Object Oriented Programming

1. What is Object Oriented Programming?

- When a complete software works as a bunch of objects talking to each other
- Entire program is organized using objects

2. What is a class?

- A Blueprint/template of objects having common properties
- User defined data type that contains data members and functions that operate on that data member
- Defines the properties (attributes) and behaviors (methods) that the objects will have

3. How much memory does a class occupy?

- Classes do not use memory. They merely serve as a template from which objects are instantiated.
- Now, objects actually initialize the class members and methods when they are created, using memory in the process.

4. What is an object?

- Instance of the class
- Real world entity
- Need to create an object of the class in order to use the data members and methods of the class

5. Difference between a 'Class' and a 'Structure'

- In C++, both structures and classes are used to define user-defined data types that can encapsulate data members and functions.
- Some of the main distinctions:
- **Default Member Accessibility**
  - In a structure, members are **public** by default. This means that all the data members of a structure can be accessed directly from outside the structure without any restriction.

```
struct Point {
        int x;  // Public by default
        int y;  // Public by default
    };
```

  - In a class, members are **private** by default. This means that, by default, the data members and member functions of a class are not accessible from outside the class.

```
class Point {
        int x;  // Private by default
        int y;  // Private by default
    };
```

- **Access Specifiers**
  - In a structure, you cannot explicitly specify access specifiers (e.g public, private, protected). All members are public by default.
  - In a class, you can use access specifiers to control the visibility of members. You can explicitly declare members as (e.g public, private, protected)
- **Inheritance**
  - Structures do not support inheritance. They are primarily used for simple data structures and do not have features like polymorphism, encapsulation, and inheritance.
  - Classes support inheritance, allowing for the creation of hierarchies and the implementation of polymorphism through virtual functions.

- **Constructor and Destructor**
  - In a structure, you cannot explicitly declare a constructor or destructor. However, you can use aggregate initialization to initialize the members.

```
// Define a Point structure
struct Point {
    int x;
    int y;
};
int main() {
    // Aggregate initialization of a Point structure
    Point p1 = {10, 20};
    Point p2{30, 40};  // Alternatively, you can use braces without the equals sign
    // Displaying the values
    cout << "p1: (" << p1.x << ", " << p1.y << ")" << endl;
    cout << "p2: (" << p2.x << ", " << p2.y << ")" << endl;
}
```

  - In a class, you can declare and define constructors and destructors, allowing for explicit initialization and cleanup of class objects.

6. Static Methods vs Non-Static Methods

- In object-oriented programming (OOP), functions associated with a class or object can be categorized as static or non-static (also known as instance methods).
- **Static Methods:**
  - Static methods are **associated with the class** rather than with an instance of the class.
  - Called on the class itself, not on an object of the class.
  - Don't have access to the instance-specific data of the class.
  - Declared using the **static** keyword.

```
class MathOperations {
public:
    static int add(int a, int b) {
        return a + b;
    }
    static double multiply(double a, double b) {
        return a * b;
    }
};
int main() {
    // Calling static methods without creating an object
    int result1 = MathOperations::add(5, 3);
    double result2 = MathOperations::multiply(2.5, 4.0);
    cout << "Result of addition: " << result1 << endl;
    cout << "Result of multiplication: " << result2 << endl;
}
```

  - Here, `add` and `multiply` are static methods of the `MathOperations` class. They can be called using the scope resolution operator `::` without creating an object of the class
- **Non-Static (Instance) Methods**:
  - Non-static methods are **associated with instances (objects) of the class**. They have access to the instance-specific data of the class. They are called on objects of the class
  - No special keyword is needed for non-static methods

```
class Circle {
private:
    double radius;
public:
    // Constructor
    Circle(double r) : radius(r) {}
```

```
    // Non-static method
    double calculateArea() {
        return 3.14 * radius * radius;
    }
};
int main() {
    // Creating an object of the Circle class
    Circle myCircle(5.0);
    // Calling the non-static method on the object
    double area = myCircle.calculateArea();
    cout << "Area of the circle: " << area << endl;
}
```

