Module #3 Introduction to OOPS Programming

1. Introduction to C++

Question: 1

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

Answer:

Key Differences Between Procedural Programming and Object-Oriented Programming (OOP)

Feature	**Procedural Programming (POP)**	**Object-Oriented Programming (OOP)**
Basic	Procedure/ Structure oriented	Object oriented
Concept	Program is divided into functions (procedures).	Program is divided into objects (which combine data and functions).
Focus	Focuses on functions and logic.	Focuses on data (objects) and how they interact.
Data Handling	Data is global and can be freely accessed by functions.	Data is hidden inside objects and accessed using methods.
Security	Less secure — data can be accidentally modified.	More secure — uses encapsulation to protect data.
Reusability	Code reusability is low; functions are reused but not data.	High reusability through inheritance and polymorphism.
Examples	C, Pascal, BASIC	C++, Java, Python, C#
Data and Functions Relationship	Data moves freely between functions.	Data and functions are bound together inside objects.
Approach	Top-down approach (start from main function → sub-functions).	Bottom-up approach (start by designing classes and objects).

❖ **Conclusion:**
Procedural programming focuses on functions and step-by-step logic, while OOP centres around objects combining data and behaviour. OOP makes programs more organized, reusable, and easier to maintain.

Question: 2

List and explain the main advantages of Object-Oriented Programming (OOP) over Procedural Programming (POP).

Answer:

❖ **Introduction:**

Object-Oriented Programming (OOP) is a modern programming approach that focuses on objects rather than functions.

It helps organize and manage complex programs easily by combining data and the methods (functions) that work on that data.

Compared to Procedural Programming (POP), OOP offers several key advantages.

❖ **Main Advantages of OOP over POP:**

1. **Data Security (Encapsulation):**

   In OOP, data is hidden inside classes and accessed only through methods.

   This prevents accidental modification and ensures data protection.

   Example: Private data members in a class cannot be accessed directly from outside.

2. **Code Reusability (Inheritance):**

   OOP allows one class to inherit properties and behaviours of another class.

   This avoids writing the same code multiple times and promotes reuse.

   Example: A “Car” class can inherit common features from a “Vehicle” class.

3. **Easier Maintenance:**

   Since OOP divides the program into small, independent objects, any change can be made easily without affecting the entire program.

   This makes programs easier to update and maintain.

4. **Modularity and Organization:**

   OOP programs are divided into classes and objects, making them more structured and organized.

   Each class performs a specific role, improving clarity and teamwork in large projects.

5. **Flexibility (Polymorphism):**

   OOP allows the same function or operator to behave differently based on context.

   Example: A function named area() can calculate the area of a circle or rectangle depending on the input.

6. **Real-World Modeling:**

   OOP is based on the concept of real-world entities like students, cars, or employees.

   This makes program design more natural, logical, and easier to understand.

7. **Improved Productivity:**

   Reusability, modularity, and easier debugging help developers work faster and more efficiently.

   Once classes are created, they can be reused in many projects.

❖ **Conclusion:**

In summary, OOP provides better structure, security, reusability, and maintainability compared to Procedural Programming.

That’s why most modern programming languages like C++, Java, and Python use the Object-Oriented approach to build robust and scalable applications.

Question: 3

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

Answer:

To write and run C++ programs, we need to set up a C++ development environment on our computer.

❖ **Here are the main steps:**

1. **Install an IDE or Compiler:**
   - Download and install a C++ compiler or IDE such as Dev C++, Code::Blocks, or Turbo C++.
   - These tools help to write, compile, and run C++ programs easily.
2. **Configure the IDE:**
   - After installation, open the IDE and check that the compiler path (like GCC or MinGW) is set correctly.
   - This ensures the program can compile properly.
3. **Create a New Project or File:**
   - Open the IDE → Click on New Project or New File → Choose C++ Source File → Save it with a .cpp extension.
   - Example: hello.cpp
4. **Write the Program:**
   - Type your C++ code in the editor window.
   - **Example:**

```
#include<iostream>

using namespace std;

  int main() {

  cout << "Hello, World!";

  return 0;

}
```

5. **Compile the Program:**
   - Click on Compile or press F9 (in most IDEs).
   - The compiler checks for errors and converts the code into machine language.
6. **Run the Program:**
   - After successful compilation, click Run or press F10 to see the output on the screen.

❖ **Conclusion:**

By installing and configuring an IDE or compiler, creating a new file, writing, compiling, and running the code, we can successfully set up and use a C++ development environment.

Question 4:

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

Answer:

- **Definition:**

  Input/Output operations in C++ are used to take data from the user (input) and show results on the screen (output).

  C++ uses special objects called cin (for input) and cout (for output).

- **These objects are defined in the iostream header file.**

1. **Input Operation – cin:**

   Used to get input from the user (keyboard).

   - **Syntax:**

     ```
     cin >> variable_name;
     ```

   - **Example:**

     ```
     int age;

     cout << "Enter your age: ";

     cin >> age;
     ```

2. **Output Operation – cout:**

   Used to display output on the screen.

   - **Syntax:**

     ```
     cout << message;
     ```

   - **Example:**

     ```
     cout << "Hello, World!";
     ```

- **Complete Example Program:**

```
#include<iostream>

using namespace std;


int main() {

   int num;

   cout << "Enter a number: ";

   cin >> num;
```

```
   cout << "You entered: " << num;
   return 0;
}
```

- **Output:**

```
Enter a number: 5
You entered: 5
```

- **Conclusion:**

  The main input/output operations in C++ are cin (for input) and cout (for output), which make communication between the user and the program easy.

2. Variables, Data Types, and Operators

Question: 1

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

Answer:

❖ **Definition:**

In C++, data types define the type of data that a variable can store.

They tell the compiler how much memory is required and what kind of value will be stored.

❖ **Types of Data Types in C++:**

C++ data types are mainly divided into three categories:

1. **Basic (or Primitive) Data Types**

These are the fundamental types used to store simple values.

Data Type	**Description**	**Example**
int	Used to store integer (whole) numbers	int age = 20;
float	Used to store decimal numbers	float price = 45.6;
double	Used to store large decimal numbers with more precision	double pi = 3.14159;
char	Used to store a single character	char grade = 'A';
bool	Used to store true or false value	bool pass = true;
void	Represents no return value (Used in functions)	void display();

2. **Derived Data Types**

These are made from basic data types.

Type	**Description**	**Example**
Array	Collection of same data type elements	int marks[5] = {80, 85, 90, 75, 95};
Pointer	Store address of another variable	int*ptr = &age;
Function	Group of statements performing a task	int add(int a, int b)

3. User-defined Data Types

Created by the programmer to represent complex data.

Type	Description	Example
Structure	Used to group different data types	struct Student {int id; char name[20];};
Class	Used in OOP to define objects	class Car {public: string brand;};
Enum	Used to define named constants	enum Color {Red, Green, Blue};

❖ **Example Program:**

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

int main() {
   int age = 18;
   float weight = 55.6;
   char grade = 'A';
   bool isStudent = true;

   cout << "Age: " << age << endl;
   cout << "Weight: " << weight << endl;
   cout << "Grade: " << grade << endl;
   cout << "Student: " << isStudent << endl;

   return 0;
}
```

○ **Output:**

```
Age: 18
Weight: 55.6
Grade: A
Student: 1
```

- **Conclusion:**

  C++ provides different data types to store various kinds of information like integers, characters, floating-point numbers, and user-defined structures, making it a powerful and flexible language.

Question: 2

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

Answer:

- **Definition:**

  In C++, type conversion means changing the data type of a variable from one type to another. It helps when we need to perform operations between different data types.

- **Types of conversion:**

  There are two types of type conversions in C++:

  1. Implicit
  2. Explicit.

1. Implicit Type Conversion (Type Casting or Type Promotion)

- Also known as Automatic Type Conversion.
- It is done automatically by the compiler without any user involvement.
- The compiler converts smaller data types into larger data types to avoid data loss.
- This follows a type promotion hierarchy: char → int → float → double
- **Example:**

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

int main() {
  int num = 10;
  float result = num + 5.5;  // int automatically converted to float

  cout << "Result = " << result;
  return 0;
}
```

  - **Output:**

```
Result = 15.5
```

  Here, int num is automatically converted to float during the addition.

