Introduction to OOPS Programming

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

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Definition	Programming is based on procedures or routines (functions).	Programming is based on objects that contain data and behavior.
Approach	Top-down approach.	Bottom-up approach.
Main Focus	Focuses on functions and the sequence of actions to be performed.	Focuses on data (objects) and encapsulation of related functions (methods).
Data Handling	Data is usually global and shared across functions.	Data is encapsulated within objects and protected from outside interference.
Reusability	Code is not easily reusable.	Promotes code reusability through inheritance and polymorphism.
Examples	C, Pascal, Fortran	C++, Java, Python, C#
Modularity	Less modular; functions operate on shared data.	Highly modular; each object is a self-contained unit.
Security	Low security for data since everything is accessible.	High security through encapsulation and access modifiers (private, public, etc.).
Inheritance	Not supported.	Supported (objects can inherit properties and behavior).
Polymorphism & Encapsulation	Not supported.	Core features of OOP.
Maintenance	More difficult as program size increases.	Easier to maintain, modify, and extend.

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

★ **Modularity (Code Reusability)**

- **OOP**: Code is organized into objects and classes. Once a class is created, it can be reused as needed (e.g., inheritance, object reuse).
- **POP**: Code is written in functions, and reusability is limited. You often need to duplicate code or write extra logic for reuse.

★ **Data Security (Encapsulation)**

- **OOP**: Data and functions are bundled into classes. Access modifiers (private, public, protected) protect data from unauthorized access.
- **POP**: Data is globally accessible by all functions, which can lead to accidental modification and lower security.

★ **Scalability and Maintainability**

- **OOP**: Easier to maintain and scale due to its modular structure. Updating a class doesn't affect other parts of the program as long as interfaces remain the same.
- **POP**: Hard to scale and maintain because the code is tightly coupled, and one change can affect many parts.

★ **Improved Collaboration**

- **OOP**: Easier for teams to work on different parts of a program independently (different classes/modules).
- **POP**: More linear and sequential, making team collaboration more complex.

★ **Real-world Modeling**

- **OOP**: Closer to how real-world systems work. Objects represent real-world entities with properties and behaviors.
- **POP**: More abstract and function-based, harder to relate to real-world entities.

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

❖ **Install a C++ Compiler**

➔ A compiler is required to convert your C++ code into machine code.

➢ **GCC (GNU Compiler Collection)**

➢ Install MinGW (Minimalist GNU for Windows)

➢ Download from: https://osdn.net/projects/mingw/

➢ Install g++, add its path to Environment Variables

❖ **Choose an Integrated Development Environment (IDE) or Text Editor**

➔ An IDE makes writing, compiling, and debugging easier.

➔ Popular IDEs:

- Code::Blocks (lightweight IDE for C++)
- Dev C++
- VS Code

❖ **Configure the IDE**

➔ Make sure the IDE is set up to use the installed compiler.

- In **Code::Blocks**:
    - Go to Settings > Compiler
    - Set the path to g++.exe if using MinGW

- In **VS Code**:
    - Install extensions: C/C++ by Microsoft
    - Configure tasks.json and launch.json for building and debugging

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

❖ **Output using cout**

➔ cout stands for Console Output.
➔ It is used to display data to the user.
➔ << is the insertion operator

★ Syntax:

```
cout << data;
```

★ Example:

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

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

❖ **Input using cin**

➔ cin stands for Console Input.
➔ It is used to take input from the user.
➔ >> is the extraction operator
➔ Multiple inputs can be taken in one line: cin >> a >> b;

★ Syntax:

```
cin >> variable;
```

★ Example:

```
#include <iostream>
using namespace std;
int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You are " << age << " years old." << endl;
    return 0;
}
```

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

Category	Data Type	Description
Basic	int	Stores integers
	float	Stores single-precision decimal numbers
	double	Stores double-precision decimal numbers
	char	Stores a single character
	bool	Stores boolean value (true/false)
Void	void	Represents absence of type (used in functions)
Derived	array	Collection of elements of same type
	pointer	Stores memory address of another variable
	function	Represents functions
	reference	Alias for another variable
User-defined	struct	Groups related variables
	union	Shares memory among its members
	enum	User-defined constants
	class	Blueprint for objects (OOP)

