Banking System Simulator Using C++ Data Structures

Bautista, Mariella
Cabrera, Gabriel
Pizarro, Jeus
Quioyo, Angelo

Technological Institute of the Philippines

TABLE OF CONTENTS

THE PROBLEM

In the financial services industry, particularly within retail banking, there is a significant gap between theoretical knowledge and practical, real-world application (Zambianchi & Genovese, 2024). New employees, such as bank tellers and branch managers, are required to master a wide array of complex procedures, adhere to strict security policies, and operate proprietary software systems where a single mistake can lead to financial loss and reputational damage (IvyPanda, 2023). Traditional training methods, which often rely on passive observation, mentorship, and manual-based learning, can be inefficient and fail to provide a safe, repeatable environment for trainees to build confidence and competence, especially when handling irregular situations or critical errors (Derrico, 2014; Penceo eLearning Provider, 2024). This creates a need for a controlled, interactive environment where trainees can practice their skills without exposing the institution or its customers to any real-world risk.

This project addresses several specific challenges inherent in the training and operational management of a banking environment. Firstly, there is a lack of a safe, sandboxed environment for tellers to practice core transaction workflows. A trainee needs to repeatedly perform deposits, withdrawals, and transfers to build muscle memory, but doing so on a live system is not feasible (IvyPanda, 2023). Secondly, teaching effective error handling is difficult. A trainee must learn how to respond to common but critical issues such as insufficient funds, invalid destination accounts during a transfer, or attempts to violate bank policy, like transferring funds to the same account. Without a system that can reliably and safely generate these errors on demand, training for these edge cases becomes purely theoretical (Hurix Digital, 2024). Lastly, for management trainees, gaining a holistic understanding of branch operations, like account lifecycle management and auditing, is often a fragmented process. They need a tool that provides a clear, top-down view of all account activities and demonstrates the importance of maintaining a complete and immutable audit trail, a feature that is often obscured in complex, production-level systems (Burns, 2023).

The Banking System Simulator is designed to directly solve these problems by serving as a high-fidelity training and simulation platform. It provides a fully functional, self-contained banking environment where all operations, from customer transactions to administrative actions, are logged and persisted without real-world financial impact. For tellers, it offers a command-line interface to practice an unlimited number of transactions and receive immediate, specific feedback when they encounter one of the system's many built-in validation errors. For managers, the simulator provides an administrative backend with tools to manage the entire account lifecycle and, most critically, to view a comprehensive, human-readable transaction log that serves as a perfect model for teaching auditing principles. By simulating core banking logic, data persistence, and a clear distinction between customer and administrative roles, the program provides a robust, hands-on solution for bridging the gap between theory and practice in a banking context (Cohen & Heames, n.d.; Shivam, 2025).

OBJECTIVES

The overall objective of this project is to create and execute a working Bank Account Management System that operates through the console, ensuring the system is very safe in handling client accounts and very accurate in keeping comprehensive records of all financial transactions. This system is designed to serve as both a functional prototype of a real world banking application and an effective training tool by demonstrating the practical application of data structures in C++ programming language.

Specifically, the project aims to:

1. Provide users with the ability to execute the principal activities of a financial institution. This includes creating a functional transaction engine that accurately processes deposits, withdrawals, balance inquiries, and account transfers, all supported by proper validation to prevent invalid operations such as over-drafting or transferring funds to a non-existent or inactive account.
2. Establish a secure administrative module for system oversight and account management. This is achieved by providing password-protected access to critical functions, including the ability to create new accounts, view a list of all active and inactive accounts, and perform a "soft delete" on existing accounts which preserves their history for auditing purposes. This module also grants exclusive access to a complete log of all system-wide transactions, ensuring full administrative accountability.
3. Make sure that the system will be capable of loading and saving account data through files persistently in order to keep state across sessions. This is achieved by developing a resilient persistence mechanism where all changes to account states, system settings, and transaction logs are immediately written to disk across multiple files, guaranteeing full data recoverability in the event of an unexpected program shutdown.

