Module 3:
Introduction to OOPS Programming

Name: Chandvaniya Abhay

1.Introduction to C++

Theory Exercise:

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

1. Programming Paradigm

- **Procedural Programming**: Follows a **top-down** approach and focuses on **procedures or routines** (i.e., functions).
- **OOP**: Follows a **bottom-up** approach and is based on the concept of **objects and classes**.

2. Core Focus

- **Procedural**: Focuses on **functions** and the **sequence of actions** to be performed.
- **OOP**: Focuses on **objects** that contain both **data (attributes)** and **methods (behaviors)**.

3. Data Handling

- **Procedural**: Data is **separate** from functions and is often **global**, making it less secure.
- **OOP**: Data is **encapsulated** within objects, enhancing **data security and integrity**.

4. Code Reusability

- **Procedural**: Less emphasis on code reuse. Reusability is limited to function calls.
- **OOP**: Promotes code reuse through **inheritance** and **polymorphism**.

5. Modularity

- **Procedural**: Code is divided into **functions**, but often less modular as data is shared.

- **OOP**: Code is divided into **classes and objects**, making it highly modular.

6. Examples of Languages

- **Procedural**: C, Pascal, Fortran
- **OOP**: Java, C++, Python, C#, Ruby

7. Ease of Maintenance and Scalability

- **Procedural**: Can become difficult to maintain and scale as project size grows.
- **OOP**: Easier to maintain, update, and scale due to encapsulation and modularity.

8. Key Concepts

- **Procedural**:
  - Functions
  - Procedures
  - Sequential execution
- **OOP**:
  - Classes and Objects
  - Inheritance
  - Polymorphism
  - Encapsulation
  - Abstraction

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

1. Encapsulation (Data Hiding)

- **Explanation**: In OOP, data and the methods that operate on it are bundled together into **classes**, and internal details can be hidden from outside access using access modifiers (private, public, protected).
- **Advantage**: This prevents accidental or unauthorized modification of data, increasing **data security and integrity**.
- **In POP**: Data is usually global and accessible by any function, making it vulnerable to unintended changes.

2. Code Reusability through Inheritance

- **Explanation**: OOP supports **inheritance**, allowing a new class (child) to inherit properties and methods from an existing class (parent).
- **Advantage**: Promotes **code reuse**, reducing duplication and making maintenance easier.
- **In POP**: No concept of inheritance; code must be rewritten or manually copied.

3. Polymorphism

- **Explanation**: OOP allows methods or functions to behave differently based on the object that invokes them (e.g., method overloading or overriding).
- **Advantage**: Enables **flexibility and extensibility** in code, allowing the same interface to work with different data types or behaviors.
- **In POP**: Functionality must be implemented separately for each data type or condition, increasing complexity.

4. Modularity

- **Explanation**: Programs are divided into smaller, self-contained units called **objects**, each representing a real-world entity.
- **Advantage**: Makes it easier to **design, debug, test, and maintain** code, especially in large projects.

- **In POP**: Modularity is limited to functions, and managing large codebases becomes harder.

5. Scalability and Maintainability

- **Explanation**: OOP's structure makes it easier to add new features or update existing ones without breaking the whole system.
- **Advantage**: OOP is **ideal for large-scale software development** where teams may work on different parts of a system.
- **In POP**: Any change may require reworking many parts of the program due to tightly coupled code.

6. Real-World Modeling

- **Explanation**: OOP allows you to model real-world objects (like Car, BankAccount, User) with attributes and behaviors.
- **Advantage**: Leads to **more intuitive and natural program design**, especially useful in GUI, games, and simulations.
- **In POP**: Abstracts tasks as procedures, which can be harder to relate to real-world entities.

7. Improved Collaboration in Teams

- **Explanation**: In OOP, different developers can work on different classes/modules independently.
- **Advantage**: Encourages **team development** and simplifies version control and integration.
- **In POP**: Tight coupling makes collaboration more difficult and increases the chance of code conflicts.

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

Step 1: Install a C++ Compiler

A compiler is required to translate C++ code into executable programs.

