**Module 3 - Introduction to OOPS Programming**

**1. Introduction to C++:**

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

**Ans:** Procedural Programming:

* Scanf, printf.
* Procedural language.
* Data not secure.
* Not supported interitance.
* Namespace functionality not supported.
* Stdio.h

Object Oriented Programming:

* Cin>>, cout>>.
* OOP.
* Data secure ( public, private, protected ).
* Interitance supported.
* Namespace functionality supported.
* Iostream.

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

**Ans:** 1. Modularity and Reusability:

**OOP:** Promotes breaking down a program into self-contained units called objects, which encapsulate both data and methods.

**POP:** Typically organizes code into functions or procedures, which may operate on global data.

2. Encapsulation and Data Security:

**OOP:** Encapsulation bundles data and the methods that operate on that data within a single unit (the object), hiding internal implementation details from external access. This enhances data security and integrity by controlling how data can be accessed and modified.

**POP:** Data and functions are often separated, and data can be directly accessed and modified by various functions, potentially leading to unintended changes and security vulnerabilities.

3. Inheritance and Code Reduction:

**OOP:** Supports inheritance, allowing new classes (subclasses) to inherit properties and behaviors from existing classes (superclasses).

**POP:** Does not inherently support inheritance, often requiring the re-implementation of similar functionalities across different parts of a program.

4. Polymorphism and Flexibility:

**OOP:** Enables polymorphism, allowing objects of different classes to be treated as objects of a common superclass.

**POP:** Lacks direct support for polymorphism, requiring explicit conditional logic to handle different data types or functionalities.

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

**Ans:** Setting Up C++ Development Environment To start programming in C++, follow these steps:

1. Install a C++ Compiler

○ Use MinGW (for Windows) or GCC (for Linux/macOS).

○ Download MinGW-w64 from mingw-w64.org or install g++ via:

■ Windows: Install MinGW or use MSYS2.

■ Linux/macOS: Run sudo apt install g++ (Linux) or brew install gcc (macOS).

Install an IDE or Code Editor

○ Code::Blocks, Dev-C++, Visual Studio, or Eclipse (for full IDEs).

○ VS Code or Sublime Text (for lightweight coding).

1. Write and Compile Code Create a .cpp file, write C++ code, and compile using: g++ filename.cpp -o output ./output

Writing and Running Your First C++ Program

1. Create a C++ File Open your text editor or IDE and create a new file called main.cpp.

Write the C++ Code

Add the following simple C++ code to print "Hello, World!" to the console:

#include // Include the input-output library

int main() { // Main function where execution begins

std::cout << "Hello, World!" << std::endl; // Output to console

return 0; // Exit status

}

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

**Ans:** In C++, the primary input/output operations involve using cin for input and cout for output. cin (pronounced "see-in") is used with the extraction operator (>>) to read data from the standard input (usually the keyboard). cout (pronounced "see-out") is used with the insertion operator (<<) to write data to the standard output (usually the console/screen).

**Example:**

#include <iostream>  
  
int main() {  
 int age;  
 std::cout << "Enter your age: "; *// Output to the console*  
 std::cin >> age; *// Input from the console*  
 std::cout << "You entered: " << age << std::endl; *// Output the entered value*  
 return 0;  
}

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

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

**Ans:** There are 3 types of a data types available:

 Data Types:

* int: Stores whole numbers (integers).

**Example:**

int age = 30;

* char: Stores a single character.

**Example:**

char grade = 'A';

* float: Stores single-precision floating-point numbers (decimals).

**Example:**

float pi\_val = 3.14f;

Derived Data Types:

* **Arrays**: Collections of elements of the same data type.

**Example:**

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

* struct (Structures): Collections of variables of different data types under a single name.

**Example:**

struct Point {  
 int x;  
 int y;  
 };  
 Point p1;  
 p1.x = 5;

* union (Unions): Allow different data types to share the same memory location.

**Example:**

union Data {  
 int i;  
 float f;  
 };  
 Data d;  
 d.i = 10;

* **Pointers**: Variables that store memory addresses.

**Example:**

int num = 10;  
 int\* ptr = &num; // ptr stores the address of num

Enum:

* enum (Enumerations): Define a set of named integer constants.

