Abhay Gajjar

Assignment-3

* Theory Questions in C++ *

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

=>

No.	**Procedural Oriented Programming**	**Object-Oriented Programming**
1	The Program is divided into small parts called functions.	The Program is divided into small parts called objects.
2	Procedural programming follows top-down approach.	Object-oriented programming follows a bottom-up approach.
3	There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
4	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.
5	In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.

6	there is no concept of data hiding and inheritance.	the concept of data hiding and inheritance is used.
7	Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
8	Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
9	Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
10	**Examples:** C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc

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

=> This Main Advantages Oops :-

- Objects help in task partitioning in the project.
- Secure programs can be built using data hiding.
- It can potentially map the objects.
- Enables the categorization of the objects into various classes.
- Object-oriented systems can be upgraded effortlessly.
- Redundant codes can be eliminated using inheritance.
- Codes can be extended using reusability.
- Greater modularity can be achieved.

- Data abstraction increases reliability.
- Flexible due to the dynamic binding concept.
- Decouples the essential specification from its implementation by using information hiding.

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

=> C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features.
C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc. Before we start programming with C++. We will need an environment to be set up on our local computer to compile and run our C++ programs successfully. If you do not want to set up a local environment you can also use online IDEs for compiling your program.

Using Online IDE :-

IDE stands for an integrated development environment. IDE is a software application that provides facilities to a computer programmer for developing software. There are many online IDEs available that you can use to compile and run your programs easily without setting up a local development environment.

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

=> In C++, the main input/output operations are handled using streams from the iostream library.

Input Operations:

1. **cin**: Used to take input from the standard input device (u sually the keyboard).

Example =

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


int main()
{
   int number;
   cout << "Enter a number: ";
   cin >> number;
   cout << "You entered: " << number << endl;
   return 0;
}
```

Output Operations:

1. **cout**: Used to output data to the standard output device (usually the screen).

Example =

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

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

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

=> C++ offers a variety of data types to help you define

variables and manage data effectively. Let's go through the pr imary data types:

1.) Basic Data Types

- **int**: Integer type to store whole numbers.

Example = int age = 25;

- **float**: Floating-point type to store decimal numbers.

Example = float temperature = 36.6;

- **double**: Double-precision floating point type, for more precision.

Example = double pi = 3.141592653589793;

- **char**: Character type to store a single character.

Example = char initial = 'A';

- **bool**: Boolean type to store true or false.

Example = bool isRaining = false;

2.) **Derived Data Types**

- **Array**: Collection of elements of the same type.

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

- **Pointer**: Stores the address of another variable.

Example = int x = 10;

Example = int* ptr = &x; // Pointer to an integer

- **Reference**: Another name for an existing variable.

Example = int y = 20;

Example = int& ref = y; // Reference to y

3.) **User-Defined Data Types**

- **Structure**: Custom data type to group different datatype.

Example = struct Person {

```
   string name;
   int age;
   float height;
```

```
};
```

- **Class**: Defines objects that encapsulate data and functions.

Example = class Car {

```
public:
   string brand;
   string model;
   int year;

   void displayInfo() {
      cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year << endl;
   }
};
```

6.) Explain the difference between implicit and explicit type conversion in C++ ?

=> **Implicit Type Conversion :-**

- Automatic conversion of data types by the compiler.
- Occurs when you perform operations involving different

data types, and the compiler automatically converts them to a common type.

Example =

```
int a = 10;

float b = 3.5;

float result = a + b;  // 'a' is implicitly converted to float
```

Explicit Type Conversion :-

- Manual conversion of data types using casting operators.
- You explicitly specify the type conversion using casting operators.

Example =

```
double x = 9.7;

int y = static_cast<int>(x);  // Explicitly converts 'x' to int.
```

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

=> **Operators in C++ can be classified into 6 types:**

1.) Arithmetic Operators

=> Arithmetic Operators are used to perform common mathematical operations.

Example =

```
   // CPP Program to demonstrate the Binary Operators

#include <iostream>

using namespace std;
```

```
int main()
{
    int a = 8, b = 3;

    // Addition operator
    cout << "a + b = " << (a + b) << endl;

    // Subtraction operator
    cout << "a - b = " << (a - b) << endl;

    // Multiplication operator
    cout << "a * b = " << (a * b) << endl;

    // Division operator
    cout << "a / b = " << (a / b) << endl;

    // Modulo operator
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

2.) Relational Operators

=> Relational Operator are used to comparison of the two operands.

