Part A: Conceptual Questions

1. What is abstraction in OOP?

- Abstraction means showing only the important things and hiding the complicated stuff. In OOP, it helps us focus on what an object does, instead of how it does it. It keeps the program simple and easy to work with.

Real-world example:
When we use a TV remote, we just press buttons to change the channel or volume. We don’t see the electronics or signals behind it. That's an abstraction — we get to use the remote without knowing what’s happening inside.

2. Abstraction vs. Encapsulation

- Abstraction hides complexity by showing only the necessary parts.
- Encapsulation hides internal data and keeps it safe from being accessed directly.

Why it’s confusing:

- Both hide things from the user, but for different reasons. Abstraction hides unneeded details, and encapsulation hides data to protect it.

3. Designing with Abstraction – Smart Thermostat

Attributes:

1. currentTemperature
2. targetTemperature
3. mode (cooling/heating)

Methods:

1. setTargetTemperature()
2. switchMode()

Why hide things:

- We wouldn't show how the sensors work or how the firmware processes temperature. Those are behind the scenes. The user just wants to set the temp and let the system do the rest.

4. Benefits of Abstraction

1. Makes big systems easier to understand and manage.
2. You can change internal code without messing up how people use the class.

Short answer:
Abstraction reduces complexity by keeping only what matters visible and hiding everything else.

Part B: Minimal Class Example (C++)

Scenario: Banking System

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

// Abstract class
class BankAccount {
public:
    virtual void deposit(double amount) = 0;
    virtual void withdraw(double amount) = 0;
    virtual ~BankAccount() {}
};

// Derived class
class SavingsAccount : public BankAccount {
private:
```

```
    double balance;

public:
    SavingsAccount() : balance(0.0) {}

    void deposit(double amount) override {
        // pretend logging, encryption, etc. happens here
        balance += amount;
        cout << "Deposited $" << amount << endl;
    }

    void withdraw(double amount) override {
        if (amount <= balance) {
            balance -= amount;
            cout << "Withdrew $" << amount << endl;
        } else {
            cout << "Not enough balance." << endl;
        }
    }
};
```

Quick explanation:

- The BankAccount class is abstract and only defines what we can do (deposit and withdraw). The SavingsAccount class gives the actual logic but hides any complex stuff like encryption or transaction logs.

Part C: Reflection & Comparison

1. What would I hide?

- I'd hide the balance variable and any private helper methods like logging or verifying transactions. The user should only see what they actually need — deposit() and withdraw().

2. Abstraction + Polymorphism

- If BankAccount is abstract, and I create a SavingsAccount object, calling withdraw() using a BankAccount* The pointer shows polymorphism. But since the base class only shows what's necessary and hides the rest, it also shows abstraction. So they work together.

3. Real-world domain

- In healthcare, abstraction is super helpful. Doctors and nurses use software to check records or schedule appointments, but they don't see how the data is stored or how servers handle it. That complexity is hidden, and only the needed tools are shown.

Additional Exploration

Interfaces vs. Abstract Classes

- Interfaces only have method names (no body), while abstract classes can have both abstract and regular methods.
- Interfaces are great when you want different classes to follow the same rules, but they're not related — like Flyable or Drivable.

Testing abstractions

- To test an abstract class, you can create a fake child class that implements the methods in a simple way. Then you test those methods to make sure they behave the way they're supposed to.