**1. Introduction to C++**

**THEORY EXERCISE:**

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

| **Aspect** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Definition** | A programming paradigm based on functions/procedures to operate on data. | A programming paradigm based on objects, which encapsulate data and behavior. |
| **Basic Concept** | Focuses on **functions** (procedures) to perform operations on data. | Focuses on **objects** that combine both data and functions (methods). |
| **Approach** | **Top-down approach** (start with main procedure, break into sub-procedures). | **Bottom-up approach** (start with objects, then build systems by interaction). |
| **Data Handling** | Data is separate from functions; functions operate on data. | Data and functions are bundled together inside objects. |
| **Access Control** | No strict control – data can be accessed freely by functions. | Provides **encapsulation** – access to data can be restricted using access modifiers (private, protected, public). |
| **Reusability** | Code reuse is mainly through functions. | Code reuse is through **classes and inheritance**. |
| **Security** | Less secure – since data is exposed globally and can be modified by any function. | More secure – encapsulation and data hiding protect sensitive information. |
| **Examples of Languages** | C, Pascal, FORTRAN, BASIC | C++, Java, Python, C#, Ruby |
| **Real-World Analogy** | Like a recipe book: each recipe (function) describes steps to manipulate ingredients (data). | Like real-world objects: e.g., a “Car” object has properties (color, speed) and behaviors (drive, brake). |

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

**Advantages of OOP over POP**

| **Advantage** | **OOP** | **POP (Procedural-Oriented Programming)** |
| --- | --- | --- |
| **Modularity** | Programs are divided into classes and objects, making them more organized and modular. | Programs are divided into functions, but data and functions are separate, reducing modularity. |
| **Code Reusability** | Supports **inheritance** – existing classes can be reused and extended. | Limited reusability – functions can be reused, but code often needs rewriting. |
| **Data Security** | Uses **encapsulation** and **access modifiers** (private, public, protected) to protect data. | Data is often global and can be accessed/modified by any function. |
| **Scalability & Maintainability** | Easier to scale large applications; objects can be modified without affecting other parts. | Difficult to manage in large projects as functions and data become harder to track. |
| **Flexibility (Polymorphism)** | Allows **method overloading/overriding**, enabling the same operation to behave differently. | No polymorphism – functions have fixed behavior and names. |
| **Real-World Mapping** | Models real-world entities (Car, Bank, Student, etc.) making programs intuitive. | No direct real-world mapping; focuses only on functions and logic. |
| **Extensibility** | New features can be added by extending existing classes. | Adding new features often requires rewriting major parts of the code. |

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

**Steps to Set Up a C++ Development Environment:**

1. **Install a C++ Compiler**
   * Most common: **GCC (g++)** for Linux/Mac, **MinGW** or **MSVC** for Windows.
   * The compiler translates C++ code into machine code.
2. **Install an Editor or IDE**
   * Editors: VS Code, Sublime Text, Notepad++.
   * IDEs (Integrated Development Environment): Code::Blocks, CLion, Dev-C++, Visual Studio.
   * IDEs provide debugging, code completion, and project management features.
3. **Set Up PATH (if needed)**
   * On Windows, add compiler path (like MinGW’s bin folder) to the system PATH environment variable.
   * This allows running g++ from the terminal/command prompt.
4. **Write Your First Program**
   * Create a file like hello.cpp with a simple C++ program.
5. **Compile the Program**
   * Open terminal/command prompt and run:
   * g++ hello.cpp -o hello
6. **Run the Program**
   * Execute the compiled file:
   * ./hello # Linux/Mac
   * hello.exe # Windows

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

**Main Input/Output (I/O) Operations in C++**

C++ uses the **iostream** library for input and output operations.

**1. Output Operation (cout)**

* Used to display (output) data on the screen.
* Works with the **insertion operator (<<)**.

**Example:**

#include <iostream>

using namespace std;

int main() {

int age = 21;

cout << "Hello, welcome to C++!" << endl;

cout << "Your age is: " << age << endl;

return 0;

}

**Output:**

Hello, welcome to C++!

Your age is: 21

**2. Input Operation (cin)**

* Used to take (input) data from the user.
* Works with the **extraction operator (>>)**.

**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;

}

**Input:**

John

25

**Output:**

Hello John, you are 25 years old.