**Example:**

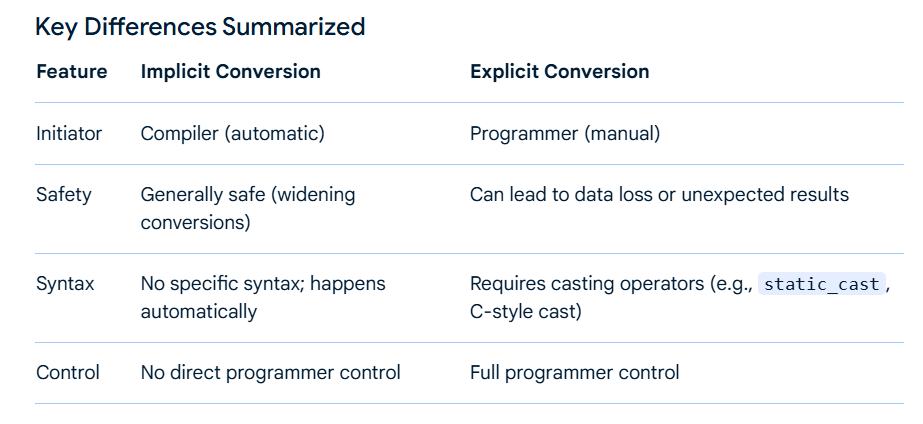
enum Day { Monday, Tuesday, Wednesday };  
 Day today = Monday;

Void:

* void: Represents the absence of a type, typically used for functions that don't return a value or for generic pointers.

**Example:**

void display\_message() {  
 *// Function does not return a value*  
 }

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

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

**Ans:**

1. Arithmetic Operators:

Used for performing mathematical calculations.

Include:

+ (addition)

- (subtraction)

\* (multiplication)

/ (division)

% (modulo - remainder of a division)

1. Relational Operators:

Used for comparing values.

Return a boolean result (true or false).

Include:

== (equal to)

!= (not equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

1. Logical Operators:

Used for combining or modifying boolean expressions.

Return a boolean result.

Include:

&& (logical AND - true if both operands are true)

|| (logical OR - true if at least one operand is true)

! (logical NOT - inverts the boolean value of an operand)

1. Assignment Operators:

Used for assigning values to variables.

The basic assignment operator is =.

Compound assignment operators combine arithmetic operations with assignment (e.g., +=, -=, \*=, /=, %=).

1. Increment/Decrement Operators:

Used to increase or decrease the value of a variable by 1.

++ (increment)

-- (decrement)

Can be used in prefix (e.g., ++x) or postfix (e.g., x++) forms, with subtle differences in behavior.

1. Bitwise Operators:

Used for performing operations on individual bits of data.

Include:

& (bitwise AND)

| (bitwise OR)

^ (bitwise XOR)

~ (bitwise NOT)

<< (left shift)

>> (right shift)

1. Conditional Operator (Ternary Operator):

A shorthand for an if-else statement.

Syntax: condition ? expression1 : expression2

If condition is true, expression1 is evaluated; otherwise, expression2 is evaluated.

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

**Ans:** In C++, constants and literals serve as fundamental constructs for representing fixed, unchanging values within a program.

Literals:

Literals are fixed values directly embedded in the source code. They represent specific data values without requiring a variable name. Examples include:

* **Integer literals:** 10, 0xFF (hexadecimal), 012 (octal)
* **Floating-point literals:** 3.14, 1.2e-5
* **Character literals:** 'A', '\n' (newline escape sequence)
* **String literals:** "Hello, World!"
* **Boolean literals:** true, false

Literals are used to initialize variables, provide values in expressions, or pass arguments to functions.

Constants:

Constants are named entities whose values cannot be modified after initialization. They provide a way to assign a meaningful name to a fixed value, improving code readability and maintainability. In C++, constants are typically declared using the const keyword.

**Example:**

const double PI = 3.14159;  
const int MAX\_USERS = 100;

**Purpose and Use:**

**Immutability:**

Both literals and constants enforce immutability, ensuring that certain values remain fixed throughout the program's execution, preventing accidental modification.

**Compile-time Optimization:**

The compiler can often perform optimizations with constants and literals, as their values are known at compile time.

**Error Prevention:**

Using constants helps prevent errors that might arise from inadvertently changing a critical value during program execution. The compiler will issue an error if an attempt is made to modify a const variable.

**Readability and Maintainability:**

Constants, in particular, enhance code readability by replacing "magic numbers" with descriptive names (e.g., PI instead of 3.14159).

