OOPS 1. Introduction to C++

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

- Focuses on **functions** or **procedures** (step-by-step instructions).
- Data and functions are **separate**.
- Programs are divided into **functions**.
- Example languages: **C**, **Pascal**.
- **Data is not hidden** — can be accessed from anywhere.
- **Reusability** is less.

Object-Oriented Programming (OOP)

- Focuses on **objects** (which combine data and functions together).
- Data and functions are **inside the object**.
- Programs are divided into **classes and objects**.
- Example languages: **C++**, **Java**, **Python**.
- **Data is hidden** (Encapsulation).
- **Reusability** is more (through inheritance).

Feature	Procedural	OOP
Focus	Functions	Objects
Data	Separate from functions	Combined in objects
Security	Less (data open)	More (data hidden)
Reusability	Low	High
Example	C	C++, Java

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

Advantages of OOP over POP

1. **Reusability** – Use existing code again using classes.
2. **Security** – Data is hidden inside objects.
3. **Easy Maintenance** – Simple to update or fix code.
4. **Modularity** – Program divided into small parts (objects).
5. **Flexibility** – Easy to add new features.
6. **Real-world Model** – Based on real objects like Car, Student, etc.

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

Steps to Set Up a C++ Development Environment

1. **Install a Compiler** ○ A compiler converts C++ code into machine code.
   - Example: **MinGW**, **Turbo C++**, or **GCC**.
2. **Install an IDE (Editor)** ○ IDE helps you write and run C++ programs easily.
   - Example: **Code: Blocks**, **Dev C++**, **Visual Studio**, or **VS Code**.
3. **Link Compiler with IDE** ○ Make sure the IDE is connected to the compiler so programs can compile and run properly.
4. **Create a New Project or File** ○ Open the IDE → Create new C++ file → Save it with **".cpp"** extension.
5. **Write the Program** ○ Type your C++ code in the editor window.
6. **Compile the Code** ○ Click on **"Compile"** or **"Build"** to check for errors.
7. **Run the Program** ○ Click **"Run"** to see the output on the screen.

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

Main Input/Output Operations in C++

1. **Input (taking data from user)** → cin
2. **Output (displaying data to user)** → cout

Both are part of the **iostream** header file.

Code: -

```
#include <iostream> using

namespace std;

int main() {     int num;     cout << "Enter a

number: ";   // Output     cin >> num;

// Input     cout << "You entered: " << num; //

Output     return 0;

}
```

Explanation:

- cout → Used to **print output** on the screen.
- cin → Used to **take input** from the user.
- << → Insertion operator (used with cout).
- >> → Extraction operator (used with cin).

2. Variables, Data Types, and Operators

Q. What are the different data types available in C++? Explain with examples. 1. Basic (Primitive) Data Types

Data Type	Description	Example
int	Stores whole numbers	int age = 20;
float	Stores decimal numbers (small)	float price = 12.5;
double	Stores large decimal numbers	double area = 45.678;
char	Stores a single character	char grade = 'A';
bool	Stores true or false	bool pass = true;

2. Derived Data Types

Made from basic types. Examples:

- **Array** → int marks[5];
- **Pointer** → int *ptr;
- **Function** → void show();
- **Reference** → int &ref = x; **3.**

User-Defined Data Types

Created by the user. Examples:

- **Structure** → struct Student { int id; char name[20]; };
- **Class** → class Car { };
- **Enum** → enum Color { Red, Green, Blue };

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

Type Conversion in C++

Changing one data type to another is called **type conversion**.

There are **two types**

1. Implicit Type Conversion (Automatic)

- Done **automatically by the compiler**.
- Also called **Type Promotion**.
- Happens when different data types are used in an expression. Example: -

```
int a = 5; float
b = 2.5;
float c = a + b;   // int 'a' is automatically converted to float
```

2. Explicit Type Conversion (Manual)

- Done **by the programmer** using **type casting**.
- You tell the compiler what type to convert.

Example: - float

```
a = 5.6;
int b = (int)a;   // Explicitly converting float to int
```

Type	Done By	Also Called	Example
Implicit	Compiler	Type Promotion	float c = a + b;

Explicit	Programmer	Type Casting	int b = (int)a;

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

Types of Operators in C++

Operators are symbols used to perform operations on variables and values.

1. Arithmetic Operators

Used for mathematical calculations.

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

```
int a = 10, b = 3; cout << a
+ b;  // Output: 13
```

2. Relational Operators

Used to compare two values.

