OOPS By Bhavy Sharma

Q1. Differentiate between << & >> operators.
Ans. In C++, `<<` and `>>` are used as stream insertion and extraction operators:
`<<` **(Stream Insertion Operator):** Used to output data to a stream (usually `cout` for console output).

- **Example:**

  cout << "Hello, World!";

`>>` **(Stream Extraction Operator):** Used to extract data from a stream (usually `cin` for console input).

- **Example:**

  int x;

  cin >> x;

Q2. What is the difference between structured and an object-oriented language?

Ans. The difference between structured and object-oriented programming languages lies in their approach to organizing code, handling data, and problem solving.

1. Structured Programming Language:

Structured programming focuses on a logical structure where code is organized into small functions. It promotes clear, understandable, and modular code but doesn't inherently deal with objects or data encapsulation.

Important Points :-

Procedures/Functions: Code is divided into procedures or functions that perform specific tasks.

Sequential Flow: Structured languages typically follow a top-down approach with sequential execution of commands.

Control Structures: Uses control structures like loops (`for`, `while`), conditionals (`if`, `else`), and switches to control program flow.

Data: Data and functions are separate, and functions operate on external data.

- **Example (C):**

```
#include <stdio.h>
void sayHello() {
    printf("Hello, World!");
}
int main() {
    sayHello();
    return 0;
}
```

2. Object-Oriented Programming Language (OOP):

Object-oriented programming organizes code around objects, which represent real-world entities. These objects combine data (attributes) and functions (methods) into a single unit and encourage reusability, modularity, and abstraction.

Important Points :-

Classes and Objects: A class defines a blueprint for creating objects (instances). An object is an instance of a class that encapsulates both data and behaviour.

Encapsulation: Combines data and methods in a single unit, hiding internal implementation and protecting data through access control (public, private, protected).

Inheritance: Allows one class to inherit properties and methods from another, promoting code reuse.

Polymorphism: Enables objects of different classes to be treated as objects of a common superclass, typically using method overriding or interfaces.

Abstraction: Focuses on hiding complexity and showing only essential details.

Example(C++):-

```
#include <iostream>

class Person {
   private:
      string name;
   public:
      void setName(string n) {
         name = n;
      }
      void introduce() {
         cout << "Hello, my name is " << name <<endl;
      }
};

int main() {
   Person p;
   p.setName("John");
   p.introduce();
   return 0;
}
```

Aspect	Structured Programming	Object-Oriented Programming (OOP)
Basic Unit	Functions/Procedures	Objects (instances of classes)
Data and Functions	Separate (functions operate on data)	Combined (data and methods encapsulated)
Approach	Top-down, logical structure	Bottom-up, real-world modeling
Modularity	Uses functions for modularity	Uses classes and objects for modularity
Reusability	Limited (mainly via functions)	High (inheritance, polymorphism)
Examples	C, Pascal	C++, Java, Python (OOP part)
Abstraction	Focuses less on abstraction	Strong abstraction (classes hide details)
Inheritance & Polymorphism	Not supported	Supported

Q3. Define Classes and Objects. Explain the Concept of Base and Derived Class using an Example?
Ans.
Classes:- A class in object-oriented programming is a user-defined data type that serves as a blueprint for creating objects. It defines attributes (data) and methods (behaviour/Functions) that the objects instantiated from the class will possess. The class itself doesn't hold data, but it provides a structure for objects, which are specific instances of that class.

Object:- An object is an instance of a class. It represents a real-world entity and holds actual data. Each object created from a class can have different values for its attributes, but it shares the same structure and behaviour defined by the class.

Example of Class and Object:

```
class Car {
public:
    string brand;
    int speed;

    void accelerate() {
```

```
        speed += 10;
    }
};

int main() {
    Car myCar;              // Creating an object 'myCar' of class 'Car'
    myCar.brand = "Swift";
    myCar.speed = 100;
    myCar.accelerate();
    return 0;
}
```

Base and Derived Classes:

In object-oriented programming, inheritance is a mechanism where one class (derived class) inherits the properties and behaviour (attributes and methods) of another class (base class). This allows for code reuse and logical hierarchy.

- Base Class (Parent Class or Superclass): The class from which properties and behaviours are inherited.
- Derived Class (Child Class or Subclass): The class that inherits from the base class and can have additional properties or methods or override existing ones.