**3. Control Flow Statements:**

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

**Ans:** Conditional statements in C++ are control structures that enable a program to execute different blocks of code based on whether a specified condition evaluates to true or false.

1. If-Else Statements:

The if-else statement allows for two possible execution paths based on a single condition.

if statement:

**Example:**

if (condition) {  
 *// Code to execute if condition is true*  
 }

* else statement: Provides an alternative block of code to execute if the if condition is false.

**Example:**

if (condition) {  
 *// Code to execute if condition is true*  
 } else {  
 *// Code to execute if condition is false*  
 }

* else if statement: Allows for checking multiple conditions sequentially. If the if condition is false, the else if condition is checked, and so on.

**Example:**

if (condition1) {  
 *// Code for condition1*  
 } else if (condition2) {  
 *// Code for condition2*  
 } else {  
 *// Code if neither condition1 nor condition2 is true*  
 }

2. Switch Statements:

The switch statement is used to select one of many possible code blocks to execute based on the value of a single variable or expression. It offers a more structured and often cleaner alternative to multiple else if statements when dealing with a fixed set of discrete values.

**Example:**

switch (expression) {  
 case value1:  
 *// Code to execute if expression equals value1*  
 break; *// Exits the switch statement*  
 case value2:  
 *// Code to execute if expression equals value2*  
 break;  
 default:  
 *// Code to execute if expression does not match any case*  
}

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

**Ans:** In C++, for, while, and do-while loops are used for repetitive execution of code blocks, but they differ in their control flow and when the loop condition is evaluated.

* for loop:
  + Used when the number of iterations is known or can be determined beforehand.
  + Syntax includes initialization, condition, and increment/decrement in a single line.
  + The condition is checked before each iteration.

**Example:**

for (int i = 0; i < 5; ++i) {  
 *// Code to be executed 5 times*  
 }

* while loop:
  + Used when the number of iterations is unknown and depends on a specific condition being met.
  + The condition is checked before each iteration. If the condition is initially false, the loop body may not execute even once.

**Example:**

int count = 0;  
 while (count < 5) {  
 *// Code to be executed as long as count is less than 5*  
 count++;  
 }

do-while loop:

* Similar to a while loop, but guarantees at least one execution of the loop body.

**Example:**

int num;  
 do {  
 *// Code to be executed at least once*  
 std::cout << "Enter a positive number: ";  
 std::cin >> num;  
 } while (num <= 0);

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

**Ans:** In C++, break and continue statements are used within loops to alter their flow of execution.

1. break Statement:

The break statement is used to terminate the loop entirely. When break is encountered inside a loop, the loop immediately stops, and program execution continues with the statement immediately following the loop.

**Example:**

#include <iostream>

Using namespace std;  
  
int main() {  
 for (int i = 0; i < 10; ++i) {  
 if (i == 5) {  
 break; *// Exit the loop when i is 5*  
 }  
 cout << i << " "<<endl;  
 }  
 cout << "\nLoop terminated." <<endl;  
 return 0;  
}

2. continue Statement:

The continue statement is used to skip the current iteration of the loop and proceed to the next iteration.

**Example:**

#include <iostream>

Using namespace std;  
  
int main() {  
 for (int i = 0; i < 5; ++i) {  
 if (i == 2) {  
 continue; *// Skip the rest of the current iteration when i is 2*  
 }  
 cout << i << " "<<endl;  
 }  
 cout << "\nLoop finished." <<endl;  
 return 0;  
}

4. Explain nested control structures with an example.

**Ans:** Nested control structures in C++ involve placing one control flow statement inside another. This allows for more complex logical operations and iterative processes. Common examples include nesting if statements within other if or else blocks, or placing for or while loops inside other loops.

**Example:** Nested for loops for a Multiplication Table

This program demonstrates nested for loops to generate a multiplication table. The outer loop iterates through the multiplicands (numbers from 1 to 5), and the inner loop iterates through the multipliers (also from 1 to 5).

#include <iostream>

Using namespace std;  
  
int main() {  
 *// Outer loop for multiplicands*  
 for (int i = 1; i <= 5; ++i) {  
 *// Inner loop for multipliers*  
 for (int j = 1; j <= 5; ++j) {  
 cout << i << " \* " << j << " = " << (i \* j) <<endl;  
 }  
 cout << endl; *// Add a blank line after each multiplicand's table*  
 }  
 return 0;  
}

**4. Functions and Scope:**

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

**Ans:** A function in C++ is a self-contained block of code designed to perform a specific task. Functions promote code reusability, modularity, and readability by encapsulating operations into named units that can be invoked multiple times.