        - Here, `calculateArea` is a non-static method of the `Circle` class. It is called on an object (`myCircle`) of the class and has access to the instance-specific data (the `radius` attribute)
    - Use static methods when the functionality is not specific to an instance, and use non-static methods when the functionality depends on the instance's data

7. Is it always necessary to create objects from class?

- **No.** If the base class includes non-static methods, an object must be constructed.
- But no objects need to be generated if the class includes static methods. In this instance, you can use the class name to directly call those static methods.

8. What is a constructor?

- A constructor is a block of code that **initializes the newly created object**. A constructor is not a method as it doesn't have a return type.
- Its purpose is to initialize the object's data members and perform any necessary setup.
- It generally is the method having the same name as the class but in some languages, it might differ.
    - In python, a constructor is named **__init__**.
    - In C++ and Java, the constructor is named the same as the class name.

9. What are the different types of constructors in C++?

- **Default Constructor**
    - A default constructor is a constructor with no parameters.
    - It is called automatically when an object is created if no other constructor is specified.
    - If a class does not explicitly define any constructor, the compiler generates a default constructor.

```
class MyClass {
public:
    // Default constructor
    MyClass() {
        cout << "Default constructor called." << endl;
    }
};
int main() {
    // Creating an object invokes the default constructor
    MyClass obj;
}
```

- **Parameterized Constructor**
    - A parameterized constructor accepts parameters to initialize the data members of an object.
    - It allows you to initialize object properties at the time of creation.

```
class Point {
public:
    // Data members
    int x;
```

```
    int y;
    // Parameterized constructor
    Point(int xVal, int yVal) : x(xVal), y(yVal) {
        cout << "Parameterized constructor called." << endl;
    }
};
int main() {
    // Creating an object with parameters
    Point myPoint(3, 4);
}
```

- **Copy Constructor**
  - A copy constructor creates a new object as a copy of an existing object.
  - It is called when an object is passed by value or returned by value.

```
class ComplexNumber {
public:
    // Data members
    double real;
    double imag;
    // Copy constructor
    ComplexNumber(const ComplexNumber& other){
        real = other.real;
        imag = other.imag;
        cout << "Copy constructor called." << endl;
    }
};
int main() {
    // Creating an object
    ComplexNumber num1;
    // Creating another object as a copy of the first one
    ComplexNumber num2 = num1;
}
```

- **Constructor Overloading**
  - Constructor overloading involves defining multiple constructors in a class with different parameter lists. Allows creating objects with different initialization options.

```
class Rectangle {
public:
    // Data members
    int width;
    int height;
    // Default constructor
    Rectangle() : width(0), height(0) {
        cout << "Default constructor called." << endl;
    }
    // Parameterized constructor
    Rectangle(int w, int h) : width(w), height(h) {
        cout << "Parameterized constructor called." << endl;
    }
    // Constructor with a default value
    Rectangle(int side) : width(side), height(side) {
        cout << "Single-parameter constructor called." << endl;
    }
};
int main() {
    Rectangle rect1;          // Default constructor
    Rectangle rect2(4, 6);    // Parameterized constructor
    Rectangle square(5);      // Single-parameter constructor
}
```

10. What is a Destructor?

- A destructor is a method that is automatically called when the object is made of scope or destroyed.
- A destructor cannot be overloaded in a class. There can only be one destructor present in a class.
  - In C++, the destructor name is the same as the class name but with the **(~)** tilde symbol as the prefix.
  - In Python, the destructor is called **__del__**.
  - In Java, the garbage collector automatically deletes the useless objects so there is no concept of destructor in Java.

```
class MyClass {
public:
    // Constructor
    MyClass() {
        cout << "Constructor called." << endl;
    }
    // Destructor
    ~MyClass() {
        cout << "Destructor called." << endl;
    }
};
int main() {
    MyClass obj;
    // Object goes out of scope, destructor is automatically called
    cout << "Exiting main function." << endl;
    return 0;
}
```