➤ *Common Compilers:*

- **GCC (GNU Compiler Collection)** – For Linux/macOS/Windows (via MinGW)
- **MSVC (Microsoft Visual C++)** – For Windows (part of Visual Studio)
- **Clang** – Popular on macOS and some Linux systems

➤ *How to Install:*

<u>Windows</u>:

- Option 1: **Install MinGW** (Minimalist GNU for Windows)
  1. Download MinGW from mingw-w64.org
  2. Install it with C++ support.
  3. Add bin folder (e.g., C:\MinGW\bin) to your **System PATH**.
- Option 2: **Install Visual Studio** (Recommended for beginners)
  1. Download Visual Studio Community Edition from visualstudio.microsoft.com
  2. During installation, select **"Desktop development with C++"** workload.

<u>Linux</u>:

```
sudo apt update
sudo apt install build-essential
```

<u>macOS:</u>

Install Xcode Command Line Tools:

```
xcode-select –install
```

Step 2: Choose and Install a Text Editor or IDE

➤ *Popular IDEs for C++:*

- **Visual Studio** (Windows only)
- **Code::Blocks** (Cross-platform)
- **CLion** (JetBrains, paid with free student license)
- **Dev C++** (Lightweight)
- **Eclipse CDT** (for C/C++)

➤ *Popular Text Editors with Extensions:*

- **VS Code** (Lightweight and powerful)
  - Install VS Code from: code.visualstudio.com
  - Add the **C/C++ extension** by Microsoft

Step 3: Set Up Environment Variables (if needed)

If using GCC or MinGW manually:

- Add the path to the compiler's bin directory to your system's **PATH** environment variable.
- This allows you to run g++ or gcc from the terminal or command prompt.

Step 4: Write Your First C++ Program

Open your IDE or text editor and create a new file, e.g., main.cpp:

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

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

Step 5: Compile and Run the Program

➤ *Using Command Line (GCC):*

1. Open terminal or command prompt
2. Navigate to the directory with main.cpp
3. Compile:
4. g++ main.cpp -o myprogram
5. Run:
   - On Windows:
   - myprogram.exe
   - On Linux/macOS:
   - ./myprogram

➤ *Using an IDE:*

- Press **Run** or **Build and Run** — the IDE handles compilation and execution.

Step 6: (Optional) Set Up Debugging and Build Tools

- Most modern IDEs include a debugger (like GDB).
- You can set breakpoints, inspect variables, and step through code.

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

Main Input/Output Operations in C++

C++ uses the **iostream** library for basic I/O operations, which includes:

- cin → Standard **input** stream (from keyboard)
- cout → Standard **output** stream (to screen)
- cerr → Standard **error** stream (for error messages)
- clog → Standard **log** stream (for general logging info)

All of these are part of the <iostream> header.

1. cout – Output Operation

Used to print output to the screen.

➤ **Syntax:**

cout << data;

➤ **Example:**

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

int main() {
   cout << "Hello, world!" << endl;
   cout << "The answer is: " << 42 << endl;
   return 0;
}
```

- << is the **insertion operator**.
- endl is used to move to a new line (can also use \n).

2. cin – Input Operation

Used to get input from the user (keyboard).

➤ **Syntax:**

cin >> variable;

➤ **Example:**

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

int main() {
   int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "You entered: " << age << endl;
   return 0;
}
```

- >> is the **extraction operator**.
- Reads input and stores it in the variable.

3. cerr – Error Output

Used to display error messages.

➤ **Example:**

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

int main() {
   cerr << "Error: Invalid input!" << endl;
   return 0;
}
```

- cerr is **unbuffered** — messages appear immediately.

4. clog – Logging Output

Used for general-purpose logging or debugging messages.

➤ **Example:**

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

int main() {
    clog << "Program started..." << endl;
    return 0;
}
```

- clog is **buffered** — messages may appear later than cerr.

2. Variables, Data Types, and Operators

Theory Exercise:

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

1. Basic (Primitive) Data Types

These are the fundamental data types provided by the language.