RATIONALE

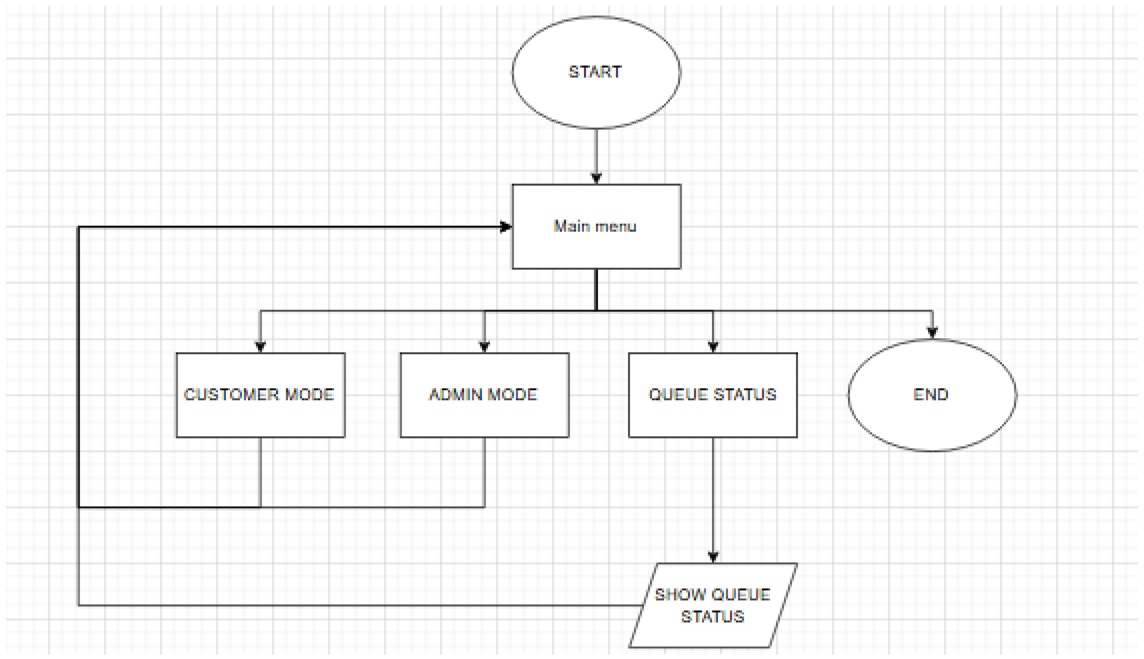
This banking system simulator is made to model fundamental financial operations, employing a selection of core data structures to manage state, process transactions, and handle client interactions efficiently. The architectural design prioritizes a clear separation of concerns, with specific data structures chosen for their performance characteristics in relation to their designated tasks. The primary storage mechanism for customer accounts is a std::vector, a dynamic array that houses smart pointers to Account objects. This choice provides contiguous memory allocation for pointers, which can be beneficial for cache performance during iteration, such as when an administrator lists all accounts. However, the most frequent operation on this structure is searching for a specific account by its number, which is implemented as a linear search. This results in a time complexity of O(N) for nearly all customer-facing transactions (e.g., deposits, withdrawals, transfers), where N is the total number of accounts in the bank. While efficient for a small-scale simulation, this approach would become a bottleneck in a system with a large number of accounts, where a hash map (std::unordered_map) offering O(1) average-case lookup time would be a more scalable alternative. The space complexity for account storage is O(N), as it grows linearly with the number of created accounts.

To manage client service flow, the system simulates a waiting line using a std::queue. When a customer authenticates, their account number is enqueued. This structure models the First-In, First-Out (FIFO) nature of a real-world queue. The operations of adding a customer to the queue (enqueueCustomer via push()) and serving the next customer (serveNext via pop()) are both highly efficient, with a constant time complexity of O(1). This ensures that the queuing mechanism remains performant and does not degrade as the number of waiting customers increases. The space complexity is O(C), where C is the current number of customers in the queue, making it very efficient for its purpose.

Within each Account object, transaction history is managed by a custom TransactionStack, which is implemented as a singly linked list. A stack was chosen because its Last-In, First-Out (LIFO) behavior is a natural fit for recording events; the most recent transaction is always the first one available. Using a linked list for the stack's underlying structure provides a key advantage: adding a new transaction (push) is an O(1) operation that involves only creating a new node and updating the head pointer, with no need for memory reallocation or shifting of existing elements. The space complexity for each account's history is O(T), where T is the number of transactions for that specific account. When a customer requests a statement, the toChronologicalVector method traverses the linked list and reverses it to present the history in chronological order. This traversal and reversal process has a time complexity of O(T), which is perfectly acceptable as it is only performed on demand and is scoped to a single account's history rather than the entire bank's dataset. This design ensures that recording transactions is maximally efficient while still allowing for correctly ordered retrieval when needed.

