Turram Toussaint

2143 OOP

Prof Griffin

**Part A: Conceptual Questions**

**Definition** Encapsulation is the principle of restricting direct access to an object's data and exposing controlled ways to interact with it. This basically prevents unintended modifications and ensures data security.

Example: In a banking application, if the balance of an account is a public variable, any part of the program could alter it incorrectly. By encapsulating it within a class and providing controlled access methods, we prevent unintended changes and enforce business rules.

**Visibility Modifiers**

| **Modifier** | **Benefit** | **Drawback** |
| --- | --- | --- |
| Public: | Provides easy access to class members, enhancing flexibility. | Increases risk of accidental or malicious modifications, leading to inconsistent states. |
| Private: | Ensures strict control over data modification, improving security. | Reduces flexibility, making inheritance and extensions more challenging. |
| Protected: | Allows controlled inheritance, enabling child classes to access necessary members. | Still exposes data to subclasses, which might lead to unintended changes if not properly handled. |

A Scenario for using protected: In an object-oriented framework where a base class represents a generic Employee, and derived classes (Manager, Engineer) need access to an ID but should not modify it arbitrarily, making it protected allows controlled access.

**Impact on Maintenance** Encapsulation simplifies debugging by isolating where and how data is modified. If a bug occurs due to incorrect balance calculations, restricting direct access ensures that only deposit and withdrawal methods need to be checked, rather than searching through the entire codebase for unintentional modifications.

An Example of breaking code: If an employee's salary is a public variable, any part of the application could alter it. A careless modification might set it to a negative number, causing unexpected errors in payroll calculations.

**Real-World Analogy** A car's dashboard provides a public interface (e.g., steering wheel, pedals, buttons), while the engine and internal mechanics remain hidden. This ensures users operate the vehicle safely without needing to manipulate the internal engine components. Keeping the private side hidden prevents misuse and accidental damage.

**Part B: Small-Class Design (Minimal Coding)**

class BankAccount {

private:

double balance;

std::string accountNumber;

public:

BankAccount(std::string 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;

}

};

**Encapsulation Justification**

* Private balance: Prevents direct modification, ensuring transactions follow controlled deposit and withdrawal processes.
* Private accountNumber: Protects account details from unauthorized changes, maintaining security.
* Public deposit(): Ensures only positive values are added to the balance.
* Public withdraw(): Prevents overdrafts by checking if sufficient funds are available before deducting money.

**Documentation**

\* @class BankAccount

\* @brief Represents a bank account with encapsulated balance and account number.

\* @details

\* - Use deposit() to add funds.

\* - Use withdraw() to remove funds.

\* - Direct access to balance is restricted to maintain data integrity.

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

**Pros and Cons**

Benefits of hiding internal data:

1. Prevents unintended modifications, ensuring data consistency.
2. Enables controlled access, allowing validation and security measures.

Limitation:

* Adds some performance overhead due to the extra method calls and validation logic.

**Encapsulation vs. Abstraction** Encapsulation focuses mainly on restricting access to an object’s internal state. Abstraction, on the other hand, simplifies complexity by exposing only relevant functionalities while hiding the implementation details.

Encapsulation and abstraction as information hiding: Both prevent unnecessary exposure of details. Encapsulation restricts access to internal data, while abstraction hides implementation complexity from the user, providing only essential features.

**Testing Encapsulated Classes** Even with private data, unit testing can be done by:

1. **Public Methods Testing:** Verify deposit and withdraw functions operate correctly.
2. **Getter Methods (if applicable):** If necessary, provide controlled read-only access.
3. **Friend Functions (C++ specific):** Allow specific test classes to access private members.
4. **Mocking & Dependency Injection:** Simulate class behavior to test encapsulation indirectly.