Type	Description	Example
Int	Integer values (whole numbers)	int age = 25;
Float	Floating point (single precision)	float price = 5.75;
Double	Double precision floating point	double pi = 3.1415;
Char	Single character	char grade = 'A';
bool	Boolean (true or false)	bool isOpen = true;

2. Derived Data Types

These are based on fundamental types.

Type	Description	Example
Array	Collection of elements of same type	int nums[5] = {1, 2, 3, 4, 5};
Pointer	Stores memory address	int* ptr = &age;
Function	Block of code that performs a task	int sum(int a, int b) { return a+b; }
Reference	An alias for another variable	int& ref = age;

3. User-defined Data Types

Created by the programmer for specific needs.

Type	Description	Example
Struct	Group of variables of different types	struct Person { string name; int age; };
class	Blueprint for objects (OOP)	class Car { public: string brand; };
union	Like struct but shares memory	union Data { int i; float f; };
enum	Named set of integer constants	enum Color { RED, GREEN, BLUE };
typedef / using	Aliases for data types	typedef int Marks; or using Marks = int;

4. Void Type

Represents the absence of any value or type.

Type	Description	Example
Void	Used for functions that return nothing	void greet() { cout << "Hi!"; }

5. Modifiers with Data Types

Used to alter the meaning/size of basic data types.

Type	Description	Example
Signed	Allows both positive and negative values	signed int x = -100;
Unsigned	Only allows positive values	unsigned int y = 200;
short	Reduces storage size	short int a = 10;
Long	Increases storage size	long int b = 1000000;
long long	Even bigger than long	long long int c = 1e12;

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

1. Implicit Type Conversion (Type Coercion)

- Performed automatically by the compiler**.**
- Happens when you assign or operate on variables of different types.
- It follows the standard type promotion rules (e.g., int to float, char to int).
- No data loss *if* the conversion is to a "larger" or compatible type.

2. Explicit Type Conversion (Type Casting)

- Performed manually by the programmer.
- Used when implicit conversion doesn't work or may cause data loss.
- Syntax involves *casting operators* or *C-style casting*.

Feature	Implicit Conversion	Explicit Conversion
Performed by	Compiler	Programmer
Syntax	Automatic	(type) or static_cast<>
Safety	Generally safe	Riskier (can lose data)
Use Case	Convenience, mixed types	Precision, overriding rules

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

1. Arithmetic Operators

Used for basic mathematical operations.

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

2. Relational (Comparison) Operators

Used to compare two values.

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

3. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
!	Logical NOT	!(a > b)

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	a = 10
+=	Add and assign	a +=5 // a = a + 5
-=	Subtract and assign	a -= 2
*=	Multiply and assign	a *= 3
/=	Divide and assign	a /= 4
%=	Modulus and assign	a %= 2

5. Unary Operators

Operate on a single operand.

Operator	Description	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	++a or a++
--	Decrement	--a or a--
!	Logical NOT	! true

6. Bitwise Operators

Operate at the binary level.

Operator	Description	Example
&	Bitwise AND	a & b
`	`	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left Shift	a << 2
>>	Right shift	a >> 1

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

1. **Constants in C++**

- **Purpose:**

Constants are named identifiers used to store values that must **not change** during program execution.

- **Why Use Constants?**

Prevent accidental modification of important values

Improve code clarity (PI is more meaningful than 3.14159)

Easier maintenance (change the value in one place)

Example:

```
const int maxScore = 100;
```

2. Literals in C++

- **Purpose:**

Literals are **fixed values** written directly in the source code — not stored in a variable.

- **Why Use Literals?**

Represent constant values like numbers, characters, and strings

Used in expressions, assignments, and function calls

- **Types of Literals:**

Literal Type	Example	Description
Integer	42, 0xFF	Decimal, hex, octal, binary
Floating-point	3.14, 2.5e3	Real numbers (float/double)
Character	'A', '9'	Enclosed in single quotes
String	"Hello"	Enclosed in double quotes
Boolean	true, false	Boolean values
Null pointer	nullptr	Null pointer literal (C++11+)

3. Control Flow Statements

Theory Exercise:

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

- **Conditional Statements in C++**

Conditional statements in C++ are **decision-making constructs** that allow a program to choose between different paths of execution based on whether a given condition is **true** or **false**.

