Module 3: C++ ASSIGNMENT

LAB EXERCISES:

Exercise 1: First C++ Program - Hello World

Question:

Write a simple C++ program to display the message **"Hello, World!"**.

Objective:

Understand the basic structure of a C++ program, including:

- Preprocessor directive (#include)
- main() function
- Output using cout

Sample Code:

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

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

Exercise 2: Basic Input/Output

Question:

Write a C++ program that accepts user input for their **name** and **age**, and then displays a personalized greeting.

Sample Code:

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

```
main() {
  string name;
  int age;

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

  cout << "Hello, " << name << "! You are " << age << " years old." << endl;

}
```

Exercise 3: POP vs. OOP Comparison Program

Question:

Write two C++ programs to calculate the **area of a rectangle**:

- One using **Procedural Programming (POP)**
- One using **Object-Oriented Programming (OOP)** with a class

(a) POP Example:

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

int main() {
  double length, width, area;

  cout << "Enter length: ";
  cin >> length;

  cout << "Enter width: ";
  cin >> width;

  area = length * width;
  cout << "Area of rectangle (POP): " << area << endl;
  return 0;
}
```

(b) OOP Example:

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

class Rectangle {
public:
  double length, width;

  void input() {
    cout << "Enter length: ";
    cin >> length;
    cout << "Enter width: ";
    cin >> width;
  }

  double area() {
    return length * width;
  }

  void display() {
    cout << "Area of rectangle (OOP): " << area() << endl;
  }
};

int main() {
  Rectangle rect;
  rect.input();
  rect.display();

  return 0;
}
```

Exercise 4: Setting Up Development Environment

Question:

Write a C++ program that asks the user for **two numbers** and displays their **sum**. Ensure this program is created and run in a C++ IDE like **Dev C++** or **Code::Blocks**.

Objective:

Learn how to:

- Install and configure a C++ IDE
- Write, compile, and run a simple program

Sample Code:

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

int main() {
  int num1, num2, sum;

  cout << "Enter first number: ";
  cin >> num1;

  cout << "Enter second number: ";
  cin >> num2;

  sum = num1 + num2;
  cout << "The sum is: " << sum << endl;

  return 0;
}
```

THEORY EXERCISES

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

Feature	**Procedural Programming (POP)**	**Object-Oriented Programming (OOP)**
Approach	Follows a top-down approach	Follows a bottom-up approach
Focus	Focuses on functions	Focuses on objects
Data Handling	Data is global and accessible to all functions	Data is encapsulated inside objects
Reusability	Low code reusability	High code reusability through inheritance
Security	Less secure, as data is exposed	More secure, due to data hiding (encapsulation)
Examples	C, early versions of BASIC	C++, Java, Python (OOP features)

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

Advantages of OOP:

1. **Encapsulation:**
2. Combines data and functions into a single unit (class) and hides internal details. This improves data security.
3. **Reusability (Inheritance):**

Classes can inherit properties and methods from other classes, allowing code reuse and reducing redundancy.

4. **Modularity:**

Code is divided into objects, making it easier to manage, update, and debug.

5. **Polymorphism:**

The same function or operator behaves differently based on the context (function overloading, operator overloading).

6. **Abstraction:**

Hides complex implementation details and shows only essential features to the user.

7. **Maintainability:**

OOP makes it easier to manage and maintain code due to better organization and modularity.

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

Steps to Set Up a C++ Development Environment:

1. **Install an IDE or Text Editor:**

1. Common IDEs: Dev C++, Code::Blocks, Visual Studio, or Eclipse.
2. Alternatively, use a text editor like VS Code with a C++ plugin.

2. **Install a C++ Compiler:**

1. For Windows: Install **MinGW** or **TDM-GCC**.

3. **Configure the IDE:**

1. Link the installed compiler to the IDE (in settings or preferences).
2. Set the directory paths if required.

4. **Write and Save the Program:**

1. Open the IDE and write your .cpp file.

5. **Compile the Program:**

1. Use the "Build" or "Compile" option to convert your code into an executable.

6. **Run the Program:**

1. Use the "Run" button or terminal command to execute your code.

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

Input Operation:

- **cin** is used to accept input from the user.

Output Operation:

- **cout** is used to display output to the console.

Example:

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

int main() {
  string name;
  int age;

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

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

  cout << "Hello, " << name << "! You are " << age << " years old." << endl;

  return 0;
}
```

Here is a **well-structured lab assignment** on **Variables, Data Types, and Operators** in C++, complete with **questions, objectives**, and **sample code** for each exercise.