2. **Explicit Type Conversion (Type Casting)**
   - Also known as Manual Type Conversion.
   - It is done manually by the programmer using casting operators.
   - The programmer specifies which type to convert the variable into.
   - **Syntax:**
     (data_type) variable;
   - **Example:**

```
#include<iostream>

using namespace std;


int main() {

   double value = 9.78;

   int result = (int)value;   // Explicit conversion


   cout << "Result = " << result;

   return 0;

}
```

   - **Output:**

```
Result = 9
```

     Here, double value is manually converted to int, so the decimal part is removed.

❖ **Difference between Implicit and Explicit Type Conversion**

Basis	**Implicit Conversion**	**Explicit Conversion**
Also called	Type Promotion / Automatic	Type Casting / Manual
Who performs it	Compiler	Programmer
Conversion type	Done automatically	Done manually
Syntax	No special syntax	Uses casting operator (type)
Risk of data loss	Less	Possible (if not handled properly)

❖ **Conclusion:**

Implicit conversion is done automatically by the compiler, while explicit conversion is controlled by the programmer using casting. Both are useful for handling different data types in C++ programs safely and effectively.

Question: 3

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

Answer:

- **Definition:**

  In C++, operators are special symbols used to perform operations on variables and values.

  For example: +, -, *, /, =, ==, etc.

- **Types of operators in C++:**

  Operators are mainly divided into the following types:

  1. Arithmetic Operators
  2. Relational Operators
  3. Logical Operators
  4. Assignment Operators
  5. Increment And Decrement Operators
  6. Conditional (Ternary) Operator
  7. Bitwise Operators
  8. Special Operators

1. Arithmetic Operators

Used to perform mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	6 / 3	2
%	Modulus (remainder)	7 % 3	1

2. Relational Operators

Used to compare two values. The result is either true (1) or false (0).

Operator	Description	Example	Result
==	Equal to	5 == 3	false
!=	Not equal to	5 != 3	true
>	Greater than	5 > 3	true
<	Less than	5 < 3	false
>=	Greater than or equal to	5 >= 3	true
<=	Less than or equal to	5 <= 3	false

3. Logical Operators

Used to combine two or more conditions.

Operator	Description	Example	Result
&&	Logical AND	(5 > 3 && 4 > 2)	true
||	Logical OR	(5 > 3 || 4 < 2)	true
!	Logical NOT	!(5 > 3)	false

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Result
=	Assign value	x = 10	x = 10
+=	Add and assign	x += 5	x = 15
-=	Subtract and assign	x -= 3	x = 12
*=	Multiply and assign	x *= 2	x = 24
/=	Divide and assign	x /= 2	x = 6
%=	Modulus and assign	x %= 3	x = 0

5. Increment and Decrement Operators

Used to increase or decrease a variable by 1.

Operator	Description	Example	Result
++x	Pre – Increment	x = 5; y = ++x;	x = 6, y = 6
x++	Post – Increment	x = 5; y = x++;	x = 6, y = 5
--x	Pre – Decrement	x = 5; y = --x;	x = 4, y = 4
x--	Post – Decrement	x = 5; y = x--;	x = 4, y = 5

6. Conditional (Ternary) Operator

The Ternary Operator in C++ is a conditional operator that used as a short form of if-else statement in a single line.

It is called “ternary” because it works with three operands.

- **Syntax:**

  ```
  condition ? expression1 : expression2;
  ```

  - If the condition is true, then expression1 is executed.
  - If the condition is false, then expression2 is executed.

- **Example:**

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

int main() {
   int a = 10, b = 20;
   int max;

   max = (a > b) ? a : b;   // Using ternary operator
   cout << "Maximum number is: " << max;
   return 0;
}
```

- **Output:**

```
Maximum number is: 20
```

7. Bitwise Operators

Used to perform operations on bits.

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

- **Example:**

```
#include<iostream>
using namespace std;
int main() {
   int a = 5, b = 3;
   cout << "a = " << a << ", b = " << b << endl;
   cout << "a & b = " << (a & b) << endl;
```

```
    cout << "a | b = " << (a | b) << endl;

    cout << "a ^ b = " << (a ^ b) << endl;

    cout << "~a = " << (~a) << endl;

    cout << "a << 1 = " << (a << 1) << endl;

    cout << "a >> 1 = " << (a >> 1) << endl;


    return 0;

}
```

- **Output:**

```
a = 5, b = 3

a & b = 1

a | b = 7

a ^ b = 6

~a = -6

a << 1 = 10

a >> 1 = 2
```

8. **Special Operators**

Operator	**Description**	**Example**
sizeof	Returns size of data type	sizeof(int)
,	Comma operator	x = (a++, b++)
::	Scope resolution operator	::x
&	Address of variable	&a
*	Pointer to variable	*ptr

❖ **Example Program: Demonstration of All Operatorsbin C++**

```
#include<iostream>

using namespace std;


int main() {

    int a = 10, b = 5, result;

    bool check;
    cout << "=== Demonstration of All Operators in C++ ===" << endl;

    // 1. Arithmetic Operator
    result = a + b;
    cout << "\n1. Arithmetic Operator (+): " << a << " + " << b << " = " << result << endl;

    // 2. Relational Operator
    check = (a > b);
    cout << "2. Relational Operator (>): " << a << " > " << b << " = " << check << endl;

    // 3. Logical Operator
    check = (a > b && b < 20);
    cout << "3. Logical Operator (&&): (" << a << " > " << b << " && " << b << " < 20) = " <<
check << endl;

    // 4. Assignment Operator
    result = a;  // Assigning value
    cout << "4. Assignment Operator (=): result = " << result << endl;

    // 5. Increment / Decrement Operator
    cout << "5. Increment Operator (++): Before = " << a;
    a++;
    cout << ", After = " << a << endl;

    // 6. Conditional (Ternary) Operator
    result = (a > b) ? a : b;
    cout << "6. Conditional Operator (?:): Greater value between " << a << " and " << b << "
= " << result << endl;
```

