Part A: Conceptual Questions

1. **Definition**
    - Abstraction is the concept of hiding complex implementation details and showing the user the necessary functionalities only.
    - Analogy: A driver does not need to know the complex inner workings of a car to be able to accelerate in a car. All the driver needs to do is interact with the pedal to accelerate. The complex systems that control acceleration (like the engine and transmission) are hidden within the car.
2. **Abstraction vs. Encapsulation**
    - Encapsulation focuses on how data is hidden and accessed within a class, whereas abstraction focuses on hiding implementation and only exposing essential functionality.
    - Some may get abstraction and encapsulation confused with each other because both concepts are forms of information hiding.
3. **Designing with Abstraction**
    - When designing a *smart thermostat,* three essential attributes to display would be the current temperature, target temperature, and mode(heating, cooling, auto). Two essential methods for this thermostat would be *Set Temperature* and *Switch Mode*.
    - Details like the internal circuit design and firmware design would be omitted/hidden because the user does not need to directly interact with these complex components.
4. **Benefits of Abstraction**
    - Two benefits of abstraction are *enhanced maintainability*, allowing developers to modify internal implementation without affecting the rest of a system, and *improved collaboration,* as developers can work on different components separately while sharing abstract interfaces.
    - Abstraction reduces code complexity by hiding complex details and only showing necessary features.

Part B: Minimal Demonstration

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

//Derived class
class SavingsAccount : public BankAccount {
public:
   void deposit(double amount){
      cout << "Deposited: $" << amount << endl;
   }

   void withdraw(double amount){
      cout << "Withdrew: $" << amount << endl;
   }
};
```

Part C: Reflection & Comparison

1. **Distilling the Essentials**
   - In the SavingsAccount class, internal data such as *balance* and private methods like *transaction logging* would be hidden from direct user interaction to maintain a clear public interface.
   - This ensures *data integrity,* as users cannot modify sensitive attributes directly, preventing unintended errors.
2. **Contrast with Polymorphism**
   - When a reference to the base class *BankAccount* calls a method like *withdraw,* the SavingsAccount implementation of it will be used at runtime, highlighting polymorphism.
   - This also highlights abstraction as the user can withdraw money how they want without worrying about how it is done.
3. **Real-World Example**
   - In gaming, abstraction is crucial for simpler API design when dealing with game physics and objects. By using abstraction, developers can define things like an object's position while hidden physics algorithms will control how the object interacts with the game world.