**3. Other Useful I/O Functions**

* endl → moves cursor to next line (like \n).
* getline(cin, variable) → reads a full line including spaces.

**Example with getline():**

string fullName;

cout << "Enter full name: ";

getline(cin, fullName);

cout << "Welcome, " << fullName << endl;

**2. Variables, Data Types, and Operators**

**THEORY EXERCISE:**

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

**Data Types in C++**

| **Category** | **Data Type** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Basic (Fundamental)** | int | Stores integers (whole numbers). | int age = 25; |
|  | float | Stores single-precision decimal numbers (6–7 digits). | float pi = 3.14; |
|  | double | Stores double-precision decimal numbers (15–16 digits). | double bigPi = 3.14159265359; |
|  | char | Stores a single character. | char grade = 'A'; |
|  | bool | Stores logical values: true or false. | bool isPassed = true; |
|  | void | Represents “no value” (used in functions). | void display(); |
| **Derived** | Arrays | Collection of elements of the same type. | int numbers[5] = {1,2,3,4,5}; |
|  | Pointers | Stores memory addresses of variables. | int x=10; int \*ptr=&x; |
|  | References | Alias for another variable. | int a=5; int &b=a; |
|  | Functions | Blocks of code that return values. | int add(int x, int y); |
| **User-Defined** | struct | Groups different data types. | struct Student {int id; char name[20];}; |
|  | class | Defines objects with data + functions. | class Car {public: string model; void drive();}; |
|  | enum | Defines a set of named constants. | enum Week {Mon, Tue, Wed}; |
|  | typedef / using | Creates a new name for a data type. | typedef unsigned int uint; |

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

| **Aspect** | **Implicit Type Conversion (Type Casting)** | **Explicit Type Conversion (Type Casting)** |
| --- | --- | --- |
| **Definition** | Also called **Type Promotion** or **Type Casting**. Conversion is done **automatically by the compiler** without programmer intervention. | Also called **Type Casting**. Conversion is done **manually by the programmer** using casting operators. |
| **Control** | Compiler-controlled. | Programmer-controlled. |
| **When it Happens** | When a smaller data type is assigned to a larger data type (to prevent data loss). | When programmer explicitly forces a conversion, even if data loss may occur. |
| **Syntax** | No special syntax – happens automatically. | (newType) value or static\_cast<newType>(value) |
| **Risk of Data Loss** | Usually safe (but may cause precision loss, e.g., float to int). | May cause data loss if types are incompatible. |
| **Example** | int x = 5; float y = x; // int → float (automatic) | float pi = 3.14; int num = (int)pi; // explicit cast ORcpp int num = static\_cast<int>(pi); |

**EXAMPLES:**

**1. Implicit Type Conversion (Type Promotion)**

Happens **automatically** by the compiler.

#include <iostream>

using namespace std;

int main() {

int num = 10; // integer

double result;

result = num; // implicit conversion (int → double)

cout << "Result = " << result << endl;

return 0;

}

**Output:**

Result = 10

Here, int (10) is automatically converted into double (10.0).

**2. Explicit Type Conversion (Type Casting)**

Done **manually** by the programmer.

#include <iostream>

using namespace std;

int main() {

double pi = 3.14159;

int num;

num = (int)pi; // C-style casting

cout << "After C-style cast: " << num << endl;

num = static\_cast<int>(pi); // C++ style casting

cout << "After static\_cast: " << num << endl;

return 0;

}

**Output:**

After C-style cast: 3

After static\_cast: 3

Here, double (3.14159) is **explicitly converted** into int (3), losing the decimal part.

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

**1. Arithmetic Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 - 3 | 2 |
| \* | Multiplication | 5 \* 3 | 15 |
| / | Division | 10 / 3 | 3 |
| % | Modulus (remainder) | 10 % 3 | 1 |

**2. Relational Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | 5 == 5 | true |
| != | Not equal to | 5 != 3 | true |
| > | Greater than | 5 > 3 | true |
| < | Less than | 5 < 3 | false |
| >= | Greater or equal | 5 >= 5 | true |
| <= | Less or equal | 3 <= 5 | true |

**3. Logical Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND | true && false | false |
| || | Logical OR | True or false | true |
| ! | Logical NOT | !true | false |