```
    // 7. Bitwise Operator
    result = a & b;
    cout << "7. Bitwise Operator (&): " << a << " & " << b << " = " << result << endl;

    return 0;
}
```

- **Output:**

```
=== Demonstration of All Operators in C++ ===
1. Arithmetic Operator (+): 10 + 5 = 15
2. Relational Operator (>): 10 > 5 = 1
3. Logical Operator (&&): (10 > 5 && 5 < 20) = 1
4. Assignment Operator (=): result = 10
5. Increment Operator (++): Before = 10, After = 11
6. Conditional Operator (?:): Greater value between 11 and 5 = 11
7. Bitwise Operator (&): 11 & 5 = 1
```

❖ **Conclusion:**

C++ provides a wide range of operators to perform different kinds of operations such as arithmetic, logical, relational, and more. Understanding them helps in writing efficient and effective programs.

Question: 4

Explain the Purpose and Use of Constants and Literals in C++.

Answer:

❖ **Definition:**

In C++, constants are values that cannot be change during the execution of a program.

Literals are the fixed values that are directly written in the program.

Literals are fixed values used directly in the code to represent constant data like numbers, characters, or strings.

❖ **Purpose of Constants:**

1. **To prevent accidental changes:**

   Constants keep important values fixed throughout the program.

2. **To make code readable and maintainable:**

   Using meaningful constant names makes code easier to understand.

3. **To avoid hardcoding values:**

   Instead of writing the same number multiple times, use a named constant.

❖ **Declaring Constants in C++**

There are two main ways to define constants:

1. **Using const keyword**

   ```
   const int MAX = 100;
   ```

   The value of MAX cannot be changed after declaration.

2. **Using #define preprocessor**

   ```
   #define PI 3.14159
   ```

   Defines a symbolic constant before compilation.

❖ **Example of Constants:**

```
#include<iostream>

using namespace std;


#define PI 3.14159   // Constant using #define


int main() {
```

```
    const int MAX_STUDENTS = 50;  // Constant using const keyword
    int radius = 5;
    float area;

    area = PI * radius * radius;   // Using constant

    cout << "Maximum Students: " << MAX_STUDENTS << endl;
    cout << "Area of Circle: " << area << endl;

    return 0;
  }
```

- **Output:**

```
Maximum Students: 50
Area of Circle: 78.5397
```

❖ **Literals in C++**

Literals are constant values used directly in the program.

Type of Literal	**Example**	**Description**
Integer Literal	10, -5	Whole numbers
Floating-point Literal	3.14, -0.5	Decimal numbers
Character Literal	'A', '@'	Single characters
String Literal	"Hello"	Sequence of characters
Boolean Literal	true, false	Logical values

❖ **Example of Literals:**

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

int main() {
  int age = 20;          // Integer literal
  float pi = 3.14;       // Floating-point literal
  char grade = 'A';      // Character literal
```

```
    string name = "Dharini"; // String literal
    bool pass = true;        // Boolean literal

    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Grade: " << grade << endl;
    cout << "PI Value: " << pi << endl;
    cout << "Passed: " << pass << endl;

    return 0;
}
```

- **Output:**

```
Name: Dharini
Age: 20
Grade: A
PI Value: 3.14
Passed: 1
```

❖ **Conclusion:**

Constants are used to store fixed values that cannot be changed.

Literals are the actual fixed values used in the program.

They both help make programs secure, clear, and easy to maintain.

3. Control Flow Statements

Question: 1

What are Conditional Statements in C++? Explain the if-else and switch statements.

Answer:

❖ **Definition:**

Conditional Statements in C++ are used to make decisions in a program based on certain conditions.

They allow the program to execute different blocks of code depending on whether a condition is true or false.

In simple words — conditional statements help the program decide "what to do next."

❖ **Types of Conditional Statements in C++**

1. **if Statement**

   - **Purpose:**

     The if statement is used to check a condition.

     If the condition is true, the statement inside the if block executes; otherwise, it is skipped.

   - **Syntax:**

     ```
     if (condition) {
         // Code executes if condition is true
     }
     ```

   - **Example:**

     ```
     int age = 20;

     if (age >= 18) {
         cout << "You are eligible to vote.";
     }
     ```

     Output:

     ```
     You are eligible to vote.
     ```

2. **if-else Statement**
   - **Purpose:**

     The if-else statement is used when we need to perform one action if the condition is true, and another action if the condition is false.
   - **Syntax:**

     ```
     if (condition) {
        // Executes when condition is true
     }
     else {
        // Executes when condition is false
     }
     ```
   - **Example:**

     ```
     int number = 5;
     if (number % 2 == 0)
     {
        cout << "Even number";
     }
     else
     {
         cout << "Odd number";
     }
     ```

     Output:

     ```
     Odd number
     ```

3. **Nested if Statement**

- **Purpose:**

  A nested if means an if statement inside another if statement.

  It is used to test multiple conditions.

- **Syntax:**

```
if (condition1) {
   if (condition2) {
      // Executes if both conditions are true
   }
}
```

- **Example:**

```
int marks = 85;

if (marks >= 40) {
   if (marks >= 75)
      cout << "Distinction";
   else
      cout << "Pass";
}
else {
   cout << "Fail";
}
```

  Output:

```
Distinction
```

4. **else-if Ladder**

- **Purpose:**

  The else-if ladder is used when there are multiple conditions to check.

  The program tests each condition one by one until one is true.

- **Syntax:**

```
if (condition1) {
   // Code 1
}
else if (condition2) {
   // Code 2
}
else if (condition3) {
   // Code 3
}
else {
   // Default code
}
```

- **Example:**

```
int marks = 68;

if (marks >= 85)
{
   cout << "Grade A";
}
else if (marks >= 70)
{
   cout << "Grade B";
}
else if (marks >= 50)
```

```
{
   cout << "Grade C";
}
Else
{
   cout << "Fail";
}
```

Output:

```
Grade C
```

5. **Switch Statement**
   - **Purpose:**

     The switch statement is used to test the value of a variable against multiple possible cases.

     It is a clean alternative to using many if-else statements.

   - **Syntax:**

```
switch (expression) {
   case value1:
      // Code to execute
      break;

   case value2:
      // Code to execute
      break;

   default:
      // Code to execute if no case matches
}
```

- **Example:**

```
int day = 3;


switch (day) {
   case 1:
      cout << "Monday";
      break;
   case 2:
      cout << "Tuesday";
      break;
   case 3:
      cout << "Wednesday";
      break;
   default:
      cout << "Invalid day";
}
```

Output:

```
Wednesday
```

❖ **Summary Table:**

Condition Type	**Use**	**Example**	**When executed**
if	To check a single condition	if (a > b)	When condition is true
if-else	To choose between two options	if (a > b) else	When one condition true or false
nested if	To check one condition inside another	if (a > 0) { if (a < 10) ... }	When multiple conditions true
else-if ladder	To test many conditions	if ... else if ... else	When multiple possible values
switch	To select one case from many	switch (choice)	When value matches a case

- **Conclusion:**

  Conditional statements are essential in C++ programming.

  They allow programs to make logical decisions, control program flow, and execute code selectively based on conditions.

Question: 2

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

Answer:

❖ **Definition:**

In C++, loops are used to repeat a block of code multiple times until a specific condition becomes false.

❖ **Types of loops:**

The three main types of loops are — for, while, and do-while.

Each loop has a different way of checking the condition and executing the statements.

1. **for Loop**
   - **Purpose:**

     Used when the number of iterations is known in advance.

   - **Syntax:**

     ```
     for(initialization; condition; increment/decrement)
     {
        // Code to execute
     }
     ```

   - **Example:**

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

     int main() {
        for(int i = 1; i <= 5; i++)
        {
           cout << i << " ";
        }
        return 0;
     }
     ```

   - **Output:**

     ```
     1 2 3 4 5
     ```

- **Explanation:**

  The loop starts with i = 1

  Runs while i <= 5

  Increases i by 1 after each iteration.

2. while Loop

- **Purpose:**

  Used when the number of iterations is not known beforehand.

  It checks the condition before executing the loop body.

