**Part A: Conceptual Questions**

1. Definition
    - Encapsulation is the principle of bundling data (attributes) and methods (functions) that operate on the data within a single unit (class) while restricting direct access to some of the object's components. This ensures that the internal state of an object is hidden from the outside world and can only be altered in controlled ways.
    - Example: A BankAccount class with a private balance ensures that users cannot directly set a negative balance. Instead, they must use methods like deposit or withdraw, which validate inputs before modifying the data.

2. Visibility Modifiers
    - Public:
        - Benefit: Allows unrestricted access, making it easy to create utility methods or interact with objects.
        - Drawback: No safeguards—external code can potentially misuse or corrupt the data.
    - Private:
        - Benefit: Completely hides the data, preventing unintended access or modification.
        - Drawback: Reduced flexibility, as access is restricted to within the class.
    - Protected:
        - Benefit: Allows derived classes to access members while keeping them hidden from external code.
        - Drawback: Still exposes internal details to subclasses, which might misuse them.
    - Scenario for Protected Members:
        - Use protected members when designing a base class meant for inheritance, where subclasses may need controlled access to certain internal data. For example, in a Shape class, protected member variables like width and height can be directly accessed in derived classes like Rectangle or Triangle.

3. Impact on Maintenance
    - Encapsulation reduces debugging complexity by ensuring that only well-defined methods can modify internal data. This creates predictable behavior and avoids side effects caused by unintended interactions.

- Example: If balance in a BankAccount is public, any external code could change it arbitrarily, potentially leading to inconsistencies (a negative balance). Debugging such issues in a large system would be time-consuming. Encapsulation ensures that modifications go through controlled methods, reducing such risks.

4. Real-World Analogy
    - Consider a car as an example:
        - Public Interface: The steering wheel, pedals, and gear shift that a driver uses to control the car.
        - Private Implementation: The engine, transmission, and internal electronics that the driver doesn't need to see or access. Hiding the private side ensures that the driver doesn't accidentally interfere with complex systems, making the car safer and easier to use.

**Part B: Small-Class Design**

1. Class Skeleton

```
class BankAccount {
private:
   double balance;
   int accountNumber;

public:
   BankAccount(int accNum, double initialBalance) : accountNumber(accNum),
balance(initialBalance) {}

   void deposit(double amount) {
      if (amount > 0) {
         balance += amount;
      }
   }

   bool withdraw(double amount) {
      if (amount > 0 && amount <= balance) {
         balance -= amount;
         return true;
      }
      return false;
```

```
    }

    double getBalance() const {
        return balance;
    }
};
```

2. Encapsulation Justification
    - Private Data Members:
        - balance: Prevents direct modification, ensuring only validated transactions (deposit/withdraw) alter the balance.
        - accountNumber: Prevents accidental changes to a critical identifier, maintaining consistency.
    - Public Methods:
        - deposit: Validates that the deposit amount is positive before adding it to the balance.
        - withdraw: Ensures the requested amount is valid and doesn't exceed the current balance.

3. Documentation

```
// Represents a bank account with controlled access to the balance.
// Direct modification of balance is prohibited to ensure data integrity.
class BankAccount {
    // Private members: Do not access directly!
    double balance;  // Tracks the account's current balance.
    int accountNumber; // Unique identifier for the account.

public:
    // Initializes the account with a unique number and starting balance.
    BankAccount(int accNum, double initialBalance);

    // Adds money to the account. Amount must be positive.
    void deposit(double amount);

    // Withdraws money if sufficient balance exists. Returns success status.
    bool withdraw(double amount);

    // Retrieves the current balance.
    double getBalance() const;
```

};

**Part C: Reflection & Short-Answer**

1. Pros and Cons of Hiding Data
    - Benefits:
        - Enforces constraints, ensuring data is always valid.
        - Simplifies debugging by limiting the scope of potential issues.
    - Limitation:
        - Introduces overhead since methods must be written to access or modify data.

2. Encapsulation vs. Abstraction
    - Encapsulation focuses on restricting access to internal details and controlling data manipulation, while abstraction hides implementation complexity by exposing only essential features.
    - Both are forms of information hiding, but abstraction deals more with what an object does, while encapsulation focuses on how it protects its data.

3. Testing Encapsulated Classes
    - Private data can be tested indirectly through public methods.
    - Strategy: Test valid and invalid deposit and withdraw operations, then use getBalance to verify that the internal state is as expected after each operation. This ensures thorough testing without exposing private data.