Example =

```
// CPP Program to demonstrate the Relational Operators
#include <iostream>
using namespace std;

int main()
{
   int a = 6, b = 4;

   // Equal to operator
   cout << "a == b is " << (a == b) << endl;

   // Greater than operator
   cout << "a > b is " << (a > b) << endl;

   // Greater than or Equal to operator
   cout << "a >= b is " << (a >= b) << endl;

   //  Lesser than operator
```

```
    cout << "a < b is " << (a < b) << endl;

    // Lesser than or Equal to operator
    cout << "a <= b is " << (a <= b) << endl;

    // true
    cout << "a != b is " << (a != b) << endl;

    return 0;
}
```

3.) Logical Operators

=> Logical operator are used to combine two or more condition**.**

Example =

```
// CPP Program to demonstrate the Logical Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;
    // Logical AND operator
    cout << "a && b is " << (a && b) << endl;

    // Logical OR operator
    cout << "a || b is " << (a || b) << endl;

    // Logical NOT operator
    cout << "!b is " << (!b) << endl;

    return 0;
}
```

4.) Bitwise Operators

=> Bitwise Operators allows precise manipulation of bits , giving you control over hardware operations.

Example =

```
// CPP Program to demonstrate the Bitwise Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;

    // Binary AND operator
    cout << "a & b is " << (a & b) << endl;

    // Binary OR operator
    cout << "a | b is " << (a | b) << endl;

    // Binary XOR operator
    cout << "a ^ b is " << (a ^ b) << endl;

    // Left Shift operator
    cout << "a<<1 is " << (a << 1) << endl;

    // Right Shift operator
    cout << "a>>1 is " << (a >> 1) << endl;

    // One’s Complement operator
    cout << "~(a) is " << ~(a) << endl;

    return 0;
}
```

5.) Assignment Operators

=> Assignment operator are used to assign values to variables.

Example =

```
// CPP Program to demonstrate the Assignment Operators
#include <iostream>
using namespace std;

int main()
{
   int a = 6, b = 4;

   // Assignment Operator
   cout << "a = " << a << endl;

   //  Add and Assignment Operator
   cout << "a += b is " << (a += b) << endl;

   // Subtract and Assignment Operator
   cout << "a -= b is " << (a -= b) << endl;

   //  Multiply and Assignment Operator
```

```
    cout << "a *= b is " << (a *= b) << endl;

    //  Divide and Assignment Operator
    cout << "a /= b is " << (a /= b) << endl;

    return 0;
}
```

6.)Ternary or Conditional Operators

=> one liner Condition.

Example =

```
 // CPP Program to demonstrate the Conditional Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 3, b = 4;

    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number is " << result << endl;
```

```
   return 0;
}
```

8.) Explain the purpose and use of constants and literals in C++ ?

=>**Constants** and **literals** play an important role in programming to ensure values remain unchanged and to make code more readable and maintainable.

1.) Constants :-

- **Purpose**: Constant are used to define values that should not change throughout the program.
- **Use**: They enhance code readability, prevent accidental modification, and make maintenance easier.

Example =

```
const int DAYS_IN_WEEK = 7;
const float PI = 3.14159;
```

2.) Literals :-

- **Purpose**: Literals represent fixed values in the code. They are used directly in the program without being assigned to a variable.
- **Use**: Literals simplify code by providing direct value representation.

Example =

```
int age = 25;       // Integer literal
char grade = 'A';   // Character literal
float height = 5.9; // Floating-point literal
bool flag = true;   // Boolean literal.
```

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

=> Conditional statements in Programming, also known as decision-making statements, allow a program to perform different actions based on whether a certain condition is true or false.