FLOWCHART and PSEUDO CODE

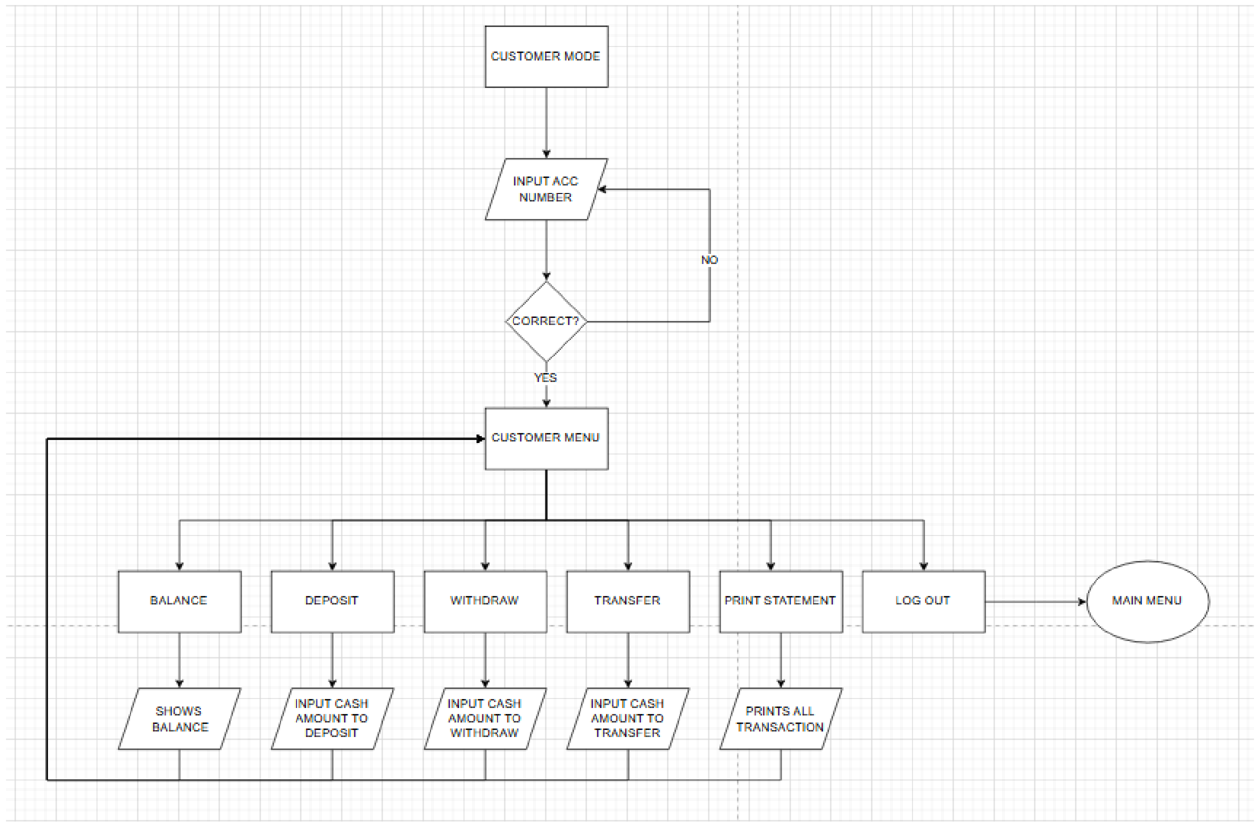
I. Main Program Loop (main.cpp logic)



```
FUNCTION Main
  INITIALIZE Bank object (triggers LoadData)
```