2.VARIABLES, DATA TYPES, AND OPERATORS

Exercise 1: Variables and Constants

Question:

Write a C++ program that demonstrates the use of **variables** and **constants**. Create variables of different data types (e.g., int, float, char, bool) and perform operations on them.

Objective:

Understand the difference between variables (changeable values) and constants (fixed values).

Sample Code:

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

int main() {
  // Variable declaration
  int age = 25;
  float height = 5.9;
  char grade = 'A';
  bool passed = true;

  // Constant declaration
  const float PI = 3.14159;

  // Displaying values
  cout << "Age: " << age << endl;
  cout << "Height: " << height << " feet" << endl;
  cout << "Grade: " << grade << endl;
  cout << "Passed: " << passed << endl;
  cout << "Value of PI (constant): " << PI << endl;

  return 0;
}
```

Exercise 2: Type Conversion

Question:

Write a C++ program that performs both **implicit** and **explicit** type conversions and prints the results.

Objective:

Practice type casting in C++ and understand how data types are converted automatically (implicit) or manually (explicit).

Sample Code:

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

int main() {
  // Implicit Conversion
  int a = 10;
  float b = a;  // int to float (implicit)
  cout << "Implicit Conversion (int to float): " << b << endl;

  // Explicit Conversion
  float x = 9.87;
  int y = (int)x;  // float to int (explicit)
  cout << "Explicit Conversion (float to int): " << y << endl;

  return 0;
}
```

Exercise 3: Operator Demonstration

Question:

Write a C++ program that demonstrates the use of:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators

Objective:

Reinforce understanding of different types of operators used in C++ programming.

Sample Code:

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

int main() {
  int a = 10, b = 3;

  // Arithmetic Operators
  cout << "Arithmetic Operators:" << endl;
  cout << "a + b = " << a + b << endl;
  cout << "a - b = " << a - b << endl;
  cout << "a * b = " << a * b << endl;
  cout << "a / b = " << a / b << endl;
  cout << "a % b = " << a % b << endl;

  // Relational Operators
  cout << "\nRelational Operators:" << endl;
  cout << "a == b: " << (a == b) << endl;
  cout << "a != b: " << (a != b) << endl;
  cout << "a > b: " << (a > b) << endl;
  cout << "a < b: " << (a < b) << endl;

  // Logical Operators
  bool x = true, y = false;
  cout << "\nLogical Operators:" << endl;
  cout << "x && y: " << (x && y) << endl;
  cout << "x || y: " << (x || y) << endl;
  cout << "!x: " << (!x) << endl;

  // Bitwise Operators
  cout << "\nBitwise Operators:" << endl;
  cout << "a & b: " << (a & b) << endl;
  cout << "a | b: " << (a | b) << endl;
  cout << "a ^ b: " << (a ^ b) << endl;
  cout << "a << 1: " << (a << 1) << endl;
  cout << "a >> 1: " << (a >> 1) << endl;

