  // Set value through public method
    cout << "Balance: " << a.getBalance();  // Output: 5000

    return 0;
}
```

Output:

```
Balance: 5000
```