1. if-else Statement

The **if-else** statement is the most basic form of conditional control structure.

Purpose:

It allows the program to evaluate a condition and:

- Execute one block of code if the condition is **true**
- Execute a different block if the condition is **false**

Key Concepts:

- The condition is a **Boolean expression** (i.e., it evaluates to true or false)
- Can be extended using else if for multiple conditions

Types:

- **Simple if**: Executes code only if the condition is true
- **if-else**: Executes one block if the condition is true, another if false
- **if-else if-else**: Handles multiple conditions in sequence

2. switch Statement

The switch statement is used for **multi-way decision-making** when a variable or expression can take on a limited set of constant values.

Purpose:

It simplifies the code when multiple if-else conditions are based on the **same variable** or **expression**.

Key Concepts:

- The expression inside the switch must be of an **integral type** (e.g., int, char, enum)
- Each case represents a possible value of the expression
- The break statement prevents fall-through to the next case
- A default case handles any unmatched values

Summary:

Feature	if-else Statement	switch Statement
Type	General-purpose condition checking	Multi-way branching based on constant value
Conditions	Boolean expressions (any logical test)	Constant integral expressions only
Flexibility	More flexible (can handle complex logic)	Less flexible (limited to discrete values)
Use Case	When conditions are complex or varied	When checking one variable for multiple values

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

- **Difference Between for, while, and do-while Loops in C++**

Loops in C++ are **control structures** that allow you to **repeat a block of code** multiple times based on a condition.

1. for Loop

Definition:

A for loop is used when the **number of iterations is known** beforehand.

Syntax:

```
for (initialization; condition; update)
{
   // loop body
}
```

Characteristics:

- Initialization, condition, and update are all part of the loop declaration.
- Best suited for **count-controlled** loops (e.g., loops that run a fixed number of times).
- Compact and readable for simple iteration

2. while Loop

Definition:

A while loop is used when the **number of iterations is not known** and depends on a condition being true.

Syntax:

```
while (condition)
{
   // loop body
}
```

Characteristics:

- The condition is checked **before** the loop body is executed.
- If the condition is false initially, the loop **may not execute at all**.
- Commonly used when the loop depends on **user input** or **external events**.

3. do-while Loop

Definition:

A do-while loop is similar to a while loop, but the **condition is checked after** executing the loop body.

Syntax:

```
do
{
   // loop body
} while (condition);
```

Characteristics:

- The loop body is **executed at least once**, regardless of the condition.
- Useful when the loop must run at least once (e.g., menus, retry prompts).

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

Break and Continue Statements in C++ Loops

In C++, break and continue are **loop control statements** used to **alter the normal flow** of loop execution.

1. break Statement

Purpose:

- Used to **immediately exit** the loop, regardless of the loop condition.
- Control moves to the **first statement after the loop**.

Use Cases:

- Exiting a loop when a certain condition is met.
- Terminating early during search operations or menu-driven programs.

Example:

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

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

2. continue Statement

Purpose:

- Used to **skip the current iteration** of the loop and move to the **next iteration**.
- The rest of the loop body **after continue is ignored** for the current iteration.

Use Cases:

- Skipping unwanted or invalid data in a loop.
- Skipping execution based on a condition.

Example:

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

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

4. Explain nested control structures with an example.

Nested Control Structures in C++

Definition:

Nested control structures are **control statements placed inside other control statements**. This means you can put a loop inside another loop, an if inside a loop, a switch inside an if, etc.

They allow more complex decision-making and repetition in your program by combining multiple control structures.