  return 0;
```

}

THEORY EXERCISES

Topic: Variables, Data Types, Type Conversion, and Operators

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

C++ provides several **data types**, broadly classified into:

1. Primary (Built-in) Data Types:

- **int** – used for integers
- Example: int age = 25;
- **float** – used for floating-point numbers

Example: float height = 5.9;

- **double** – used for double-precision floating-point numbers

Example: double pi = 3.14159;

- **char** – used for single characters

Example: char grade = 'A';

- **bool** – used for boolean values (true or false)

Example: bool passed = true;

2. Derived Data Types:

- **Arrays** – collection of similar data types

Example: int numbers[5];

- **Pointers** – store memory address

Example: int* ptr = &age;

- **Functions** – block of reusable code

3. User-defined Data Types:

- **Structures (struct)**

- **Unions (union)**
- **Enumerations (enum)**
- **Classes (class)**

4. Void Type:

- Represents no value or no return type

Example: void display();

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

Type conversion refers to changing a variable from one data type to another.

Type	**Description**	**Example**
Implicit Conversion (Type Promotion)	Automatically done by the compiler when converting a smaller data type to a larger one.	int a = 10; float b = a; // int to float
Explicit Conversion (Type Casting)	Manually done by the programmer using casting syntax.	float x = 9.75; int y = (int)x; // float to int

Key Difference:

- Implicit is **automatic** and done safely by the compiler.
- Explicit is **manual** and may lead to data loss.

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

C++ supports various **operators**:

Operator Type	**Description**	**Example**
Arithmetic Operators	Perform mathematical operations	+, -, *, /, %Example: a + b
Relational Operators	Compare values	==, !=, <, >, <=, >=Example: a < b

Logical Operators	Combine multiple conditions	&&, `
Bitwise Operators	Operate on bits	&, `
Assignment Operators	Assign values	=, +=, -=, *=, /=, %=, etc.Example: a += 5
Increment/Decrement	Increase or decrease values	++, --Example: a++
Conditional (Ternary) Operator	Short-hand for if-else	condition ? expr1 : expr2Example: (a > b) ? a : b
Comma Operator	Separates multiple expressions	a = (x = 5, x + 10);

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

Constants:

Constants are **fixed values** that do not change during the execution of the program. Declared using the const keyword.

Example:

const float PI = 3.14159;

- You cannot modify PI later in the program.
- Used to define fixed values like math constants, limits, or configuration settings.

Literals:

Literals are the **actual values assigned** to variables or constants.

Types of Literals:

- **Integer Literal:** 10, -5
- **Float Literal:** 3.14, 0.001f
- **Character Literal:** 'A', 'z'

- **String Literal:** "Hello", "C++"
- **Boolean Literal:** true, false

Example:

```
int age = 21;         // 21 is an integer literal
char grade = 'A';     // 'A' is a character literal
string name = "Tom";  // "Tom" is a string literal
```

Purpose:

- Enhance code clarity
- Prevent accidental value changes
- Represent fixed data values

Here is the **well-organized lab assignment** for the topic **Control Flow Statements** in C++. It includes clearly defined **exercises, objectives**, and **sample code** for each task.

3. CONTROL FLOW STATEMENTS

Exercise 1: Grade Calculator

Question:

Write a C++ program that takes a student's marks as input and calculates the grade based on the following conditions using **if-else**:

- Marks >= 90 → Grade A
- Marks >= 80 → Grade B
- Marks >= 70 → Grade C
- Marks >= 60 → Grade D
- Below 60 → Grade F

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

```
int main() {
  int marks;
  cout << "Enter your marks: ";
  cin >> marks;

  if (marks >= 90)
     cout << "Grade: A" << endl;
  else if (marks >= 80)
     cout << "Grade: B" << endl;
  else if (marks >= 70)
     cout << "Grade: C" << endl;
  else if (marks >= 60)
     cout << "Grade: D" << endl;
  else
     cout << "Grade: F" << endl;

  return 0;
}
```

Exercise 2: Number Guessing Game

Question:

Write a C++ program that asks the user to guess a number between 1 and 100. Use a **while loop** to allow multiple attempts. Give hints if the guess is too high or too low. Stop when the correct number is guessed.

```
#include <iostream>
#include <cstdlib>  // for rand() and srand()
#include <ctime>    // for time()
using namespace std;

int main() {
  srand(time(0));  // seed random number generator
  int secret = rand() % 100 + 1;
  int guess;
  cout << "Guess a number between 1 and 100: ";

  while (true) {
    cin >> guess;

    if (guess == secret) {
      cout << "Congratulations! You guessed it right." << endl;
      break;
    } else if (guess < secret) {
      cout << "Too low! Try again: ";
    } else {
      cout << "Too high! Try again: ";
    }
  }

  return 0;
}
```

Exercise 3: Multiplication Table

Question:

Write a C++ program to display the multiplication table of a given number using a **for loop**.

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

int main() {
  int num;
  cout << "Enter a number to display its multiplication table: ";
  cin >> num;

  for (int i = 1; i <= 10; ++i) {
    cout << num << " x " << i << " = " << num * i << endl;
  }
  return 0;
}
```

Exercise 4: Nested Control Structures

Write a program that prints a **right-angled triangle** using * symbols. Use **nested loops**.