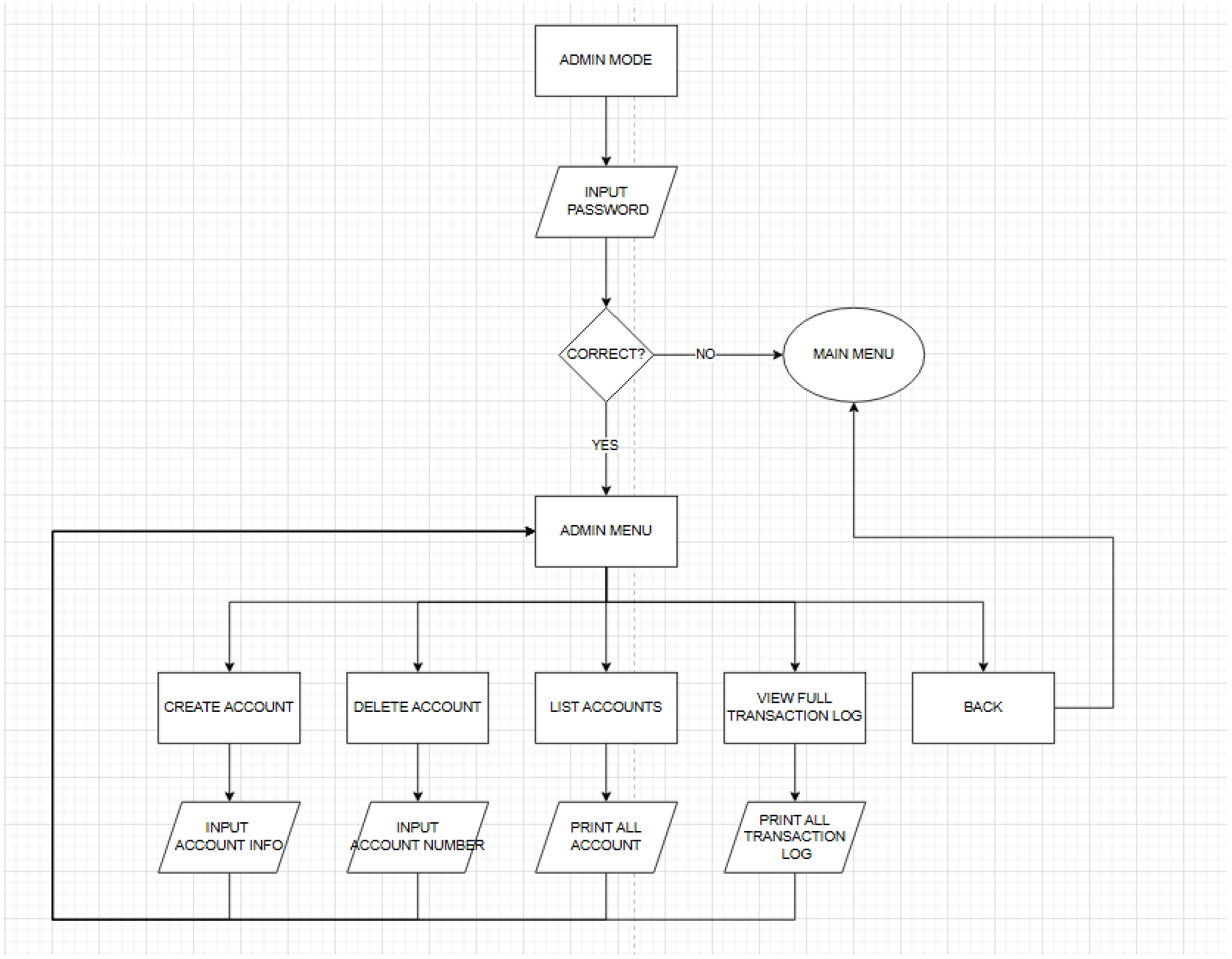
```
    SET running = TRUE

    WHILE running IS TRUE:
        Display Main Menu (Customer, Admin, Status, Exit)
        READ choice

        IF choice = 1 (Customer Mode):
            CALL HandleCustomerMode(Bank)
        ELSE IF choice = 2 (Admin Mode):
            CALL HandleAdminMode(Bank)
        ELSE IF choice = 3 (Queue Status):
            CALL DisplaySystemStatus(Bank)
        ELSE IF choice = 0 (Exit):
            SET running = FALSE
        ELSE:
            Display "Invalid choice"

    // Bank destructor automatically calls SaveData() here
    Display "Exiting."
END FUNCTION

FUNCTION DisplaySystemStatus(Bank)
    GET currentlyServingAcc FROM Bank
    GET totalTransactionCount FROM Bank
    GET hasQueuedCustomers FROM Bank

    Display "Now Serving: " + currentlyServingAcc
    Display "Total Transactions: " + totalTransactionCount
    Display "Customers in Queue: " + (IF hasQueuedCustomers THEN "Yes" ELSE "No")
END FUNCTION
```