1.) If-else Statement :-

- To execute different blocks of code based on whether a condition is true or false.

Syntax =

```
if (condition) {
   // Block of code to execute if the condition is true
} else {
   // Block of code to execute if the condition is false
}
```

Example =

```
#include <iostream>
```

```
using namespace std;

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

    if (number > 0) {
        cout << "The number is positive." << endl;
    } else if (number < 0) {
        cout << "The number is negative." << endl;
    } else {
        cout << "The number is zero." << endl;
    }

    return 0;
}
```

2.) Switch Statement :-

- To execute one block of code out of multiple blocks based on the value of a variable.

Syntax =

```
switch (expression) {
    case value1:
        // Block of code to execute if expression equals value1
        break;
    case value2:
        // Block of code to execute if expression equals value2
        break;
    // More cases...
    default:
        // Block of code to execute if expression doesn't match any case
}
```

Example =

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

int main() {
    int day;
    cout << "Enter a number (1-7) for the day of the week: ";
    cin >> day;

    switch (day) {
        case 1:
```

```
    cout << "Monday" << endl;
    break;
case 2:
    cout << "Tuesday" << endl;
    break;
case 3:
    cout << "Wednesday" << endl;
    break;
case 4:
    cout << "Thursday" << endl;
    break;
case 5:
    cout << "Friday" << endl;
    break;
case 6:
    cout << "Saturday" << endl;
    break;
case 7:
    cout << "Sunday" << endl;
    break;
default:
    cout << "Invalid day number!" << endl;
    }

    return 0;
}
```

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

=>

No.	For-Loop	While-Loop	Do-While Loop
1	for (initialization; condition; increment/decrement) {}	while (condition) { }	do { } while (condition);
2	Declared within the loop structure and executed once at the beginning.	Declared outside the loop; should be done explicitly before the loop.	Declared outside the loop structure
3	Checked before each iteration.	Checked before each iteration.	Checked after each iteration.
4	Executed after each iteration.	Executed inside the loop; needs to	Executed inside the loop; needs to

		be handled explicitly.	be handled explicitly.
5	For loop is entry controlled loop.	while loop is entry controlled loop.	do-while loop is exit controlled loop.

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

=> **Break and Continue Statements in Loops :-**

Break Statement:

- Terminates the loop immediately and transfers control to the statement following the loop.
- Often used to exit the loop when a specific condition is met.

Example =

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

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break;  // Exit the loop when i equals 5
        }
        cout << i << " ";
    }
    // Output: 0 1 2 3 4
    return 0;
}
```

Continue Statement:

- Skip the current iteration of the loop and continues with the next iteration.
- Often used to skip specific conditions within a loop with out terminating the entire loop.

Example =

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

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            continue;  // Skip the iteration when i equals 5
        }
        cout << i << " ";
    }
    // Output: 0 1 2 3 4 6 7 8 9
    return 0;
}
```

12.) Explain nested control structures with an example ?

=>

Nested control structures are simply control structure placed inside other control structure.They allow for complex decision making and repeated actions within different levels of conditions.

Example: Nested if-else and for loops

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

int main() {
    int limit;

    // Asking user for the limit
    cout << "Enter the limit for the multiplication table: ";
    cin >> limit;
    // Outer loop to iterate through each number up to the limit
    for (int i = 1; i <= limit; i++) {
        // Nested if-else to check if the number is even
        if (i % 2 == 0) {
            cout << "Multiplication table for " << i << ":\n";
            // Inner loop to generate the multiplication table
            for (int j = 1; j <= 10; j++) {
                cout << i << " x " << j << " = " << i * j << endl;
            }
            cout << endl; // Adding a blank line for better readability
        } else {
            cout << i << " is an odd number, skipping...\n";
        }
    }

    return 0;
}
```

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

=>

A function is a block of code designed to perform a speci fic task. It helps in organizing code, reducing redundancy, and improving readability.