**4. Assignment Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| = | Assign value | x = 10 | x = 10 |
| += | Add & assign | x += 5 | x = x + 5 |
| -= | Subtract & assign | x -= 3 | x = x - 3 |
| \*= | Multiply & assign | x \*= 2 | x = x \* 2 |
| /= | Divide & assign | x /= 2 | x = x / 2 |
| %= | Modulus & assign | x %= 3 | x = x % 3 |

**5. Increment/Decrement Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| ++x | Pre-increment | x=5; ++x; | x=6 |
| x++ | Post-increment | x=5; x++; | x=6 (after use) |
| --x | Pre-decrement | x=5; --x; | x=4 |
| x-- | Post-decrement | x=5; x--; | x=4 (after use) |

**6. Bitwise Operators**

| **Operator** | **Meaning** | **Example (a=5=0101, b=3=0011)** | **Result** |
| --- | --- | --- | --- |
| & | AND | a & b | 1 (0001) |
| | | OR | a|b | 7(0111) |
| ^ | XOR | a ^ b | 6 (0110) |
| ~ | NOT | ~a | -6 |
| << | Left shift | a << 1 | 10 (1010) |
| >> | Right shift | a >> 1 | 2 (0010) |

**7. Conditional (Ternary) Operator**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| ?: | Shorthand if-else | (age >= 18) ? "Adult" : "Minor" | "Adult" |

**8. sizeof Operator**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| sizeof | Gives size of type/variable | sizeof(int) | 4 (system dependent) |

**9. Comma Operator**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| , | Evaluates expressions left to right | int a=(1,2,3); | a = 3 |

**10. Pointer Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| & | Address-of | &x | Address of x |
| \* | Dereference | \*p | Value stored at pointer p |

**11. Type Cast Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| (type) | C-style cast | (int)3.14 | 3 |
| static\_cast<>() | C++ style cast | static\_cast<int>(3.14) | 3 |

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

**Constants and Literals in C++**

| **Aspect** | **Constants** | **Literals** |
| --- | --- | --- |
| **Definition** | A constant is a **named variable whose value cannot be changed** during program execution. | A literal is a **fixed value written directly in the program** (without a variable name). |
| **Purpose** | Used to store values that should remain unchanged (e.g., Pi value, configuration values). | Provide constant values directly in code (e.g., numbers, characters, strings). |
| **Declaration** | Declared using const keyword or #define. | Written directly (e.g., 10, 'A', "Hello"). |
| **Example** | cpp const float PI = 3.14; #define MAX 100 | cpp int x = 10; char grade = 'A'; string msg = "Hello"; |
| **Usage** | - Increases code readability. - Prevents accidental modification of important values. | - Used as actual values in assignments, conditions, loops, etc. |

**3. Control Flow Statements**

**THEORY EXERCISE:**

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

**Conditional Statements in C++**

| **Aspect** | **Explanation** |
| --- | --- |
| **Definition** | Conditional statements are used to make decisions in a program. They allow the program to execute different blocks of code based on certain conditions (true/false). |
| **Purpose** | - Control the flow of execution. - Perform different actions based on different conditions. |
| **Types** | - if - if-else - if-else if-else ladder - switch |

**1. if-else Statement**

| **Aspect** | **Explanation** |
| --- | --- |
| **Definition** | Executes one block of code if a condition is true, otherwise executes another block. |
| **Syntax** | cpp if(condition) { // code if true } else { // code if false } |
| **Example** | cpp int age = 20; if(age >= 18) { cout << "Eligible to vote"; } else { cout << "Not eligible to vote"; } |
| **Use Case** | Useful when decision has **two possible outcomes** (true/false). |

**Purpose**: Executes a block of code if a condition is true, otherwise another block.

**Example**:

#include <iostream>

using namespace std;

int main() {

int num = 10;

if (num > 0)

cout << "Positive";

else

cout << "Non-positive";

}

**2. switch Statement**

| **Aspect** | **Explanation** |
| --- | --- |
| **Definition** | Allows selection among **multiple possible cases** based on a single variable/expression. |
| **Syntax** | cpp switch(expression) { case value1: // code; break; case value2: // code; break; default: // code; } |
| **Example** | cpp int day = 3; switch(day) { case 1: cout << "Monday"; break; case 2: cout << "Tuesday"; break; case 3: cout << "Wednesday"; break; default: cout << "Invalid day"; } |
| **Use Case** | Useful when you need to **choose from multiple options** (instead of many if-else conditions). |

* **Purpose**: Selects one block of code to execute from many options based on a value.

