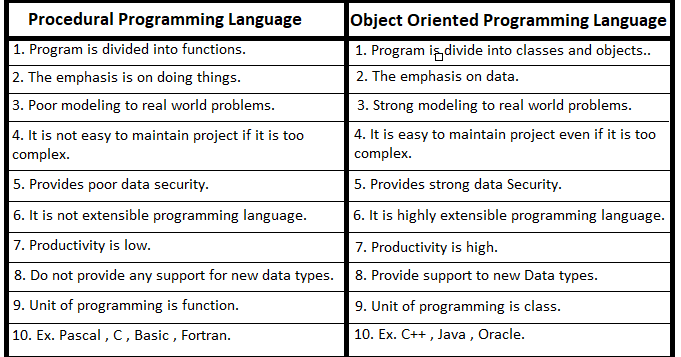
1. **What are the key differences between Procedural Programming and Object Oriented Programming (OOP)?**

**Ans :** 

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

**Ans** : Encapsulation (Data Security)

* + OOP: Data and functions are bundled together in a class. Access modifiers (private, public, protected) control visibility.
  + POP: Data is usually global and can be accessed by any function, making it less secure.  
    Reusability (Code Reuse)
  + OOP: Classes and objects can be reused across programs. Inheritance allows one class to acquire properties of another.
  + POP: Functions can be reused, but data reusability is difficult.

1. Inheritance (Avoids Redundancy)
   * OOP: New classes can be created from existing ones, avoiding duplication of code.
   * POP: No direct way to inherit properties; functions must be rewritten.
2. Polymorphism (Flexibility)
   * OOP: Same function name can be used for different purposes (method overloading/overriding).
   * POP: No direct support for polymorphism.
3. Modularity (Organized Structure)
   * OOP: Programs are divided into classes/objects, making them modular.
   * POP: Programs are divided into functions, but managing large projects becomes complex.
4. Real-world Modeling
   * OOP: Closer to real-world concepts (objects → entities like Student, Car, BankAccount).
   * POP: Does not directly model real-world entities.
5. Maintainability & Scalability
   * OOP: Large projects are easier to maintain and extend due to its modular structure.
   * POP: Harder to scale and maintain as code size grows.
6. **Explain the steps involved in setting up a C++ development environment.**

**Ans:** 1. Install a C++ Compiler

* A compiler translates C++ code into machine code.
* Popular compilers:
  + GCC (GNU Compiler Collection) → Linux, Windows (via MinGW or Cygwin).
  + MSVC (Microsoft Visual C++) → Comes with Visual Studio.
  + Clang → macOS, Linux, Windows.
* On Windows → install MinGW or Visual Studio.

2. Choose an IDE or Text Editor

* An IDE (Integrated Development Environment) helps you write, edit, compile, and debug code.
* Popular IDEs/editors for C++:
  + Code::Blocks (lightweight, beginner-friendly).
  + Dev C++ (basic, for beginners).
  + Visual Studio (powerful, professional).
  + CLion (JetBrains, advanced).
  + VS Code (lightweight editor with extensions).

3. Configure the IDE/Editor

* Set up the compiler path inside the IDE (e.g., tell Code::Blocks/VS Code where g++ is installed).
* Install required extensions/plugins if using VS Code (like *C/C++ extension by Microsoft*).

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

**Ans:** 1. Input (cin)

* Used to take input from the user.
* Extracts data from standard input (usually the keyboard).

Example:

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age; // input operation

cout << "You entered: " << age;

return 0;

}

2. Output (cout)

* Used to display output on the screen.
* Inserts data into standard output (usually the monitor).

Example:

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!"; // output operation

return 0;

}

3. End of Line (endl)

* Used to insert a newline in output.
* Works like \n but also flushes the buffer.

Example:

cout << "First Line" << endl;

cout << "Second Line";

4. Chaining Input/Output

* Multiple values can be input/output in a single statement.