1.) Function Declaration :-

Syntax :

```
returnType functionName(parameters);
```

2.) Function Calling :-

Syntax :

```
functionName(arguments);
```

3.) Function Declaration :-

Syntax :

```
returnType functionName(parameters) {
   // Body of the function
}
```

Example =

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

// Function declaration
int add(int, int);

int main() {
    int num1 = 10, num2 = 20;

    // Function call
    int sum = add(num1, num2);

    cout << "Sum: " << sum << endl;
    return 0;
}

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

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

=> scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes:

1. Local Variables
2. Global Variables

No.	Local Scope	Global Scope
1	Limited to the block of code	Accessible throughout the program
2	Typically within functions or specific blocks	Outside of any function or block
3	Accessible only within the block where they are declared	Accessible from any part of the program
4	Created when the block is entered and destroyed when it exits	Retain their value throughout the lifetime of the program
5	Can have the same name as variables in other blocks	Should be used carefully to avoid unintended side effects
6	Temporary storage, specific to a block of code	Values that need to be accessed and modified by multiple parts of the program

15.) Explain recursion in C++ with an example ?

=> Recursion in C++ is a technique in which a function calls itself repeatedly until a given condition is satisfied. recursion is the process of solving a problem by breaking it down into smaller, simpler sub-problems.

Syntax =

return_type ***recursive_func***

```
{
  ....
    //BaseCondition
    //RecursiveCase
    ....
}
```

1. **Base Case**: A condition under which the recursive function stops calling itself to prevent infinite recursion.
2. **Recursive Case**: The part of the function that calls itself.

Example =

```
#include <iostream>

using namespace std;


// Recursive function to calculate factorial

int factorial(int n) {

    if (n <= 1) {

        return 1;  // Base case

    } else {

        return n * factorial(n - 1);  // Recursive case

    }

}
```

```
int main() {
   int number;
   cout << "Enter a number to find its factorial: ";
   cin >> number;

   int result = factorial(number);
   cout << "Factorial of " << number << " is " << result << endl;

   return 0;
}
```

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

=> **Function Prototypes** :-

A function prototype is a declaration of a function that specifies the function's name, return type, and parameters, without the body of the function.

Syntax =

```
returnType functionName(parameterType1, parameterType2, ...);
```

Example =

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

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

int main() {
   int a = 5, b = 10;
   int sum = add(a, b);  // Function call
   cout << "Sum: " << sum << endl;
   return 0;
}

// Function definition
int add(int x, int y) {
   return x + y;
}
```

1. **Forward Declaration**: Ensures that the compiler knows about the function before its actual definition. This allows the function to be called before it is defined in the code.
2. **Type Checking**: Helps the compiler to check for correct function usage and parameter types, catching errors early in the compilation process.

3. **Improved Readability**: Provides a clear, concise overview of all the functions used in the program at the beginning, making the code easier to understand and maintain.

17.) What are arrays in C++? Explain the difference between single-dimensional and multidimensional arrays ?

=> An array is a collection of elements of the same type,

stored in contiguous memory locations. Arrays allow you to store multiple values of the same type and access them using an index.

1.) Single-Dimensional Arrays :-

- A single-dimensional array is a linear list of elements.

- Elements arranged in a single row.

Syntax =

datatype variable_name[row]

Example =

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

2.) Multi-Dimensional Arrays :-

- Arrays containing arrays, forming a matrix-like structure.

- Elements arranged in rows and columns.

Syntax =

```
dataType arrayName[size1][size2];
```

Example =

```
int matrix[3][3] = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

18.) Explain string handling in C++ with examples ?

=> strings can be handled using two main approaches : C-style strings (arrays of characters) and C++ Standard Library strings (std::string).

1.) C-Style Strings =

- An array of characters terminated by a null character ('\0').

Declaration =

```
char str[20] = "Hello, World!";
```

Example =

```
#include <cstring>
cout << strlen(str);  // Output: 13 (length of the string)
```

2.) **Standard Library Strings (std::string) =**

- More Flexible and powerful compared to C-Style Strings.