Example of Base and Derived Class:

```
// Base Class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." <<endl;
    }
};

// Derived Class
```

```
class Dog : public Animal {  // Dog inherits from Animal
public:
   void bark() {
      cout << "The dog barks." <<endl;
   }
};

int main() {
   Dog myDog;              // Create object of Derived Class 'Dog'
   myDog.eat();           // Inherited from Animal class
   myDog.bark();          // Method of Dog class
   return 0;
}
```

Q4. Explain Various DataTypes Used in C?

Ans. In C programming, data types specify the type of data that a variable can store. C has a rich set of data types that can be classified into the following categories:

1. Basic (Primary) Data Types:

1. **Int :-** Used to store integer (whole) numbers, both positive and negative.
2. **Float :-** Used to store single-precision floating-point numbers (decimal numbers).
3. **Double :-** Used to store double-precision floating-point numbers for greater accuracy.
4. **Char :-** Used to store individual characters.

2. Derived Data Types:

1. **Array :-** A collection of elements of the same data type, stored in contiguous memory locations.
   Example :- **int arr[5] = {1, 2, 3, 4, 5};**

2. **Pointers :-** A variable that stores the memory address of another variable.
   Example :- **int *ptr;**
3. **Structure :-** A user-defined data type that groups variables of different data types under a single name.
   Example : - **struct Person {**
   char name[50];
   int age;
   };
4. **Union :-** Similar to a structure, but all members share the same memory location, meaning only one member can be used at a time.
   Example :- **union Data {**
   int i;
   float f;
   char str[20];
   };
5. **Enum :-** A user-defined data type that assigns names to integral constants, improving code readability.
   Example:- **enum Days { MONDAY, TUESDAY, WEDNESDAY };**

Q5. Define Array and its Type. Explain One Dimensional Array?
Ans.
Array:- An array is a collection of variables of the same data type that are stored in contiguous memory locations. It allows multiple values to be stored under a single variable name, and each element in the array can be accessed using an index. Arrays are useful for storing and managing large amounts of data efficiently.

Types of Arrays:

1. **One-Dimensional Array**: A linear array where data is stored in a single row or line.
2. **Two-Dimensional Array**: An array that stores data in rows and columns, like a table or matrix.

3. **Multi-Dimensional Array**: An array with more than two dimensions, often used in more complex scenarios like 3D arrays.

One-Dimensional Array:

A **one-dimensional array** is the simplest form of an array, where elements are stored in a single row or line. You can think of it as a list of elements stored one after another.

Example of One-Dimensional Array:

```
#include <stdio.h>
int main() {
    int numbers[5];  // Declare an array of 5 integers
    // Initializing elements
    numbers[0] = 10;
    numbers[1] = 20;
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;

    // Accessing and printing array elements
    for(int i = 0; i < 5; i++) {
        printf("Element at index %d: %d\n", i, numbers[i]);
    }
    return 0;
}
```

Advantages of One-Dimensional Arrays:

1. **Efficient Access**: You can access any element in the array in constant time using its index.
2. **Grouped Data**: You can store multiple values of the same type under one name.

3. **Easy to Iterate**: Arrays are easy to loop through using a `for` or `while` loop.

Disadvantages of One-Dimensional Arrays:

1. **Fixed Size**: The size of the array must be known at compile-time and cannot be changed later.
2. **Same Data Type**: Arrays can only store elements of the same data type.

Q6. Define algorithm and flowcharts. Write an algorithm to check whether a given number is prime or not? Draw its flow chart as well Ans.

Algorithm:

An algorithm is a step-by-step procedure or a set of rules to be followed to solve a specific problem. Algorithms are widely used in programming and computer science to perform tasks efficiently.

Flowchart:

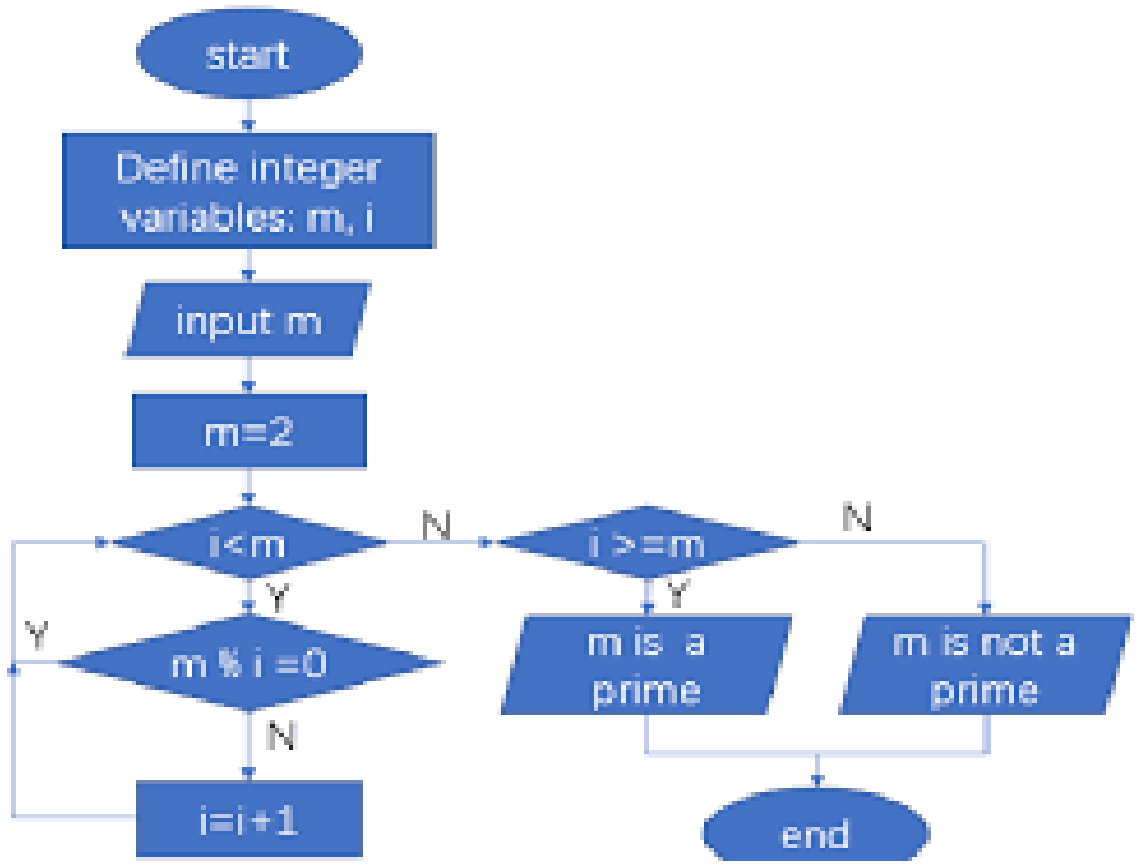
A flowchart is a diagram that represents an algorithm, process, or workflow. It shows steps in the form of boxes connected by arrows, illustrating the sequence of operations. Flowcharts help visualize the logic of a process.

Algorithm to Check Whether a Number is Prime:

1. **Start**
2. Input the number `n`
3. If `n` is less than or equal to 1, print "Not Prime" and **End**
4. Set `i = 2`
5. While `i` is less than or equal to √n:
    - If `n % i == 0`, print "Not Prime" and **End**
    - Increment `i` by 1

6. If no divisor is found, print "Prime"
7. **End**

FlowChart:



Q7. What is constructor? Explain the various types of constructor with Example

Ans. A constructor is a special type of method used in object-oriented programming to initialize objects. When an object is created, the constructor is automatically invoked to set initial values for its attributes or to perform any setup steps.

Constructors have the following characteristics:

- They have the same name as the class.
- They do not have a return type, not even `void`.

- They are invoked automatically when an object is instantiated.

Types of Constructors:

Default Constructor: A constructor that takes no parameters.

Parameterized Constructor: A constructor that takes parameters to initialize object attributes with specific values provided by the user.

Copy Constructor: A constructor that creates a new object as a copy of an existing object.

Default Copy Constructor: If no copy constructor is defined, the compiler provides a default copy constructor that performs a shallow copy (copies only the values of data members, not the objects they point to).

Q8. What do you mean by nesting of classes? also explain how friend function is important in C++

Ans. Nesting of Classes

Nesting of classes refers to defining a class within another class. The inner class (nested class) is a member of the outer class and has access to the outer class's data members and functions, including private members.

Nesting classes can be useful in cases where a class is tightly coupled with another class or represents a subpart of the outer class.