**Example**:

#include <iostream>

using namespace std;

int main() {

int day = 2;

switch(day) {

case 1: cout << "Monday"; break;

case 2: cout << "Tuesday"; break;

default: cout << "Other day";

}

}

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

| **Loop Type** | **When Condition is Checked** | **Syntax Style** | **Executes At Least Once?** | **Example** |
| --- | --- | --- | --- | --- |
| **for** | Before each iteration | Initialization, condition, update in one line | No | for(int i=0; i<5; i++) cout<<i; |
| **while** | Before each iteration | Only condition in loop header | No | int i=0; while(i<5){ cout<<i; i++; } |
| **do-while** | After each iteration | Condition checked at the end | **Yes** | int i=0; do{ cout<<i; i++; } while(i<5); |

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

| **Statement** | **Purpose** | **Effect in Loop** |
| --- | --- | --- |
| **break** | Immediately terminates the loop | Control exits the loop completely |
| **continue** | Skips the current iteration | Control moves to the next iteration of the loop |

**🔹 Example of break**

#include <iostream>

using namespace std;

int main() {

for(int i=1; i<=5; i++) {

if(i == 3) break; // loop ends when i=3

cout << i << " ";

}

return 0;

}

**Output:**

1 2

**🔹 Example of continue**

#include <iostream>

using namespace std;

int main() {

for(int i=1; i<=5; i++) {

if(i == 3) continue; // skips printing when i=3

cout << i << " ";

}

return 0;

}

**Output:**

1 2 4 5

**4. Explain nested control structures with an example.**

**Definition**

* **Nested control structures** mean placing one control structure (like if, for, while, switch) **inside another**.
* This is used when decision-making or repetition requires multiple levels of checks or iterations.

**Types of Nesting**

| **Outer Structure** | **Inner Structure (can be)** |
| --- | --- |
| if / if-else | another if / if-else |
| Loop (for, while, do-while) | another loop or if |
| switch | if, switch, or loop |

**Example 1: Nested if-else**

#include <iostream>

using namespace std;

int main() {

int num = 25;

if (num > 0) {

if (num % 2 == 0)

cout << "Positive Even Number";

else

cout << "Positive Odd Number";

} else {

cout << "Number is Negative or Zero";

}

return 0;

}

**Output:**

Positive Odd Number

**Example 2: Nested Loop**

#include <iostream>

using namespace std;

int main() {

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

for(int j=1; j<=2; j++) {

cout << "i=" << i << ", j=" << j << endl;

}

}

return 0;

}

**Output:**

i=1, j=1

i=1, j=2

i=2, j=1

i=2, j=2

i=3, j=1

i=3, j=2

* Nested control structures = one control inside another.
* Common in **decision making** (nested if) and **repeated patterns** (nested loops).

**4. Functions and Scope**

**THEORY EXERCISE:**

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

| **Aspect** | **Explanation** | **Example** |
| --- | --- | --- |
| **Function** | A block of code that performs a specific task, can be reused. | int add(int a, int b) |
| **Declaration (Prototype)** | Tells the compiler about the function’s name, return type, and parameters (placed before main() or in header files). | int add(int, int); |
| **Definition** | Actual body of the function where logic is written. | int add(int x, int y) { return x+y; } |
| **Calling** | Invoking the function to execute its code. | int sum = add(5, 10); |

#include <iostream>

using namespace std;

// Declaration

int add(int, int);

int main() {

int result = add(4, 6); // Calling

cout << "Sum = " << result;

return 0;

}

// Definition

int add(int a, int b) {

return a + b;

}

**Output:**

Sum = 10

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

| **Type** | **Definition** | **Lifetime** | **Accessibility** | **Example** |
| --- | --- | --- | --- | --- |
| **Local Variable** | Declared **inside** a function/block. | Exists only while the function/block is running. | Accessible **only within** that function/block. | void func(){ int x=5; } |
| **Global Variable** | Declared **outside** all functions. | Exists for the **entire program execution**. | Accessible by **all functions** (unless shadowed). | int g=10; void func(){ cout<<g; } |

#include <iostream>

using namespace std;

int globalVar = 100; // Global

int main() {

int localVar = 20; // Local

cout << "Local = " << localVar << endl;

cout << "Global = " << globalVar << endl;

return 0;

}

**Output:**

Local = 20

Global = 100

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

**Recursion in C++**

Recursion is a process where a function **calls itself** directly or indirectly until a **base condition** is met.