11. What is a Virtual Function?

- A virtual function is a member function of a class that can be overridden by a function with the same signature in a derived class.
- Virtual functions are essential for providing abstraction where different derived classes can provide their own implementations of a common interface defined in the base class.
  - In C++, a virtual function is declared using the **virtual** keyword. **@override** annotation is used to explicitly indicate method overriding.
  - In Java, every public, non-static, and non-final method is a virtual function
  - In Python, methods are always virtual
- In a non-virtual function, the function called is determined at compile-time based on the type of the pointer or reference.
- In contrast, with virtual functions, the function called is determined at runtime based on the actual type of the object being pointed to or referenced.

C++

```
// Base class with a virtual function
class Shape {
public:
    virtual void draw() const {
        cout << "Drawing a shape." << endl;
    }
};
// Derived class with a specific implementation of the virtual function
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};
```

Java

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape.");
    }
}
class Circle extends Shape {
    // Overrides the draw method in the base class
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

Python

```
class Shape:
    # Virtual function in Python is achieved without any special keyword
    def draw(self):
        print("Drawing a shape.")
class Circle(Shape):
    # Overrides the draw method in the base class
    def draw(self):
        print("Drawing a circle.")
```

12. What is a Pure Virtual Function?

- A Function that doesn't contain any statements. This function is defined in the derived class if needed.
  - In C++, a pure virtual function is marked as "pure" using the = 0 syntax.
  - In Java, abstract classes are used to define abstract methods (pure virtual functions), and the abstract keyword is used.
  - In Python, the abc module is used to create abstract base classes with abstract methods (pure virtual functions). The @abstractmethod decorator is used to declare an abstract method.

C++

```
class Shape {
public:
    // Pure virtual function
    virtual void draw() const = 0;
};
// Derived class implementing the pure virtual function
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};
```

Java

```
abstract class Shape {
    // Pure virtual function (abstract method)
    abstract void draw();
}
// Derived class implementing the pure virtual function
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

Python

```
from abc import ABC, abstractmethod
# Abstract base class with a pure virtual function
class Shape(ABC):
    # Pure virtual function
    @abstractmethod
    def draw(self):
        pass
# Derived class implementing the pure virtual function
class Circle(Shape):
    def draw(self):
        print("Drawing a circle.")
```

13. What are the access specifiers? What is its significance in OOPS?

- Keywords that define the visibility and accessibility of class members (attributes and methods) from outside the class.
- Three types of access specifiers:
  - Public
  - Private
  - Protected
- **Public**
  - Members declared as public are accessible from any part of the program. (From with in the class and outside the class)
  - Public members are part of the class interface, and their details can be **freely accessed and modified by any code that uses the class.**

```
class MyClass {
   public:
       int publicVar;
       void publicMethod() {
           // Implementation}
   };
```

- **Private**
  - Members declared as private are only accessible **within the same class.**
  - Private members are hidden from the outside world, and **access to them is restricted to the methods of the same class.**
  - Private members enhance security by preventing external code from directly accessing or modifying sensitive data. This helps in maintaining the integrity of the class.

```
 class MyClass {
   private:
       int privateVar;
       void privateMethod() {
           // Implementation
       }
   public:
       void publicMethod() {
           // Can access privateVar and privateMethod
       }
   };
```

- **Protected**
  - Members declared as protected are accessible **within the same class and its derived classes.**