Types of Nested Structures

1. **Nested if statements**
2. **if inside loops (or vice versa)**
3. **Nested loops (for, while, do-while)**
4. **Nested switch statements** (less common)

Example: Nested if Statement

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

int main()
{
   int number;
   cout << "Enter a number: ";
   cin >> number;

   if (number > 0)
   {
      if (number % 2 == 0)
      {
         cout << "The number is positive and even." << endl;
      }
      else
```

```
        {
            cout << "The number is positive and odd." << endl;
        }
    }
    else
    {
        cout << "The number is not positive." << endl;
    }

    return 0;
}
```

Example: Nested for **Loops** (Printing a pattern)

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

int main()
{
    for (int i = 1; i <= 3; i++)
    {
        for (int j = 1; j <= 5; j++)
        {
            cout << "* ";
        }
        cout << endl;
    }
    return 0;
}
```

4. Functions and Scope

Theory Exercise:

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 performs a specific task.
Functions help **organize code**, **avoid repetition**, and **make programs easier to read and maintain**.

⇒ **3 Key Parts of a Function**

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

1. Function Declaration (Prototype)

- Tells the compiler **what the function looks like**.
- Usually placed **above main()**, or in a header file.
- Includes the return type, function name, and parameters (if any).

Syntax:

```
returnType functionName(parameterType1, parameterType2, ...);
```

Example:

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

2. Function Definition

- This is where you **write the actual code** that the function performs.
- Must match the declaration.

Syntax:

```
returnType functionName(parameters)
{
    // function body (code to run)
}
```

Example:

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

3. Function Call

- This is how you **use** the function in your program.
- You "call" the function by its name and pass required arguments.

Example:

```
int result = add(5, 3);  // Function call
```

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

What is *Scope* in C++?

In C++, the **scope of a variable** defines **where in the program the variable can be accessed or used**.

Types of Variable Scope in C++

There are mainly two types:

1. **Local Scope**
2. **Global Scope**

1. Local Scope

- A variable declared **inside a function or a block** ({ }) is called a **local variable**.
- It can **only be accessed within that function or block**.
- It is **created when the function/block runs** and **destroyed when it ends**.

Example:

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

2. Global Scope

- A variable declared **outside all functions**, typically at the top of the program.
- It can be accessed **from any function in the same file** (after its declaration).
- It exists **throughout the program's lifetime**.

Example:

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

void show()
{
    cout << x << endl;  // Can access x here
}
```

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

What is Recursion?

Recursion is a programming technique where a function **calls itself** to solve a smaller part of the problem.

In C++, a recursive function must have:

1. A **base case** – to stop the recursion.
2. A **recursive case** – where the function calls itself.

Simple Real-Life Analogy

Imagine you have a stack of plates. To count them:

- You take one plate.
- Ask someone to count the rest.
- When no plates are left, you stop.

That's recursion! Breaking a big problem into smaller versions of itself.

Example: Factorial Using Recursion

The **factorial** of a number n (written as n!) is:

```
n! = n × (n-1) × (n-2) × ... × 1
```

With recursion:

```
n! = n × (n-1)!
Base Case: 0! = 1
```

Example:

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

int factorial(int n) // Recursive function
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);  // recursive call
}

int main()
{
    int num;

    cout << "Enter a positive number: ";
    cin >> num;

    if (num < 0)
    {
        cout << "Factorial is not negative numbers." << endl;
    }
    else
    {
        int result = factorial(num);
        cout << "Factorial of " << num << " is: " << result << endl;
    }
    return 0;
}
```

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

What Are Function Prototypes in C++?

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

- The **function name**
- The **return type**
- The **parameter types** (but not necessarily the names)
- **Without providing the function body**

It lets the compiler know **how the function will be used later in the code**, even if the actual function definition comes after the call.

Syntax of a Function Prototype:

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

Example:

```
int add(int, int);  // This is a function prototype
```

Why Are Function Prototypes Used?

Purpose	Explanation
Inform the compiler early	So it knows the function's signature before its actual definition appears.
Enable type checking	Ensures the function is called with the correct number and type of arguments.
Allow flexible code structure	Lets you place main() at the top and actual function code later.
Useful in multiple files	Prototypes can be placed in header files (.h) for sharing between files.