- **Syntax:**

```
while(condition)
{
  // Code to execute
}
```

- **Example:**

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

int main() {
  int i = 1;
  while(i <= 5) {
    cout << i << " ";
    i++;
  }
  return 0;
}
```

- **Output:**

```
1 2 3 4 5
```

- **Explanation:**

  The condition i <= 5 is checked before executing the loop.

  If the condition is false initially, the loop never executes.

3. **do-while Loop**

- **Purpose:**

  Used when you want the loop to run at least once, even if the condition is false.

  It checks the condition after executing the loop body.

- **Syntax:**

```
do
{
   // Code to execute
}
while(condition);
```

- **Example:**

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

int main() {
   int i = 1;
   do {
      cout << i << " ";
      i++;
   } while(i <= 5);
   return 0;
}
```

  - **Output:**

```
1 2 3 4 5
```

- **Explanation:**

  The loop body runs first, then checks the condition.

  Ensures the code executes at least once.

- **Difference Between for, while, and do-while Loops**

Feature	**for Loop**	**while Loop**	**do-while Loop**
Condition Checking	Before loop execution	Before loop execution	After loop execution
Use Case	When number of iterations is known	When number of iterations is unknown	When loop must run at least once
Syntax Simplicity	All parts (init, condition, increment) in one line	Parts written separately	Condition checked at end
Minimum Execution	0 times	0 times	1 time
Example	for(i=1;i<=5;i++)	while(i<=5)	do { } while(i<=5);

- **Conclusion:**

  Use for loop when you know how many times to repeat.

  Use while loop when you don’t know the number of repetitions.

  Use do-while loop when the code must execute at least once, regardless of condition.

Question: 3

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

Answer:

❖ **Definition:**

In C++, break and continue are jump statements used to control the flow of loops.

They help in altering the normal execution of for, while, or do-while loops.

❖ **Types of Statements:**

1. **break Statement**
   - **Purpose:**

     The break statement is used to exit the loop immediately, even if the loop condition is still true.

     When break executes, control moves out of the loop.

   - **Syntax:**

     break;

   - **Example of break:**

```
#include<iostream>

using namespace std;


int main() {

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

      if(i == 6)

         break; // stop loop when i becomes 6

      cout << i << " ";

   }

   return 0;

}
```

   - **Output:**

```
1 2 3 4 5
```

- **Explanation:**

  The loop runs from 1 to 10.

  When i becomes 6, the break statement stops the loop immediately.

  So, only numbers 1 to 5 are printed.

2. **continue Statement**

- **Purpose:**

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

  It does not exit the loop; it just skips the remaining statements in the loop body for that iteration.

- **Syntax:**

  continue;

- **Example of continue:**

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

int main() {
  for(int i = 1; i <= 10; i++) {
    if(i == 5)
      continue; // skip printing 5
    cout << i << " ";
  }
  return 0;
}
```

- **Output:**

```
1 2 3 4 6 7 8 9 10
```

- **Explanation:**

  The loop runs from 1 to 10.

  When i equals 5, the continue statement skips that iteration.

  The loop does not stop; it continues printing the rest of the numbers.

❖ **Difference Between break and continue**

Feature	break	continue
Meaning	Exits the loop completely	Skips current iteration and continues next
Control Flow	Transfers control outside the loop	Transfers control to the next iteration
Use Case	When loop must stop early	When you want to skip certain values
Example Result	Stops printing after a value	Skips a specific value

❖ **Conclusion:**

break is used to terminate a loop early.

continue is used to skip certain iterations.

Both help in controlling the flow of loops effectively and make programs more flexible.

Question: 4

Explain Nested Control Structures with an example.

Answer:

❖ **Definition:**

In C++, a nested control structure means one control structure inside another.

It allows more complex decision-making and repetition by combining loops and conditional statements.

In simple words, “nested” means one inside another, such as an if statement inside another if, or a loop inside another loop.

❖ **Types of Nested Control Structures:**

1. Nested if statement
2. Nested loops (for, while, do-while)
3. Nested if-else Ladder
4. Nested Loop with if Condition

1. **Nested if Statement**

A nested if means placing one if statement inside another.

It is used when one condition depends on another condition.

- **Syntax:**

```
if (condition1) {
    if (condition2) {
        // Executes if both conditions are true
    }
}
```

- **Example:**

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

int main() {
    int marks;
    cout << "Enter your marks: ";
```

```
   cin >> marks;


   if (marks >= 40) {
      if (marks >= 75)
         cout << "Grade: Distinction";
      else
         cout << "Grade: Pass";
   }
   else {
 cout << "Grade: Fail";
        }
        return 0;
     }
```

- **Output:**

```
Enter your marks: 82
Grade: Distinction
```

- **Explanation:**

  The outer if checks if the student passed (marks ≥ 40).

  The inner if checks if the student scored distinction (marks ≥ 75).

  If neither condition is met, it prints “Fail.”

2. **Nested Loop**

   A loop inside another loop is called a nested loop.

   Used for tables, patterns, and matrix operations.

   - **Syntax:**

```
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= m; j++) {
      // Inner loop statements
   }
}
```

- **Example:**

```
#include<iostream>
using namespace std;
int main() {
  for(int i = 1; i <= 3; i++) {      // Outer loop
    for(int j = 1; j <= 2; j++) {    // Inner loop
      cout << "i = " << i << ", j = " << j << endl;
    }
  }
  return 0;
}
```

  - **Output:**

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```

  - **Explanation:**

    The outer loop controls the variable i.

    For every single value of i, the inner loop runs fully for all values of j.

    This structure is useful for tables, matrices, and pattern programs.

3. Nested if-else Ladder

An if-else ladder inside another if structure is used when you have multiple related decisions to make.

- **Example:**

```
#include<iostream>
using namespace std;
int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;

    if (age >= 18) {
        if (age >= 60)
            cout << "You are a Senior Citizen.";
        else
            cout << "You are an Adult.";
    }
    else {
        cout << "You are a Minor.";
    }

    return 0;
}
```

- **Output:**

```
Enter your age: 65
You are a Senior Citizen.
```

4. Nested Loop with if Condition

You can also use if statements inside loops for conditional execution during repetition.

❖ **Example:**

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

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i % 2 == 0)
            cout << i << " is Even" << endl;
        else
            cout << i << " is Odd" << endl;
    }
    return 0;
}
```

- **Output:**

```
1 is Odd
2 is Even
3 is Odd
4 is Even
5 is Odd
```

❖ **Summary Table**

Type	**Description**	**Use Case Example**
Nested if	if inside another if	Checking multiple conditions
Nested if-else	Multiple dependent conditions	Age or Grade checking
Nested loops	Loop inside another loop	Patterns, tables, matrices
if inside loop	Condition checking during loop	Even/Odd check, filtering values

❖ **Conclusion:**

Nested control structures are useful when:

You need to make multiple decisions.

You need repeated actions within another loop or condition.

They make the program more flexible, powerful, and organized for solving complex problems.

4. Functions and Scope

Question: 1

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

Answer:

❖ **Definition:**

A function in C++ is a block of code that performs a specific task and can be reused many times in a program.

It helps in modular programming — dividing a large program into smaller, manageable parts.

❖ **Purpose of Functions:**

To avoid repetition of code.

To make programs modular and readable.

To debug and maintain programs easily.

To reuse the same logic multiple times.

❖ **Types of Functions in C++:**

1. Library (Built-in) Functions — Provided by C++ (like sqrt(), pow(), cout, etc.)

2. User-defined Functions — Created by the programmer.

❖ **Syntax of a Function:**

