QUESTION NO 01:

A) Explain the concept of pure virtual functions , abstract base classes and interfaces. Discuss the role of abstract classes in designing class hierarchies.

Pure Virtual Functions:

A pure virtual function is a function in a base class that has no implementation in that class and is marked with the "pure virtual" specifier "= 0." This makes the base class abstract, meaning it cannot be instantiated on its own. Instead, it serves as a blueprint for derived classes to implement the pure virtual function.

Abstract Base Classes:

An abstract base class (ABC) is a class that contains one or more pure virtual functions and may also include concrete (implemented) functions. It cannot be instantiated on its own, but it can be used as a base class for derived classes. Abstract base classes are meant to provide a common interface for a group of related classes while allowing each derived class to implement its own behaviour.

Interfaces:

An interface is a programming concept that defines a contract for classes. It specifies a set of methods that a class must implement, but it does not provide any implementation details. Interfaces are used to achieve abstraction and ensure that classes that implement the interface adhere to a specific set of behaviors.

In some programming languages like Java or C#, interfaces are explicitly defined using keywords like **interface**. In C++, abstract classes with pure virtual functions can be used to achieve a similar effect.

ROLE OF ABSTRACT CLASSES IN DESIGINIG CLASS HIERARCHIES:

Abstract classes play a crucial role in designing class hierarchies by providing a common base for related classes. They allow you to define a shared interface and behavior for a group of classes while leaving certain methods to be implemented by derived classes. This promotes code reusability, extensibility, and the ability to work with a group of related objects through a common interface.

Abstract classes help in achieving abstraction, as they define what derived classes should do without specifying how they should do it. This separation of interface and implementation is a fundamental principle in object-oriented design and promotes the creation of modular, flexible, and maintainable code.

B) Select the scenario of your choice and create abstract class with at least one pure virtual function, implement concrete derived classes that inherits from the abstract and provide implementation for pure virtual functions.

```
#include <iostream>


// Abstract base class
class Shape {
public:
    // Pure virtual function for calculating area
    virtual double calculateArea() const = 0;

    // Virtual function for displaying information
    virtual void displayInfo() const {
        std::cout << "This is a shape." << std::endl;
    }

    // Virtual destructor (important when dealing with polymorphism)
    virtual ~Shape() {}
};
/ Concrete derived class: Circle
class Circle : public Shape {
private:
    double radius;

public:
    // Constructor
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function
    double calculateArea() const override {
        return 3.14159 * radius * radius;
```

```
    }

    // Override the displayInfo function
    void displayInfo() const override {
        std::cout << "This is a circle with radius " << radius << "." << std::endl;
    }
};

// Concrete derived class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    // Constructor
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function
    double calculateArea() const override {
        return length * width;
    }

    // Override the displayInfo function
    void displayInfo() const override {
        std::cout << "This is a rectangle with length " << length << " and width " << width
<< "." << std::endl;
    }
};

int main() {
```

```
    // Create instances of the derived classes
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);

    // Call the pure virtual function and displayInfo for each object
    std::cout << "Circle Area: " << circle.calculateArea() << std::endl;
    circle.displayInfo();

    std::cout << "\nRectangle Area: " << rectangle.calculateArea() << std::endl;
    rectangle.displayInfo();

    return 0;
}
```

In this example, the **Shape** class serves as an abstract base class with a pure virtual function **calculateArea()**. The **Circle** and **Rectangle** classes are concrete derived classes that provide their own implementations of the **calculateArea()** function. The **displayInfo()** function is also overridden in each class to demonstrate polymorphic behaviour.

This scenario allows for a common interface (**calculateArea()**) to be shared among different shapes while allowing each shape to calculate its area differently.

C) Explain the concepts of association , aggregation and composition in opp. Provide example for each type of relationship in real world context.

1. <u>ASSOCIATION:</u>

Association represents a bi-directional relationship between two classes where one class is related to another, but there is no strong ownership or lifecycle dependency. The classes are independent, and changes in one class do not necessarily affect the other.

<u>EXAMPLE: TEACHER AND STUDENT:</u>

```
class Teacher {
public:
   void teach() {
      // Teaching logic
   }
};

class Student {
public:
```

```
    void learn() {
        // Learning logic
    }
};

// Association
class Classroom {
public:
    void addTeacher(Teacher* t) {
        teachers.push_back(t);
    }

    void addStudent(Student* s) {
        students.push_back(s);
    }

private:
    std::vector<Teacher*> teachers;
    std::vector<Student*> students;
};
```

2) AGGREGATION:

Aggregation is a specialized form of association where one class is part of another class, but the parts can exist independently of the whole. The parts have a weaker relationship with the whole, and they can be shared among different wholes.