❖ **Example**

```
#include <iostream>
using namespace std;
struct Student {
    int id;
    char grade;
};
int main() {
    int age = 20;
    float height = 5.9;
    double pi = 3.14159;
    char gender = 'M';
    bool isStudent = true;

    int marks[3] = {85, 90, 95};    // Array
    int* ptr = &age;                // Pointer
    int &ref = age;                 // Reference
    Student s1 = {1, 'A'};          // Struct
    enum Color { Red, Blue };       // Enum
    Color c = Red;

    cout << "Age: " << age << ", Height: " << height << ", PI: " << pi << endl;
    cout << "Gender: " << gender << ", Is Student: " << isStudent << endl;
```

```
    cout << "Pointer Value: " << *ptr << ", Reference: " << ref << endl;
    cout << "Student ID: " << s1.id << ", Grade: " << s1.grade << endl;
    return 0;
}
```

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

❖ **Implicit Type Conversion (Type Promotion)**

➔ Also called automatic conversion, this is done by the compiler when you mix different data types.

➢ Key Points:

- Happens automatically.
- Converts a smaller data type to a larger one (to prevent data loss).
- Also known as type promotion.

★ Example:

```
int a = 10;
double b = a;  // int to double (automatic)
cout << b;     // Output: 10.0
```

❖ **Explicit Type Conversion (Type Casting)**

➔ Also called manual conversion, this is done by the programmer using casting operators.

➢ Key Points:

- Must be done manually.
- Can convert larger to smaller types (may lose data).
- Use casting syntax.

★ Example:

```
double x = 10.75;
int y = (int)x;  // double to int (manual cast)
cout << y;       // Output: 10
```

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

Operator Type	Description	Example
1. Arithmetic Operators	Perform mathematical operations	+, -, *, /, % int c = a + b;
2. Relational Operators	Compare two values (return true or false)	==, !=, >, <, >=, <= if (a > b)
3. Logical Operators	Combine multiple conditions	&&, || if (a > 0 && b < 10)
4. Assignment Operators	Assign values to variables	=, +=, -=, *=, /=, %= x += 5; // x = x + 5
5. Increment/Decrement	Increase or decrease value by 1	++, -- a++;, --b;
6. Bitwise Operators	Perform bit-level operations	&, | int x = a & b;
7. Conditional (Ternary)	Short-hand for if-else	condition ? expr1 : expr2 int max = (a > b) ? a : b;

★ Example

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

int main() {
    int a = 10, b = 5;

    // Arithmetic
    cout << "Sum: " << (a + b) << endl;

    // Relational
    cout << "Is a > b? " << (a > b) << endl;

    // Logical
    cout << "Logical AND: " << (a > 0 && b > 0) << endl;

    // Assignment
    a += 2;
    cout << "After a += 2: " << a << endl;

    // Increment
    b++;
    cout << "After b++: " << b << endl;

    // Bitwise
    cout << "a & b: " << (a & b) << endl;

    return 0;
}
```

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

❖ **Constants in C++**

★ Purpose:

➔ Constants are used to define unchangeable values to improve code readability, prevent accidental modification, and enhance reliability.

➢ How to Define Constants:

a) Using const keyword:

```
const int MAX_USERS = 100;
```

b) Using #define preprocessor directive:

```
#define PI 3.14159
```

➔ **Note :-** const has type checking; #define does not.

★ Example

```
#include <iostream>

using namespace std;

int main() {

   const int MAX = 50;

   cout << "Maximum limit: " << MAX << endl;

   return 0;

}
```

❖ **Literals in C++**

★ Purpose**:**

➔ Literals are actual fixed values used in the code - like numbers, characters, and strings.

➔ **Types of Literals**

- Integer Literal
- Floating-point Literal
- Character Literal
- String Literal
- Boolean Literal
- Null Pointer Literal

★ Example