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

int main() {
  int rows;
  cout << "Enter the number of rows: ";
  cin >> rows;

  for (int i = 1; i <= rows; ++i) {
     for (int j = 1; j <= i; ++j) {
        cout << "* ";
     }
     cout << endl;
  }

  return 0;
}
```

Here is the **Theory Exercise** section for the topic **Control Flow Statements** in C++. Each question includes clear explanations and examples suitable for assignments or lab records.

THEORY EXERCISES

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

Conditional statements allow the program to make decisions based on certain conditions. They control the flow of execution depending on whether a condition is true or false.

a) if-else Statement:

Used to execute a block of code when a condition is true, and optionally another block when it is false.

Syntax:

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

Example:

```
int marks = 85;
if (marks >= 50) {
  cout << "Pass";
} else {
  cout << "Fail";
}
```

b) switch Statement:

Used to select one of many code blocks to be executed based on the value of a variable.

Syntax:

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

Example:

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

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

All three are **loop control structures** used for repetition, but they differ in how and when they check the condition.

Loop Type	Condition Check	Executes At Least Once?	Use Case
for	Before loop body	No	Known number of iterations
while	Before loop body	No	Condition-controlled loops
do-while	After loop body	Yes	Run at least once regardless of condition

Example of each:

- **for loop**

```
for (int i = 1; i <= 5; i++) {
  cout << i << " ";
}
```

- **while loop**

```
int i = 1;
while (i <= 5) {
  cout << i << " ";
  i++;
}
```

- **do-while loop**

```
int i = 1;
do {
  cout << i << " ";
  i++;
} while (i <= 5);
```

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

break Statement:

- Used to **exit** a loop or switch statement **immediately**.
- Often used when a specific condition is met.

Example:

```
for (int i = 1; i <= 10; i++) {
  if (i == 5) break;
  cout << i << " ";
}
// Output: 1 2 3 4
```

continue Statement:

- Skips the **rest of the loop** for the current iteration and continues with the next iteration.

Example:

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

4. Explain nested control structures with an example.

Nested control structures refer to placing one control structure (like if, for, or while) **inside another**.

Common Use Cases:

- Nested if statements for multiple condition checks.
- Nested loops for multi-level tasks like printing patterns.

Example – Nested for loop (printing a triangle):

```
int rows = 5;
for (int i = 1; i <= rows; i++) {
  for (int j = 1; j <= i; j++) {
     cout << "* ";
  }
  cout << endl;
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

4 – FUNCTIONS AND SCOPE

Exercise 1: Simple Calculator Using Functions

Question:

Write a C++ program that defines functions for basic arithmetic operations:

- Addition
- Subtraction
- Multiplication
- Division

The main function should take user input and call the appropriate function based on the selected operation.

Objective:

Practice defining and calling user-defined functions in C++.

Sample Code:

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

float add(float a, float b) {
  return a + b;
}

float subtract(float a, float b) {
  return a - b;
}

float multiply(float a, float b) {
  return a * b;
}

float divide(float a, float b) {
  if (b != 0)
     return a / b;
  else {
     cout << "Error: Division by zero!" << endl;
     return 0;
  }
}
```

```
int main() {
  float num1, num2;
  char op;

  cout << "Enter two numbers: ";
  cin >> num1 >> num2;
  cout << "Enter operation (+, -, *, /): ";
  cin >> op;

  switch (op) {
     case '+': cout << "Result: " << add(num1, num2); break;
     case '-': cout << "Result: " << subtract(num1, num2); break;
     case '*': cout << "Result: " << multiply(num1, num2); break;
     case '/': cout << "Result: " << divide(num1, num2); break;
     default: cout << "Invalid operation!";
  }

  return 0;
}
```

Exercise 2: Factorial Calculation Using Recursion

Question:

Write a C++ program to calculate the **factorial of a number** using a recursive function.

Objective:

Understand and implement **recursion** in C++ functions.

Sample Code:

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

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

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

  if (num < 0)
     cout << "Factorial not defined for negative numbers.";
  else
     cout << "Factorial of " << num << " = " << factorial(num);

  return 0;
}
```

Exercise 3: Variable Scope

Question:

Write a C++ program to demonstrate the difference between **local and global variables** using functions.

Objective:

Reinforce the concept of **variable scope** in C++.