```
class MyBaseClass {
   protected:
       int Var;
```

```
};

class MyDerivedClass : public MyBaseClass {
public:
    void accessProtected() {
        Var = 42;  // Accessing protected member from the derived class
    }
};
```

14. List main features of OOPS?

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

15. What is Inheritance?

- Allows a new class to inherit properties and behaviors from an existing class. The existing class is called the base class or parent class, and the new class is called the derived class or child class.
- Inheritance provides a way to model an "is-a" relationship between classes, where a derived class is a specialized version of the base class.
- Inheritance promotes code reuse and helps create a hierarchical structure in the code.

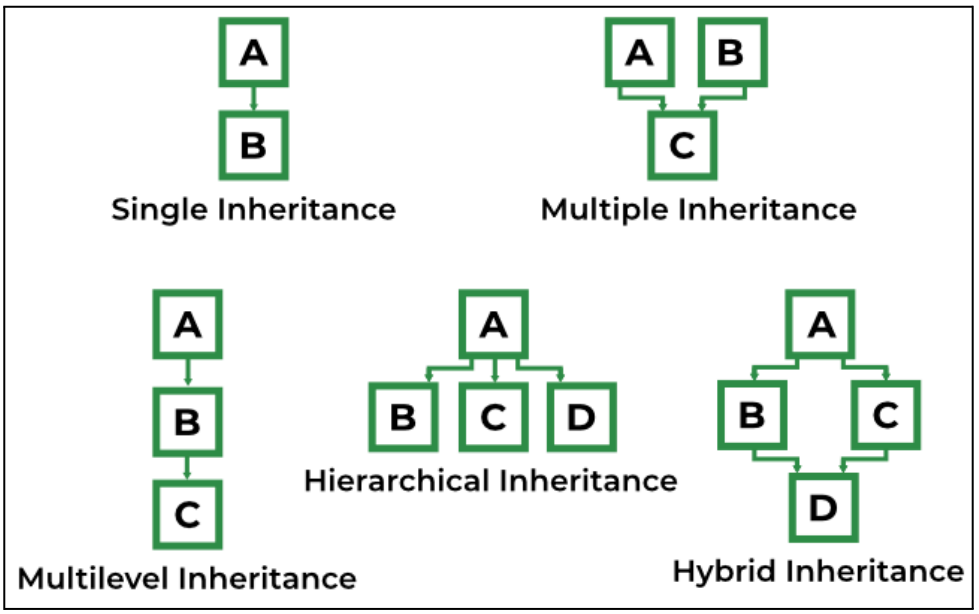
```
// Base class
class Animal {
protected: // To allow derived classes to access the `name` attribute.
    string name;
public:
    Animal(const std::string& n) : name(n) {}
    void eat() {
        cout << name << " is eating." << endl;
    }
    void sleep() {
        cout << name << " is sleeping." << endl;
    }

// Derived class 1
class Dog : public Animal {
public:
    Dog(const std::string& n) : Animal(n) {}
    void bark() {
        cout << name << " is barking." << endl;
    }
};

// Derived class 2
class Cat : public Animal {
public:
    Cat(const std::string& n) : Animal(n) {}
    void meow() {
        cout << name << " is meowing." << endl;
    }
};

int main() {
    // Creating objects of derived classes
    Dog myDog("Buddy");
    Cat myCat("Whiskers");
    // Using inherited methods
```

```
    myDog.eat();
    myDog.sleep();
    myDog.bark();
    myCat.eat();
    myCat.sleep();
    myCat.meow();
}
```

- `Animal` is the base class with attributes and methods common to all animals.
- `Dog` and `Cat` are derived classes that inherit from the `Animal` class.
- The `Dog` and `Cat` classes have their own specific methods (`bark` and `meow`, respectively) in addition to the inherited methods from the base class.

16. What different types of Inheritance are there?

- **Single Inheritance:** Child class derived directly from the base class
- **Multiple Inheritance:** Child class derived from multiple base classes.
- **Multilevel Inheritance:** Child class derived from the class which is also derived from another base class and so on...
- **Hierarchical Inheritance:** Multiple child classes derived from a single base class.
- **Hybrid Inheritance:** Inheritance consisting of multiple inheritance types of the above specified.



17. Overloading vs Overriding

- **Function Overloading**
  - Function overloading refers to the ability to define multiple functions in the same scope with the same name but different parameter lists.
  - Overloading involves defining multiple functions with the same name but different parameters.
- Key Points:
  - Overloaded functions must have different parameter lists (either in terms of the number or types of parameters).
  - The compiler determines which function to call based on the number and types of arguments provided during the function call.
  - Overloading is resolved at compile-time and is also known as compile-time polymorphism or static polymorphism.

