Part A: Conceptual Questions

1. Definition of Abstraction
    - Abstraction is the process of hiding the implementation details of an object and exposing only the essential details to the user.
    - Real-world analogy: Think of driving a car. You only need to know how to operate the steering wheel, accelerator, and brake. The internal workings of the engine, transmission, and electrical systems are abstracted away.

2. Abstraction vs. Encapsulation
    - Both abstraction and encapsulation involve hiding private or protected details of an object or class from the user.
    - Someone may confuse the two concepts because both involve hiding details, but abstraction focuses on exposing only the relevant behaviors and attributes while hiding the internal workings, while encapsulation focuses on bundling data and methods together and controlling access to them via access modifiers (private, protected, public).

3. Designing with Abstraction
    - Essential attributes of a smart thermostat:
        - Current temperature
        - Desired temperature
        - Power status (on/off)
    - Essential methods:
        - setTemperature()
        - getTemperature()
    - Why omit details: Internal details like circuit design or firmware routines are irrelevant to the user—they only interact with the interface to control the thermostat.

4. Benefits of Abstraction
    - Simplifies complex systems by focusing only on relevant features.
    - Reduces dependency on specific implementations, improving maintainability and scalability.
    - Reducing code complexity: Abstraction allows developers to work with a clear and concise interface, avoiding the clutter of implementation details.

Part B: Minimal Class Example (Pseudo-code)

```
#include <iostream>
using namespace std;

// Abstract class
class BankAccount {
```

```
public:
    virtual void deposit(double amount) = 0;  // Abstract method
    virtual void withdraw(double amount) = 0; // Abstract method
    virtual ~BankAccount() {} // Virtual destructor for cleanup
};

// Derived class
class SavingsAccount : public BankAccount {
private:
    double balance;

    // Internal mechanism
    void logTransaction(const string& type, double amount) {
        // Logging details hidden from the user
        cout << "Transaction: " << type << ", Amount: " << amount << endl;
    }

public:
    SavingsAccount() : balance(0) {}

    void deposit(double amount) override {
        balance += amount;
        logTransaction("Deposit", amount);
    }

    void withdraw(double amount) override {
        if (amount <= balance) {
            balance -= amount;
            logTransaction("Withdraw", amount);
        } else {
            cout << "Insufficient funds!" << endl;
        }
    }

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

Part C: Reflection & Comparison

1. Distilling the Essentials
    - Hidden data/methods: Data like balance is private to ensure it can only be accessed or modified via controlled methods (deposit, withdraw). Internal

functions like logTransaction are hidden to keep the public interface clean and concise.
    - Explanation: To maintain security and enforce a clear separation between the user interface and internal logic.

2. Contrast with Polymorphism
    - The method calls (deposit and withdraw) on a BankAccount pointer invoke the SavingsAccount implementations. This highlights both abstraction (hiding mechanics) and polymorphism (runtime method resolution).

3. Real-World Example
    - Gaming domain: Abstraction is crucial for designing game engines. For instance, a GameCharacter abstract class may expose methods like move() and attack() without detailing the specific implementation for each character type (Knight, Archer, Wizard).