Theory Exercise

1. Introduction to C++

A-1:

Procedural Programming (POP): Procedural programming is a programming style where the code is organized into functions or procedures that operate on data.
Code Structure: The program is divided into small parts called functions, and these functions are called in a specific sequence.
Security: Data is less secure, as there is no concept of hiding data.
Examples: Languages like C, Pascal use procedural programming.

Object-Oriented Programming (OOP): Object-Oriented Programming is a style of programming that uses objects and classes to organize code.
Code Structure: The program is made up of objects that interact with each other. Each object has its own data and behavior.
Security: Data is more secure, as access to data can be controlled.
Examples: Languages like C++, Java, Python (OOP), and C# support object-oriented programming.

A-2:

Main Advantages of OOP over POP are as follows:

- **Encapsulation**: Data and functions are bundled together.
- **Reusability**: Use of classes and objects makes code reusable.
- **Inheritance**: New classes can reuse features of existing ones.
- **Polymorphism**: Same function behaves differently based on context.
- **Modularity**: Easier to manage and debug large programs.

A-3:

Steps to Set Up a C++ Development Environment are as follows:

1. **Install a compiler**: e.g., GCC (MinGW for Windows).
2. **Install an IDE or editor**: e.g., Code::Blocks, Dev C++, Visual Studio, or VS Code.
3. **Set path (if needed)**: Add compiler to system PATH.
4. **Create and save a C++ file**: Use .cpp extension.
5. **Compile and run**: Use IDE buttons or terminal commands.

A-4:

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

int main() {
    int num;
    cout << "Enter a number: ";   // Output
    cin >> num;                   // Input
    cout << "You entered: " << num << endl;
    return 0;
}
```

2. Variables, Data Types, and Operators

A-1:

C++ has several **data types** used to store different kinds of values:

- **int** → for whole numbers
  Example: int age = 25;
- **float** → for decimal numbers
  Example: float weight = 55.5;
- **double** → for decimal numbers
  Example: double price = 99.99;
- **char** → for a single character
  Example: char grade = 'A';
- **bool** → for true or false values
  Example: bool isPassed = true;
- **string** (from <string> library) → for words/text
  Example: string name = "Alice";

A-2:

Implicit Conversion : The compiler **automatically** converts one data type to another.
Example:

```
int a = 5;

float b = a;
```

Explicit Conversion : You **manually** convert one data type to another.
Example:

```
float a = 5.5;

int b = (int)a;
```

A-3:

Types of Operators in C++ :

- **Arithmetic Operators**: + - * / %
  Example: int sum = a + b;
- **Relational Operators**: == != > < >= <=
  Example: if (a > b)
- **Logical Operators**: && || !
  Example: if (a > 0 && b > 0)
- **Assignment Operators**: = += -= *= /=
  Example: x += 5; (means x = x + 5;)
- **Increment/Decrement**: ++ --
  Example: i++; or i--;
- **Bitwise Operators**: & | ^ << >>
  (Used for bit-level operations)

A-4:

Constants:
These are fixed values that **do not change** during the program.
Declared using const keyword.
Example: const float PI = 3.14;

Literals:
These are the actual values used directly in the code.
Examples:

- 100 (int literal)
- 'A' (char literal)
- 3.14 (float/double literal)
- "Hello" (string literal)

3. Control Flow Statements

A-1:

Conditional statements are used to make decisions in a program - they let the program choose what to do based on certain conditions.

1. if-else Statement: Used to run code if a condition is **true**, and run something else if it is **false**.

Example:

```
int age = 18;
if (age >= 18) {
    cout << "You can vote.";
} else {
    cout << "You cannot vote.";
}
```

2. switch Statement: Used when you have many conditions to check for **one variable** (like a menu).

Example:

```
int choice = 2;
switch(choice)
{
    case 1: cout << "Option 1"; break;
    case 2: cout << "Option 2"; break;
    default: cout << "Invalid choice";
}
```

A-2:

Difference Between for, while, and do-while Loops

Loops are used to **repeat code**.

Loop	When to Use	Example
for loop	When you know how many times to loop	for(int i = 0; i < 5; i++)
while loop	When you don't know how many times	while(condition)
do-while loop	Same as while, but runs at least once	do { } while(condition);

A-3:

break and continue in Loops :

- **break**: Stop the loop immediately.
- **continue**: Skips the current loop step and moves to the next one.

Example with break:

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) break;
   cout << i << " ";
}
// Output: 1 2
```

Example with continue:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    cout << i << " ";
}
// Output: 1 2 4 5
```

A-4:

Nested Control Structures : This means using one control structure **inside another** (like a loop inside a loop or an if inside a loop).

Example: Nested if

```
int a = 5, b = 10;
if (a < 10) {
    if (b > 5) {
        cout << "Both conditions are true";
    }
}
```

Example: Nested loop

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        cout << i << "," << j << endl;
    }
}
```

4. Functions and Scope

A-1:

A **function** is a block of code that performs a specific task. Instead of writing the same code again and again, we can just call the function whenever needed.

