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

* Procedural programming focuses on procedures (functions) to manipulate data, while object-oriented programming (OOP) uses objects that encapsulate data and methods, promoting modularity and reusability through concepts like inheritance and polymorphism.
* In procedural programming, the program is divided into small parts called functions. In object-oriented programming, the program is divided into small parts called objects.
* Procedural programming follows a top-down approach. Object-oriented programming follows a bottom-up approach.
* There is no access specifier in procedural programming. Object-oriented programming has access specifiers like private, public, protected, etc.
* Adding new data and functions is not easy. Adding new data and function is easy. Procedural programming does not have any proper way of hiding data so it is less secure. Object-oriented programming provides data hiding so it is more secure.
* In procedural programming, overloading is not possible. Overloading is possible in object-oriented programming.
* In procedural programming, there is no concept of data hiding and inheritance. In object-oriented programming, the concept of data hiding and inheritance is used.
* In procedural programming, the function is more important than the data. In object-oriented programming, data is more important than function.
* Procedural programming is used for designing medium-sized programs. Object-oriented programming is used for designing large and complex programs.
* Procedural programming uses the concept of procedure abstraction. Object-oriented programming uses the concept of data abstraction.
* Code reusability absent in procedural programming, Code reusability present in object-oriented programming.

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

* Object-Oriented Programming (OOP) offers significant advantages over Procedural Oriented Programming (POP), primarily through its ability to model real-world entities as objects, leading to enhanced code organization, reusability, and maintainability, especially for complex projects.
* Here's explain advantages: Modularity and Organization:- OOP promotes modularity by grouping data and functions into objects, making code easier to understand, maintain, and debug.
* Code Reusability: OOP supports code reusability through inheritance, allowing developers to create new classes based on existing ones, reducing redundancy and development time.
* Data Encapsulation and Security: - OOP uses encapsulation to hide data and implementation details within objects, enhancing security and preventing unintended access or modification.
* Flexibility and Extensibility: - OOP's features like polymorphism and abstraction allow for flexible and extensible code, enabling easier adaptation to changing requirements.
* Improved Problem Solving: - OOP's object-oriented approach aligns well with real-world problem-solving, making it easier to model complex systems and their interactions.
* Easier Debugging and Maintenance: - OOP's modular structure and encapsulation make it easier to identify, isolate, and fix issues, leading to faster debugging and simpler maintenance.
* Scalability: - OOP's modular and reusable nature makes it easier to scale projects, adding new features and functionalities without disrupting existing code.

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

* To set up a C++ development environment, you will need a C++ compiler, a text editor or IDE, and potentially a build system. Here is a breakdown of the steps:
* Choose a Compiler:

GCC (GNU Compiler Collection): A widely used, open-source compiler available on most platforms (Linux, macOS, Windows).

Windows: Install MinGW (Minimalist Gnu for Windows) or use the Visual Studio C++ compiler.

macOS: Xcode's command-line tools include GCC.

Linux: GCC is usually pre-installed or available through package managers.

Clang: Another popular compiler, often used as an alternative to GCC.

Visual Studio C++ Compiler: A powerful compiler integrated with the Visual Studio IDE.

* Choose a Text Editor or IDE:- Text Editors:

Visual Studio Code: A free, versatile editor with excellent C++ support through extensions.

Sublime Text: A popular, paid editor known for its speed and customization options.

vim/Emacs: Powerful, command-line text editors.

* IDEs (Integrated Development Environments): Visual Studio: A comprehensive IDE with a built-in C++ compiler and debugger.

CLion: A dedicated C/C++ IDE from JetBrains.

Eclipse: A versatile IDE with C/C++ support through plugins.

Choose a Build System (Optional but Recommended):

CMake:- A cross-platform build system that generates build files for various IDEs and compilers.

* Make: A traditional build system used on Unix-like systems.
* MSBuild: The build system used by Visual Studio.
* Install the Tools: Download and install the chosen compiler, editor/IDE, and build system (if needed).

Follow the instructions provided by the respective software vendors.

* Configure the Environment: Compiler Path: Ensure the compiler's executable path is added to your system's environment variables (PATH) so that you can run the compiler from the command line.
* IDE/Editor Settings: Configure your IDE/editor to use the chosen compiler and build system.
* Project Setup: Create a new project in your IDE/editor and set up the necessary build configurations.

Write and Compile Your First C++ Program: Create a new C++ file (e.g., main.cpp). Write a simple C++ program (e.g., "Hello, World!").