5.Arrays And Strings

Theory Exercise:

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

What Are Arrays in C++?

An **array** in C++ is a **collection of elements of the same data type**, stored in **contiguous memory locations**.
Each element is accessed using an **index**, which starts from 0.

Basic Syntax of an Array:

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

Example:

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

Types of Arrays in C++

There are mainly two types of arrays:

Type	Meaning
Single-dimensional	A linear list of elements
Multi-dimensional	Arrays with 2 or more dimensions (like a table or matrix)

1. Single-Dimensional Array

A **single-dimensional array** stores data in a **linear form (1D)** — like a list.

Example:

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

Index	Value
0	10
1	20
2	30
3	40

2. Multi-Dimensional Array

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

Declaration:

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

Example Initialization:

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

	Col 0	**Col 1**	**Col 2**
Row 0	1	2	3
Row 1	4	5	6

Key Differences: 1D vs 2D Arrays

Feature	**1D Array**	**2D Array**
Structure	Linear (like a list)	Tabular (like a grid/matrix)
Declaration example	int a[5];	int a[2][3];
Access element	a[i]	a[i][j]
Use case examples	Marks, prices, names list	Matrices, tables, game boards
Memory layout	Continuous in 1 direction	Continuous in row-major order

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

String Handling in C++

In C++, **strings** are sequences of characters. There are **two main ways** to handle strings:

1. C-style Strings (Old way)

- Based on character arrays.
- End with a **null character** '\0'.
- Uses functions from <cstring> like strcpy(), strlen(), etc.

Example:

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

int main()
{
    char name[20];

    cout << "Enter your name: ";
    cin >> name;

    cout << "Length = " << strlen(name) << endl;  // Counts characters

    return 0;
}
```

2. C++ String Class (Modern way)

- Provided by the **<string>** header.
- Safer and easier to use.
- Supports many built-in operations like concatenation, comparison, length, etc.

Common String Operations (C++ String Class)

1. Declare and Initialize

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

int main()
{
   string str1 = "Hello";
   string str2("World");

   cout << str1 << " " << str2 << endl;  // Output: Hello World
   return 0;
}
```

2. Input and Output

```
string name;
cout << "Enter your name: ";
cin >> name;  // Only reads a single word
cout << "Hello, " << name << "!" << endl;
For full lines (including spaces), use getline():
string fullName;
getline(cin, fullName);
```

3. Concatenation

```
string a = "Good";
string b = "Morning";
string result = a + " " + b;
cout << result;  // Output: Good Morning
```

4. Length of String

```
string text = "example";
cout << "Length = " << text.length();  // Output: 7
```

5. Access Characters

```
string word = "Apple";
cout << word[0];  // Output: A
word[1] = 'u';    // Changes 'p' to 'u'
cout << word;     // Output: Auple
```

6. Compare Strings

```
string a = "hello";
string b = "hello";

if (a == b)
{
   cout << "Strings are equal";
}
else
{
   cout << "Strings are not equal";
}
```

7. Other Useful Functions

Function	Description
s.length()	Returns length of string
s.empty()	Checks if string is empty
s.substr(pos, n)	Returns substring
s.find("text")	Finds position of substring
s.erase(pos, n)	Removes part of the string
s.insert(pos, str)	Inserts string at position

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

How Are Arrays Initialized in C++?

In C++, **arrays can be initialized** at the time of declaration or later in the program. Let's look at examples of **both 1D and 2D arrays**.

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

Syntax:

```
data_type array_name[size] = {values};
```

Example 1: Full Initialization

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

- This creates an array of size 5 and fills it with the values.

Example 2: Partial Initialization

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

- Only the first two elements are set.
- Remaining values are automatically initialized to **0**.

Example 3: Size Inferred from Values

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

- Compiler automatically sets the size to 3.