Example:

#include <iostream>

using namespace std;

int main() {

int x, y;

cout << "Enter two numbers: ";

cin >> x >> y; // multiple input

cout << "Sum = " << x + y; // multiple output

return 0;

}

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

**Ans:** **These are the fundamental types:**

* int → stores integers (whole numbers).
* float → stores decimal numbers (single precision).
* double → stores decimal numbers (double precision).
* char → stores a single character.
* bool → stores true or false.
* void → represents no value (used in functions).

**These are built using basic types.**

* Array → collection of similar data.
* Pointer → stores address of a variable.
* Reference → alias for another variable.
* Function → returns a data type.

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

**Ans: Implicit Type Conversion (Type Casting / Type Promotion)**

* Also called Type Promotion or Type Casting by Compiler.
* Done automatically by C++ compiler without programmer’s intervention.
* Converts a smaller data type into a larger one to prevent data loss.

Example:

#include <iostream>

using namespace std;

int main() {

int a = 10;

double b = a; // implicit conversion (int → double)

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

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

* Also called Type Casting.
* Done manually by the programmer using casting operators.
* Can be done using:
  + C-style casting: (type) expression
  + Function-style casting: type(expression)
  + Casting operators: static\_cast<>, dynamic\_cast<>, const\_cast<>, reinterpret\_cast<>

Example:

#include <iostream>

using namespace std;

int main() {

double pi = 3.14159;

int x = (int)pi; // explicit conversion (C-style)

int y = int(pi); // explicit conversion (function-style)

cout << "pi = " << pi << endl;

cout << "x = " << x << endl;

cout << "y = " << y << endl;

return 0;

}

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

**Ans: 1. Arithmetic Operators**

Used for mathematical calculations.

| Operator | Meaning | Example |
| --- | --- | --- |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (remainder) | a % b |

Example:

int a = 10, b = 3;

cout << a + b; // 13

cout << a % b; // 1

**2. Relational (Comparison) Operators**

Used to compare values. Result → true (1) or false (0).

| Operator | Meaning | Example |
| --- | --- | --- |
| == | Equal to | a == b |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater or equal | a >= b |
| <= | Less or equal | a <= b |

Example:

int x = 5, y = 10;

cout << (x < y); // 1 (true)

**3. Logical Operators**

Used to combine conditions.

| Operator | Meaning | Example |
| --- | --- | --- |
| && | Logical AND | (x>0 && y>0) |
| ` |  | ` |
| ! | Logical NOT | !(x>0) |

Example:

int x = 5, y = 10;

cout << (x>0 && y>0); // 1 (true)

**4. Assignment Operators**

Used to assign values to variables.

| Operator | Example | Meaning |
| --- | --- | --- |
| = | a = 10 | assign value |
| += | a += 5 | a = a + 5 |
| -= | a -= 3 | a = a - 3 |
| \*= | a \*= 2 | a = a \* 2 |
| /= | a /= 2 | a = a / 2 |
| %= | a %= 3 | a = a % 3 |

**5. Increment and Decrement Operators**

Increase or decrease a variable by 1.

* ++a (pre-increment) → increases first, then uses the value.
* a++ (post-increment) → uses the value, then increases.

Example:

int a = 5;

cout << ++a; // 6 (pre-increment)

cout << a++; // 6 (post-increment, then becomes 7)

**6. Bitwise Operators**

Work at the bit (binary) level.

| Operator | Meaning | Example |
| --- | --- | --- |
| & | Bitwise AND | a & b |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR | a ^ b |
| ~ | Bitwise NOT | ~a |
| << | Left shift | a << 1 |
| >> | Right shift | a >> 1 |

Example:

int a = 5, b = 3; // (binary: a=0101, b=0011)

cout << (a & b); // 1

cout << (a | b); // 7

**7. Conditional (Ternary) Operator**

Short form of if-else.

int a = 10, b = 20;

int max = (a > b) ? a : b;

cout << "Max = " << max;

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

**Ans**: **Constants**

* A constant is a variable whose value cannot be changed during program execution.
* Declared using const keyword or #define preprocessor directive.
* Purpose:
  + Makes program more reliable (no accidental changes).
  + Increases readability and maintainability.