```
returnType functionName(parameters)
{
   // function body
}
```

❖ **Example:**

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

Here,

int → return type

add → function name

(int a, int b) → parameters

return a + b; → function body

❖ **Function Structure in C++**

A function in C++ has three main parts:

Part	**Meaning**	**Example**
Function Declaration	Tells the compiler that a function exists before it is used.	int add(int, int);
Function Definition	Contains the actual code or logic of the function.	int add(int x, int y) { return x + y; }
Function Calling	Executes the function when needed in the program.	sum = add(5, 10);

❖ **Example Program:**

```
#include<iostream>

using namespace std;


// Function Declaration

int add(int, int);


int main() {

   int num1 = 10, num2 = 20, result;


   // Function Call

   result = add(num1, num2);

   cout << "Sum = " << result;

   return 0;

}


// Function Definition

int add(int a, int b) {

   return a + b;
}
```

- **Output:**

```
Sum = 30
```

- **Explanation:**

  1. Declaration: int add(int, int); informs the compiler about the function.

  2. Definition: Defines what the function does (adds two numbers).

  3. Calling: Executes the function in main() using add(num1, num2).

❖ **Conclusion:**

A function in C++ is a reusable block of code designed to perform a specific task.

It improves code clarity, reusability, and maintenance.

Functions are defined once but can be called multiple times whenever needed.

Question: 2

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

Answer:

❖ **Definition:**

In C++, the scope of a variable refers to the part or region of the program where that variable can be accessed or used.

In simple terms, scope defines the visibility and lifetime of a variable in a program.

❖ **Types of Variable Scope in C++**

Type	**Description**
Local Scope	The variable is declared inside a function or block and can only be used within that block.
Global Scope	The variable is declared outside all functions and can be accessed from any part of the program.

1. **Local Variables**

Declared inside a function or block ({ }).

Created when the function starts and destroyed when it ends.

Cannot be accessed outside the function.

❖ **Example:**

```
#include<iostream>

using namespace std;


void display() {

   int x = 10;    // Local variable

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

}

int main() {

   display();

   // cout << x;   // Error: x is not accessible here

   return 0;

}
```

- **Output:**

```
Local x = 10
```

- **Explanation:**

  The variable x exists only inside the function display().

  It is not known outside that function.

2. **Global Variables**

   Declared outside all functions.

   Can be used by all functions in the program.

   Its value remains the same throughout program execution.

- **Example:**

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

int x = 50;   // Global variable

void show() {
   cout << "Global x = " << x << endl;
}

int main() {
   cout << "Accessing global x in main: " << x << endl;
   show();
   return 0;
}
```

- **Output:**

```
Accessing global x in main: 50
Global x = 50
```

- **Explanation:**

  The variable x is declared globally, so both main() and show() can access it.

- ❖ **Difference Between Local and Global Variables**

Feature	**Local Variable**	**Global Variable**
Declaration Place	Inside a function or block	Outside all functions
Scope	Limited to the block where it is declared	Accessible throughout the program
Lifetime	Created when function starts and destroyed when it ends	Exists till the program ends
Default Value	Garbage (undefined)	Zero (0)
Access from Other Functions	Not possible	Possible

- ❖ **Example Showing Both Local and Global Variables**

```
#include<iostream>

using namespace std;


int x = 100;   // Global variable


int main() {

   int x = 20; // Local variable

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

   cout << "Global x = " << ::x << endl; // :: (scope resolution) accesses global variable

   return 0;

}
```

- o **Output:**

```
Local x = 20

Global x = 100
```

- ❖ **Conclusion:**

The scope of a variable determines where it can be accessed.

Local variables are limited to their block or function.

Global variables can be accessed by all functions.

The scope resolution operator (::) is used when both local and global variables have the same name.

Question: 3

Explain Recursion in C++ with an Example

Answer:

- **Definition:**

  Recursion in C++ is a process in which a function calls itself directly or indirectly to solve a problem.

  It breaks a complex problem into smaller subproblems of the same type.

- **Key Concept:**

  A recursive function has two main parts:

  1. Base Case – The condition that stops the recursion.

  2. Recursive Case – The function calls itself to continue the process.

- **Syntax of a Recursive Function:**

```
returnType functionName(parameters)
{
   if (base_condition)
      return value;        // Base case
   else
      return functionName(modified_parameters); // Recursive call
}
```

- **Example:** Factorial of a Number Using Recursion

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


// Recursive function to find factorial
int factorial(int n) {
   if (n == 0 || n == 1)
      return 1;      // Base case
   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;
}
```

- **Output:**

```
Enter a number: 5
Factorial of 5 = 120
```

- **Explanation:**

When you call factorial(5), the function works as follows:

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

factorial(1) = 1 (Base case reached)

Then the results return back:

factorial(2) = 2 * 1 = 2

factorial(3) = 3 * 2 = 6

factorial(4) = 4 * 6 = 24

factorial(5) = 5 * 24 = 120

❖ **Advantages of Recursion:**

Makes code simpler and cleaner.

Reduces code repetition.

Useful in problems like factorial, Fibonacci, searching, and tree traversal.

- **Disadvantages of Recursion:**

  Uses more memory due to function call stack.

  May cause stack overflow if the base case is missing.

  Sometimes slower than iterative solutions.

- **Conclusion:**

  Recursion in C++ is a powerful concept where a function calls itself to solve smaller parts of a problem.

  It continues until a base condition is met, making it ideal for problems that can be defined in terms of smaller subproblems (like factorial, Fibonacci, etc.).

Question: 4

What are Function Prototypes in C++? Why are They Used?

Answer:

- **Definition:**

  A function prototype in C++ is a declaration of a function that tells the compiler about the function’s name, return type, and parameters before it is used in the program.

  It acts as a blueprint for the function — it informs the compiler that the function will be defined later in the code.

- **Syntax:**

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

  - **Example:**

  int add(int, int);

  Here:

  int → return type

  add → function name

  (int, int) → parameters

- **Purpose of a Function Prototype:**

  It tells the compiler what kind of function will appear later.

  It allows the function to be called before its definition.

  It helps the compiler check for errors such as wrong number or type of arguments.

- **Example Program Without Prototype (May Cause Error)**

```
#include<iostream>

using namespace std;


int main() {

   int result = add(10, 20); // Function called before declaration

   cout << "Sum = " << result;

   return 0;

}
```

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

- **Output:**

  Error (in some compilers): Function ‘add’ was not declared before use.

❖ **Example Program with Function Prototype**

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

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

int main() {
    int result = add(10, 20); // Function Call
    cout << "Sum = " << result;
    return 0;
}

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

- **Output:**

```
Sum = 30
```

- **Explanation:**

  1. Function Prototype: int add(int, int);

  → Tells compiler that a function named add exists and returns an int.

  2. Function Call: add(10, 20);

  → The compiler allows this call because it already knows the function signature.

3. Function Definition:

→ Actual body of the function written later.

❖ **Advantages of Using Function Prototypes**

Advantage	**Explanation**
Helps compiler check correctness	Detects mismatched parameters or wrong data types.
Allows flexible function order	You can call functions before they are defined.
Improves code clarity	Provides an overview of all functions at the beginning of the program.

❖ **Conclusion:**

A function prototype in C++ is a declaration that specifies a function's return type, name, and parameters before its actual definition.

It ensures type safety, error checking, and allows functions to be called before they are defined, making programs more organized and reliable.

5. Arrays and Strings

Question: 1

What are Arrays in C++? Explain the Difference Between Single-Dimensional and Multi-Dimensional Arrays.

Answer:

- **Definition:**

  An array in C++ is a collection of elements of the same data type, stored in contiguous (continuous) memory locations.

  Each element of an array is accessed using an index number.

  In simple words, an array is used to store multiple values of the same type in a single variable.

- **Syntax:**