Function Declaration (Prototype):

A function declaration, also known as a function prototype, informs the compiler about the function's existence before its actual definition. It specifies the function's return type, name, and the types and order of its parameters. It does not include the function's body.

**Example:**

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

Function Definition:

The function definition provides the actual implementation of the function. It includes the function header (same as the declaration) followed by the function body enclosed in curly braces, containing the statements that perform the function's task.

**Example:**

*// Function definition*  
int add(int a, int b) {  
 return a + b;  
}

Function Calling:

Function calling is the process of executing the code within a function. To call a function, its name is used followed by parentheses containing the arguments (actual values passed to the parameters). When a function is called, the program control transfers to the function's definition, executes its statements, and then returns control to the point where it was called.

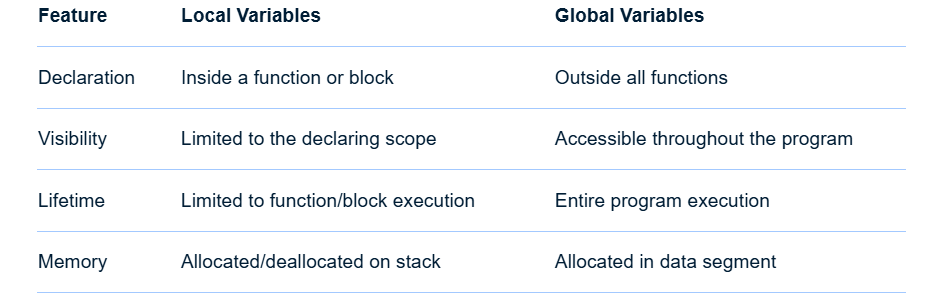
**Example:**

*// Function call*  
int result = add(5, 3); // Calls the 'add' function with arguments 5 and 3

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

**Ans:** In C++, the scope of a variable defines the region of the program where that variable is accessible and valid. Variables can have different scopes, determining their visibility and lifetime.

Differentiation between Local and Global Scope:



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

**Ans:** Recursion in C++ is a programming technique where a function calls itself, either directly or indirectly, to solve a problem. This technique is particularly useful for problems that can be broken down into smaller, self-similar subproblems.

A recursive function must have two key components:

* **Base Case:**

This is a condition that stops the recursion, preventing an infinite loop of function calls. When the base case is met, the function returns a value without making further recursive calls.

* **Recursive Case:**

This is where the function calls itself with a modified input, moving closer to the base case.

**Example:** Calculating Factorial using Recursion

The factorial of a non-negative integer n (denoted as n!) is the product of all positive integers less than or equal to n. For example, 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120.

#include <iostream>  
  
*// Recursive function to calculate factorial*  
int factorial(int n) {  
 *// Base Case: If n is 0 or 1, the factorial is 1.*  
 if (n == 0 || n == 1) {  
 return 1;  
 }   
 *// Recursive Case: n! = n \* (n-1)!*  
 else {  
 return n \* factorial(n - 1); *// Function calls itself*  
 }  
}  
  
int main() {  
 int num;  
 std::cout << "Enter a non-negative integer: ";  
 std::cin >> num;  
  
 if (num < 0) {  
 std::cout << "Factorial is not defined for negative numbers." << std::endl;  
 } else {  
 std::cout << "Factorial of " << num << " is: " << factorial(num) << std::endl;  
 }  
  
 return 0;  
}

**Explanation of the Factorial Example:**

When factorial(5) is called:

* factorial(5) returns 5 \* factorial(4)
* factorial(4) returns 4 \* factorial(3)
* factorial(3) returns 3 \* factorial(2)
* factorial(2) returns 2 \* factorial(1)
* factorial(1) hits the base case and returns 1.

Then, the results are multiplied back up the call stack: factorial(2) becomes 2 \* 1 = 2, factorial(3) becomes 3 \* 2 = 6, factorial(4) becomes 4 \* 6 = 24, and factorial(5) becomes 5 \* 24 = 120.

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

Ans: A function prototype in C++ is a declaration of a function's signature before its actual definition. It specifies the function's name, its return type, and the types and order of its parameters. A function prototype ends with a semicolon and does not include the function's body.