Examples:

#include <iostream>

using namespace std;

int main() {

const double PI = 3.14159; // constant variable

int radius = 5;

double area = PI \* radius \* radius;

cout << "Area = " << area;

return 0;

}

Here, PI is a constant → its value cannot be modified.

**2. Literals**

* A literal is a fixed value directly used in the program.
* It represents data directly instead of storing it in a variable.
* Types of literals in C++:

1. Integer Literals → 10, -45, 0
2. Floating-point Literals → 3.14, -0.99, 2.5e3
3. Character Literals → 'A', 'z', '\n'
4. String Literals → "Hello", "C++ Programming"
5. Boolean Literals → true, false
6. Pointer Literal → nullptr

Example:

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl; // "Hello, World!" is a string literal

cout << 42 << endl; // 42 is an integer literal

cout << 3.14 << endl; // 3.14 is a floating literal

cout << true << endl; // true is a boolean literal

return 0;

}

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

**Ans:** Conditional statements are used to make decisions in a program. They allow the program to execute certain blocks of code only if specific conditions are true.

The main conditional statements in C++ are:

* if
* if-else
* if-else if-else
* switch

1. if-else Statement

* The if condition checks whether an expression is true.
* If it is true, the code inside if runs.
* If it is false, the code inside else runs.

**switch Statement**

* The switch statement is used when we have multiple conditions to check for a single variable.
* Instead of writing many if-else if statements, we can use switch.
* Each possible value is handled by a case.
* The break statement stops execution after a case is matched.
* The default case runs if none of the cases match.

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

**Ans : 1. for loop**

* Used when the number of iterations is known in advance.
* Initialization, condition, and increment/decrement are all in one line.

Syntax:

for(initialization; condition; update) {

// code block

}

**while loop**

* Used when the number of iterations is not known in advance.
* First checks the condition, then executes the loop.
* Entry-controlled loop (condition checked before execution).

Syntax:

while(condition) {

// code block

}

**do-while loop**

* Similar to while but the condition is checked after executing the loop body.
* This means the loop runs at least once, even if the condition is false.
* Exit-controlled loop.

Syntax:

do {

// code block

} while(condition);

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

**Ans: 1. break Statement**

* Used to terminate the loop immediately.
* Control jumps out of the loop and continues after the loop body.

Syntax:

break;

Example (break inside a for loop):

#include <iostream>

using namespace std;

int main() {

for(int i = 1; i <= 10; i++) {

if(i == 5) {

break; // loop ends when i = 5

}

cout << i << " ";

}

return 0;

}

**2. continue Statement**

* Used to skip the current iteration of the loop.
* Control jumps to the next iteration (condition check in while / do-while, update in for).

Syntax:

continue;

Example (continue inside a for loop):

#include <iostream>

using namespace std;

int main() {

for(int i = 1; i <= 10; i++) {

if(i == 5) {

continue; // skip when i = 5

}

cout << i << " ";

}

return 0;

}

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

**Ans:** A *nested control structure* means placing one control structure inside another.  
Control structures include:

* Conditional statements (if-else, switch)
* Looping statements (for, while, do-while)

So nested control structures = *one inside another*.  
For example:

* if inside if (nested if-else)
* loop inside loop (nested loops, e.g., printing patterns)
* loop inside if, etc.