```
#include <iostream>

using namespace std;

int main() {

   int age = 21;             // 21 is an integer literal

   char grade = 'A';         // 'A' is a character literal

   float pi = 3.14;          // 3.14 is a float literal

   string name = "Alice";    // "Alice" is a string literal

   bool isPass = true;       // true is a boolean literal

   cout << "Name: " << name << ", Age: " << age << ", Grade: " << grade <<
endl;

   return 0;

}
```

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

➔ In C++, conditional statements are used to make decisions in the program based on certain conditions.

➔ They control the flow of execution by executing different blocks of code depending on whether a condition is true or false.

❖ **if-else Statement**

➔ Used to execute one block of code if a condition is true, and another block if it's false.

★ Syntax:

```
if (condition) {
    // Executes if condition is true
}
else {
    // Executes if condition is false
}
```

★ Example:

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

int main() {
    int age;
    cout << "Enter Age : ";
    cin >> age;

  if (age >= 18){
    print("Adult");
  }
  else{
    print("Minor");
  }
    return 0;
}
```

❖ **switch Statement**

➔ Used to execute one block of code based on the value of a variable or expression (usually int or char).

★ Syntax:

```
switch (expression) {
    case value1:
        // Code to execute
        break;
    case value2:
        // Code to execute
        break;
    default:
        // Code to execute 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;
}
```

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

❖ **for Loop**

➔ Used when the number of iterations is known in advance.

★ Syntax:

```
for (initialization; condition; Incre/Decre) {
    // Code to execute
}
```

★ Example:

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
// Output: 1 2 3 4 5
```

❖ **while Loop**

➔ Used when the number of iterations is not known and the loop should run as long as a condition is true.

★ Syntax:

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

★ Example:

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
// Output: 1 2 3 4 5
```

❖ **do-while Loop**

➔ Similar to while, but it guarantees at least one execution of the loop body even if the condition is false initially.

★ Syntax:

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

★ Example:

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

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

❖ **break Statement**

➔ Used to exit a loop or switch statement immediately, even if the loop condition is still true.

★ Example:

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5)
            break;  // exits the loop when i == 5
        cout << i << " ";
    }
    return 0;
}
// Output: 1 2 3 4
```

❖ **continue Statement**

➔ Used to skip the current iteration of the loop and continue with the next iteration.

★ Example:

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3)
            continue;  // skips when i == 3
        cout << i << " ";
    }
    return 0;
}
// Output: 1 2 4 5
```

12. Explain nested control structures with an example.

➔ Nested control structures occur when one control structure (like an if, for, while, or switch) is placed inside another.

➔ This allows you to perform more complex decision-making and looping.

★ **Nested if Statement**

```
#include <iostream>
using namespace std;
int main() {
    int age = 18;
    int marks = 85;

    if (age >= 18) {
        if (marks > 80) {
            cout << "Eligible for scholarship.";
        } else {
            cout << "No scholarship.";
        }
    } else {
        cout << "Not eligible due to age.";
    }

    return 0;
}
```

★ **Nested for Loop (Multiplication Table)#include <iostream>**

```
#include <iostream>
using namespace std;                                    //Output:
int main() {                                            *
    int rows = 5;                                       * *
    for (int i = 1; i <= rows; i++) {                   * * *
        for (int j = 1; j <= i; j++) {                  * * * *
            cout << "* ";                               * * * * *
        }
        cout << endl;
    }
    return 0;
}
```

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

➔ A function in C++ is a block of code that performs a specific task and can be called (invoked) whenever needed in a program. It helps in:

- Breaking a large program into smaller, manageable pieces
- Code reuse
- Improving readability and maintenance

❖ **Function Declaration (or Prototype)**

➔ It tells the compiler about the function's name, return type, and parameters.

➔ Placed before main() or in a header file.

★ **Syntax:**

```
return_type function_name(parameter_list);
```

★ **Example:**

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

❖ **Function Definition**

➔ This is the actual implementation of the function - the block of code that runs when the function is called.

★ **Syntax:**

```
return_type function_name(parameter_list) {
    // body of the function
}
```

★ **Example:**

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

❖ **Function Calling**

➔ You call or invoke the function by writing its name with required arguments.

