PART A

1. Definition:In your own words, what is abstraction in OOP?Provide a short real-world analogy illustrating how abstraction hides unnecessary details.

Abstraction in Object Oriented Programming (OOP) refers to the concept of simplifying complex systems by focusing only on the essential features and hiding unnecessary details.

Abstraction in OOP simplifies complex systems by exposing only essential functionalities and hiding the internal details. In the case of a Vehicle class, we interact with common methods like start_engine() or drive(), but we don't need to know how each vehicle implements them. The specifics are hidden in subclasses like Car or Bike, which implement those details.

2. Abstraction vs. Encapsulation: Briefly compare abstraction and encapsulation. Why might someone confuse the two concepts?

Abstraction hides complexity by exposing only essential features (what an object does), while encapsulation hides internal data and restricts direct access, providing controlled interaction through methods (how an object works).

They are often confused because both involve hiding details, but abstraction simplifies interfaces, while encapsulation focuses on protecting and managing data.

3. Designing with Abstraction:Imagine you are modeling a smart thermostat in a home automation system.List three attributes and two methods you consider essential to the thermostat (from the user's or system's perspective).Explain why you would omit certain internal details (e.g., circuit design, firmware routines).

Essential Attributes:

1.Current Temperature

2.Set Temperature

3.Mode

Essential Methods:

SetTemperature()

SwitchMode()

Details like circuit design or firmware routines are hidden because the user only cares about setting the temperature and mode, not the technical workings behind them. This simplifies the user experience and focuses on what's essential.

4. Benefits of Abstraction:Name two benefits of abstraction in large-scale software projects.In a short sentence, how can abstraction reduce code complexity?

Simplified Maintenance: Changes can be made internally without affecting other parts of the system.

Improved Reusability: Common interfaces can be reused across different components.

Abstraction simplifies code by hiding unnecessary details and providing a clear interface, making the code easier to understand and manage.

PART B

1. Model a Banking System where you only want certain core operations exposed (like deposit and withdraw), hiding internal complexities (like encryption, logging, or ledger balancing).The BankAccount could be an abstract class, specifying only the interface methods (no concrete details).The user interacts with derived classes (like SavingsAccount), which implement the abstract methods but hide the internal mechanics.

```
// Abstract base class defining the interface
class BankAccount {
public:
    virtual void deposit(double amount) = 0;
    virtual void withdraw(double amount) = 0;
    virtual double getBalance() const = 0;
    virtual ~BankAccount() {}
};

// Derived class implementing the abstract methods
class SavingsAccount : public BankAccount {
private:
    double balance;

    // Internal methods hidden from user
    void logTransaction(const string& details) {}
    void encryptData() {}

public:
    SavingsAccount(double initialBalance) : balance(initialBalance) {}

    void deposit(double amount) override {

    }

    void withdraw(double amount) override {

        }
    }

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

int main() {

}
```

PART C

1. Distilling the Essentials:In your SavingsAccount class, which data or methods would you hide from direct user calls to maintain a clear public interface?Provide a brief explanation.
2. Contrast with Polymorphism:If BankAccount is abstract, how does a method call on SavingsAccount highlight polymorphism while also showcasing abstraction?
3. Real-World Example:Briefly name another real-world domain (e.g., gaming or healthcare) where abstraction is crucial for a simpler API design.

In the SavingsAccount class, internal methods like logTransaction() and encryptData(), as well as the balance, should be hidden to maintain a clear and simple public interface. This keeps the user interaction focused only on highlevel operations such as deposit(), withdraw(), and getBalance(), without delving into implementation details.

Contrast with Polymorphism: Polymorphism allows a method call like deposit() to invoke the correct implementation, even if the object is of type BankAccount, but points to a SavingsAccount. This showcases abstraction by allowing interaction with the general BankAccount interface, while the underlying SavingsAccount provides the specific behavior.

In gaming, abstraction is crucial for simplifying interactions with complex game engines. For example, a game might expose highlevel methods like moveCharacter(), attack(), or jump(), while hiding the underlying complexities of physics calculations, collision detection, and rendering. This allows game developers to work with simpler, more intuitive interfaces without worrying about the complex details.