Example 4: Default Initialization

```
int numbers[3] = {};  // All values will be 0
```

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

Syntax:

```
data_type array_name[rows][columns] = { {row1}, {row2}, ... };
```

Example 1: Full Initialization

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

	Col 0	**Col 1**	**Col 2**
Row 0	1	2	3
Row 1	4	5	6

Example 2: Flat Initialization

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

- Elements are filled row by row.

Example 3: Partial Initialization

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

- Missing values are initialized to 0.

	Col 0	**Col 1**	**Col 2**
Row 0	1	2	0
Row 1	4	0	0

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

String Operations and Functions in C++

In C++, strings are commonly handled using the **std::string** class, which provides a wide range of **operations** and **built-in functions** to manipulate and analyze strings easily.

You need to include:

```
#include <string>
```

⇒ Basic String Operations

1. Declaration and Initialization

```
string s1 = "Hello";
string s2("World");
```

2. Input/Output

```
string name;
cin >> name;          // Reads single word
getline(cin, name);  // Reads full line (with spaces)

cout << "Hello, " << name;
```

3. Concatenation

```
string a = "Good";
string b = "Morning";
string result = a + " " + b;

cout << result;  // Output: Good Morning
```

4. String Length

```
string word = "example";
cout << word.length();  // Output: 7
```

5. Access Characters

```
string text = "Hello";
cout << text[0];      // Output: H
text[1] = 'a';        // Changes 'e' to 'a'
cout << text;         // Output: Hallo
```

⇒ Useful String Functions

1. length() / size()

Returns the number of characters in the string.

```
string s = "apple";
cout << s.length();  // Output: 5
```

2. empty()

Checks if the string is empty.

```
string s = "";
if (s.empty())
{
    cout << "String is empty";
}
```

3. append()

Adds another string at the end.

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

4. substr(start, length)

Extracts a substring from the string.

```
string s = "Programming";
string sub = s.substr(0, 4);  // Output: "Prog"
```

5. find(substring)

Finds the position of the first occurrence of a substring.

```
string s = "banana";
int pos = s.find("na");  // Output: 2
```

6. replace(pos, len, new_str)

Replaces part of the string.

```
string s = "I like apples";
s.replace(7, 6, "oranges");
cout << s;  // Output: I like oranges
```

7. erase(pos, len)

Removes characters from the string.

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

8. insert(pos, str)

Inserts characters at a given position.

```
string s = "Hello";
s.insert(5, " World");
cout << s;  // Output: Hello World
```

9. compare()

Compares two strings (returns 0 if equal).

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

if (a.compare(b) == 0)
   cout << "Equal";
else
   cout << "Not Equal";
```

6. Introduction to Object-Oriented Programming

Theory Exercise:

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

1. Encapsulation

- **Definition**: Wrapping data (variables) and functions (methods) into a single unit — the **class**.
- **Goal**: Hide the internal details of an object and only expose what is necessary (e.g., through public methods).

Example:

```
class BankAccount
{
private:
   double balance; // Hidden from outside

public:
   void deposit(double amount)
   {
      balance += amount;
   }

   double getBalance()
   {
      return balance;
   }
};
```

2. Abstraction

- **Definition**: Hiding complex implementation details and showing only the essential features.
- **Goal**: Make code easier to use and maintain.

Example:

  - You **use cout** to print, but don’t need to know how it works internally.
  - In your own class, you provide simple methods like startCar() instead of showing all engine processes.

3. Inheritance

- **Definition**: One class (child or derived class) inherits properties and behaviors from another (base class).
- **Goal**: Reuse code and build relationships between classes.

Example:

```
class Person
{
public:
   string name;
};

class Student : public Person
{
public:
   int studentID;
};
```

4. Polymorphism

- **Definition**: "Many forms" – the ability to use the same function name with different behaviors.
- **Types**:
  - **Compile-time** (Function Overloading)
  - **Run-time** (Function Overriding with Virtual Functions)

Example:

```
class Shape
{
public:
    virtual void draw()
    {
        cout << "Drawing shape" << endl;
    }
};

class Circle : public Shape
{
public:
    void draw() override
    {
        cout << "Drawing circle" << endl;
    }
};
```

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

What Are Classes and Objects in C++?

In C++, **classes** and **objects** are fundamental concepts of **Object-Oriented Programming (OOP)**.

Class:

A **class** is a **blueprint or template** for creating objects.
It defines **data members** (variables) and **member functions** (methods) that operate on the data.

Think of a class like a "recipe" or "design".

Object:

An **object** is a **real-world instance** of a class.
It has its own values for the variables defined in the class.

□ Think of an object like a "cake" made from the "recipe" (class).

Syntax of a Class in C++

```
class ClassName
{
  // Access specifier
public:
  // Data members (variables)
  int value;
  // Member functions
  void display()
  {
    cout << "Value is: " << value << endl;
  }
};
```

Example: Class and Object in C++

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

// Define a class
class Car
{
public:
    string brand;
    int year;

    void displayInfo()
    {
        cout << "Brand: " << brand << endl;
        cout << "Year: " << year << endl;
    }
};

int main()
{
    // Create an object of the class Car
    Car myCar;

    // Assign values to the object's data members
    myCar.brand = "Toyota";
    myCar.year = 2022;

    // Call a member function
    myCar.displayInfo();

    return 0;
}
```

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

What is Inheritance in C++?

Inheritance is a core concept of Object-Oriented Programming (OOP) that allows a class (**derived class**) to **inherit**properties (data members) and behaviors (member functions) from another class (**base class**).

Why Use Inheritance?

- **Code reusability**: You don't have to rewrite code for common functionality.
- **Extensibility**: Easily extend or customize behavior in derived classes.
- **Logical hierarchy**: Models real-world relationships (e.g., Car is a type of Vehicle).

Syntax

```
class Base
{
  // base class members
};

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

Example: Inheritance in C++

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

// Base class
class Person
{
public:
    string name;
    int age;

    void displayInfo()
    {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

// Derived class
class Student : public Person
{
public:
    string studentID;

    void displayStudent()
    {
        displayInfo();  // call base class function
        cout << "Student ID: " << studentID << endl;
    }
};

int main()
{
    Student s1;

    // Accessing base class members
    s1.name = "Alice";
    s1.age = 20;

    // Accessing derived class member
    s1.studentID = "S123";

    // Display all info
    s1.displayStudent();

    return 0;
}
```

Types of Inheritance in C++:

Type	**Description**
Single	One base → one derived class
Multiple	One derived class → inherits from multiple base classes
Multilevel	Derived from a derived class
Hierarchical	Multiple derived classes from one base
Hybrid	Combination of two or more types

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

What is Encapsulation in C++?

Encapsulation is one of the fundamental concepts of **Object-Oriented Programming (OOP)**.
It refers to the **bundling of data (variables)** and **functions (methods)** that operate on that data into a **single unit (class)**. It also restricts **direct access** to some of the object's components — which is known as **data hiding**.

Goal of Encapsulation:

- **Protect data** from unauthorized access or modification.
- Control how data is accessed or changed using methods.
- Improve code **security, maintainability, and modularity**.

How is Encapsulation Achieved in C++?

Encapsulation is achieved by:

1. **Declaring data members as private** (cannot be accessed directly outside the class).

2. **Providing public getter/setter functions** to access or update the private data safely.

Example: Encapsulation in C++

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

class Employee
{
private:
   int salary;  //private data member

public:
   //Setter: sets value of salary
   void setSalary(int s)
   {
      if (s > 0)
         salary = s;
      else
         cout << "Invalid salary!" << endl;
   }

   //Getter: returns value of salary
   int getSalary()
   {
      return salary;
   }
};

int main()
{
   Employee emp;

   emp.setSalary(50000);           //Safe access via setter
   cout << "Salary: " << emp.getSalary() << endl;  //Access via getter

   return 0;
}
```

Why Use Encapsulation?

Benefit	Explanation
Data Protection	Prevents accidental or unauthorized access
Controlled Access	Use logic in setters/getters to validate data
Code Maintainability	Internal implementation can change without affecting external code
Modularity	Keeps code organized and easier to manage