Sample Code:

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

int globalVar = 100; // Global variable

void showScope() {
  int localVar = 50; // Local variable
  cout << "Inside function - Local variable: " << localVar << endl;
  cout << "Inside function - Global variable: " << globalVar << endl;
}

int main() {
  int localVar = 25; // Local variable in main
  cout << "Inside main - Local variable: " << localVar << endl;
  cout << "Inside main - Global variable: " << globalVar << endl;

  showScope();

  return 0;
}
```

Output Example:

```
Inside main - Local variable: 25
Inside main - Global variable: 100
Inside function - Local variable: 50
Inside function - Global variable: 100
```

THEORY EXERCISES

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

A **function** in C++ is a block of code that performs a specific task. Functions help break down a program into smaller, manageable, and reusable parts.

Function Components:

- **Declaration (Prototype):**
- Tells the compiler about the function name, return type, and parameters **before** it is used.
- Example:

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

- **Definition:**

Contains the actual body of the function.

Example:

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

- **Calling a Function:**

The function is used in main() or another function.

Example:

```
int result = add(5, 3);
```

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

Scope refers to the region in a program where a variable can be accessed.

Local Scope:

- A variable declared inside a function or block.
- Accessible **only** within that function or block.

Example:

```
void func() {
  int x = 10; // local to func
}
```

Global Scope:

- A variable declared **outside** all functions.
- Accessible **throughout** the entire program.

Example:

```
int x = 10; // global
```

```
void func() {
  cout << x; // accessible
}
```

Feature	Local Variable	Global Variable
Declared in	Inside a function/block	Outside all functions
Scope	Limited to the function	Entire program
Lifetime	Until function ends	Till the program ends

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

Recursion is a process in which a function calls **itself** directly or indirectly to solve a problem.

Every recursive function must have:

- A **base condition** to stop recursion.
- A **recursive call** that moves toward the base condition.

Example: Factorial using Recursion

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

Explanation:

- factorial(5) → 5 * factorial(4) → 5 * 4 * 3 * 2 * 1 = 120

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

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

- The function name
- Its return type
- Number and type of parameters

Syntax:

```
returnType functionName(parameterList);
```

Example:

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

Here’s a **well-structured Lab Assignment** for the topic **Arrays and Strings** in C++. It includes each lab exercise with clear descriptions, objectives, and sample code for practice and submission.

5. ARRAYS AND STRINGS

SExercise 1: Array Sum and Average

Question:

Write a C++ program that:

- Accepts n integer values from the user into an array
- Calculates the **sum** and **average**
- Displays the results

Objective:

Understand basic array input, traversal, and arithmetic operations.

Sample Code:

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

```
int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;

  int arr[100], sum = 0;
  for (int i = 0; i < n; i++) {
     cout << "Enter element " << i + 1 << ": ";
     cin >> arr[i];
     sum += arr[i];
  }

  float average = (float)sum / n;

  cout << "Sum = " << sum << endl;
  cout << "Average = " << average << endl;

  return 0;
}
```

Exercise 2: Matrix Addition

Question:

Write a C++ program to **add two 2x2 matrices** entered by the user and display the result.

Objective:

Practice using two-dimensional arrays for matrix operations.

Sample Code:

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

int main() {
  int a[2][2], b[2][2], sum[2][2];
  cout << "Enter elements of first 2x2 matrix:\n";
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < 2; j++) {
      cout << "a[" << i << "][" << j << "]: ";
      cin >> a[i][j];
    }

  cout << "Enter elements of second 2x2 matrix:\n";
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < 2; j++) {
      cout << "b[" << i << "][" << j << "]: ";
      cin >> b[i][j];
    }

  cout << "Resultant matrix (a + b):\n";
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      sum[i][j] = a[i][j] + b[i][j];
      cout << sum[i][j] << " ";
    }
    cout << endl;
  }

  return 0;
}
```

Exercise 3: String Palindrome Check

Question:

Write a C++ program that checks whether a **given string** is a **palindrome** or not.

(A palindrome reads the same forwards and backwards, like "madam", "racecar")

Objective:

Practice string manipulation and comparison.