Compile the program using the chosen compiler (e.g., g++ main.cpp -o hello). Run the executable (e.g., ./hello or hello.exe).

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

* In C++, the primary input/output operations are handled using streams, with cin for standard input and cout for standard output. cin reads data from the keyboard, while cout displays data to the console.

Example: C++

#include <iostream>

using namespace std;

int main() {

int age;

string name;

// Input

cout << "What is your name? ";

cin >> name;

cout << "What is your age? ";

cin >> age;

// Output

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

  return 0;

}

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

* In C++, data types are broadly classified into primitive, derived, and user-defined, with primitive types including int, float, char, bool, and double for integers, floating-point numbers, characters, booleans, and double-precision floating-point numbers, respectively.
* Primitive Data Types: int :- Stores integer values (whole numbers).

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

double: Stores double-precision floating-point numbers (larger decimal numbers).

char: Stores single characters (letters, numbers, symbols).

bool: Stores boolean values (true or false).

void: Represents the absence of a value or type.

* Derived Data Types: Arrays: Collections of elements of the same data type.

Pointers: Variables that store the memory address of another variable.

References: Aliases for existing variables.

* Function Types: Data types that represent functions.
* User-Defined Data Types:

Classes: User-defined data types that can contain data (member variables) and functions (member functions).

Structures: Similar to classes, but with default access specifier as public.

Unions: Data types where multiple members share the same memory location.

Enumerations: User-defined data types that consist of a set of named integer constants.

Typedef: Used to create aliases for existing data types.

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

* In C++, implicit type conversion happens automatically by the compiler when a value of one type is used where a different type is expected, while explicit type conversion (or casting) is done manually by the programmer using operators like static\_cast.
* Implicit Type Conversion (Automatic Type Conversion): The compiler automatically converts a value from one type to another without any explicit instructions from the programmer.

This happens when Assigning a value of one type to a variable of another type.

Performing operations on values of different types.

Examples: Assigning an int to a double variable: double num = 5; (The int value 5 is implicitly converted to a double value 5.0).

Adding an int and a double: int a = 5; double b = 2.5; double result = a + b; (The int value 5 is implicitly converted to a double before the addition).

* Explicit Type Conversion (Casting): The programmer explicitly instructs the compiler to convert a value from one type to another using a cast operator.

Use the static cast, dynamic cast, const cast, or reinterpret cast operators.

Examples: int num int = static cast<int>(3.14); (The double value 3.14 is explicitly converted to an int, resulting in 3).

double num\_ double = static\_ cast<double>(5); (The int value 5 is explicitly converted to a double, resulting in 5.0).

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

* In C++, operators are categorized into arithmetic, relational, logical, bitwise, assignment, and other types, which are used to perform various operations on data.
* Arithmetic Operators: It is used for performing basic mathematical operations.

Examples: + (addition), - (subtraction), \* (multiplication), / (division), % (modulo).

* Relational Operators: It is used to compare two values and return a boolean result (true or false).

Examples: == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to).

* Logical Operators: It is used to combine or modify boolean expressions.

Examples: && (logical AND), || (logical OR), ! (logical NOT).

* Assignment Operators: Used to assign values to variables.
* Examples: = (assignment), += (add and assign), -= (subtract and assign), \*= (multiply and assign), /= (divide and assign), %= (modulo and assign).
* Increment/Decrement Operators: ++ (increment), -- (decrement).
* Conditional Operator: ? : (ternary operator).

Comma Operator: , (used to separate expressions in a comma-separated list).

* Type Casting Operators: Used to convert one data type to another.

Sizeof Operator: Used to determine the size of a data type or variable.

* Pointer Operators: & (address-of), \* (dereference).

Member Access Operators: . (dot), -> (arrow).

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

* In C++, constants are values that remain unchanged throughout the program's execution, while literals are the fixed values directly used in the code.
* Constants, declared using const, improve readability and prevent accidental modification, whereas literals represent data directly within the code.

Purpose of constatnts to define named values that are intended to be read-only and not modified during program execution.

* Purpose of literals to represent fixed values directly in the code, such as numbers, characters, or strings. Literals are used to initialize variables, pass arguments to functions, and perform calculations.

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

* In C++, conditional statements, also known as decision-making statements, allow programs to execute different code blocks based on whether a condition is true or false, enabling dynamic and logical program flow. The primary conditional statements are if, if-else, else if, and switch.
* if-else Statement: It executes one block of code if the condition is true and another block if the condition is false.

if (condition) {

// Code to be executed if the condition is true

} else {

// Code to be executed if the condition is false

    }

* switch Statement: Provides a more efficient way to select one of several code blocks to execute based on the value of a variable or expression.