```
class MathOperations {
public:
    int add(int a, int b) {
```

```
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
};
int main() {
    MathOperations math;
    cout << math.add(3, 4) << endl;      // Calls int add(int, int)
    cout << math.add(2.5, 3.7) << endl; // Calls double add(double, double)
}
```

- **Function Overriding:**
  - Function overriding occurs when a derived class provides a specific implementation for a function that is already defined in its base class.
  - Overriding involves providing a specific implementation for a function in a derived class that is already defined in its base class.
- Key Points:
  - The base class must declare the function as virtual, and the derived class must use the`override` keyword to indicate that it intends to override that function.
  - The function signature (name, return type, and parameters) in the derived class must match the function signature in the base class.
  - Overriding is resolved at runtime, and it is associated with runtime polymorphism

```
class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing a shape." << std::endl;
    }
};

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

int main() {
    Shape* shape1 = new Shape();
    Shape* shape2 = new Circle();
    shape1->draw();  // Calls Shape::draw()
    shape2->draw();  // Calls Circle::draw()
}
```

18. What is Polymorphism?

- Property of the code to behave differently in different contexts
- Polymorphism can be classified in 2 types:
  - Compile-time (or static) polymorphism
  - Runtime (or dynamic) polymorphism.
- **Compile-time Polymorphism (Function Overloading)**
  - Achieved through function overloading, where multiple functions in the same scope have the same name but different parameter lists.
  - Binding of the function call to its code happens at compile time. The compiler determines the appropriate function to call based on the number and types of arguments during compile-time.

```
  // Function Overloading
```

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

int main() {
    cout << add(5, 7) << endl;      // Calls int add(int, int)
    cout << add(3.5, 2.7) << endl; // Calls double add(double, double)
    return 0;
}
```

- **Runtime Polymorphism (Function Overriding)**
  - Achieved through function overriding and virtual functions in C++.
  - Binding of the function call to it's code happens at the runtime
  - It allows a derived class to provide a specific implementation of a function that is already defined in its base class. This is accomplished using pointers or references to base class objects.
  - Enables code to work with objects of various types through a common interface.

```
// Base class with a virtual function
class Shape {
public:
    virtual void draw() const {
        cout << "Drawing a shape." << endl;
    }
};

// Derived class with a specific implementation of the virtual function
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};

int main() {
    Shape* shape1 = new Shape();
    Shape* shape2 = new Circle();
    shape1->draw();    // Calls Shape::draw()
    shape2->draw();    // Calls Circle::draw()
}
```

- In this example, the `Shape` class has a virtual function `draw()`. The `Circle` class overrides this function with a specific implementation. When calling the `draw` function through a pointer to the base class, the appropriate derived class version is executed, demonstrating runtime polymorphism.

19. What is Encapsulation?

- Involves bundling the **data (attributes or properties)** and **methods (functions) that operate on the data** within a single unit, known as a class.
- The data is kept private from the outside world, and access to it is controlled through public methods (getters and setters). This helps in hiding the internal details of the implementation and protecting the integrity of the object

```
class Student {
private:
    // Private members are not directly accessible outside the class
    string name;
    float gpa;
```

```
public:
    // Public methods provide controlled access to private members
    void setName(const std::string& n) name = n;
    void setGPA(float g) {
        if (g >= 0.0 && g <= 4.0) {
            gpa = g;
        } else {
            cout << "Invalid GPA. Setting GPA to 0.0." << std::endl;
            gpa = 0.0;
        }
    }
    // Public methods to retrieve private data
    string getName() const {return name};
    float getGPA() const {return gpa};
};

int main() {
    // Create an object of the Student class
    Student student1;
    // Use public methods to set private data
    student1.setName("John Doe");
    student1.setAge(20);
    student1.setGPA(3.5);
    // Use public methods to retrieve private data
    cout << "Student Name: " << student1.getName() << endl;
    cout << "Student Age: " << student1.getAge() << endl;
    cout << "Student GPA: " << student1.getGPA() << endl;
}
```