★ **Syntax:**

function_name(arguments);

★ **Example:**

int result = add(5, 3);

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

➔ The scope of a variable in C++ refers to the region of the program where that variable can be accessed or used.

❖ **Local Scope**

➔ A local variable is declared inside a function or block ({ }).

➔ It can only be accessed within that function or block.

➔ It is created when the block is entered and destroyed when the block ends.

★ Example:

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

void show() {
    int a = 10;  // Local variable
    cout << "Local a = " << a << endl;
}

int main() {
    show();
    return 0;
}
```

❖ **Global Scope**

➔ A global variable is declared outside all functions, usually at the top of the program.

➔ It can be accessed from any function in the program.

➔ Its lifetime is the entire duration of the program.

★ Example:

```
#include <iostream>
using namespace std;
int a = 50;  // Global variable

void show() {
    cout << "Global a from show() = " << a << endl;
}
```

```
int main() {
    cout << "Global a from main() = " << a << endl;
    show();
    return 0;
}
```

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

➔ Recursion is a programming technique where a function calls itself in order to solve a problem.

➔ A recursive function solves a small part of the problem and then calls itself to solve the remaining part. It must have:

1. Base Case – The condition where the recursion stops.
2. Recursive Case – The part where the function calls itself.

★ Syntax

```
return_type function_name(parameters) {

    if (base_condition) {

        // stop recursion

    } else {

        // recursive call

        function_name(modified_parameters);

    }

}
```

★ Example

```
#include <iostream>
using namespace std;
int factorial(int n) {
    if (n == 0 || n == 1)  // Base case
        return 1;
    else
        return n * factorial(n - 1);  // Recursive call
}
int main() {
    int number = 5;
    cout << "Factorial of " << number << " is " << factorial(number);
    return 0;
}
//Output : Factorial of 5 is 120
```

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

➔ A function prototype is a declaration of a function that tells the compiler:

- The function name
- Its return type
- The number and type of parameters

➔ It does not contain the function body.

➔ Why are Function Prototypes Used?

- Forward Declaration :- Allows calling a function before its definition.
- Type Checking :- Compiler ensures the function is called with the correct number and types of arguments.
- Code Organization :- Improves readability and modularity by separating declaration and definition (especially in large programs or multi-file projects).

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

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

➔ Each element can be accessed using its index.

Feature	Single-Dimensional Array	Multidimensional Array
Structure	Linear list	Table-like (2D), cube (3D), etc.
Syntax	int arr[5];	int arr[2][3];
Storage	In a single row	In rows and columns
Accessing Elements	arr[i]	arr[i][j]
Use Cases	Storing lists, marks, prices	Storing matrices, tables, grids

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

➔ Comes from the <string> header

➔ Safer and more flexible

➔ Supports many built-in functions

Function	Description
length() or size()	Returns number of characters
empty()	Returns true if the string is empty
append()	Appends to the string
substr(pos, len)	Returns substring starting at pos
find(str)	Finds position of str in string
compare(str)	Compares strings lexicographically
insert(pos, str)	Inserts str at position pos
erase(pos, len)	Erases len characters from pos
getline(cin, str)	Reads a full line including spaces

★ Example:

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

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

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

    cout << "Result: " << result << endl;

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

    // Substring
    cout << "Substring: " << result.substr(6, 5) << endl;

    // Comparison
    if (str1 == "Hello")
        cout << "str1 is Hello" << endl;

    // Input
    string name;
    cout << "Enter your name: ";
    getline(cin, name);   // reads entire line including spaces
    cout << "Welcome " << name << "!" << endl;

    return 0;
}
```

```
//Output :
Result: Hello World
Length: 11
Substring: World
str1 is Hello
Enter your name: Akhil Kumar
Welcome Akhil Kumar!
```

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

➔ **One-Dimensional (1D) Array**

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

int main() {
   int arr[5] = {5, 10, 15, 20, 25};

   for (int i = 0; i < 5; i++) {
      cout << arr[i] << " ";
   }

   return 0;
}

//Output : 5 10 15 20 25
```

➔ **Two-Dimensional (2D) Array**

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

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

   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 3; j++) {
         cout << matrix[i][j] << " ";
      }
      cout << endl;
   }

   return 0;
}

// Output : 1 2 3
            4 5 6
```

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

➔ Modern C++ uses the string class from the <string> header.