Syntax:

C++

switch (expression) {

case value1:

// Code to be executed if expression == value1

break; // Optional: prevents execution of the next case

case value2:

// Code to be executed if expression == value2

break;

// ... more cases

default:

// Code to be executed if none of the cases match

}

expression: An integer or character expression.

case: Specifies a value to compare with the expression.

break: Optional statement to exit the switch statement.

default: Optional block to execute if no case matches.

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

* There are three primary types of loops: the "for" loop, which is used when you know in advance how many iterations are needed;
* the "while" loop, which repeats as long as a given condition is true; and the "do-while" loop, which guarantees at least one execution of the loop body before checking the condition.
* In C++, for, while, and do-while loops are used for repetitive tasks, but they differ in how they control the loop's execution: for is best for a known number of iterations, while checks the condition before each iteration, and do-while checks the condition after each iteration, guaranteeing at least one execution.
* For &While loop is Entry controlled Loop and do while is Exit controlled Loop.

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

* In programming loops, break terminates the loop immediately, while continue skips the current iteration and proceeds to the next, useful for controlling loop execution flow.
* break Statement:

Function: Terminates the execution of the loop (for, while, or do-while) and transfers control to the statement immediately following the loop.

* #include <iostream>
* int main() {

for (int i = 0; i < 10; ++i) {

if (i == 5) {

break; // Exit the loop when i is 5

}

std::cout << i << std::endl;

}

std::cout << "Loop finished" << std::endl;

return 0;

}

* In this example, the loop will print numbers from 0 to 4, and then the break statement will terminate the loop, and the "Loop finished" message will be printed after the loop.
* Continue Statement:

Function: Skips the current iteration of the loop and proceeds to the next iteration.

* #include <iostream>

int main() {

for (int i = 0; i < 10; ++i) {

if (i == 5) {

continue; // Skip the current iteration when i is 5

}

std::cout << i << std::endl;

}

std::cout << "Loop finished" << std::endl;

return 0;

}

In this example, the loop will print numbers from 0 to 4 and then from 6 to 9, skipping the number 5 because of the continue statement.

1. Explain nested control structures with an example.

* In C++, nested control structures occur when you place one control structure (like if, else, for, while) inside another. This allows for creating complex logic and handling multiple conditions or iterations.
* #include <iostream>

int main() {

int age;

bool hasLicense;

// Get input from the user

std::cout << "Enter your age: ";

std::cin >> age;

std::cout << "Do you have a driver's license? (true/false): ";

std::cin >> hasLicense;

// Nested if-else structure

if (age >= 18) {

std::cout << "You are an adult." << std::endl;

if (hasLicense) {

std::cout << "You can drive." << std::endl;

} else {

std::cout << "You should get a license." << std::endl;

}

} else {

std::cout << "You are a minor." << std::endl;

}

  return 0;

}

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

* In C++, a function is a reusable block of code that performs a specific task, improving code organization and readability. It's defined through a declaration (specifying name, return type, and parameters), a definition (containing the code to execute), and is executed by calling it.
* Function Declaration: Purpose: Tells the compiler about the function's existence, including its name, return type, and parameters, without providing the actual code.
* Function Definition: Purpose: Provides the actual code (body) that the function executes when called.
* Function Calling: Purpose: Executes the code within the function's definition.

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

* In C++, the scope of a variable determines where it is visible and accessible within your program, influencing when it's created and destroyed. Variables can have local (within a function or block) or global (outside any function) scope, impacting their lifetime and accessibility.
* In C++, local scope refers to variables declared inside a function or block, accessible only within that scope, while global scope refers to variables declared outside any function, accessible from anywhere in the program.
* Local Scope: Definition: Variables declared inside a function or within a block of code (using curly braces {}) have local scope.

Visibility: They are only accessible within the function or block where they are declared.

Lifetime: Their existence is limited to the duration of the function or block's execution.

* Global Scope:

Definition: Variables declared outside of any function or block have global scope.

Visibility: They are accessible from anywhere in the program after they are declared.

Lifetime: They exist for the entire duration of the program's execution.