  ```
  dataType arrayName[size];
  ```

  - **Example:**

    ```
    int marks[5];  // Array to store 5 integers
    ```

- **Example Program (Single Array):**

  ```
  #include<iostream>
  using namespace std;
  int main() {
    int marks[5] = {80, 70, 90, 85, 75}; // Array declaration and initialization

    cout << "Student Marks: " << endl;
    for(int i = 0; i < 5; i++) {
      cout << marks[i] << " ";
    }
    return 0;
  }
  ```

- **Output:**

  ```
  Student Marks:
  80 70 90 85 75
  ```

❖ **Types of Arrays in C++**

Type	**Description**	**Example**
Single-Dimensional Array (1D)	A list of elements stored in one row.	int a[5];
Multi-Dimensional Array (2D or more)	An array of arrays — elements arranged in rows and columns (like a table or matrix).	int b[3][3];

1. **Single-Dimensional Array (1D Array)**
   - Store data in a single line (row).
   - Accessed using one index.
   - Best used for lists, marks, or scores.
   - **Example:**

```
#include<iostream>

using namespace std;


int main() {

   int num[4] = {10, 20, 30, 40};


   cout << "1D Array Elements: ";

   for(int i = 0; i < 4; i++) {

      cout << num[i] << " ";

   }

   return 0;

}
```

   - **Output:**

```
1D Array Elements: 10 20 30 40
```

2. **Multi-Dimensional Array (2D Array Example)**
    - Used to store data in rows and columns (table form).
    - Accessed using two or more indexes.
    - Commonly used in matrices, tables, or grids.
    - **Example:**

```
#include<iostream>

using namespace std;

int main() {

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


  cout << "2D Array (Matrix) Elements:" << endl;

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

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

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

    }

    cout << endl;

  }

  return 0;

}
```

    - **Output:**

```
2D Array (Matrix) Elements:

1 2 3

4 5 6
```

❖ **Difference Between 1D and Multi-Dimensional Arrays**

Feature	**1D Array**	**2D / Multi-Dimensional Array**
Structure	Single row of elements	Elements arranged in rows and columns
Number of Indexes	One index (e.g., a[i])	Two or more indexes (e.g., a[i][j])
Usage	Lists, marks, scores, etc.	Tables, matrices, or grids
Declaration Example	int a[5];	int a[3][3];
Access Example	a[2];	a[1][2];

❖ **Conclusion:**

An array in C++ is used to store multiple values of the same data type efficiently.

A single-dimensional array stores data in a linear form, while a multi-dimensional array stores data in tabular form.

Arrays make it easier to manage and process large amounts of related data using loops.

Question: 2

Explain String Handling in C++ with Examples.

Answer:

❖ **Definition:**

A string in C++ is a sequence of characters used to store and manipulate text.

For example, "Hello", "C++ Programming", and "1234" are all strings.

❖ **C++ provides two main ways to handle strings:**

1. Character Arrays (C-Style Strings)

2. String Class (from <string> library)

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

These are arrays of characters terminated by a null character ('\0').

- **Syntax:**

```
char str[size];
```

- **Example:**

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

int main() {
   char name[20] = "Dharini";  // C-style string

   cout << "Name: " << name << endl;
   return 0;
}
```

- **Output:**

```
Name: Dharini
```

- **Common C-String Functions (from <cstring> library):**

Function	Description	Example
strlen(str)	Returns string length	strlen("Hello") → 5
strcpy(dest, src)	Copies one string to another	strcpy(b, a)
strcat(str1, str2)	Concatenates (joins) two strings	strcat(a, b)
strcmp(str1, str2)	Compares two strings	strcmp(a, b)

- **Example Using C-String Functions:**

```
#include<iostream>
#include<cstring> // for string functions
using namespace std;

int main() {
  char str1[20] = "Hello";
  char str2[20] = "World";

  strcat(str1, str2);  // Concatenate strings

  cout << "Combined String: " << str1 << endl;
  cout << "Length: " << strlen(str1) << endl;
  return 0;
}
```

○ **Output:**

```
Combined String: HelloWorld
Length: 10
```

2. String Class (C++ Standard Library)

C++ provides a more powerful and flexible way to handle strings using the string class from the <string> header.

- **Syntax:**

```
#include<string>

string str = "Hello";
```

- **Example:**

```
#include<iostream>

#include<string>

using namespace std;


int main() {

  string name = "Dharini";

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

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

  return 0;

}
```

- **Output:**

```
Name: Dharini

Length: 7
```

- **Common string Class Functions:**

Function	Description	Example / Result
length() or size()	Returns length of string	"Hello".length() → 5
append(str)	Adds another string	s1.append("World")
substr(pos, len)	Extracts part of string	"Hello".substr(1,3) → "ell"
compare(str)	Compares two strings	s1.compare(s2)
+ operator	Concatenates strings	"Hello" + "World" → "HelloWorld"

- **Example Using String Class:**

```
#include<iostream>

#include<string>

using namespace std;


int main() {

   string str1 = "Hello";

   string str2 = "World";+

   string result = str1 + " " + str2;  // Concatenation

   cout << "Combined String: " << result << endl;


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


   return 0;

}
```

  - **Output:**

```
Combined String: Hello World

Length: 11
```

❖ **Difference Between C-Style and String Class**

Feature	**C-Style String**	**String Class (C++)**
Header File	<cstring>	<string>
Type	Character array	Object (class-based)
Operations	Use functions like strcpy(), strlen()	Use methods like .length(), .append()
Ease of Use	Manual management	Easier and safer
Example	char name[10] = "Hi";	string name = "Hi";

❖ **Conclusion:**

String handling in C++ can be done using C-style character arrays or the C++ string class.

The C-style method uses functions from <cstring>.

The string class (from <string>) provides simpler and more powerful operations for handling text.

Most modern C++ programs prefer using the string class for safety and convenience.

Question: 3

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

Answer:

❖ **What is Array Initialization?**

Array initialization means assigning values to array elements when the array is declared or later in the program.

➢ **Arrays can be initialized in two main ways:**

1. At the time of declaration
2. After declaration

1. One-Dimensional (1D) Array

A 1D array stores a list of values of the same type in a single row.

- **Syntax:**

```
data_type array_name[size] = {value1, value2, value3, ...};
```

- **Example 1: Initialize at the time of declaration**

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

int main() {
  int numbers[5] = {10, 20, 30, 40, 50};

  cout << "1D Array Elements: ";
  for(int i = 0; i < 5; i++) {
    cout << numbers[i] << " ";
  }
  return 0;
}
```

○ **Output:**

```
1D Array Elements: 10 20 30 40 50
```

- **Example 2: Initialize later**

```
int marks[3];
marks[0] = 85;
marks[1] = 90;
marks[2] = 95;
```

2. Two-Dimensional (2D) Array

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

- **Syntax:**

```
data_type array_name[rows][columns] = {{value1, value2, ...}, {value3, value4, ...}};
```

- **Example:**

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

int main() {
   int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

 cout << "2D Array Elements:\n";
   for(int i = 0; i < 2; i++) {
      for(int j = 0; j < 3; j++) {
         cout << matrix[i][j] << " ";
      }
      cout << endl;
   }
   return 0;
}
```

- **Output:**

```
2D Array Elements:
1 2 3
4 5 6
```

❖ Difference Between 1D and 2D Arrays

Feature	**1D Array**	**2D Array**
Structure	Single row (list)	Rows and columns (table)
Syntax	int a[5];	int a[2][3];
Example	{10, 20, 30}	{{1,2,3}, {4,5,6}}
Access Element	a[0]	a[0][1]

❖ Conclusion:

In C++, arrays can be initialized with values at the time of declaration or later.