Example 1: Nested if-else

#include <iostream>

using namespace std;

int main() {

int age, marks;

cout << "Enter age: ";

cin >> age;

cout << "Enter marks: ";

cin >> marks;

if(age >= 18) { // outer if

if(marks >= 50) { // inner if

cout << "You are eligible for admission." << endl;

} else {

cout << "You must score at least 50 marks." << endl;

}

} else {

cout << "You must be at least 18 years old." << endl;

}

return 0;

}

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

**Ans:** A function is a block of code that performs a specific task.

* It helps in code reusability (write once, use many times).
* It makes the program more organized and readable.

For example, instead of writing code to calculate the sum in many places, you can write a function sum() and call it whenever needed.

Parts of a Function in C++

A function in C++ has 3 main parts:

1. Function Declaration (Prototype)
2. Function Definition
3. Function Calling

**1. Function Declaration (Prototype)**

* Tells the compiler about the function name, return type, and parameters.
* Written before main() (or in a header file).

Syntax:

return Type functionName(parameter\_list);

Example:

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

**2. Function Definition**

* Actual block of code that performs the task.
* Contains the body of the function.

Syntax:

returnType functionName(parameter\_list) {

// function body

}

Example:

int add(int a, int b) {

return a + b;

}

**3. Function Calling**

* To use the function, we call it inside main() or another function.

Syntax:

functionName(arguments);

Example:

#include <iostream>

using namespace std;

// Function declaration

int add(int a, int b);

// Function definition

int add(int a, int b) {

return a + b;

}

int main() {

int x = 5, y = 7;

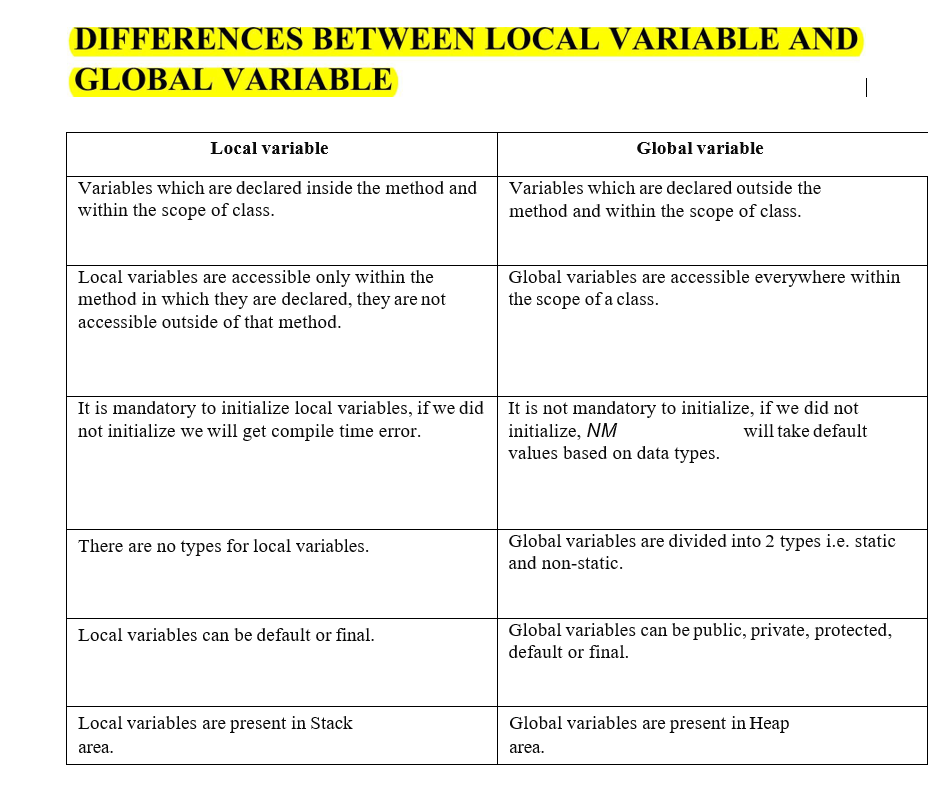
int result = add(x, y); // function call

cout << "Sum = " << result;

return 0;

}

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

**Ans:** 

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

**Ans:** Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem.

* Every recursive function has:
  1. Base Case → condition to stop recursion (prevents infinite calls).
  2. Recursive Case → function calling itself with modified arguments.