* It is useful for problems that can be divided into smaller sub-problems (like factorial, Fibonacci, tree traversal).
* Every recursive function must have a **base case** to stop infinite calls.

✅ **Example: Factorial using Recursion**

#include <iostream>

using namespace std;

int factorial(int n) {

if (n == 0) // Base case

return 1;

return n \* factorial(n - 1); // Recursive call

}

int main() {

cout << "Factorial of 5 = " << factorial(5);

return 0;

}

**Output:**

Factorial of 5 = 120

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

**Function Prototypes in C++**

A **function prototype** in C++ is a **declaration of a function** that tells the compiler about the function’s **name, return type, and parameters** before its actual definition.

* It acts as a **forward declaration**.
* It helps the compiler check whether the function is **called correctly** (right number and type of arguments).
* The actual **definition** of the function can appear later in the program.

**Why are they used?**

1. Allow calling a function **before it is defined**.
2. Enable **type checking** of function calls.
3. Improve **code readability** and organization (especially in large programs).

**Example:**

#include <iostream>

using namespace std;

// Function prototype

int add(int, int);

int main() {

cout << "Sum = " << add(5, 3);

return 0;

}

// Function definition

int add(int a, int b) {

return a + b;

}

**Output:**

Sum = 8

**5. Arrays and Strings**

**THEORY EXERCISE:**

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

**Arrays in C++**

An **array** in C++ is a collection of elements of the **same data type** stored in **contiguous memory locations**.

* Arrays allow storing and accessing multiple values using a **single variable name** with an **index**.
* The index starts from **0**.

**Difference between Single-Dimensional and Multi-Dimensional Arrays**

| **Aspect** | **Single-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| **Definition** | Stores elements in a **single row/line**. | Stores elements in **rows and columns (matrix form)** or more dimensions. |
| **Declaration** | int arr[5]; | int arr[3][3]; |
| **Storage** | Linear sequence of elements. | Elements stored in a grid (2D), cube (3D), etc. |
| **Access** | Accessed using **one index** (e.g., arr[2]). | Accessed using **two or more indices** (e.g., arr[1][2]). |
| **Usage** | Useful for storing a list of items (marks, ages, numbers). | Useful for tables, matrices, 3D graphics, etc. |

**Example:**

#include <iostream>

using namespace std;

int main() {

// Single-dimensional array

int marks[5] = {90, 85, 88, 92, 75};

cout << "Single-D array element: " << marks[2] << endl;

// Multi-dimensional array

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

cout << "Multi-D array element: " << matrix[1][0] << endl;

return 0;

}

**Output:**

Single-D array element: 88

Multi-D array element: 3

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

**String Handling in C++**

In C++, strings can be handled in **two main ways**:

1. **C-style strings** – using **character arrays**.
2. **C++ string class** – using the **std::string** class from the <string> library.

**1. C-style Strings**

* A C-style string is a **character array** ending with a **null character \0**.
* You can use functions from <cstring> like strcpy, strlen, strcmp, etc.

**Example:**

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char name[20] = "Alice"; // C-style string

cout << "Name: " << name << endl;

cout << "Length: " << strlen(name) << endl; // strlen counts characters

return 0;

}

**Output:**

Name: Alice

Length: 5

**2. C++ string Class**

* std::string is part of the **C++ Standard Library**.
* It is **easier to use** than C-style strings and supports **operators** like + for concatenation.

**Example:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string firstName = "Alice";

string lastName = "Smith";

string fullName = firstName + " " + lastName; // Concatenation

cout << "Full Name: " << fullName << endl;

cout << "Length: " << fullName.length() << endl; // length() function

return 0;

}

**Output:**

Full Name: Alice Smith

Length: 11

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

**Array Initialization in C++**

Arrays in C++ can be **initialized at the time of declaration** or later by assigning values.