Example:

int add(int a, int b); // Function prototype

Reasons for their use:

* **Type Checking and Error Prevention:**

Function prototypes allow the compiler to perform strong type checking. This means the compiler can verify that the arguments passed to a function during a call match the types and number of parameters declared in its prototype. This helps catch potential errors, such as incorrect argument types or missing arguments, at compile-time rather than at runtime.

* **Function Usage Before Definition:**

In C++, a function must be declared before it can be called. Function prototypes enable you to call a function before its full definition appears in the code, which is particularly useful when functions are defined after main() or when functions call each other in a mutually recursive manner.

**5. Arrays and Strings:**

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

**Ans:** In C++, an array is a data structure used to store a collection of elements of the same data type in contiguous memory locations. These elements are accessed using an index, which typically starts from 0.

**Single-Dimensional Arrays (1D Arrays):**

* + Represent a linear sequence of elements.
  + Can be visualized as a single row or a list of items.
  + Elements are accessed using a single index.
  + **Example:** int numbers[5]; declares an array named numbers that can store 5 integers. You access elements like numbers[0], numbers[1], etc.

**Multi-Dimensional Arrays (MD Arrays):**

* + Represent data in a tabular or grid-like structure, such as matrices or tables.
  + Can have two or more dimensions (e.g., 2D arrays, 3D arrays).
  + Elements are accessed using multiple indices, one for each dimension.
  + **Example (2D Array):** int matrix[3][4]; declares a 2D array named matrix with 3 rows and 4 columns. You access elements like matrix[0][0], matrix[1][2], etc.
  + **Example (3D Array):** int cube[2][3][4]; declares a 3D array named cube with 2 "layers," each having 3 rows and 4 columns. You access elements like cube[0][1][2].

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

**Ans:** String handling in C++ primarily involves two approaches: C-style strings (character arrays) and the std::string class. The std::string class from the <string> header is generally preferred due to its dynamic nature and rich set of member functions for manipulation.

1. C-Style Strings (Character Arrays):

C-style strings are null-terminated character arrays. They offer low-level control but require manual memory management and are less convenient for dynamic operations.

**Example:**

#include <iostream>  
#include <cstring> *// For C-style string functions*  
  
int main() {  
 char greeting[20] = "Hello"; *// Declare a character array*  
 char name[] = "World"; *// Size determined by initializer*  
  
 *// Concatenation using strcat*  
 strcat(greeting, ", ");  
 strcat(greeting, name);  
 std::cout << "Concatenated: " << greeting << std::endl; *// Output: Hello, World*  
  
 *// Length using strlen*  
 std::cout << "Length of greeting: " << strlen(greeting) << std::endl; *// Output: 12*  
  
 return 0;  
}

2. std::string Class:

The std::string class provides a robust and convenient way to handle strings in C++. It automatically manages memory and offers various member functions for common string operations.

**Example:**

#include <iostream>  
#include <string>  
  
int main() {  
 std::string s1 = "C++";  
 std::string s2 = "Programming";  
  
 *// Concatenation using + operator*  
 std::string combined = s1 + " " + s2;  
 std::cout << "Combined string: " << combined << std::endl; *// Output: C++ Programming*  
  
 *// Appending using += operator*  
 s1 += " is fun!";  
 std::cout << "Modified s1: " << s1 << std::endl; *// Output: C++ is fun!*  
  
 *// Length using length() or size()*  
 std::cout << "Length of combined: " << combined.length() << std::endl; *// Output: 15*  
  
 *// Substring extraction using substr()*  
 std::string sub = combined.substr(4, 5); *// Start at index 4, length 5*  
 std::cout << "Substring: " << sub << std::endl; *// Output: Progr*  
  
 *// Finding a substring using find()*  
 size\_t pos = combined.find("Program");  
 if (pos != std::string::npos) {  
 std::cout << "'Program' found at position: " << pos << std::endl; *// Output: 'Program' found at position: 4*  
 }  
  
 *// Comparison using == operator*  
 if (s1 == "C++ is fun!") {  
 std::cout << "s1 is equal to 'C++ is fun!'" << std::endl;  
 }  
  
 return 0;  
}

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

**Ans:** Arrays in C++ can be initialized during their declaration by providing a list of values enclosed in curly braces {}. This is known as list initialization.

1D Array Initialization:

For a one-dimensional array, the values are listed sequentially. If the number of initializers is less than the array size, the remaining elements are initialized to zero. If the size is omitted, the compiler automatically determines the size based on the number of initializers.