**Example 1: Factorial using Recursion**

#include <iostream>

using namespace std;

int factorial(int n) {

if(n == 0 || n == 1) { // base case

return 1;

} else {

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

}

}

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

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

return 0;

}

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

**Ans:** A function prototype in C++ is a declaration of a function that tells the compiler:

* The function’s name
* The function’s return type
* The number and type of parameters

It does not include the function body.

Syntax

returnType functionName(parameter\_list);

Why are Function Prototypes Used?

1. Inform the compiler about the function before it is used.
   * Ensures type checking (correct number and type of arguments).
   * Prevents errors if a function is called before being defined.
2. Allows flexibility in writing code:
   * We can write main() first and define functions later.
3. Improves readability by showing what functions exist in a program.
4. **What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.**

Ans: An array is a collection of elements of the same data type, stored in contiguous memory locations, and accessed using an index.

* Indexing in C++ arrays starts from 0.
* Useful when we need to store and process multiple values of the same type.

Syntax:

dataType arrayName[size];

Example:

int marks[5] = {90, 85, 76, 88, 95};

cout << marks[0]; // prints 90

1. Single-Dimensional Array

* Also called a 1D array or linear array.
* Stores elements in a single row (like a list).

Syntax:

dataType arrayName[size];

Example:

#include <iostream>

using namespace std;

int main() {

int numbers[5] = {10, 20, 30, 40, 50}; // 1D array

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

cout << numbers[i] << " ";

}

return 0;

}

Output:

10 20 30 40 50

2. Multi-Dimensional Array

* Stores data in rows and columns (or more dimensions).
* The most common is the 2D array, which looks like a matrix (table).

Syntax (2D array):

dataType arrayName[rows][columns];

Example:

#include <iostream>

using namespace std;

int main() {

int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} }; // 2D array

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

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

cout << matrix[i][j] << " ";

}

cout << endl;

}

return 0;

}

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

**Ans:** 1. C-Style Strings (Character Arrays)

* These are simple arrays of characters terminated by a null character ('\0').
* Header file: <cstring> provides functions like strlen(), strcpy(), strcat(), strcmp(), etc.

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

// Concatenate strings

strcat(str1, str2);

cout << "Concatenated String: " << str1 << endl;

// Find length

cout << "Length of str1: " << strlen(str1) << endl;

// Copy string

strcpy(str2, "C++");

cout << "Copied String: " << str2 << endl;

// Compare strings

if(strcmp(str1, str2) == 0)

cout << "Strings are equal";

else

cout << "Strings are not equal";

return 0;

}

2. C++ string Class (Preferred Way)

* Provided by the Standard Template Library (STL) in <string>.
* Easier and safer than C-style strings.
* Supports operators like +, =, ==, etc.
* Member functions: length(), substr(), find(), append(), insert(), erase(), etc.

#include <iostream>

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = "World";

// Concatenation using +

string s3 = s1 + " " + s2;

cout << "Concatenated String: " << s3 << endl;

// Length of string

cout << "Length of s3: " << s3.length() << endl;

// Substring

cout << "Substring (0,5): " << s3.substr(0, 5) << endl;

// Find

cout << "Position of 'World': " << s3.find("World") << endl;

// Insert

s3.insert(6, "C++ ");

cout << "After Insertion: " << s3 << endl;

// Erase

s3.erase(6, 4);

cout << "After Erase: " << s3 << endl;

return 0;

}

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

**Ans:** One-Dimensional (1D) Array Initialization

Syntax:

datatype array\_name[size] = {values};

Examples:

#include <iostream>

using namespace std;

int main() {

// Method 1: Full initialization

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

// Method 2: Partial initialization (rest elements become 0)

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

// Method 3: Compiler determines size automatically

int arr3[] = {100, 200, 300};

cout << "1D Array arr1: ";

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

cout << arr1[i] << " ";

}

return 0;

}

2. Two-Dimensional (2D) Array Initialization

A 2D array is like a matrix with rows and columns.