1D arrays store data in a single line.

2D arrays store data in a table form (rows and columns).

Both help to organize and manage multiple data values efficiently.

Question: 4

Explain String Operations and Functions in C++.

Answer:

❖ **Definition:**

A string in C++ is a sequence of characters used to store and manipulate text such as words or sentences.

Example: "Hello", "C++ Programming", "Apple"

➢ **Strings in C++ can be handled in two ways:**

1. C-style strings – Character arrays (old method)

2. String class – Provided by C++ <string> library (modern and easier method)

1. String Operations in C++

String operations are basic actions we can perform on strings, such as assigning, joining, comparing, or finding length.

Operation	**Description**	**Example / Code**	**Output**
Assignment (=)	Assign one string to another	string a = "Hello";	a = Hello
Concatenation (+)	Joins two strings together	"Hello" + "World"	HelloWorld
Comparison (==, !=)	Compares two strings	if(a == b)	True / False
Access characters	Access individual characters	a[0]	H
Input (getline)	Reads full line input	getline(cin, name);	—

- **Example Program:**

```
#include<iostream>

#include<string>

using namespace std;


int main() {

    string str1 = "Hello";

    string str2 = "World";
    // Concatenation
    string result = str1 + " " + str2;
    cout << "Concatenated String: " << result << endl;

    // Length of string
    cout << "Length: " << result.length() << endl;

    // Substring
    cout << "Substring (0-5): " << result.substr(0,5) << endl;

    // Comparison
    if(str1 == str2)
        cout << "Strings are Equal";
    else
        cout << "Strings are Not Equal";

    return 0;
}
```

- **Output:**

```
Concatenated String: Hello World
Length: 11
Substring (0-5): Hello
Strings are Not Equal
```

2. String Functions in C++

C++ provides many built-in string functions to perform various tasks easily.

Function	Definition / Use	Example	Output
length()	Returns the total number of characters in a string	"Hello".length()	5
append(str)	Adds another string at the end	"Hi".append("All")	HiAll
insert(pos, str)	Inserts a string at a specific position	"Hello".insert(5,"!")	Hello!
erase(pos, len)	Removes characters from the string	"Hello".erase(2,2)	Hlo
substr(pos, len)	Returns a part (substring) of the string	"Hello".substr(1,3)	ell
find(str)	Finds position of substring	"Hello".find("lo")	3
replace(pos, len, str)	Replaces part of string with another	"Hello".replace(0,2,"Hi")	Hilo

- **Example Using String Functions:**

```
#include<iostream>

#include<string>

using namespace std;

int main() {

   string s = "Programming";


   cout << "Original String: " << s << endl;

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

   cout << "Substring (0,4): " << s.substr(0,4) << endl;

   cout << "Position of 'gram': " << s.find("gram") << endl;


   s.replace(0,4,"Code");

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


   return 0;

}
```

- **Output:**

```
Original String: Programming

Length: 11

Substring (0,4): Prog

Position of 'gram': 3

After Replace: Codeming
```

❖ **Difference Between C-style and String Class**

Feature	**C-Style String**	**String Class**
Header File	<cstring>	<string>
Type	Character Array	String Object
Ease of Use	Harder	Easier
Example	char name[10] = "Hi";	string name = "Hi";

❖ **Conclusion:**

String operations and functions in C++ help to store, modify, and process text easily.

Using the string class, programmers can perform operations like concatenation, comparison, length checking, substring extraction, and replacement efficiently — making code simpler and more readable than using traditional C-style strings.

6. Introduction to Object-Oriented Programming

Question: 1

Explain the Key Concepts of Object-Oriented Programming (OOP).

Answer:

❖ **Definition:**

Object-Oriented Programming (OOP) is a programming approach that organizes software design around objects rather than functions or logic.

Each object represents a real-world entity with its data (attributes) and functions (behaviours).

In C++, OOP helps in writing modular, reusable, and organized code.

❖ **Key Concepts of OOP:**

Concept	**Definition**	**Example / Explanation**
1. Class	A class is a blueprint or template for creating objects. It defines data members and member functions.	cpp class Student { public: int roll; void show() { cout << roll; } };
2. Object	An object is an instance of a class. It represents a real-world entity like a car, student, or employee.	cpp Student s1; s1.roll = 101;
3. Encapsulation	Wrapping data and functions together inside a class. It hides internal details from outside access.	Example: Using private variables and public functions.
4. Abstraction	Showing only essential features and hiding unnecessary details.	Example: A car's driver uses steering and pedals, but doesn't see the engine working.
5. Inheritance	One class inherits the properties and functions of another class. It helps in code reusability.	cpp class A {}; class B : public A {};
6. Polymorphism	It means many forms. It allows one function or operator to behave differently based on the situation.	Example: Function Overloading and Overriding.
7. Data Hiding	Protecting data from unauthorized access using private and protected access specifiers.	Example: Private data can be accessed only within the class.

- **Example Program:**

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

class Student {
private:
   int rollNo;
   string name;

public:
   void setData(int r, string n) {   // Encapsulation
      rollNo = r;
      name = n;
   }
   void display() {   // Abstraction
      cout << "Roll No: " << rollNo << ", Name: " << name << endl;
   }
};

int main() {
   Student s1;        // Object
   s1.setData(101, "Dharini");
   s1.display();
   return 0;
}
```

- **Output:**

```
Roll No: 101, Name: Dharini
```

- **Conclusion:**

  The key concepts of OOP — Class, Object, Encapsulation, Abstraction, Inheritance, and Polymorphism — make C++ programs more organized, reusable, secure, and easy to maintain.

  They help in building real-world models in software development.

Question: 2

What are Classes and Objects in C++? Provide an Example.

Answer:

- **Definition of Class:**

  A class in C++ is a blueprint or template for creating objects.

  It defines data members (variables) and member functions (methods) that describe the properties and behaviors of an object.

  In simple terms, a class is like a plan, and an object is the actual thing built from that plan.

- **Definition of Object:**

  An object is an instance of a class.

  It is a real entity that uses the variables and functions defined inside the class.

  Each object has its own copy of the class's data members.

  - **Example Analogy:**

    Class → Car blueprint

    Object → Actual cars like Honda, BMW, or Tata made using that blueprint

- **Syntax of Class and Object:**

```
class ClassName

{

   // Data members

   // Member functions

};


ClassName objectName;  // Creating object
```

- **Example Program:**

```
#include<iostream>

using namespace std;


// Class Definition

class Student {
```

```
public:
    int rollNo;         // Data member
    string name;        // Data member

    void display() {    // Member function
        cout << "Roll No: " << rollNo << ", Name: " << name << endl;
    }
};

// Main Function
int main() {
    Student s1;         // Object creation
    s1.rollNo = 101;
    s1.name = "Dharini";

    s1.display();       // Function call using object
    return 0;
}
```

- **Output:**

```
Roll No: 101, Name: Dharini
```

- **Explanation:**

  1. class Student → Blueprint that defines what data (rollNo, name) and functions (display) a student has.

  2. Student s1; → Creates an object s1 of the class.

  3. s1.rollNo and s1.name → Access data members using the dot . operator.

  4. s1.display() → Calls the function defined inside the class.

❖ **Conclusion:**

A class is a template that defines data and behaviour.

An object is an instance of a class that stores actual values.

Together, they form the foundation of Object-Oriented Programming in C++.

Question: 3

What is Inheritance in C++? Explain with an Example.

Answer:

- **Definition:**

  Inheritance in C++ is an Object-Oriented Programming (OOP) concept that allows a new class to reuse the properties and functions of an existing class.

  It helps in code reusability and creates a relationship between classes.

  - The existing class is called the Base Class (Parent Class)
  - The new class is called the Derived Class (Child Class)

- **Syntax:**