**Example:**

*// Explicit size, full initialization*  
int arr1[5] = {1, 2, 3, 4, 5};   
  
*// Explicit size, partial initialization (remaining elements are 0)*  
int arr2[5] = {1, 2};   
  
*// Implicit size (size determined by the number of elements)*  
int arr3[] = {10, 20, 30};

2D Array Initialization:

For a two-dimensional array, the initialization involves nested curly braces, where each inner set of braces represents a row.

**Example:**

*// Explicit size, full initialization*  
int matrix1[2][3] = {{1, 2, 3}, {4, 5, 6}};   
  
*// Explicit size, partial initialization (remaining elements in rows are 0)*  
int matrix2[2][3] = {{1, 2}, {4}};   
  
*// Implicit row size (column size must be specified)*  
int matrix3[][3] = {{10, 20, 30}, {40, 50, 60}};

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

**Ans:** C++ provides robust support for string manipulation through both C-style character arrays and the more modern and feature-rich std::string class.

**1. std::string Operations and Functions:**

The std::string class offers a wide range of member functions for various string operations:

* **Construction and Assignment:**
  + std::string s1; (default constructor, empty string)
  + std::string s2 = "Hello"; (initialization with a C-style string literal)
  + std::string s3 = s2; (copy constructor)
  + s1 = "World"; (assignment operator)
* **Concatenation:**
  + std::string result = s1 + s2; (using the + operator)
  + s1.append(s2); (appending one string to another)
* **Length and Capacity:**
  + s.length(); or s.size(); (returns the number of characters)
  + s.empty(); (checks if the string is empty)
* **Accessing Characters:**
  + s[index]; (using the [] operator for direct character access)
  + s.at(index); (provides bounds checking, throws std::out\_of\_range if invalid index)
* **Comparison:**
  + s1 == s2;, s1 != s2;, s1 < s2; etc. (using comparison operators)
  + s1.compare(s2); (returns an integer indicating comparison result)
* **Substrings:**
  + s.substr(pos, len); (extracts a substring starting at pos with length len)
* **Searching:**
  + s.find(substring); (returns the starting position of the first occurrence of substring)
  + s.rfind(substring); (returns the starting position of the last occurrence)
* **Modification:**
  + s.insert(pos, substring); (inserts a substring at pos)
  + s.erase(pos, len); (removes characters from pos with length len)
  + s.replace(pos, len, new\_string); (replaces a portion of the string)
  + s.push\_back(char); (adds a character to the end)
  + s.pop\_back(); (removes the last character)
* **Input/Output:**
  + std::cin >> s; (reads a single word)
  + std::getline(std::cin, s); (reads an entire line, including spaces)

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

While std::string is generally preferred, C-style string functions (operating on char arrays terminated by a null character \0) are still available:

* strlen(char\_array); (returns the length of the string)
* strcpy(destination, source); (copies source to destination)
* strcat(destination, source); (concatenates source to destination)
* strcmp(str1, str2); (compares two strings lexicographically)
* strncpy, strncat, strncmp (safer versions that limit the number of characters)

**Example:**

#include <iostream>  
#include <string>  
  
int main() {  
 std::string s1 = "Hello";  
 std::string s2 = " World";  
  
 std::string s3 = s1 + s2; *// Concatenation*  
 std::cout << "Concatenated: " << s3 << std::endl;  
  
 std::cout << "Length of s3: " << s3.length() << std::endl;  
  
 std::string sub = s3.substr(6, 5); *// Substring*  
 std::cout << "Substring: " << sub << std::endl;  
  
 size\_t pos = s3.find("World"); *// Searching*  
 if (pos != std::string::npos) {  
 std::cout << "'World' found at position: " << pos << std::endl;  
 }  
  
 s1.append(" C++"); *// Appending*  
 std::cout << "Appended s1: " << s1 << std::endl;  
  
 return 0;  
}

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

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

**Ans:** Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code that manipulates the data. The key concepts of OOP are:

* **Classes and Objects:**
  + A class serves as a blueprint or template for creating objects. It defines the structure (attributes/data) and behavior (methods/functions) that objects of that class will possess.
  + An object is an instance of a class, representing a specific entity with its own unique set of attribute values and the ability to perform the behaviors defined by its class.

**Encapsulation:**

This principle involves bundling data (attributes) and the methods that operate on that data within a single unit (the object).