Example of Nested Class : -

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

class Outer {
public:
    int outerVar;
```

```
    // Nested class (Inner class)
    class Inner {
    public:
        void display() {
            cout << "I am the nested Inner class!" << endl;
        }
    };

    void show() {
        cout << "I am the outer class!" << endl;
    }
};

int main() {
    // Creating object of outer class
    Outer outerObj;
    outerObj.show();

    // Creating object of nested inner class
    Outer::Inner innerObj;
    innerObj.display();

    return 0;
}
```

Friend Function in C++

A **friend function** is a special type of function that is **not a member of a class** but can still access the private and protected members of the class. It is declared using the keyword `friend` within the class. This allows the function to "friend" the class, giving it privileged access to its private/protected data.

Importance of Friend Function:

1. **Access to Private and Protected Data:**
    - Friend functions can access all members of the class, even private and protected ones, which is not possible for regular functions. This allows more flexibility when you need external functions to work with class members.
2. **Interfacing Multiple Classes:**

- A friend function can be useful when a single function needs to interact with multiple classes by accessing their private members.

3. **Non-Member Function with Special Access:**
   - A friend function is not bound to any specific object of the class, making it useful when the function needs to operate on more than one object or when an object's data needs to be manipulated externally.

Q9. Explain static data members & statics data members function.

Ans. ## Static Data Members in C++

A static data member is a class variable that is shared by all objects of the class. It belongs to the class itself, rather than to any specific object, meaning there is only one copy of the static member, regardless of how many objects are created from that class.

Key Characteristics of Static Data Members:

1. **Shared Across All Objects:**
   - All objects of the class share the same static data member. Modifying it in one object affects all other objects.
2. **Memory Allocation:**
   - Static data members are allocated memory only once, when the class is loaded. They do not consume extra memory for each object.
3. **Accessing Static Members:**
   - Static data members can be accessed using the class name directly (e.g., `ClassName::staticMember`), or through an object of the class.
4. **Initialization Outside the Class:**
   - Static members must be defined and initialized **outside** the class (except when they're declared as `const` integral types).

Static Member Functions in C++

A **static member function** is a class function that can be called without needing to create an object of the class. Static member functions can only access **static data members** and other **static member functions**.

Key Characteristics of Static Member Functions:

1. **No Object Required:**
    - Static member functions can be called using the class name without creating an object.
2. **Cannot Access Non-Static Members:**
    - Since static functions do not belong to any specific object, they can only access static members. They **cannot access non-static** data members or functions because these are tied to objects.
3. **Access Static Data:**
    - They are mainly used to manipulate static data members of the class.

Q10. Define Encapsulation.

Ans. Encapsulation

Encapsulation is one of the core principles of Object-Oriented Programming (OOP) that refers to the concept of binding data (variables) and functions (methods) that manipulate that data into a single unit, typically a class. It also involves restricting access to certain details of an object and only allowing certain parts of it to be accessible through well-defined interfaces, such as public methods.

In simple terms, encapsulation is the technique of hiding internal details (data) of a class from the outside world and exposing only the necessary parts through methods. This helps to protect data from unauthorized access or accidental modification.

Q11. Define Polymorphism. what are different method of implementation polymorphism in C++

Ans. Polymorphism in C++

Polymorphism is one of the core principles of Object-Oriented Programming (OOP) that refers to the ability of a function, object, or method to behave in multiple ways depending on the context. In simple terms, polymorphism allows one interface to be used for different types of objects.

The word "polymorphism" comes from the Greek words "poly" (meaning many) and "morph" (meaning forms), indicating that a single entity (like a function or operator) can take on different forms.

Types of Polymorphism in C++

There are two main types of polymorphism in C++:

1. **Compile-time (Static) Polymorphism**
2. **Run-time (Dynamic) Polymorphism**

1. Compile-time (Static) Polymorphism

Compile-time polymorphism is achieved by method overloading or operator overloading, and the decision about which method or operator to invoke is made during compilation. This is called static binding or early binding.

Methods of Implementing Compile-time Polymorphism:

- **Function Overloading**
- **Operator Overloading**

a) Function Overloading:

Function overloading is when multiple functions have the **same name** but differ in the number or type of parameters. The appropriate function is selected based on the function signature at compile time.

b) Operator Overloading:

Operator overloading allows operators like `+`, `-`, `*`, etc., to be redefined so they work with objects of user-defined types (e.g., classes). This provides a way to extend the meaning of operators for custom types.

2. Run-time (Dynamic) Polymorphism

Run-time polymorphism occurs when the **function to be executed is determined at runtime**, based on the type of the object. This is implemented using **inheritance** and **virtual functions**. This is also called **dynamic binding** or **late binding**.

