# Final Project Phase 3

### Linked Lists and Special Classes

1. CustomerNode and CustomerList:

struct CustomerNode {

unique\_ptr<BankAccount> account;

CustomerNode\* next;

CustomerNode(unique\_ptr<BankAccount> acc)

: account(move(acc)), next(nullptr) {}

};

class CustomerList {

private:

CustomerNode\* head;

public:

CustomerList() : head(nullptr) {}

~CustomerList() {

while (head) {

CustomerNode\* temp = head;

head = head->next;

delete temp;

}

}

void addCustomer(unique\_ptr<BankAccount> account) {

CustomerNode\* newNode = new CustomerNode(move(account));

newNode->next = head;

head = newNode;

}

BankAccount\* getCustomerByName(const string& name) {

CustomerNode\* curr = head;

while (curr) {

if (curr->account->getOwner() == name)

return curr->account.get();

curr = curr->next;

}

return nullptr;

}

void displayAll() const {

CustomerNode\* curr = head;

while (curr) {

curr->account->display();

curr = curr->next;

}

}

};

1. Special Classes for Banking Functionalities - InterestCalculator Class:

Encapsulates logic for interest calculations, keeping SavingsAccount clean.

a. class InterestCalculator {

public:

static double calculateInterest(double balance, double rate) {

return balance \* (rate / 100.0);

}

};

b. void applyInterest() {

double interest = InterestCalculator::calculateInterest(balance, interestRate);

deposit(interest); // reuse existing deposit function

addTransaction("Interest Applied: $" + formatAmount(interest));

}

1. OverdraftProtection Helper -

a. Encapsulates overdraft logic:

class OverdraftProtection {

public:

static bool canWithdraw(double balance, double overdraft, double amount) {

return amount <= (balance + overdraft);

}

};

b. Modify CheckingAccount::withdraw() to use this:

void withdraw(double amount) override {

if (!OverdraftProtection::canWithdraw(balance, overdraftLimit, amount))

throw runtime\_error("Overdraft limit exceeded");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

#include <iostream>

#include <vector>

#include <list>

#include <memory>

#include <stdexcept>

#include <iomanip>

#include <sstream>

using namespace std;

/\*\*

\* Utility class to handle interest calculation for savings accounts.

\*/

// InterestCalculator class

class InterestCalculator {

public:

static double calculateInterest(double balance, double rate) {

return balance \* (rate / 100.0);

}

};

/\*\*

\* Utility class to enforce overdraft rules in checking accounts.

\*/

// OverdraftProtection class

class OverdraftProtection {

public:

static bool canWithdraw(double balance, double overdraft, double amount) {

return amount <= (balance + overdraft);

}

};

/\*\*

\* Base class representing a generic bank account.

\* Implements polymorphic behavior through virtual functions.

\* Provides common interface for all account types.

\*/

// Base class for bank accounts

class BankAccount {

protected:

string owner;

double balance;

list<string> transactionHistory;

// Helper method to format currency with two decimal places

string formatAmount(double amount) const {

ostringstream stream;

stream << fixed << setprecision(2) << amount;

return stream.str();

}

public:

BankAccount(string name, double initialBalance) : owner(name), balance(initialBalance) {}

virtual ~BankAccount() = default; // Virtual destructor for proper cleanup of derived classes

// Pure virtual methods to be implemented by derived classes

virtual void deposit(double amount) = 0;

virtual void withdraw(double amount) = 0;

virtual void display() const = 0;

// Records a transaction in the account's history

void addTransaction(const string& transaction) {

transactionHistory.push\_back(transaction);

}

// Displays all past transactions for this account

void displayTransactionHistory() const {

cout << "Transaction History for " << owner << ":\n";

for (const auto& transaction : transactionHistory) {

cout << transaction << endl;

}

}

// Accessors

double getBalance() const { return balance; }

string getOwner() const { return owner; }

};

/\*\*

\* Derived class representing a savings account.

\* Inherits from BankAccount and provides specific functionality

\* including interest calculation and standard deposit/withdrawal.

\*/

// Derived class for savings accounts

class SavingsAccount : public BankAccount {

private:

double interestRate;

public:

SavingsAccount(string name, double balance, double rate)

: BankAccount(name, balance), interestRate(rate) {

}

// Adds funds to the account

void deposit(double amount) override {

balance += amount;

addTransaction("Deposited: $" + formatAmount(amount));

}

// Withdraws funds from the account if sufficient balance exists

void withdraw(double amount) override {

if (amount > balance)

throw runtime\_error("Insufficient funds");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

// Applies interest to the account using InterestCalculator

void applyInterest() {

double interest = InterestCalculator::calculateInterest(balance, interestRate);

deposit(interest);

addTransaction("Interest Applied: $" + formatAmount(interest));

}

// Displays account information

void display() const override {

cout << "Savings Account: " << owner << " | Balance: $" << fixed << setprecision(2) << balance

<< " | Interest Rate: " << interestRate << "%\n";

}

};

/\*\*

\* Derived class representing a checking account.

