Part A: Conceptual Questions

1. **Definition**
    - Encapsulation - The grouping of data/methods together in the same construct (e.g. classes). It can prevent unintended changes to data by enabling us to hide/limit direct access to specific data through *private access modifiers.*
2. **Visibility Modifiers**
    - Public Access
        - Members are accessible from both within and outside of a class.
        - Benefit: Allows external methods to directly access these members.
        - Drawback: Reduces data safety as external methods can freely modify public members, potentially leading to unintended behavior.
    - Private Access
        - Members can only be accessed by the class itself and friend classes/methods.
        - Benefit: Enhances data safety by restricting external access and modification of members.
        - Drawback: Limits code flexibility, requiring getter and setter functions to retrieve private data when necessary.
    - Protected Access
        - Members are accessible from the class itself and its sub-classes, but not by external code.
        - Benefit: Balances data safety and flexibility by allowing sub-classes to manipulate inherited data without exposing them to outside code.
        - Drawback: Less safe than private access as derived classes can access these members, potentially leading to unintended behavior if not used carefully.
        - Protected members may intentionally be used instead of private members when implementing *inheritance* in a program, as derived classes may need to access and modify certain base class members without exposing them to external code.
3. **Impact on Maintenance**
    - Encapsulation can reduce debugging complexity when maintaining a large codebase because it isolates and protects the internal state of objects. By restricting direct access to attributes, encapsulation minimizes the risk of unintended modifications that could cause errors within the codebase.
    - If internal data is made public, code could potentially break. For example, if you had a BankAccount class where the *balance* member is made

public, outside parts of the program could modify the balance and unintentionally set it to an invalid number.

4. **Real-World Analogy**
    - Using the example of a *car,* its "public interface" would include things like the steering wheel, pedals, dashboard, and gear stick. The driver can interact with these elements without needing to know their internal workings. The car's "private implementation" includes things like the engine, brake system, and other electrical components. The driver does not directly interact with the components as they are hidden. Keeping the "private" side of a car hidden is useful because it makes the driving experience simple and intuitive. The driver does not need to worry about how the internal components of a car work to be able to drive.

Part B: Small-Class Design (Minimal Coding)

```
class BankAccount{
private:
    double balance; //balance member is private to prevent direct external modification
    int accountNum;

public:
    BankAccount(double initBal = 0.0, int accNum);
    ~BankAccount();

    //public method to deposit money into account
    //Direct modification of balance is not allowed to ensure data integrity.
    //Use this method to safely interact with private members when depositing.
    void deposit(double amount);
};
```

- The private member 'balance' should be kept private to ensure that the bank account's financial data cannot be directly accessed by outside methods that may lead to security issues.
- The private member 'accountNum' should be kept private to ensure the security of the unique account identifier while avoiding modifications that could disrupt account tracking.
- The public method 'deposit' enforces constraints by validating the input amount before modifying balance, preventing invalid operations.

Part C: Reflection & Short-Answer

1. **Pros and Cons**
    - Benefits of hiding internal data behind methods
        - Enforces data integrity and validation by preventing unwanted modifications
        - Allows for coding flexibility by allowing you to modify internal implementation without affecting external code
    - Potential limitation/overhead of hiding internal data behind methods
        - Using methods to access and modify data is slightly less efficient and more complex than direct access.
2. **Encapsulation vs. Other Concepts**
    - Encapsulation differs from abstraction as encapsulation focuses on how data is hidden and accessed within a class, whereas abstraction focuses on hiding implementation and only exposing essential functionality.
    - Both encapsulation and abstraction are considered forms of “information hiding”. Encapsulation hides the internal state of an object and its data by restricting access, while abstraction hides implementation complexity.
3. **Testing Encapsulated Classes**
    - To unit test a class with private data members thoroughly without exposing private data, we can focus on testing *public methods* that interact with the private data. By declaring a getter function or display function that corresponds with the private data members, we can test this method by providing different values to ensure that the private members are being accessed correctly.