Function Declaration: This tells the compiler that a function **exists**. It includes the function name, return type, and parameters (if any), but **no code body**.

Example:

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

Function Definition: This is where we **write the actual code** for the function.

Example:

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

Function Calling: This is where we **use the function** in our program to get results.

Example:

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

A-2:

Scope means where in the program a variable can be used.

Local Scope:

- A local variable is declared **inside a function** or block.
- It can only be used **within that function**.
- It gets destroyed when the function ends.

Example:

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

Global Scope:

- A global variable is declared **outside all functions**.
- It can be used in **any function** of the program.
- It exists until the program ends.

Example:

```
int x = 100;  // global variable
void show() {
   cout << x;
}
```

A-3:

Recursion means a function **calls itself** to solve a problem.It is used when a task can be broken into smaller similar tasks.

Example:

```
int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

A- 4:

A **function prototype** is a **declaration** of the function that tells the compiler about the function name, return type, and parameters **before the function is actually defined**.

Example:

```
int sum(int, int);  // This is a prototype
```

Why it is used:

- It **lets the compiler know** about the function before its definition.
- It allows us to **define the function later** in the program.
- It helps avoid errors related to missing function declarations.

5. Arrays and Strings

A- 1:

An **array** is a collection of **similar data items** stored at **continuous memory locations**. It allows us to store **multiple values of the same type** (like all integers or all floats) in a single variable name.

Single-Dimensional Array:

- It is like a list or row of values.
- It uses **one index** to access elements.

Example:

```
int marks[5] = {80, 90, 70, 85, 95};
cout << marks[0];  // prints 80
```

Multidimensional Array:

- It is like a **table** (rows and columns).
- It uses **two or more indices** to access elements.

Example :

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

A- 2:

Strings are used to store **text** (like names, sentences). There are **two ways** to handle strings in C++:

Using Character Arrays: Stored as an array of characters ending with '\0'.

Example:

```
char name[10] = "John";
cout << name;  // prints John
```

Using string class : Easier and safer way to handle strings. Need to include #include <string> and use std::string.

Example:
string city = "London";
cout << city;

A- 3:

1D Array (Single Dimensional):

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

2D Array (Multidimensional):

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

A- 4:

String Operations and Functions in C++

In C++, strings are used to store sequences of characters and offer various built-in functions for manipulation.

You can find the number of characters in a string using functions like .length() or .size().

- .len() – Get the number of characters in a string

Characters within the string can be accessed or modified using index notation, such as s[i]. To join two strings, the + operator or .concat() function is commonly used.

- .cat() – Add one string to the end of another

Strings can be compared for equality using == or using .compare().

- .cmp() – To compare two string

6. Introduction to Object-Oriented Programming

A- 1:

Object-Oriented Programming (OOP) :

- OOP is a way of writing programs using **objects**.
- Objects are like real-world things – they have **data** (like color, size) and **actions** (like walk, drive).
- OOP makes code **easier to understand, reuse, and manage**.

A- 2:

Classes and Objects in C++:

- A **class** is like a **blueprint** or plan. It defines what an object can do.
- An **object** is like a **real thing** made using the blueprint (class).

Example:

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

class Vehicle{       // This is a class
public:
   string color;
   void drive() {
      cout << "Car is driving" << endl;
   }
};
```

```
int main() {
    Vehicle v;          // This is an object
    v.color = "Red";
    v.drive();      // Using the object's function
    return 0;
}
```

A- 3:

Inheritance means a class can **get features (data and functions) from another class**.

- It's like a child getting features from parents.

Example:

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

class Vehicle{
public:
    void color() {
        cout << "Color is grey" << endl;
    }
};
class Tata: public Vehicle{
public:
    void model() {
        cout << "Tata Nexon 2025" << endl;
    }
};
```

```
int main() {
    Tata t1;
    t1.color();   // Inherited from Vehicle
    t1.model();  // Tata's own function
    return 0;
}
```

A- 4:

Encapsulation means **hiding details** and only showing what is necessary.

- It helps **protect data** inside a class.
- In C++, it is done by using **private** and **public** keywords.

Example:

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

class Person {
private:
    int age;
public:
    void setAge(int a) {
        if (a > 0) {
            age = a;
        }
    }
    int getAge() {
        return age;
    }
};
```

```
int main() {
    Person p;
    p.setAge(25);         // Set age using function
    cout << p.getAge();   // Get age using function
    return 0;
}
```