Part of Standard Library.

Declaration =

```
#include <string>

string str = "Hello, World!";
```

Example =

```
string str1 = "Hello, ";

string str2 = "World!";

string result = str1 + str2;

cout << result;  // Output: Hello, World!
```

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

=> Array initialization is the process of assigning/storing elements to an array. The initialization can be done in a single statement or one by one. Note that the first element in an array is stored at index 0, while the last element is stored at index n-1, where n is the total number of elements in the array.

1.) One-Dimensional Arrays =

Initialization =

```
int numbers[5] = {1, 2, 3, 4, 5};  // Declares and initializes an array
```

2.) Two-Dimensional Arrays =

Initialization =

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};  // Declares and initializes a 3x3 array
```

20.) Explain string operations and functions in C++ ?

=>   string handling in C++ with a focus on the std::string class from the Standard Library. It provides versatile and powerful way to work with strings.

String Initialization =

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

int main() {
    string str1 = "Hello";
    string str2("World");
    string str3 = str1 + " " + str2;  // Concatenation
    cout << str3 << endl;  // Output: Hello World
    return 0;
}
```

Common String Operation and Function =

1.) String Length

Function = length() , size().

Example =

```
string str = "Hello, World!";
cout << "Length: " << str.length() << endl;  // Output: 13
```

2.) **Concatenation**

Function = +

Example =

```
string str1 = "Hello";
string str2 = "World";
string result = str1 + " " + str2;
cout << result << endl;  // Output: Hello World
```

3.) Substring

Function = substr(startIndex , length)

Example=

```
string str = "Hello, World!";
string sub = str.substr(7, 5);
cout << sub << endl;  // Output: World
```

4.) **Find**

Function = find(substring)

Example =

```
size_t pos = str.find("World");
if (pos != string::npos) {
   cout << "'World' found at: " << pos << endl;  // Output: 'World' found at: 7
}
```

5.) Replace

Function = replace(startIndex , length, newString)

Example =

```
str.replace(7, 5, "Universe");
cout << str << endl;  // Output: Hello, Universe!
```

6.) Comparison

Function = ==, >,< ,>=,<= , !=

Example =

```
string str1 = "Hello";
string str2 = "World";
if (str1 != str2) {
   cout << "Strings are not equal." << endl;  // Output: Strings are not equal.
}
```

7.) Insertion

Function = insert (position , substring)

Example =

```
str.insert(5, " Beautiful");
cout << str << endl;  // Output: Hello Beautiful, Universe!
```

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

=> **1.) Class =** A class is a data-type that has its own members i.e. data members and member functions. It is the blueprint for an object

Properties =

- **Class** is a user-defined data-type.
- A class contains members like data members and member functions.
- **Data members** are variables of the class.
- **Member functions** are the methods that are used to manipulate data members.

Syntax =

```
class class_name {
  data_type data_name;
  return_type method_name(parameters);
}
```

2.) Object = An object is an instance of a class.

- An object is the entity that is created to allocate memory.

Syntax =

```
class_name object_name;
```

3.) Encapsulation = Encapsulation is defined as wrapping up data and information under a single unit.

Example =

```
class Car {
private:
   int speed;
public:
   void setSpeed(int s) { speed = s; }
   int getSpeed() { return speed; }
};
```

4.) Polymorphism = The Word polymorphism means having manyforms.Allows objects of different classes to be treated a s objects of a common base class.

Example =

```
class Shape {
public:
    virtual void draw() {
        cout << "Drawing Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

void displayShape(Shape* shape) {
    shape->draw();
}

Shape* shape = new Circle();
displayShape(shape);  // Output: Drawing Circle
```

5.) Inheritance =

A mechanism where a new class inherits properties and behavior (methods) from an existing class.

Example =

```
class Vehicle {
public:
   int wheels;
};


class Car : public Vehicle {
public:
   string model;
};
```

6.) Abstraction =

Hiding the complex implementation details and showing only the essential features of the object.