Sample Code:

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

int main() {
  char str[100];
  cout << "Enter a string: ";
  cin >> str;

  int len = strlen(str);
  bool isPalindrome = true;

  for (int i = 0; i < len / 2; i++) {
     if (str[i] != str[len - 1 - i]) {
        isPalindrome = false;
        break;
     }
  }

  if (isPalindrome)
     cout << "The string is a palindrome." << endl;
  else
     cout << "The string is not a palindrome." << endl;

  return 0;
}
```

Here’s a clear and concise **Theory Exercise** section for the topic **Arrays and Strings** in C++. Each question is explained with examples, suitable for assignment or study notes.

THEORY EXERCISES

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

Arrays in C++ are collections of elements of the same data type stored in contiguous memory locations. Arrays help to store multiple values under a single variable name, accessed by indices.

- **Single-Dimensional Array (1D Array):**
- A linear list of elements accessed by one index.
- Example:

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

Access element: arr[2] gives 3.

- **Multi-Dimensional Array:**

Arrays with more than one dimension (like matrices). Accessed using multiple indices.

Example of 2D array (matrix):

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

Access element: matrix[1][2] gives 6.

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

Strings in C++ can be handled in two main ways:

- **C-Style Strings:**

Character arrays terminated by a null character '\0'.

Example:

```
char str[10] = "hello";
```

Functions like strlen(), strcpy(), strcmp() from <cstring> are used for operations.

- **C++ String Class:**

The std::string class from the C++ Standard Library allows easier string handling.

Example:

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

int main() {
  string s = "hello";
  cout << s.length(); // prints 5
  s += " world";
  cout << s; // prints "hello world"
}
```

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

Arrays can be initialized at the time of declaration in C++.

- **1D Array Initialization:**

```
int numbers[5] = {10, 20, 30, 40, 50};
// Partial initialization
int nums[5] = {1, 2};  // remaining elements are 0 by default
```

- **2D Array Initialization:**

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

// Partial initialization: remaining elements will be zero
```

```
int mat[2][3] = { {1, 2}, {3} }; // rest elements = 0
```

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

Common string operations in C++ (using C-style strings and string class):

Operation	**C-style String Example**	**C++ String Class Example**
Length	strlen(str)	str.length() or str.size()
Copy	strcpy(dest, src)	str2 = str1;
Concatenation	strcat(str1, str2)	str1 + str2 or str1.append(str2)
Comparison	strcmp(str1, str2)	str1 == str2
Find substring	strstr(str, substr)	str.find("substring")
Convert case	Custom loop (no standard func)	Use <algorithm> with transform

Code:

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

int main() {
    char str1[20] = "Hello";
    char str2[20];
    strcpy(str2, str1);    // Copy str1 to str2
    cout << "Copied string: " << str2 << endl;

    strcat(str1, " World"); // Concatenate
    cout << "Concatenated string: " << str1 << endl;


    return 0;
}
```

6. INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Objective: Understand basic OOP concepts including class definition, encapsulation, and inheritance.

Exercise 1: Class for a Simple Calculator

Question:

Define a class Calculator with member functions for:

- Addition
- Subtraction
- Multiplication
- Division

Create objects of the class and call these functions to perform calculations.

Objective:

Learn how to create and use classes and objects in C++.

Sample Code:

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

class Calculator {
public:
  float add(float a, float b) {
     return a + b;
  }
  float subtract(float a, float b) {
        return a - b;
    }
    float multiply(float a, float b) {
        return a * b;
    }
    float divide(float a, float b) {
        if (b != 0)
            return a / b;
        else {
            cout << "Division by zero error!" << endl;
            return 0;
        }
    }
};

int main() {
    Calculator calc;
    float x = 10, y = 5;

    cout << "Add: " << calc.add(x, y) << endl;
    cout << "Subtract: " << calc.subtract(x, y) << endl;
    cout << "Multiply: " << calc.multiply(x, y) << endl;
    cout << "Divide: " << calc.divide(x, y) << endl;

    return 0;
}
```

Exercise 2: Class for Bank Account (Encapsulation)

Question:

Create a class BankAccount with:

- Private data member: balance
- Public member functions: deposit(), withdraw(), and getBalance()

Ensure that the balance cannot be accessed directly from outside the class.

Objective:

Understand encapsulation by controlling access to class members.

Sample Code:

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

class BankAccount {
private:
  double balance;

public:
  BankAccount() {
    balance = 0.0;
  }

  void deposit(double amount) {
    if (amount > 0)
      balance += amount;
    else
      cout << "Invalid deposit amount." << endl;
  }

  void withdraw(double amount) {
    if (amount > 0 && amount <= balance)
      balance -= amount;
    else
      cout << "Invalid or insufficient funds for withdrawal." << endl;
  }

  double getBalance() {
    return balance;
  }
};

int main() {
  BankAccount acc;
  acc.deposit(500);
  acc.withdraw(200);
  cout << "Current balance: $" << acc.getBalance() << endl;

  return 0;
}
```