- Here, `Student` class encapsulates the private data members `name` and `gpa`
- Public methods (`setName`, `setGPA`, `getName`, `getGPA`) provide controlled access to these private members

20. What is Abstraction?

- Allows to focus on the essential aspects of an object or system while hiding the unnecessary or irrelevant details
- Implemented using **'classes'** and **'interfaces'**
- **Abstract Class**
  - A class that cannot be instantiated on its own and is meant to serve as a base for other classes.
  - It may have abstract methods, which are declared but not implemented in the abstract class. Subclasses must provide concrete implementations for these methods.
  - Abstract classes may also have concrete methods with fully implemented functionality.

JAVA

```
abstract class Animal {
    abstract void walk();  // Abstract method
    void eat() {
        System.out.println("The animal is eating.");
    }
    // Constructor
    Animal() {
        System.out.println("An Animal is going to be created.");
    }
}
```

```
class Cow extends Animal {
    // Constructor
    Cow() {
        System.out.println("You have created a Cow");
    }
    // Implementation of the abstract method
    void walk() {
        System.out.println("Cow is walking.");
    }
}

public class OOPS {
    public static void main(String args[]) {
        // Creating objects of derived classes
        Cow cow = new Cow();
        cow.walk();
        cow.eat();
    }
}
```

C++

```
class Animal {
public:
    // Pure virtual function makes the class abstract
    virtual void walk() = 0;
    void eat() {
        cout << "The animal is eating." << endl;
    }
    Animal() {
        cout << "An Animal is going to be created." << endl;
    }
};


// Derived class Cow
class Cow : public Animal {
public:
    // Constructor
    Cow() {
        cout << "You have created a Cow" << endl;
    }
    // Implementation of the pure virtual function
    void walk() override {
        cout << "Cow is walking." << endl;
    }
};

int main() {
    // Creating objects of derived classes
    Cow cow;
    cow.walk();
    cow.eat();
}
```

- In C++, the equivalent of **Java's abstract class** and **pure virtual function** is achieved by declaring the function as pure virtual in the base class. The derived classes (Cow) provide concrete implementations for the pure virtual function walk()

- **Interfaces**
  - A unique class type.
  - An interface is **a collection of abstract methods** but not their implementation.
  - Inside an interface, only method declaration is permitted. You cannot make objects using an interface.
  - A class that implements an interface must provide concrete implementations for all the methods declared in that interface.

```
// Interface class
class Printable {
public:
    // Pure virtual function makes the class an interface
    virtual void print() const = 0;
};

// Concrete class implementing the interface
class Book : public Printable {
public:
    // Implementation of the interface method
    void print() const override {
        cout << "Printing a book." << endl;
    }
};

int main() {
    // Using objects of classes implementing the interface
    Book myBook;
    // Invoking the interface method
    myBook.print();
}
```

21. How is an **abstract class** different from an **interface**?

- Both abstract classes and interfaces are **special types of classes** that just include the declaration of the methods, not their implementation.

Abstract Class	Interface
When an abstract class is inherited, however, the subclass is not required to supply the definition of the abstract method until and unless the subclass actually uses it.	When an interface is implemented, the subclass is required to specify all of the interface's methods as well as their implementation.
A class that is abstract can have both abstract and non-abstract methods.	An interface can only have abstract methods.
Abstract class doesn't support multiple inheritance.	An interface supports multiple inheritance.

22.