Example =

```
class CoffeeMachine {
public:
   void makeCoffee() {
      boilWater();
      brewCoffee();
      pourCoffee();
   }
private:
   void boilWater() { cout << "Boiling water" << endl; }
   void brewCoffee() { cout << "Brewing coffee" << endl; }
   void pourCoffee() { cout << "Pouring coffee" << endl; }
};
```

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

=> **1.) Class =** A class is a data-type that has its own members i.e. data members and member functions. It is the blueprint for an object

Properties =

- **Class** is a user-defined data-type.
- A class contains members like data members and member functions.
- **Data members** are variables of the class.
- **Member functions** are the methods that are used to manipulate data members.

Syntax =

```
class class_name {
  data_type data_name;
  return_type method_name(parameters);
};
```

2.) Object = An object is an instance of a class.

- An object is the entity that is created to allocate memory.

Syntax =

class_name object_name;

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

=> The inheritance can be classified on the basis of the relationship between the derived class and the base class.

=> we have 5 types of inheritances:

1. **Single inheritance**
2. **Multilevel inheritance**
3. **Multiple inheritance**
4. **Hierarchical inheritance**
5. **Hybrid inheritance**

1.) Single Inheritance =

- single inheritance, a class is allowed to inherit from only one class. i.e. one base class is inherited by one derived class only.

Syntax =

Class *subclass_name* : *access_mode* *base_class*
{

// body of subclass
};

Example =

```
Class A
{
        .....
};
Class B : public A
{
        .....
};
```

2.) Mutilevel Inheritance =

- derived class is created from another derived class and that derived class can be derived from a base class or any other derived class. There can be any number of levels.

Syntax =

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier derived_class1
{
... .. ...
}
```

Example =

```
Class A
{
        .....
};
Class B : public A
{
        ……
};
Class C : public A
{
        ……
};
```

3.) Mutiple Inheritance =

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.

Syntax =

```
Class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
 // body of subclass
};
```

Example =

```
Class A
{
        …..
};
Class B
{
        …..
};
Class C : public B , public A
{
        …..
};
```

4.) Hierarchical Inheritance =

- more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

Example =

```
Class A
{
};
Class B : public A
{
};
Class C : public A
{
};
```

5.) Hybrid Inheritance =

- Hybrid Inheritance is implemented by combining more than one type of inheritance.

Example =

```
Class F
{
};
Class G
{
};
Class B : public F
{
};
Class E : public F, public G
{
};
Class A : Class B
```

```
{
};
Class C : Class B
{
};
```

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

=>Encapsulation in C++ is defined as the wrapping up of data and information in a single unit.

- **It Achieved in classes :-**

1. **Access Specifiers**: Encapsulation is implemented using access specifiers to define the visibility and accessibility of class members. The three access specifiers are:
   - **private**: Members declared as private are accessible only within the same class.
   - **protected**: Members declared as protected are accessible within the same class and derived classes.
   - **public**: Members declared as public are accessible from any part of the program.
2. **Getters and Setters**: These are public methods used to access and modify private data members. They provide controlled access to the private data.

Example =

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

class Rectangle {
private:
    double length;
    double width;

public:
    // Setter for length
    void setLength(double len) {
        if (len > 0) {
            length = len;
        } else {
            cout << "Length must be positive." << endl;
        }
    }

    // Getter for length
    double getLength() {
        return length;
    }
};



    // Setter for width
    void setWidth(double wid) {
        if (wid > 0) {
            width = wid;
        } else {
            cout << "Width must be positive." << endl;
        }
    }

    // Getter for width
    double getWidth() {
        return width;
    }

    // Method to calculate area
    double area() {
        return length * width;
    }
};

int main() {
    Rectangle rect;
    rect.setLength(5.0);
    rect.setWidth(3.0);
    cout << "Length: " << rect.getLength() << endl;
    cout << "Width: " << rect.getWidth() << endl;
    cout << "Area: " << rect.area() << endl;

    return 0;
}
```