**Abstraction:**

Abstraction focuses on presenting only the essential information to the user while hiding the complex implementation details. It allows developers to create simplified representations of real-world entities, focusing on what an object does rather than how it does it.

**Inheritance:**

Inheritance enables a new class (subclass or derived class) to inherit properties and behaviors from an existing class (superclass or base class). This promotes code reusability and establishes a hierarchical relationship between classes, modeling "is-a" relationships (e.g., a "Car" is a "Vehicle").

**Polymorphism:**

Polymorphism, meaning "many forms," allows objects of different classes to be treated as objects of a common type. It enables a single interface to be used for different underlying data types or behaviors, leading to more flexible and extensible code. This is often achieved through method overriding (where a subclass provides its own implementation of a method inherited from its superclass) or method overloading.

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

**Ans:** In C++, a class serves as a blueprint or a template for creating objects. It is a user-defined data type that encapsulates data members (variables) and member functions (methods) into a single unit. Data members represent the attributes or state of an object, while member functions define the behaviors or actions that an object can perform.

An object is an instance of a class. When a class is defined, no memory is allocated. However, when an object of that class is created, memory is allocated to store the object's specific data members and to allow it to utilize the member functions defined in its class.

**Example:**

#include <iostream>  
#include <string>

Using namespace std;  
  
class Car {  
public: *// Access specifier: members are accessible from outside the class*  
 *// Data members (attributes)*  
 string brand;  
 string model;  
 int year;  
  
 *// Member function (behavior)*  
 void displayInfo() {  
 cout << "Brand: " << brand <<endl;  
 cout << "Model: " << model <<endl;  
 cout << "Year: " << year <<endl;  
 }  
};  
  
int main() {  
 *// Create objects (instances) of the Car class*  
 Car car1; *// Declares an object named car1*  
 car1.brand = "Maruti";  
 car1.model = "Suzuki";  
 car1.year =2009;  
  
 Car car2; *// Declares another object named car2*  
 car2.brand = "Honda";  
 car2.model = "Breza";  
 car2.year =2025;  
  
 *// Call member functions on the objects*  
 cout << "Car 1 Information:" <<endl;  
 car1.displayInfo();  
  
 cout << "\nCar 2 Information:" <<endl;  
 car2.displayInfo();  
  
 return 0;  
}

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

**Ans:** Inheritance in C++ is a core concept of Object-Oriented Programming (OOP) that allows a new class to be created based on an existing class. The new class, known as the derived class (or child class), inherits the properties (data members) and behaviors (member functions) of the existing class, known as the base class (or parent class). This mechanism promotes code reusability and establishes a hierarchical "is-a" relationship between classes.

**Example:**

#include <iostream>  
#include <string>

Using namespace std;  
  
*// Base class*  
class Vehicle {  
public:  
 string brand;  
 void honk() {  
 cout << "Tuut, tuut!" <<endl;  
 }  
};  
  
*// Derived class*  
class Car : public Vehicle { *// Car inherits publicly from Vehicle*  
public:  
 string model;  
};  
  
int main() {  
 Car myCar;  
 myCar.brand = "Ford"; *// Inherited from Vehicle*  
 myCar.model = "Mustang";  
   
 cout << "My car is a " << myCar.brand << " " << myCar.model << endl;  
 myCar.honk(); *// Inherited from Vehicle*  
   
 return 0;  
}

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

**Ans:** Encapsulation in C++ is an object-oriented programming concept that involves bundling data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class. It is a mechanism for restricting direct access to some of an object's components, which helps in data hiding and maintaining data integrity.

**Private Access Specifier**:

* Data members (variables) that are intended to be hidden from external access are declared with the private access specifier.

**Public Access Specifier**:

* Member functions that are designed to provide controlled access to the private data members are declared with the public access specifier.

**Example:**

class BankAccount {  
private:  
 double balance; *// Private data member*  
  
public:  
 *// Public setter method to deposit money*  
 void deposit(double amount) {  
 if (amount > 0) {  
 balance += amount;  
 }  
 }  
  
 *// Public getter method to get the balance*  
 double getBalance() const {  
 return balance;  
 }  
};

In this example, balance is a private data member, ensuring it cannot be directly modified from outside the BankAccount class. Access and modification are only permitted through the public deposit() and getBalance() methods, which can include validation or other logic to maintain data integrity.