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

```
 if(a >
b)
   cout << "a is greater";
```

3. Logical Operators

Used to combine conditions.

Operator	Description	Example
&&	Logical AND	(a > 5 && b < 10)

||	Logical OR	`
!	Logical NOT	!(a == b)

4. Assignment Operators

Used to assign or modify values.

Operator	Description	Example
=	Assign	a = 5
+=	Add and assign	a += 2
-=	Subtract and assign	a -= 2
*=	Multiply and assign	a *= 2
/=	Divide and assign	a /= 2

5. Increment and Decrement Operators

Used to increase or decrease value by 1.

Operator	Description	Example
++	Increment	a++
--	Decrement	b--

6. Bitwise Operators

Used to perform operations on bits.

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left Shift	a << 1
>>	Right Shift	a >> 1

7. Conditional (Ternary) Operator

Used as a short form of if-else.

Operator	Description	Example

?:	Conditional	(a > b) ? a : b

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

Constants and Literals in C++ 1. Constants

- **Constants** are fixed values that **do not change** during program execution.
- Used to make programs **more readable and secure**.

Example:

```
const int age = 18;   // 'age' cannot be changed later
```

- Integer constant → 10
- Floating constant → 12.5
- Character constant → 'A'
- String constant → "Hello"
- Boolean constant → true, false

2. Literals

- **Literals** are the **actual constant values** used directly in the program.
- They represent fixed data.

Example:

```
int x = 5;        // 5 is a literal char
grade = 'A'; // 'A' is a literal
```

Constants → named fixed values
Literals → actual fixed values written in code

3. Control Flow Statements

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

Conditional Statements in C++

Conditional statements are used to **control the flow of a program**.
They let the program **make decisions** based on certain conditions (true or false).

1. if–else Statement

The if–else statement checks a condition.

- If the condition is **true**, it runs the **if block**.
- If the condition is **false**, it runs the **else block**.

```
Syntax: - if
(condition) {
  // Code to execute if condition is true
} else
{
  // Code to execute if condition is false
}
Example: - #include
<iostream> using
namespace std;

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

   if (marks >= 50) {
      cout << "You passed the exam!";
   }
else {
      cout << "You failed the exam.";
   }
   return 0;
}
```

2. switch Statement

The switch statement is used when you have **many possible choices** based on a single value.

It's easier to read than using many if–else statements. Syntax: - switch(expression) {   case

```
value1:       // Code for value1        break;     case value2:       // Code for value2        break;
default:
      // Code if no case matches
}


Example: - #include
<iostream> using
namespace std; int
main() {     int day;
   cout << "Enter day number (1-3): ";
cin >> day;

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

```
  }
  return 0; }
```

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

1. for Loop

 Used when we **know** how many times to repeat.

 The condition is checked **before** running.

Syntax: - for(initialization; condition;

update) {

  // code to repeat

}

Example: - **for(int i = 1; i**

<= 5; i++) {   cout << i

<< " ";

}

2. while Loop

- Used when we **don't know** how many times to repeat.
- The condition is also checked **before** running. **Syntax: -** while(condition) {

  // code to repeat

}

Example; -

int i = 1; while(i

<= 5) { cout <<

i << " "; i++;

}

3. do–while Loop

 Used when we want the loop to **run at least once**.

 The condition is checked **after** running.

Syntax: - do {

  // code to repeat

} while(condition); Example:

-

int i = 1; do {

cout << i << " ";

i++;

} while(i <= 5);

Feature	for Loop	while Loop	do–while Loop
Use	When number of repetitions is known	When repetitions are unknown	When code must run at least once
Condition Check	Before loop	Before loop	After loop
Syntax Style	All in one line	Separate initialization	Separate initialization

Executes At Least Once?	✗ No	✗ No	✓ Yes

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

break statement

 Used to **stop (exit)** a loop **immediately**.

 Control moves **outside** the loop. Example:

-

```
for(int i = 1; i <= 5; i++) {     if(i == 3)

break;  // loop stops when i = 3

cout << i << " ";

}
```

2. continue statement

 Used to **skip** the current loop iteration.

 The loop continues with the **next** value.

Example: -

```
for(int i = 1; i <= 5; i++) {     if(i
== 3)         continue;  // skips
printing 3     cout << i << " ";
}
```

Q. Explain nested control structures with an example.

Nested Control Structures in C++

Meaning:
When one control structure (like if, for, while) is placed **inside another**, it is called a **nested control structure**.

It helps in checking **multiple conditions** or performing **complex decisions**.

Example 1: Nested if Statement

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


int main() {     int
age = 20;     char
gender = 'M';


    if (age >= 18) {
if (gender == 'M')
            cout << "You are an adult male.";
else            cout << "You are an adult
female.";
    } else {        cout << "You are not
an adult.";
    }
    return 0;
}
```

Output:
You are an adult male.