II. Customer Mode Logic



```
FUNCTION HandleCustomerMode(Bank)
  IF Bank HAS currentlyServingAcc:
    SET accNo = Bank.getServingAcc()
    CALL CustomerMenuLoop(Bank, accNo)
    RETURN

  // Serving a new customer
  Display "Welcome, please enter account details."
  READ accNo
  READ pin

  IF Bank.AuthenticateCustomer(accNo, pin) IS TRUE AND Account IS Active:
    IF Bank.isAccCurrentlyServed(accNo) IS TRUE:
       Display "Account already being served."
    ELSE IF Bank.hasQueuedCustomers():
      // Simulate joining queue if someone else is being served
      Bank.EnqueueCustomer(accNo)
      Display "You have been added to the queue."
    ELSE:
      // Immediate service
      Bank.SetCurrentlyServingAcc(accNo)
      CALL CustomerMenuLoop(Bank, accNo)
  ELSE:
    Display "Authentication failed or account inactive."
END FUNCTION

FUNCTION CustomerMenuLoop(Bank, accNo)
  SET serving = TRUE
  WHILE serving IS TRUE:
    Display Customer Menu (Deposit, Withdraw, Transfer, Balance, Statement, End Session)
    READ actionChoice

    SWITCH actionChoice:
      CASE 1 (Deposit):
        READ amount
        IF Bank.Deposit(accNo, amount) IS TRUE: Display "Success"
        ELSE: Display "Failed"
      CASE 2 (Withdraw):
        READ amount
        IF Bank.Withdraw(accNo, amount) IS TRUE: Display "Success"
        ELSE: Display "Failed"
      CASE 3 (Transfer):
        READ toAccNo, amount
        IF Bank.Transfer(accNo, toAccNo, amount) IS TRUE: Display "Success"
        ELSE: Display "Failed"
      CASE 4 (Balance):
        Display Bank.GetBalance(accNo)
      CASE 5 (Statement):
        CALL Bank.PrintStatement(accNo)
      CASE 0 (End Session):
        Bank.ClearCurrentlyServingAcc()
        IF Bank.HasQueuedCustomers():
          // Automatically serve next customer
          nextAcc = Bank.ServeNext()
          Display "Serving next customer: " + nextAcc
        SET serving = FALSE
      DEFAULT:
        Display "Unknown action"
END FUNCTION
```

III. Admin Mode Logic



```
FUNCTION HandleAdminMode(Bank)
  READ adminPassInput
  IF Bank.AuthenticateAdmin(adminPassInput) IS FALSE:
    Display "Admin authentication failed."
    RETURN

  SET adminRunning = TRUE
  WHILE adminRunning IS TRUE:
    Display Admin Menu (Create, Delete, Print All, View Log, Back)
    READ actionChoice

    SWITCH actionChoice:
      CASE 1 (Create Account):
        READ name, pin, initialDeposit
        newAccNo = Bank.CreateAccount(name, pin, initialDeposit)
        Display "Account created with ID: " + newAccNo
      CASE 2 (Delete Account):
         READ accToDelete
         // This function typically includes confirmation logic
         Bank.DeleteAccount(accToDelete)
      CASE 3 (Print All):
         Bank.PrintAllAccounts()
      CASE 4 (View Log):
         Display raw contents of "transaction_log.txt"
      CASE 0 (Back):
         SET adminRunning = FALSE
      DEFAULT:
         Display "Unknown action"
END FUNCTION
```

