PART A

1. Definition

Abstraction in object-oriented programming (OOP) is the concept of concealing complex implementation details and displaying only the most important properties of an object to the user. Real world analogy: When driving a car, you use the steering wheel and pedals without needing to understand the engine or transmission- abstraction hides the complexity so you can focus on driving.

2. Abstraction vs. Encapsulation

Abstraction hides what is done, focusing on relevant features, while Encapsulation hides how it is accomplished by combining data and methods into a class and restricting access. Someone may be confused since both involve hiding details, yet abstraction simplifies interfaces, whereas encapsulation ensures data integrity.

3. Designing with Abstraction

Smart Thermostat

Attributes: desired temperature, current temperature and operating mode.
Methods: set desired temperature, switch mode.
Explanation: We exclude internal features like sensor calibration or firmware routines since users and mist system components do not need them. They only interface with high-level functions, making the design simpler and more focused.

4. Benefits of abstraction
    - Promotes code reuse and modularity
    - Easier maintenance and updates

Abstraction decreases code complexity by allowing developers to work with simplified models without having to worry about low-level details.

PART B

Abstract Class: BankAccount

```
#include <iostream>
using namespace std;
// Abstract class defining only 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: SavingsAccount

```
class SavingsAccount : public BankAccount {
private:
    double balance;

    // Internal (hidden) methods
    void encryptTransaction(double amount) {
        cout << "[Encrypting transaction of " << amount << "]" << endl;
    }

    void logTransaction(const string& type, double amount) {
        cout << "[Logging " << type << " of " << amount << "]" << endl;
    }

    void updateLedger() {
        cout << "[Ledger balanced]" << endl;
    }

public:
    SavingsAccount(double initialBalance = 0.0) : balance(initialBalance) {}

    // Public method exposed to user
    void deposit(double amount) override {
        if (amount > 0) {
            encryptTransaction(amount);
            balance += amount;
            logTransaction("deposit", amount);
            updateLedger();
        } else {
            cout << "Invalid deposit amount.\n";
        }
    }
    // Public method exposed to user
    void withdraw(double amount) override {
        if (amount > 0 && amount <= balance) {
            encryptTransaction(amount);
            balance -= amount;
            logTransaction("withdraw", amount);
            updateLedger();
        } else {
```

```
            cout << "Invalid withdrawal.\n";
        }
    }
    double getBalance() const override {
        return balance;
    }
};
```

Driver code

```
int main() {
    BankAccount* account = new SavingsAccount(1000.0);

    account->deposit(200.0);    // Exposed
    account->withdraw(150.0);   // Exposed

    cout << "Current balance: $" << account->getBalance() << endl;

    delete account;
    return 0;
}
```

PART C

1. Distilling the essentials

   In the SavingsAccount class, this should be hidden from the direc user access:

   - Private data: balance
   - Internal methods: encryptTransaction(), logTransaction(), updateLedger()

   Explanation:

   The internal workings and security of the account include these internal procedures and information. By exposing them, encapsulation would be broken and possible security or consistency problems could arise due to misuse or confusion. Users just require access to getBalance, deposit, and withdraw.

2. Constract with polymorphism

If BankAccount is an abstract class and SavingsAccount is a subclass that implements or overrides its methods, then using withdraw() on a BankAccount reference that links to a SavingsAccount object exemplifies both polymorphism and abstraction.

The user or calling code interacts with the abstract type (BankAccount) without knowing or needing to understand the precise implementation details in SavingsAccount. This hides complexity and focuses solely on the fundamental behaviors defined in the abstract class. When the function withdraw() is invoked on a BankAccount reference that refers to a SavingsAccount object, runtime polymorphism guarantees that the SavingsAccount version of withdraw() is

executed. This enables distinct account types (e.g., SavingsAccount, CheckingAccount) to define their unique behaviors while yet being handled consistently as BankAccount.

3. Real-world example

Abstraction is critical in healthcare, as simpler APIs can provide patient records, appointments,and billing information without revealing underlying data storage, encryption, or compliance methods such as HIPAA.