Example 2: Nested Loop

```
for(int i = 1; i <= 3; i++) {
for(int j = 1; j <= 2; j++) {
cout << i << "," << j << "  ";
    }
}
```

Output:

```
1,1 1,2 2,1 2,2 3,1 3,2
```

4. Functions and Scope

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

→ A **function** is a block of code that performs a specific task. It helps in reusing code and makes programs easier to read and manage.

1. Function Declaration

It tells the compiler about the function's name, return type, and parameters (no body).
Syntax: returnType functionName(parameter1, parameter2,

...); Example:

```
int add(int a, int b);
```

2. Function Definition

It contains the actual code (body) that defines what the function does. **Syntax:**

```
returnType functionName(parameter1, parameter2, ...) {
// function body
}
```

Example: -

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

3. Function Calling

It means using the function to perform its task. **Example:**

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

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

int main() {
    int result = add(5, 3);  // Calling
cout << "Sum = " << result;     return
0;
}

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

Output:

```
Sum = 8
```

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

Scope of Variables in C++:
The **scope** of a variable means the part of the program where the variable can be accessed or used.

1. Local Scope

- Declared **inside** a function or block.
- Can be used **only within** that function or block.
- Automatically destroyed when the function ends.

Example:

```
void display() {
    int x = 10;   // Local variable
cout << x;

}
```

2. Global Scope

- Declared **outside** all functions.
- Can be used **anywhere** in the program.
- Exists for the **entire program** duration.

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

int x = 20;   // Global variable

void show() {
cout << x;
}

int main() {
show();     cout
<< x;
}
```

Feature	Local Variable	Global Variable
Declared in	Inside function/block	Outside all functions
Access	Only within that function	Anywhere in the program
Lifetime	Ends when function ends	Exists till program ends
Memory	Created when function runs	Created at program start

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

Recursion in C++:
Recursion is a process in which a function **calls itself** directly or indirectly to solve a problem. It continues until a **base condition** is met (to stop the function calls).

Syntax:

```
returnType functionName(parameters) {
if (base condition)        return value;
else
      // recursive call
      functionName(updated parameters);
}
```

Example: Factorial using Recursion

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

int factorial(int n) {    if (n == 0)
// base condition        return 1;
else
      return n * factorial(n - 1);  // recursive call
}

int main() {
   cout << "Factorial of 5 = " << factorial(5);
return 0;
}
```

Output:

```
Factorial of 5 = 120
```

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

Function Prototype in C++:
A **function prototype** is a **declaration** of a function that tells the compiler about the **function's name, return type, and parameters — before** the function is actually defined.

Syntax: returnType functionName(parameter1Type,

parameter2Type, ...); Example:

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

Purpose / Why Used:

1. To tell the compiler that a function **exists** before it is used.
2. To enable **calling a function** before its definition.
3. To help the compiler **check correct arguments** and **return type**.

Example:

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

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

int main() {     cout << add(5, 3);   //
Function call     return 0;
}
```

int add(int a, int b) { // Function definition

return a + b;

}

Output:

8

5. Arrays and Strings

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

Arrays in C++:
An **array** is a collection of **similar data types** (like int, float, char) stored in **contiguous memory locations**. It allows storing multiple values in one variable name.

Example of Array: int numbers[5] =

{10, 20, 30, 40, 50};

Types of Arrays:

1. **Single-Dimensional Array**

- Stores data in a **single row (line)**.
- Accessed using **one index**.

Example:

int arr[3] = {1, 2, 3}; cout

<< arr[0];   // Output: 1

2. **Multi-Dimensional Array**

- Stores data in **rows and columns** (like a table).
- Accessed using **two or more indexes**.
- Most common is the **2D array**.

Example: -

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

<< matrix[1][2];   // Output: 6

Difference Between Single and Multi-Dimensional Arrays

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	Stores elements in one row	Stores elements in rows and columns
Indexing	One index (e.g., arr[i])	Two or more indexes (e.g., arr[i][j])
Example	int a[5];	int a[3][3];
Use	Simple list of data	Tabular or matrix data

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

String Handling in C++:

A **string** is a sequence of characters used to store and manipulate text. C++ provides two main ways to handle strings:

1. Using **character arrays**
2. Using the **string class** (from <string> library)

1. Using Character Arrays

Strings can be represented as an array of characters ending with a **null character ('\0')**.

Example:

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

int main() {
    char name[10] = "Vrujal";
cout << "Name: " << name;
return 0;
}
```

2. Using string Class (C++ Standard Library)

Easier and safer way to handle strings.