Function / Operation	Description
s.length() / s.size()	Returns the length of the string
s.empty()	Checks if the string is empty
s.append(str)	Appends str to the string
s + str	Concatenates strings
s.substr(pos, len)	Returns substring starting at pos
s.find(str)	Finds position of str
s.insert(pos, str)	Inserts str at position pos
s.erase(pos, len)	Removes len characters from position pos
getline(cin, s)	Takes entire line input with spaces
s == str	Compares two strings

★ Example

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

int main() {
   string s1 = "Hello";
   string s2 = "World";

   string s3 = s1 + " " + s2;  // Concatenation
   cout << "s3: " << s3 << endl;

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

```
    cout << "Substring (6-5): " << s3.substr(6, 5) << endl;

    s3.insert(5, ",");
    cout << "After insert: " << s3 << endl;

    s3.erase(5, 1);
    cout << "After erase: " << s3 << endl;

    if (s1 == "Hello")
        cout << "s1 is Hello" << endl;

    return 0;
}

// Output :
s3: Hello World
Length: 11
Substring (6-5): World
After insert: Hello, World
After erase: Hello World
s1 is Hello
```

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

➔ Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of "objects", which contain data (attributes) and functions (methods).

➔ Here are the main pillars of OOP:

I. Class and Object

➔ Class

- A class is a blueprint or template for creating objects.
- It defines attributes (variables) and methods (functions).

➔ Object

- An object is an instance of a class.
- It accesses the class's members using the dot (.) operator.

II. Encapsulation

- Wrapping data and methods together into a single unit (class)
- Keeps data safe from outside interference
- Achieved using access specifiers: private, public, protected

III. Abstraction

- Hiding internal details and showing only essential features
- Simplifies complexity by exposing only what is necessary

IV. Inheritance

- Acquiring properties and behaviors of one class (base) into another (derived)
- Promotes code reuse

V. Polymorphism

- Same function name behaves differently based on context
- Two types:
    - Compile-time (Static) Polymorphism → Function Overloading
    - Run-time (Dynamic) Polymorphism → Function Overriding using virtual functions

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

➔ **Class**

- A class in C++ is a user-defined data type that acts as a blueprint for creating objects.
- It contains data members (variables) and member functions (methods) that operate on the data.

➔ **Object**

- An object is an instance of a class.
- When a class is defined, no memory is allocated. Memory is allocated only when an object is created.

★ Syntax:

```
class ClassName {
  // Access specifier
public:
  // Data members
  // Member functions
};
```

★ Example:

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

// Class definition
class Student {
public:
```

```
    string name;
    int age;

    void display() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

int main() {
    // Creating an object of Student
    Student s1;

    // Accessing data members and member function
    s1.name = "Akhil";
    s1.age = 20;

    s1.display();  // Calling member function

    return 0;
}


//Output:
Name: Akhil
Age: 20
```

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

➔ Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) in C++.

➔ It allows a new class (derived class) to acquire properties and behaviors (data members and member functions) from an existing class (base class).

★ Syntax:

```
class Base {
  // base class members
};

class Derived : access_specifier Base {
  // derived class members
};
```

★ Example:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

// Derived class
class Dog : public Animal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};
int main() {
    Dog d;
    d.eat();   // inherited from Animal
    d.bark();  // own function
    return 0;
}
```

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

➔ Encapsulation is an Object-Oriented Programming (OOP) concept that refers to binding data and the functions that operate on that data into a single unit - typically a class.

➔ It also means restricting direct access to some of an object's components to protect data from unauthorized modification.

➔ Encapsulation is achieved using:

1. Classes to wrap data and methods together
2. Access specifiers:
    - private: accessible only inside the class
    - public: accessible from outside the class
    - protected: accessible in the class and derived classes