Exercise 3: Inheritance Example

Question:

Implement a base class Person with data members name and age.

Derive two classes Student and Teacher from Person with additional data members like studentID and subject respectively.

Demonstrate reusability and inheritance by creating objects and accessing members.

Objective:

Learn the concept and syntax of inheritance in C++.

Sample Code:

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

class Person {
public:
  string name;
  int age;

  void inputPerson() {
     cout << "Enter name: ";
     getline(cin, name);
     cout << "Enter age: ";
     cin >> age;

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

class Student : public Person {
public:
  int studentID;

  void inputStudent() {
     inputPerson();
     cout << "Enter Student ID: ";
     cin >> studentID;

  }

  void displayStudent() {
     displayPerson();
     cout << "Student ID: " << studentID << endl;
  }
};

class Teacher : public Person {
public:
  string subject;

  void inputTeacher() {
     inputPerson();
     cout << "Enter subject: ";
     getline(cin, subject);
  }

  void displayTeacher() {
     displayPerson();
     cout << "Subject: " << subject << endl;
  }
};

int main() {
  Student stud;
  Teacher teach;

  cout << "--- Enter Student Details ---" << endl;
  stud.inputStudent();

  cout << "\n--- Enter Teacher Details ---" << endl;
  teach.inputTeacher();

  cout << "\n--- Student Details ---" << endl;
  stud.displayStudent();

  cout << "\n--- Teacher Details ---" << endl;
  teach.displayTeacher();

  return 0;
}
```

THEORY EXERCISES

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

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**, which are instances of classes. The main concepts of OOP include:

- **Class:** A blueprint or template for creating objects, defining attributes (data members) and behaviors (member functions).
- **Object:** An instance of a class that contains actual data and can use the class functions.

- **Encapsulation:** Bundling of data and functions that operate on the data into a single unit (class), and restricting direct access to some of the object’s components.
- **Inheritance:** Mechanism by which one class (derived class) inherits properties and behaviors from another class (base class), enabling code reuse.
- **Polymorphism:** Ability of functions or methods to behave differently based on the object that invokes them, often achieved via function overloading or overriding.
- **Abstraction:** Hiding the complex implementation details and showing only the necessary features of an object.

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

- **Class:** A user-defined data type in C++ that encapsulates data members and member functions. It serves as a blueprint for creating objects.
- **Object:** A variable of a class type that holds data and can use the class’s member functions.

Example:

```
class Car {
public:
  string brand;
  int year;

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

int main() {
  Car car1;
  car1.brand = "Toyota";
  car1.year = 2020;
  car1.display();
  return 0;
}
```

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

Inheritance is a feature in C++ where a new class (derived class) inherits attributes and behaviors (data members and member functions) from an existing class (base class). It promotes code reusability and establishes a relationship between classes.

Example:

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

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

int main() {
  Student s;
  s.name = "Alice";
  s.age = 20;
  s.studentID = 101;
  cout << s.name << " is " << s.age << " years old, Student ID: " << s.studentID
<< endl;
  return 0;
}
```

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

Encapsulation is the concept of wrapping data (variables) and methods (functions) that operate on data into a single unit or class, and restricting access to some of the object's components to protect data integrity.

In C++, encapsulation is achieved using:

- **Access Specifiers:** private, protected, and public control the visibility of class members.
- private members are accessible only within the class.
- public members are accessible from outside the class.
- **Getter and Setter Functions:** Public functions that provide controlled access to private data members.

Example:

```
class BankAccount {
private:
  double balance;

public:
  void deposit(double amount) {
     if (amount > 0)
        balance += amount;
  }

  double getBalance() {
     return balance;
  }

  BankAccount() {
     balance = 0;
  }
};
```