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

* In C++, recursion is a programming technique where a function calls itself to solve a problem, breaking it down into smaller, self-similar subproblems until a base case is reached, at which point the function stops calling itself and returns a result.
* Calculating Factorial

#include <iostream>

using namespace std;

// Recursive function to calculate factorial

int factorial(int n) {

// Base case: factorial of 0 is 1

if (n == 0) {

return 1;

}

// Recursive case: n! = n \* (n-1)!

else {

return n \* factorial(n - 1);

}

}

int main() {

int number = 5;

int result = factorial(number);

cout << "Factorial of " << number << " is: " << result << endl;

return 0;

}

Explanation: Base Case: The if (n == 0) statement defines the base case, which is the condition that stops the recursion. When n becomes 0, the function returns 1, preventing an infinite loop.

Recursive Case: The else statement defines the recursive case. It calculates the factorial of n by multiplying n with the factorial of n-1, effectively calling the factorial function again with a smaller value of n.

How it works: When factorial(5) is called, it returns 5 \* factorial(4).

factorial(4) returns 4 \* factorial(3).This continues until factorial(1) returns 1 \* factorial(0).factorial(0) returns 1 (base case).The values are then multiplied back up the chain of calls, resulting in 5 \* 4 \* 3 \* 2 \* 1 = 120.

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

* Function Prototype: A function prototype provides the compiler with information about the function's return type, name, and parameters. It is declared at the beginning of the program before the main function, and its purpose is to inform the compiler about the function's signature. This allows the compiler to properly validate and integrate function calls in your code, especially when a function is defined after it's called.
* Function prototypes in C++ are used to inform the compiler about a function's signature (return type, name, and parameters) before its actual definition, allowing functions to be called before they are defined, improving code organization, and enabling type checking.

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

* In C++, an array is a data structure that stores a collection of elements of the same data type in contiguous memory locations. Single-dimensional arrays store elements in a linear fashion, while multi-dimensional arrays store elements in a table-like format (rows and columns).
* Definition: An array is a data structure used to store a collection of elements of the same data type, where each element is accessed using an index (a numerical position).
* A single-dimensional array stores elements in a linear, sequential manner (like a list), accessed by a single index, while a multi-dimensional array stores elements in a table-like format (like a grid or matrix), accessed using multiple indices, one for each dimension.
* Single-Dimensional Array: - Structure: Elements are arranged in a single row or column, like a list.

Accessing Elements: Each element is accessed using a single index (a number representing its position in the array).

Example: int numbers [5] = {1, 2, 3, 4, 5}; (An array of 5 integers)

To access the third element (which is 3), you would use numbers [2].

* Multi-Dimensional Array: - Structure: Elements are arranged in multiple dimensions, like a table or grid (2D array) or a cube (3D array).

Accessing Elements: Each element is accessed using multiple indices, one for each dimension.

Example: int matrix [2][3] = {{1, 2, 3}, {4, 5, 6}}; (A 2x3 matrix)

To access the element in the second row and first column (which is 4), you would use matrix [1][0].

Common Uses:

2D Arrays: Representing tables, matrices, images, or other grid-like data.

3D Arrays: Representing cubes, 3D models, or other 3D data structures.

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

* From C, C++ inherited the convention of using null-terminated strings that are handled by a pointer to their first element, and a library of functions that manipulate such strings. In modern standard C++, a string literal such as "hello" still denotes a NUL-terminated array of characters.
* The C++ programming language has support for string handling, mostly implemented in its standard library. The language standard specifies several string types, some inherited from C, some designed to make use of the language's features, such as classes and RAII. The most-used of these is std::string.

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

* The initializer for an array is a comma-separated list of constant expressions enclosed in braces ({ }). The initializer is preceded by an equal sign (=). You do not need to initialize all elements in an array.
* Here are examples of 1D (one-dimensional) and 2D (two-dimensional) arrays in C++.
* 1D Array:

#include <iostream>

int main() {

// Declare a 1D array of integers with a size of 5

int numbers[5];

// Initialize the array elements

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

// Access and print the array elements

std::cout << "1D Array Elements:" << std::endl;

for (int i = 0; i < 5; ++i) {

std::cout << "numbers[" << i << "] = " << numbers[i] << std::endl;

}

  return 0;

}

* 2D Array:

#include <iostream>

int main() {

// Declare a 2D array of integers with 2 rows and 3 columns

int matrix[2][3];

// Initialize the array elements

matrix[0][0] = 1;

matrix[0][1] = 2;

matrix[0][2] = 3;

matrix[1][0] = 4;

matrix[1][1] = 5;

matrix[1][2] = 6;

// Access and print the array elements

std::cout << "2D Array Elements:" << std::endl;

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

std::cout << "matrix[" << i << "][" << j << "] = " << matrix[i][j] << std::endl;

}

}

  return 0;

}

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

* In C++, string operations and functions are performed using the std:: string class, offering a wide range of functionalities for manipulating and processing text data, including length retrieval, comparison, substring extraction, searching, and more.
* string operations and functions: Basic Operations:
* Length: myString.length() or myString.size() returns the number of characters in the string.