Example:

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

int main() {
    string name = "Rana Vrujal";
cout << "Name: " << name << endl;

    // String operations
    cout << "Length: " << name.length() << endl;     cout
<< "Uppercase first letter: " << name[0] << endl;     cout
<< "Full name: " << name + " Dama";     return 0;
}
```

Function	Description	Example
length()	Returns number of characters	name.length()
append()	Adds text to the end	name.append(" Dama")
substr(pos, len)	Extracts substring	name.substr(0, 4)
compare()	Compares two strings	name.compare("Vrujal")

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

Array Initialization in C++:

→ Array initialization in C++ means giving initial values to array elements at the time of declaration. A **1D array** stores data in a single row, while a **2D array** stores data in rows and columns.

→ In C++, **initializing an array** means assigning values to its elements when the array is declared. Arrays can be **one-dimensional (1D)** or **two-dimensional (2D)**.

1. One-Dimensional (1D) Array

A **1D array** stores a list of elements in a single row.

Syntax:

dataType arrayName[size] = {value1, value2, value3, ...};

Example:

```
#include <iostream> using

namespace std;


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

Initialization     cout << "Second Element: " <<

numbers[1];

    return 0;

}
```

Output:

```
Second Element: 20
```

Other Examples:

```
int marks[5] = {80, 90};        // Remaining elements become 0 int
values[] = {1, 2, 3, 4, 5}; // Size automatically decided
```

2. Two-Dimensional (2D) Array

A **2D array** stores elements in **rows and columns** (like a table).

Syntax:

dataType arrayName[rows][cols] = { {row1}, {row2}, ... };

Example:

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

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

    cout << "Element at [1][2]: " << matrix[1][2];
return 0; }
```

Output:

```
Element at [1][2]: 6
```

Type	Declaration	Example	Access Example
1D Array	int a[5];	{10, 20, 30, 40, 50}	a[2] → 30
2D Array	int b[2][3];	{{1,2,3},{4,5,6}}	b[1][2] → 6

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

→ String operations in C++ allow performing tasks like joining, comparing, searching, and modifying text. The **string class** provides built-in functions like length(), append(), substr(), and compare() to make string handling easy and efficient.

They can be handled in two ways:

1. **Character arrays** (C-style strings)
2. **string class** from the **<string>** library (modern C++ way)

1. Basic String Operations (using string class)

Example:

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

int main() {
    string str1 = "Hello";
    string str2 = "World";

    // Concatenation
    string str3 = str1 + " " + str2;

    // Length
    int len = str3.length();

    // Access character
    char ch = str3[1];

    // Substring
    string part = str3.substr(0, 5);

    // Comparison
if (str1 == "Hello")
        cout << "Strings are equal\n";

    cout << "Combined: " << str3 << endl;
cout << "Length: " << len << endl;     cout
<< "Second letter: " << ch << endl;     cout
<< "Substring: " << part;
}
```

Output:

```
Strings are equal
Combined: Hello World
Length: 11
Second letter: e
Substring: Hello
```

2. Common String Functions

Function	Description	Example
length() or size()	Returns number of characters	str.length()
append(str)	Adds another string at the end	str1.append(" World")
substr(pos, len)	Extracts part of the string	str.substr(0,5)
compare(str)	Compares two strings	str1.compare(str2)
find(str)	Finds position of substring	str.find("lo")
erase(pos, len)	Removes characters	str.erase(2,3)
insert(pos, str)	Inserts substring	str.insert(2, "Hi")

3. Example of Common Functions

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

int main() {
    string s = "RanaVrujal";

    cout << "Length: " << s.length() << endl;
cout << "Substring: " << s.substr(0, 4) << endl;
s.append(" Dama");     cout << "After Append: "
<< s << endl;     s.erase(4, 2);     cout << "After
Erase: " << s << endl;
}
```

Output:

```
Length: 10
Substring: Rana
After Append: RanaVrujal Dama
```

After Erase: Ranaujal Dama

Summary

Operation	Example	Description
Concatenation	str1 + str2	Joins two strings
Comparison	str1 == str2	Checks if equal
Substring	str.substr(2,4)	Extracts part of string
Length	str.length()	Gets size
Insert / Erase	str.insert(), str.erase()	Modify string content

6. Introduction to Object-Oriented Programming

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

Key Concepts of Object-Oriented Programming (OOP):

OOP in C++ is based on several important concepts that make programming easier, more organized, and reusable.