```
class DerivedClass : accessSpecifier BaseClass
{
  // additional members of derived class
};
```

  - **Access Specifiers:**

    public → Public and protected members of base class remain accessible

    private → All members of base class become private in derived class

    protected → All members become protected in derived class

- **Example Program:**

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

// Base Class
class Person {
public:
  string name;
  int age;

  void getInfo() {
    cout << "Enter name and age: ";
```

```
        cin >> name >> age;
    }
};

// Derived Class
class Student : public Person {
public:
    int rollNo;

    void display() {
        cout << "Name: " << name << ", Age: " << age << ", Roll No: " << rollNo << endl;
    }
};

// Main Function
int main() {
    Student s1;
    s1.getInfo();       // Function from Base Class
    s1.rollNo = 101;    // Data from Derived Class
    s1.display();       // Function from Derived Class
    return 0;
}
```

- **Output** (Example Input: Dharini 20):

  ```
  Enter name and age: Dharini 20
  Name: Dharini, Age: 20, Roll No: 101
  ```

- **Explanation:**

  1. Person is the base class containing data and a function.

  2. Student is the derived class that inherits properties from Person.

  3. The object s1 of the derived class can access both getInfo() (base class function) and display() (its own function).

❖ **Types of Inheritance in C++:**
There are 5 main types of inheritance in C++:

1. **Single Inheritance**

   In single inheritance, one derived class inherits from one base class.

   - **Diagram:**
     Base → Derived

   - **Example:**

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

class A {
  public:
    void displayA() {
      cout << "Class A called" << endl;
    }
};

class B : public A {
  public:
    void displayB() {
      cout << "Class B called" << endl;
    }
};

int main() {
  B obj;
  obj.displayA();
  obj.displayB();
  return 0;
}
```

2. **Multiple Inheritance**

   In multiple inheritance, one derived class inherits from two or more base classes.

   - **Diagram:**

```
Base1 →
         → Derived
Base2 →
```

   - **Example:**

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

```
class A {
  public:
    void displayA() {
      cout << "Class A called" << endl;
    }
};

class B {
  public:
    void displayB() {
      cout << "Class B called" << endl;
    }
};

class C : public A, public B {
  public:
    void displayC() {
      cout << "Class C called" << endl;
    }
};

int main() {
  C obj;
  obj.displayA();
  obj.displayB();
  obj.displayC();
  return 0;
}
```

3. **Multilevel Inheritance**

   In multilevel inheritance, a class is derived from another derived class (a chain of inheritance).

   ○ **Diagram:**
   Base → Derived1 → Derived2

   ○ **Example:**

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

class A {
  public:
    void displayA() {
      cout << "Class A called" << endl;
    }
};

class B : public A {
  public:
    void displayB() {
      cout << "Class B called" << endl;
    }
};

class C : public B {
  public:
    void displayC() {
      cout << "Class C called" << endl;
    }
};

int main() {
  C obj;
  obj.displayA();
  obj.displayB();
  obj.displayC();
  return 0;
}
```

4. **Hierarchical Inheritance**

   In hierarchical inheritance, multiple derived classes inherit from a single base class.

   - **Diagram:**

```
         → Derived1
Base →
         → Derived2
```

   - **Example:**

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

class A {
  public:
    void displayA() {
      cout << "Class A called" << endl;
    }
};
```

```
class B : public A {
  public:
    void displayB() {
      cout << "Class B called" << endl;
    }
};

class C : public A {
  public:
    void displayC() {
      cout << "Class C called" << endl;
    }
};

int main() {
  B obj1;
  C obj2;
  obj1.displayA();
  obj1.displayB();
  obj2.displayA();
  obj2.displayC();
  return 0;
}
```

5. **Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance.

It may include multiple and multilevel inheritance together.

- **Diagram:**

```
      Class A
       /   \
      /     \

Class B    Class C

      \     /
       \   /
      Class D
```

- **Explanation:**
    - Class A → is the base class.
    - Class B and Class C → both inherit from Class A (→ Hierarchical Inheritance).
    - Class D → inherits from both Class B and Class C (→ Multiple Inheritance).

- **Example:**

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

class A {
  public:
    void displayA() {
      cout << "Class A called" << endl;
    }
};

class B : public A {
  public:
    void displayB() {
      cout << "Class B called" << endl;
    }
};

class C {
  public:
    void displayC() {
      cout << "Class C called" << endl;
    }
};

class D : public B, public C {
  public:
    void displayD() {
      cout << "Class D called" << endl;
    }
};

int main() {
  D obj;
  obj.displayA();
  obj.displayB();
  obj.displayC();
  obj.displayD();
  return 0;
}
```

❖ **Summary Table**

Type	**Description**	**Example**
Single Inheritance	One base class → one derived class	A → B
Multiple Inheritance	One derived class → multiple base classes	A, B → C
Multilevel Inheritance	Derived class becomes base for another	A → B → C
Hierarchical Inheritance	One base → multiple derived classes	A → B, C
Hybrid Inheritance	Combination of two or more types	Mix of above

❖ **Conclusion:**

Inheritance allows one class to reuse and extend the features of another class.

It promotes code reusability, reduces redundancy, and establishes a relationship between parent and child classes in C++.

Question: 4

What is Encapsulation in C++? How is it Achieved in Classes?

Answer:

- **Definition:**

  Encapsulation in C++ is one of the main Object-Oriented Programming (OOP) concepts.

  It means binding data and functions together inside a single unit called a class and restricting direct access to the data from outside the class.

  In simple words, encapsulation = data hiding + data protection.

- **Example in Real Life:**

  Think of a capsule — it hides the medicine inside.

  Similarly, in programming, encapsulation hides the internal data and allows access only through specific functions.

- **How Encapsulation is Achieved in C++**

  Encapsulation is achieved by:

  1. Using Classes to bind data and functions together.

  2. Using Access Specifiers to control access to data:

     - private → accessible only within the class
     - public → accessible outside the class
     - protected → accessible in derived classes

- **Example Program:**

```
#include<iostream>

using namespace std;


class Student {

private:

   int rollNo;       // Private data (hidden)

   string name;


public:

   // Function to set data (write access)

   void setData(int r, string n) {
       rollNo = r;
       name = n;
   }

   // Function to get data (read access)
   void display() {
       cout << "Roll No: " << rollNo << ", Name: " << name << endl;
   }
};

int main() {
   Student s1;
   s1.setData(101, "Dharini");   // Access through public function
   s1.display();                 // Display details
   return 0;
}
```

- **Output:**

```
Roll No: 101, Name: Dharini
```

- **Explanation:**

  1. Private members rollNo and name cannot be accessed directly from main().

  2. Access is given only through public functions (setData() and display()), which ensures data safety.

  3. This is encapsulation — combining data + functions and controlling access.

❖ **Advantages of Encapsulation:**

Advantage	**Explanation**
Data Hiding	Keeps data safe from direct modification.
Code Reusability	Class can be reused in other programs.
Easy Maintenance	Changes can be made without affecting other code.
Security	Only specific functions can access private data.

❖ **Conclusion:**

Encapsulation in C++ is the process of wrapping data and functions together and protecting data from direct access.

It is achieved using classes and access specifiers (private, public), which makes the program secure, modular, and easy to manage.