Syntax:

datatype array\_name[rows][cols] = { {row1}, {row2}, ... };

Examples:

#include <iostream>

using namespace std;

int main() {

// Method 1: Full initialization

int arr1[2][3] = { {1, 2, 3}, {4, 5, 6} };

// Method 2: Flattened initialization

int arr2[2][3] = {1, 2, 3, 4, 5, 6};

// Method 3: Partial initialization

int arr3[2][3] = { {1}, {4, 5} }; // => {{1,0,0}, {4,5,0}}

cout << "2D Array arr1:" << endl;

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

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

cout << arr1[i][j] << " ";

}

cout << endl;

}

return 0;

}

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

**Ans:** String Operations with C-style Strings

Stored in a char array and terminated with '\0'.

Common functions (from <cstring>):

* strlen(str) → returns length
* strcpy(dest, src) → copy
* strcat(str1, str2) → concatenate
* strcmp(str1, str2) → compare

Example:

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char s1[20] = "Hello";

char s2[20] = "World";

cout << "Length of s1: " << strlen(s1) << endl;

strcat(s1, s2); // Concatenate

cout << "Concatenated: " << s1 << endl;

strcpy(s2, "C++"); // Copy

cout << "Copied: " << s2 << endl;

if(strcmp(s1, s2) == 0)

cout << "Strings are equal";

else

cout << "Strings are not equal";

return 0;

}

2. String Operations with C++ string Class

Header file: <string>.  
Strings are treated as objects with many built-in functions.

Common Operations & Functions:

* Concatenation: s1 + s2, s1.append(s2)
* Assignment: s1 = "Hello";
* Comparison: ==, !=, <, >
* Length: s1.length() or s1.size()
* Access character: s1[i]
* Substring: s1.substr(pos, length)
* Find: s1.find("word")
* Insert: s1.insert(pos, "text")
* Erase: s1.erase(pos, length)
* Replace: s1.replace(pos, length, "new")

Example:

#include <iostream>

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = "World";

// Concatenation

string s3 = s1 + " " + s2;

cout << "Concatenated: " << s3 << endl;

// Length

cout << "Length: " << s3.length() << endl;

// Substring

cout << "Substring (0,5): " << s3.substr(0, 5) << endl;

// Find

cout << "Position of 'World': " << s3.find("World") << endl;

// Insert

s3.insert(6, "C++ ");

cout << "After Insert: " << s3 << endl;

// Replace

s3.replace(6, 3, "Java");

cout << "After Replace: " << s3 << endl;

// Erase

s3.erase(6, 5);

cout << "After Erase: " << s3 << endl;

return 0;

}

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

* **Ans**: A class is a blueprint or template for creating objects.
* It defines the properties (data members/attributes) and behaviors (methods/functions) of the object.
* Example in C++:
* class Car {
* public:
* string brand;
* int speed;
* void drive() {
* cout << "Car is driving" << endl;
* }
* };

2. Object

* An object is an instance of a class.
* It represents real-world entities and is created using the class definition.
* Example:
* Car c1; // c1 is an object of class Car
* c1.brand = "BMW";
* c1.drive();

3. Encapsulation

* Wrapping of data (variables) and methods (functions) together into a single unit (class).
* Helps in data hiding and controlling access using access specifiers (private, protected, public).
* Example:
* class BankAccount {
* private:
* double balance;
* public:
* void deposit(double amount) { balance += amount; }
* double getBalance() { return balance; }
* };