1. Class

A **class** is a **blueprint** or **template** for creating objects.
It defines data (variables) and actions (functions) that the objects will have.

2. Object

An **object** is an **instance of a class**.
It represents a real-world entity and can use the class's data and functions.

3. Encapsulation

Encapsulation means **binding data and functions together** into a single unit (class).
It helps to **protect data** from direct access using access specifiers (private, public, protected).

4. Abstraction

Abstraction means **showing only the necessary details** and **hiding complex implementation**. It simplifies the user's view of the system.

5. Polymorphism

Polymorphism means **one name, many forms**.
It allows the same function or operation to behave **differently** for different objects.

6. Inheritance

Inheritance allows a class to **acquire properties and behaviors of another class**. It supports **code reusability** and **hierarchical relationships** between classes.

7. Data Binding

Data binding refers to the **linking of data and functions** together in a program at runtime or compile time. It connects the code (methods) with the data they operate on.

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

Class

A **class** is a **blueprint** or **template** that defines how an object will look and behave.
It contains **data members** (variables) and **member functions** (methods) that operate on the data.

Object

An **object** is an **instance of a class**.
It represents a **real-world entity** and can access the data and functions defined in the class.

Example:

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

class Student {        // Class definition
public:     string name;
    int age;

    void display() {   // Member function         cout <<
"Name: " << name << ", Age: " << age;
    }
};

int main() {
    Student s1;        // Object creation
    s1.name = "Vrujal";
s1.age = 20;
s1.display();     return
0;
}
```

Output:

```
Name: Vrujal, Age: 20
```

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

Definition:
Inheritance is one of the main concepts of Object-Oriented Programming (OOP).
It allows a **class (child/derived class)** to **inherit properties and behaviors** (data and functions) from another **class (parent/base class)**.

This helps in **code reusability** and **avoids repetition**.

Types of Inheritance:

1. **Single Inheritance** – One base and one derived class.
2. **Multiple Inheritance** – One derived class with more than one base class.
3. **Multilevel Inheritance** – A class derived from another derived class.
4. **Hierarchical Inheritance** – Multiple classes inherit from one base class.
5. **Hybrid Inheritance** – Combination of two or more types of inheritance.

Example:

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

// Base class class
Animal { public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

// Derived class class Dog
: public Animal { public:
void bark() {
        cout << "Barking..." << endl;
    }
};

int main() {     Dog d1;     d1.eat();
// Inherited from Animal     d1.bark();
// Defined in Dog
    return 0; }
```

Output:

```
Eating...
Barking...
```

1 Single Inheritance:

```
A
|
B
```

2▢Multiple Inheritance:

```
A    B
 \   /
   C
```

3▢Multilevel Inheritance:

```
A
|
B
|
C
```

4▢Hierarchical Inheritance:

```
  A
/ | \
B  C  D
```

5 ▢Hybrid Inheritance:

```
A
|
B     C
 \   /
   D
```

Q. What is encapsulation in C++? How isit achieved in classes?

Definition:
Encapsulation is the process of **wrapping data (variables)** and **functions (methods)** together into a **single unit** called a **class**. It is used to **protect data from unauthorized access** and **maintain data security**.

How Encapsulation is Achieved:

1. **Using Classes:**
   Data and functions are combined inside a class.
2. **Using Access Specifiers:**
   Access to data is controlled using:
   - private → Accessible only inside the class.
   - public → Accessible from outside the class.
   - protected → Accessible in derived classes.

Example:

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

class BankAccount { private:
int balance;   // Private data

public:     void deposit(int amount) {   // Public
method         balance += amount;
    }

    int getBalance() {           // Public method
return balance;
    }
};

int main() {
    BankAccount acc;
acc.deposit(1000);
    cout << "Balance: " << acc.getBalance();
return 0; }
```

Output:

```
Balance: 1000
```