Object Oriented Programming

1. 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.

2. 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.
- **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.
  - In a class, you can define constructors and destructors, allowing for explicit initialization and cleanup of class objects.

    ```
    // 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;
    }
    ```

3. 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.

4. 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.")
```

5. Diamond Problem

- Occurs when a class inherits from **two** classes that both inherit from a **common** base class
- Example:

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

  - Class A is the base class. Class B and Class C inherit from A. Class D inherits from both B and C
  - If D tries to access a method from A, it is unclear whether it comes from B or C, leading to ambiguity

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

class B : public A {};
class C : public A {};

// D inherits from both B and C
class D : public B, public C {};

int main() {
    D obj;
    obj.show();  ❌ Error: Ambiguous access to 'show' from A
    return 0;
}
```

- To **resolve** the diamond problem, we use **virtual inheritance**, which ensures only one instance of **A** exists in **D**
  - How Virtual Inheritance Solves It?
    - B and C inherit ‘A’ virtually, ensuring only **one shared instance** of ‘A’ exists
    - D now calls the only instance of A, preventing ambiguity

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

class B : virtual public A {};
class C : virtual public A {};

// D inherits from both B and C
class D : public B, public C {};

int main() {
    D obj;
    obj.show();  ✅ No ambiguity, calls A::show()
    return 0;
}
```

6. Interface

- 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();
}
```

7. 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

8.

9.