Accessing Characters: You can access individual characters using the array-like syntax: myString[0] (accesses the first character).

* Concatenation: Use the + operator or the append() function: myString += " and C++"; or myString.append(" and C++");.
* Comparison: Use the relational operators (==, !=, <, >, <=, >=) to compare strings.
* Assignment: Use the = operator to assign one string to another: myString = "New string";.
* Key String Functions: substr(): Extracts a substring from a given position and length: myString.substr(start\_index, length).

find(): Finds the first occurrence of a substring: myString.find("substring").

rfind(): Finds the last occurrence of a substring: myString.rfind("substring").

replace(): Replaces a substring with another: myString.replace(start\_index, length, "replacement").

insert(): Inserts a string at a specific position: myString.insert(index,"string\_to\_insert").

erase(): Deletes characters from a string: myString.erase(start\_index, length).

empty(): Checks if the string is empty: myString.empty().

clear(): Removes all characters from the string: myString.clear().

swap(): Swaps the contents of two strings: myString.swap(anotherString).

getline(): Reads a line of input from a stream into a string: std::getline(std::cin, myString).

push\_back(): Adds a single character to the end of the string: myString.push\_back('a').

pop\_back(): Removes the last character from the string: myString.pop\_back().

at(): Accesses a character at a specific index, throwing an exception if the index is out of bounds: myString.at(index).

c\_str(): Returns a pointer to a C-style string representation of the std::string object: myString.c\_str().

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

* Object-Oriented Programming (OOP) organizes software design around data, using objects that contain data (attributes) and methods (actions). The key concepts are encapsulation, abstraction, inheritance, and polymorphism, enabling modular, reusable, and maintainable code.
* Objects:
* The fundamental unit of OOP, representing real-world entities or concepts, containing both data (attributes) and behavior (methods).
* Classes:
* A blueprint or template for creating objects, defining the structure and behavior of objects of that type.
* Encapsulation:
* Bundling data (attributes) and the methods that operate on that data into a single unit (class or object), protecting data from external access and modification.
* Abstraction:
* Hiding complex implementation details and exposing only the essential information or functionality to the user or other parts of the program.
* Inheritance:
* Allowing a class (subclass or child class) to inherit properties and methods from another class (superclass or parent class), promoting code reuse and hierarchical organization.
* Polymorphism:
* Enabling objects of different classes to respond differently to the same method call, allowing for flexible and adaptable code.

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

* In C++, a class is a blueprint or template for creating objects, encapsulating data (attributes) and functions (methods) that operate on that data, while an object is a concrete instance of a class, with its own unique data and behavior.
* Example: Consider a class named "Car". It could have data members like "model", "color", and "year", and member functions like "startEngine()", "accelerate()", and "brake()".

Example: If you create two objects of the "Car" class, they would be two separate cars, each with its own model, color, year, etc.

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

* Inheritance in C++ is a mechanism that allows a class to inherit properties from another class. It is a key feature of object-oriented programming.
* Examples of inheritance: A class called "Baby" can inherit properties from a class called "Mother"
* A class called "Bat" can inherit properties from both a class called "Mammal" and a class called "WingedAnimal”
* In C++, inheritance allows a class (derived or child class) to inherit properties (attributes and methods) from another class (base or parent class), promoting code reusability and a hierarchical relationship between classes.
* Example: #include <iostream>

// Base class

class Animal {

public:

void eat() {

std::cout << "I can eat!" << std::endl;

}

void sleep() {

std::cout << "I can sleep!" << std::endl;

}

};

// Derived class inheriting from Animal

class Dog : public Animal {

public:

void bark() {

std::cout << "Woof woof!!" << std::endl;

}

};

int main() {

Dog dog1; // Create an object of the derived class

dog1.eat(); // Accessing inherited method from Animal

dog1.sleep(); // Accessing inherited method from Animal

dog1.bark(); // Accessing the method defined in Dog

    return 0;

}

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

* In C++, encapsulation, a core Object-Oriented Programming (OOP) principle, involves bundling data (attributes) and methods (functions) that operate on that data within a single unit, a class, and restricting direct access to some of the object's components, thereby protecting data from external interference.
* In C++, encapsulation is achieved by bundling data (attributes) and the functions (methods) that operate on that data within a class and controlling access to them using access specifiers (public, private, protected).