\* Includes overdraft protection logic and basic deposit/withdraw functionality.

\*/

// Derived class for checking accounts

class CheckingAccount : public BankAccount {

private:

double overdraftLimit;

public:

CheckingAccount(string name, double balance, double overdraft)

: BankAccount(name, balance), overdraftLimit(overdraft) {

}

// Adds funds to the account

void deposit(double amount) override {

balance += amount;

addTransaction("Deposited: $" + formatAmount(amount));

}

// Withdraws funds within allowed overdraft limit

void withdraw(double amount) override {

if (!OverdraftProtection::canWithdraw(balance, overdraftLimit, amount))

throw runtime\_error("Overdraft limit exceeded");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

// Displays account information

void display() const override {

cout << "Checking Account: " << owner << " | Balance: $" << fixed << setprecision(2) << balance

<< " | Overdraft Limit: $" << overdraftLimit << "\n";

}

};

/\*\*

\* Struct representing a node in the customer linked list.

\* Stores a unique bank account pointer and a link to the next node.

\*/

// Linked list node for customer accounts

struct CustomerNode {

unique\_ptr<BankAccount> account;

CustomerNode\* next;

CustomerNode(unique\_ptr<BankAccount> acc)

: account(move(acc)), next(nullptr) {

}

};

/\*\*

\* Manages a singly linked list of bank accounts (customers).

\* Supports add, delete, search, and display operations.

\*/

// Linked list management for customer accounts

class CustomerList {

private:

CustomerNode\* head;

public:

CustomerList() : head(nullptr) {}

~CustomerList() {

while (head) {

CustomerNode\* temp = head;

head = head->next;

delete temp;

}

}

// Adds a new customer node to the front of the list

void addCustomer(unique\_ptr<BankAccount> account) {

CustomerNode\* newNode = new CustomerNode(move(account));

newNode->next = head;

head = newNode;

}

// Deletes a customer by name; returns true if found and removed

bool deleteCustomer(const string& name) {

CustomerNode\* curr = head;

CustomerNode\* prev = nullptr;

while (curr) {

if (curr->account->getOwner() == name) {

if (prev) prev->next = curr->next;

else head = curr->next;

delete curr;

return true;

}

prev = curr;

curr = curr->next;

}

return false;

}

// Searches for a customer by name and returns a pointer to their account

BankAccount\* getCustomerByName(const string& name) {

CustomerNode\* curr = head;

while (curr) {

if (curr->account->getOwner() == name)

return curr->account.get();

curr = curr->next;

}

return nullptr;

}

// Displays all accounts in the list

void displayAll() const {

CustomerNode\* curr = head;

while (curr) {

curr->account->display();

curr = curr->next;

}

}

};

/\*\*

\* Interface for interacting with customer accounts.

\* Supports various banking operations using runtime polymorphism.

\*/

// Banking operations

void performBankingOperations(CustomerList& customers) {

string name;

char choice;

double amount;

while (true) {

cout << "\nEnter account owner name (or 'exit' to quit): ";

cin >> name;

if (name == "exit") break;

BankAccount\* account = customers.getCustomerByName(name);

if (!account) {

cout << "Account not found.\n";

continue;

}

cout << "\nChoose operation: \nD - Deposit\nW - Withdraw\nS - Show Account\nH - Show Transaction History\nI - Apply Interest (Savings Only)\nE - Exit\nChoice: ";

cin >> choice;

switch (choice) {

case 'D': case 'd':

cout << "Enter deposit amount: ";

cin >> amount;

account->deposit(amount);

cout << "Deposit successful.\n";

break;

case 'W': case 'w':

cout << "Enter withdrawal amount: ";

cin >> amount;

try {

account->withdraw(amount);

cout << "Withdrawal successful.\n";

}

catch (const exception& e) {

cout << "Error: " << e.what() << endl;

}

break;

case 'S': case 's':

account->display();

break;

case 'H': case 'h':

account->displayTransactionHistory();

break;

case 'I': case 'i': {

SavingsAccount\* savings = dynamic\_cast<SavingsAccount\*>(account);

if (savings) {

savings->applyInterest();

cout << "Interest applied.\n";

}

else {

cout << "Interest can only be applied to savings accounts.\n";

}

break;

}

case 'E': case 'e':

return;

default:

cout << "Invalid choice.\n";

}

}

}

/\*\*

\* Entry point: Initializes customers and starts the banking interface.

\*/

int main() {

CustomerList customers;

customers.addCustomer(make\_unique<SavingsAccount>("Laurie", 5000, 2.5));

customers.addCustomer(make\_unique<CheckingAccount>("Larry", 1000, 500));

customers.addCustomer(make\_unique<SavingsAccount>("David", 10000, 2.5));

customers.addCustomer(make\_unique<CheckingAccount>("Luis", 2000, 500));

performBankingOperations(customers);

return 0;

}