**1. Initializing a 1D Array**

* A **1D array** is a simple list of elements of the same type.

**Examples:**

#include <iostream>

using namespace std;

int main() {

// Method 1: Explicit initialization

int arr1[5] = {10, 20, 30, 40, 50};

// Method 2: Partial initialization (remaining set to 0)

int arr2[5] = {1, 2}; // becomes {1, 2, 0, 0, 0}

// Method 3: Automatic size deduction

int arr3[] = {5, 10, 15}; // size = 3 automatically

cout << "arr1[0]: " << arr1[0] << ", arr2[2]: " << arr2[2] << ", arr3[1]: " << arr3[1] << endl;

return 0;

}

**Output:**

arr1[0]: 10, arr2[2]: 0, arr3[1]: 10

**2. Initializing a 2D Array**

* A **2D array** is like a **table (rows and columns)**.

**Examples:**

#include <iostream>

using namespace std;

int main() {

// Method 1: Row-wise initialization

int matrix1[2][3] = { {1, 2, 3}, {4, 5, 6} };

// Method 2: Flattened initialization (row-major order)

int matrix2[2][3] = { 1, 2, 3, 4, 5, 6 };

// Method 3: Partial initialization

int matrix3[2][3] = { {1}, {4, 5} }; // Remaining filled with 0

cout << "matrix1[0][1]: " << matrix1[0][1] << endl;

cout << "matrix2[1][2]: " << matrix2[1][2] << endl;

cout << "matrix3[1][0]: " << matrix3[1][0] << endl;

return 0;

}

**Output:**

matrix1[0][1]: 2

matrix2[1][2]: 6

matrix3[1][0]: 4

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

**String Operations and Functions in C++**

In C++, strings can be handled using:

1. **C-Style Strings** (character arrays with \0)
2. **C++ string Class** (from <string> library, safer and more powerful)

**Common String Operations (using string class)**

| **Operation** | **Function / Operator** | **Example** | **Output** |
| --- | --- | --- | --- |
| Length of string | str.length() or str.size() | string s="Hello"; cout<<s.length(); | 5 |
| Concatenation | + operator or append() | s1="Hi"; s2="All"; cout<<s1+s2; | HiAll |
| Access character | [] operator | string s="World"; cout<<s[0]; | W |
| Substring | substr(pos, len) | string s="Programming"; cout<<s.substr(0,4); | Prog |
| Compare strings | compare() (returns 0 if equal) | s1="abc"; s2="abc"; cout<<s1.compare(s2); | 0 |
| Insert text | insert(pos, str) | string s="Hello"; s.insert(5,"World"); | HelloWorld |
| Erase text | erase(pos, len) | string s="HelloWorld"; s.erase(5,5); | Hello |
| Replace text | replace(pos, len, str) | string s="HelloWorld"; s.replace(5,5,"C++"); | HelloC++ |
| Find substring | find(str) | string s="C++ Programming"; cout<<s.find("Pro"); | 4 |
| Reverse string | reverse(s.begin(), s.end()) | string s="C++"; reverse(s.begin(), s.end()); cout<<s; | ++C |

**C-Style String Functions (from <cstring>)**

| **Function** | **Purpose** | **Example** | **Output** |
| --- | --- | --- | --- |
| strlen(s) | Finds length | char s[]="Hello"; cout<<strlen(s); | 5 |
| strcpy(dest, src) | Copy string | strcpy(s2,s1); | Copies s1 into s2 |
| strcat(s1, s2) | Concatenation | strcat(s1,s2); | Joins s2 to s1 |
| strcmp(s1, s2) | Compare strings | strcmp("abc","abc"); | 0 (equal) |
| strchr(s, ch) | Find first occurrence | strchr("Hello",'l'); | Points to first l |
| strstr(s, sub) | Find substring | strstr("HelloWorld","World"); | Points to "World" |

**6. Introduction to Object-Oriented Programming**

**THEORY EXERCISE:**

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

**🔑 Key Concepts of OOP**

1. **Class**
   * A blueprint or template for creating objects.
   * It defines attributes (data members) and behaviors (member functions).