4. Abstraction

* Showing only the essential details and hiding the complex implementation.
* Achieved using abstract classes and interfaces (in Java) or pure virtual functions (in C++).
* Example:
* class Shape {
* public:
* virtual void draw() = 0; // pure virtual function
* };

5. Inheritance

* Mechanism of acquiring properties and behaviors of one class (base class/parent) into another (derived class/child).
* Promotes code reusability.
* Example:
* class Vehicle {
* public:
* void start() { cout << "Vehicle started" << endl; }
* };
* class Car : public Vehicle {
* public:
* void drive() { cout << "Car is driving" << endl; }
* };

6. Polymorphism

* Means “many forms.”
* The ability of a function, operator, or object to behave differently in different contexts.
* Two types:
  + Compile-time (Static) Polymorphism → Function Overloading, Operator Overloading.
  + Run-time (Dynamic) Polymorphism → Achieved using virtual functions.
* Example:
* // Compile-time polymorphism
* class Math {
* public:
* int add(int a, int b) { return a+b; }
* double add(double a, double b) { return a+b; }
* };
* // Run-time polymorphism
* class Animal {

public:

virtual void sound() { cout << "Animal sound" << endl; }

};

class Dog : public Animal {

public:

void sound() override { cout << "Bark" << endl; }

};

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

**Ans: Class**

* A class is a user-defined data type that acts as a blueprint for creating objects.
* It groups data members (variables) and member functions (methods) together.
* Syntax:
* class ClassName {
* // Access specifiers: public, private, protected
* public:
* // data members
* // member functions
* };

**Object**

* An object is an instance of a class.
* Memory is allocated to an object when it is created.
* Using the object, we can access the class’s data members and functions.

Example: Class and Object in C++

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

string brand;

int speed;

// Member function

void drive() {

cout << brand << " is driving at " << speed << " km/h." << endl;

}

};

int main() {

// Creating objects of class Car

Car car1;

car1.brand = "BMW";

car1.speed = 120;

car1.drive();

Car car2;

car2.brand = "Audi";

car2.speed = 150;

car2.drive();

return 0;

}

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

**Ans:** Inheritance is an Object-Oriented Programming (OOP) feature in which one class (called the derived/child class) acquires the properties and behaviors of another class (called the base/parent class).

It allows code reusability and helps to establish a relationship between classes.

Types of Inheritance in C++

1. Single Inheritance → One base class, one derived class
2. Multiple Inheritance → One derived class inherits from multiple base classes
3. Multilevel Inheritance → A class derived from another derived class
4. Hierarchical Inheritance → Multiple derived classes inherit from the same base class
5. Hybrid Inheritance → Combination of two or more types

Syntax

class Base {

// members

};

class Derived : access\_specifier Base {

// members of derived class

};

* access\_specifier can be public, private, or protected.
  + public inheritance → Base class public members remain public in derived class.
  + private inheritance → Base class public members become private in derived class.
  + protected inheritance → Base class public members become protected in derived class.

Example: Single Inheritance

#include <iostream>

using namespace std;

// Base class

class Vehicle {

public:

void start() {

cout << "Vehicle started!" << endl;

}

};

// Derived class

class Car : public Vehicle {

public:

void drive() {

cout << "Car is driving!" << endl;

}

};

int main() {

Car myCar;

// Accessing base class function through derived class object

myCar.start(); // inherited from Vehicle

myCar.drive(); // defined in Car

return 0;

}

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

**Ans:** Encapsulation is the process of wrapping data (variables) and methods (functions) together into a single unit (class).

* It is one of the four pillars of OOP.
* It helps in data hiding → restricting direct access to class data from outside.

How is Encapsulation Achieved in C++?

1. Using Classes → Data members and functions are defined inside a class.
2. Access Specifiers → Control visibility of class members.
   * private → Accessible only within the class.
   * protected → Accessible within the class and derived classes.
   * public → Accessible from anywhere.
3. Getters and Setters → Provide controlled access to private data members