Methods of Implementing Run-time Polymorphism:

- **Virtual Functions**
- **Function Overriding**

a) Virtual Functions:

Virtual functions allow derived classes to override a base class function. When a virtual function is called using a base class pointer or reference, the derived class version of the function will be executed, if it exists.

b) Function Overriding:

Function overriding occurs when a derived class provides its own implementation of a function that is already defined in the base class. The function signature (name, parameters) in both the base and derived classes must be the same.

Q12. Define Inheritance. What are the various types of Inheritance?

Ans. Inheritance in C++

Inheritance is one of the key features of Object-Oriented Programming (OOP) that allows one class (called a derived class) to inherit the properties and behaviors (methods) of another class (called a base class). This promotes code reusability, as it enables a new class to reuse

the properties and functionalities of an existing class without rewriting the code.

Inheritance helps in establishing a relationship between the base class and the derived class, where the derived class extends or modifies the behavior of the base class.

Types of Inheritance in C++

There are several types of inheritance in C++, depending on how the classes are related to each other. The most common types are:

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

1. Single Inheritance

In single inheritance, one class inherits from a single base class. The derived class gets access to the properties and methods of the base class.

2. Multiple Inheritance

In multiple inheritance, a derived class inherits from more than one base class. This means the derived class can inherit properties and methods from multiple base classes.

3. Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class, creating a chain of inheritance levels.

4. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from the same base class. This means multiple classes share the same parent class.

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example, it can be a combination of multiple and hierarchical inheritance.

Q13. What is file?

Ans. What is a File?

A file is a collection of data or information stored on a disk. Files allow us to store data persistently, meaning that the data remains available even after the program that created or modified it has terminated. Files are often categorized based on their type, such as text files (which store human-readable text) and binary files (which store data in a format that is not human-readable).

In C++, file operations are supported by the `<fstream>` library, which includes:

1. **`ifstream`: Used to read data from a file.**
2. **`ofstream`: Used to write data to a file.**
3. **`fstream`: Can be used for both reading and writing.**

Random Access in Files

Random access refers to the ability to access any part of the file directly, without reading the file sequentially from the beginning. This is useful when you need to update or read data from a specific location in a file. In C++, you can achieve random access in files using functions like `seekg()` (for input) and `seekp()` (for output) to move the file pointer to a desired location.

Key Functions:

- **`seekg()`: Moves the input file pointer to a specific location.**
- **`seekp()`: Moves the output file pointer to a specific location.**
- **`tellg()`: Returns the current position of the input file pointer.**
- **`tellp()`: Returns the current position of the output file pointer.**

Q14. Define Functions. What are the advantages of using functions? what are the various methods of parameter passing to the functions? Explain

Ans. ## What is a Function?

A function is a block of code that performs a specific task and can be reused multiple times throughout a program. Functions take inputs (parameters), perform operations, and may return an output. Functions allow code to be organized, reduce repetition, and make programs more modular and easier to maintain.

Advantages of Using Functions:

1. **Modularity**: Functions break the code into smaller, manageable sections. Each function performs a specific task, making the code easier to understand and maintain.
2. **Reusability**: Once a function is defined, it can be called multiple times without rewriting the same code.
3. **Code Readability**: Using functions makes programs easier to read and understand. It allows programmers to focus on high-level logic by separating complex tasks into individual functions.
4. **Debugging**: Functions allow errors to be isolated and fixed more easily. If something goes wrong, you can debug a small part of the program rather than the entire codebase.
5. **Code Maintenance**: By using functions, updating or modifying a program becomes simpler. If a change is required in a task, it only needs to be made in the function definition, affecting all instances where the function is used.

Methods of Parameter Passing to Functions:

Parameters can be passed to functions in several ways. The method used determines how the function accesses or manipulates the passed data. The most common methods are:

1. **Pass by Value**
2. **Pass by Reference**
3. **Pass by Pointer**

1. Pass by Value:

In **pass by value**, the function receives a copy of the argument's value. Any changes made to the parameters inside the function do not affect the original values outside the function.

2. Pass by Reference:

In **pass by reference**, the function receives the memory address of the argument. Any changes made to the parameter inside the function directly affect the original value outside the function.

3. Pass by Pointer:

In **pass by pointer**, a pointer (the memory address) of the variable is passed to the function. This method allows functions to modify the actual data in memory, similar to pass by reference.