EXAMPLE: UNIVERSITY AND DEPARTMENTS

```
class Department {
public:
    void offerCourses() {
        // Course offering logic
    }
};

// Aggregation
class University {
public:
    void addDepartment(Department* d) {
        departments.push_back(d);
    }

private:
    std::vector<Department*> departments;
};
```

3) COMPOSITION:

Composition is a strong form of aggregation where the parts are strongly tied to the whole, and the whole is responsible for the creation, destruction, and lifetime of its parts. If the whole is destroyed, its parts are also destroyed.

EXAMPLE: CAR AND ENGINE

```
// Composition
class Car {
public:
    Car() : engine(new Engine()) {
        // Car constructor logic
    }

    ~Car() {
        delete engine;
    }

    void start() {
        engine->start();
        // Other car start logic
    }

private:
    Engine* engine;
};
```

QUESTION NO 02:

You are tasked with designing a simplified shopping cart system in C++ . consider the following requirments;

- **Product class**
- **Shopping cart class**
- **User class**
- **Association**
- **Aggregation**
- **Composition**

```
#include <iostream>

#include <vector>


// Forward declarations

class Product;

class ShoppingCart;
```

```
class User;

// Product class
class Product {
public:
    Product(std::string name, double price) : name(name), price(price) {}

    std::string getName() const {
        return name;
    }

    double getPrice() const {
        return price;
    }

private:
    std::string name;
    double price;
};

// User class
class User {
public:
    User(std::string username) : username(username) {}

    std::string getUsername() const {
        return username;
    }

private:
```

```
    std::string username;
};

// Association between User and ShoppingCart
class UserShoppingCartAssociation {
public:
    UserShoppingCartAssociation(User* user, ShoppingCart* cart) : user(user), cart(cart) {}

    User* getUser() const {
        return user;
    }

    ShoppingCart* getShoppingCart() const {
        return cart;
    }

private:
    User* user;
    ShoppingCart* cart;
};

// Aggregation between ShoppingCart and Product
class ShoppingCartAggregation {
public:
    void addProduct(Product* product) {
        products.push_back(product);
    }

    void displayCart() const {
        std::cout << "Shopping Cart Contents:\n";
```

```
        for (const auto& product : products) {
            std::cout << "- " << product->getName() << " ($" << product->getPrice() << ")\n";
        }
        std::cout << "Total: $" << calculateTotal() << "\n";
    }

private:
    double calculateTotal() const {
        double total = 0.0;
        for (const auto& product : products) {
            total += product->getPrice();
        }
        return total;
    }

    std::vector<Product*> products;
};

// Composition between User and ShoppingCart
class UserShoppingCartComposition {
public:
    UserShoppingCartComposition(User* user) : user(user), cart(new
ShoppingCartAggregation()) {}

    ~UserShoppingCartComposition() {
        delete cart;
    }

    void addToCart(Product* product) {
        cart->addProduct(product);
    }
```

```
    void viewCart() const {
        std::cout << "User: " << user->getUsername() << "\n";
        cart->displayCart();
    }

private:
    User* user;
    ShoppingCartAggregation* cart;
};

int main() {
    // Creating instances of User, Product, and UserShoppingCartComposition
    User user1("JohnDoe");
    User user2("AliceSmith");

    Product product1("Laptop", 1200.0);
    Product product2("Headphones", 80.0);

    UserShoppingCartComposition userCart1(&user1);
    UserShoppingCartComposition userCart2(&user2);

    // Adding products to shopping carts
    userCart1.addToCart(&product1);
    userCart1.addToCart(&product2);

    userCart2.addToCart(&product1);

    // Viewing shopping carts
    userCart1.viewCart();
```

```
    userCart2.viewCart();


    return 0;
}
```

In this design:

1. The **Product** class represents a product with a name and price.
2. The **User** class represents a user with a username.
3. The **UserShoppingCartAssociation** class represents an association between a user and a shopping cart.
4. The **ShoppingCartAggregation** class represents an aggregation between a shopping cart and products.
5. The **UserShoppingCartComposition** class represents a composition between a user and a shopping cart.

The **main** function demonstrates creating users, products, and adding products to the shopping carts. The association, aggregation, and composition relationships are reflected in the design as described in the comments. Note that the composition ensures that the shopping cart is deleted when the user is deleted.