Example:

class Car {

public:

string brand;

void drive() { cout << "Car is driving"; }

};

1. **Object**
   * An instance of a class.
   * Objects use the variables and methods defined in the class.

Example:

Car c1;

c1.brand = "BMW";

c1.drive();

1. **Encapsulation**
   * Wrapping data (variables) and functions into a single unit (class).
   * Protects data using **access specifiers** (private, public, protected).

Example:

class Student {

private: int marks;

public: void setMarks(int m) { marks = m; }

int getMarks() { return marks; }

};

1. **Abstraction**
   * Hiding implementation details and showing only essential features.
   * Achieved using **abstract classes** or **interfaces** (pure virtual functions).

Example:

class Shape {

public: virtual void draw() = 0; // pure virtual function

};

1. **Inheritance**
   * Acquiring properties and behaviors of one class into another.
   * Promotes reusability.

Example:

class Animal { public: void eat() { cout << "Eating"; } };

class Dog : public Animal { public: void bark() { cout << "Barking"; } };

1. **Polymorphism**
   * Ability of a function or object to behave differently in different contexts.
   * Types:
     + **Compile-time (Function/Operator Overloading)**
     + **Runtime (Function Overriding using virtual functions)**

Example:

class Animal { public: virtual void sound() { cout << "Animal sound"; } };

class Dog : public Animal { public: void sound() override { cout << "Bark"; } };

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

### **Classes and Objects in C++**

* **Class**
  + A **class** is a user-defined data type that acts as a blueprint for creating objects.
  + It defines **attributes (data members)** and **behaviors (member functions)** of objects.
* **Object**
  + An **object** is an instance of a class.
  + It represents a real-world entity (like student, car, bank account).
  + Objects use class data and methods.

**Example:**

#include <iostream>

using namespace std;

// Class definition

class Student {

public:

string name;

int age;

void display() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

// Object creation

Student s1;

s1.name = "Alice";

s1.age = 20;

s1.display(); // Calling member function

return 0;

}

In this example:

* **Student** is a **class**.
* **s1** is an **object** of the class.
* **name** and **age** are data members.
* **display()** is a member function.

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

**Inheritance in C++**

* **Definition**:  
  Inheritance is an **OOP feature** in C++ that allows one class (**derived class**) to acquire the **properties and behaviors** (data members and member functions) of another class (**base class**).
* It promotes **reusability** and **hierarchical relationships**.

**Types of Inheritance**

1. **Single Inheritance** – One base class, one derived class.
2. **Multiple Inheritance** – Derived class inherits from more than one base class.
3. **Multilevel Inheritance** – Derived class becomes base class for another.
4. **Hierarchical Inheritance** – Multiple classes inherit from one base class.
5. **Hybrid Inheritance** – Combination of different types.

**Example: Single Inheritance**

#include <iostream>

using namespace std;

// Base class

class Person {

public:

string name;

int age;

void displayPerson() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

// Derived class

class Student : public Person {

public:

int rollNo;

void displayStudent() {

cout << "Roll No: " << rollNo << endl;

}

};

int main() {

Student s1;

s1.name = "Alice";

s1.age = 20;

s1.rollNo = 101;

s1.displayPerson(); // Inherited from Person

s1.displayStudent(); // Defined in Student

return 0;

}

**Explanation**:

* **Person** = Base class.
* **Student** = Derived class (inherits name, age, displayPerson()).
* Reusability: Student class does not need to redefine name or age.

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

**Encapsulation in C++**

* **Definition:** Encapsulation is the process of **wrapping data (variables) and functions (methods) together into a single unit (class)**.
* It **restricts direct access** to data and provides access through **public methods (getters/setters)**.
* It helps in **data hiding**, **security**, and **modularity**.

**How Encapsulation is Achieved**

1. **By using classes** to group data and functions.
2. **Access Specifiers** control access to data:
   * **private:** accessible only within the class.
   * **public:** accessible from outside the class.
   * **protected:** accessible within class & derived classes.

**Example:**

#include <iostream>

using namespace std;

class Student {

private:

int age; // private data (hidden)

string name;

public:

void setData(string n, int a) { // public method (setter)

name = n;

age = a;

}

void display() { // public method (getter/display)

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Student s;

s.setData("John", 20); // Access via public function

s.display();

return 0;

}