LIBRARIES USED

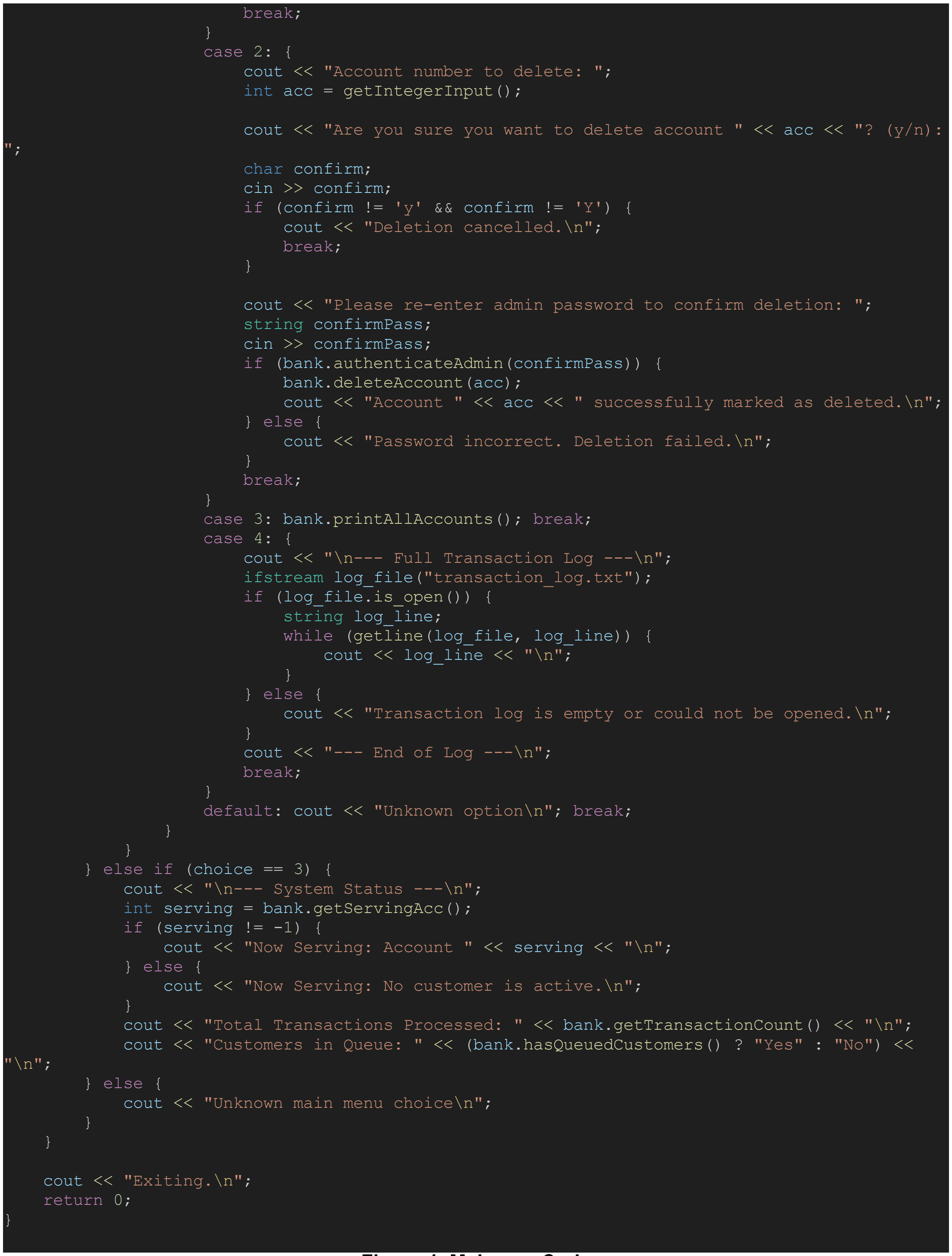
Header	Usage	Purpose in program
iostream	Used across main.cpp, Account.cpp, Bank.cpp	Provides input/output operations (std::cout, std::cin, std::cerr) for the console interface and error logging.
string	Used across Bank.h, Account.h, main.cpp	Supports string manipulation and storage (e.g., account holder names, admin password, reading file lines).
vector	Used in Bank.h, Account.h, TransactionStack.h	Stores the main collection of all Account objects (std::vector<std::unique_ptr<Account>>) and is used temporarily to retrieve chronological transaction history.
queue	Used in Bank.h	Implements the First-In, First-Out (FIFO) structure (std::queue<int>) for managing the customer waiting queue.
memory	Used in Bank.h	Provides smart pointers (std::unique_ptr) to manage dynamically allocated Account objects safely.
fstream	Used in Bank.cpp, main.cpp	Handles file input/output operations (std::ifstream, std::ofstream) for data persistence (reading/writing accounts.csv, bank_state.txt,

		transaction_log.txt).
sstream	Used in Bank.cpp	Supports string streams (std::stringstream) for parsing lines read from files, such as the transaction_log.txt.
iomanip	Used in Account.cpp, Bank.cpp	Provides I/O manipulators like std::fixed and std::setprecision(2) for precise, formatted display of currency amounts.
stdexcept	Used in Account.cpp, Bank.cpp	Provides standard exception classes like std::runtime_error for robust error handling (e.g., insufficient funds, invalid deposit amount).
limits	Used in main.cpp	Used with std::numeric_limits to clear error flags and discard bad input from std::cin for robust input handling.
algorithm	Used in TransactionStack.h	Contains general algorithms like std::reverse (likely used within the implementation of TransactionStack to sort history).

CODES AND OUTPUT

This section presents the system's architecture and demonstrates its core functionalities, validating the technical solution concepts discussed in the Rationale. The program is built around the Bank class, which manages a collection of Account objects. The central data structure is the TransactionStack (implemented as a Linked List) used within each account to record all deposits, withdrawals, and transfers chronologically.

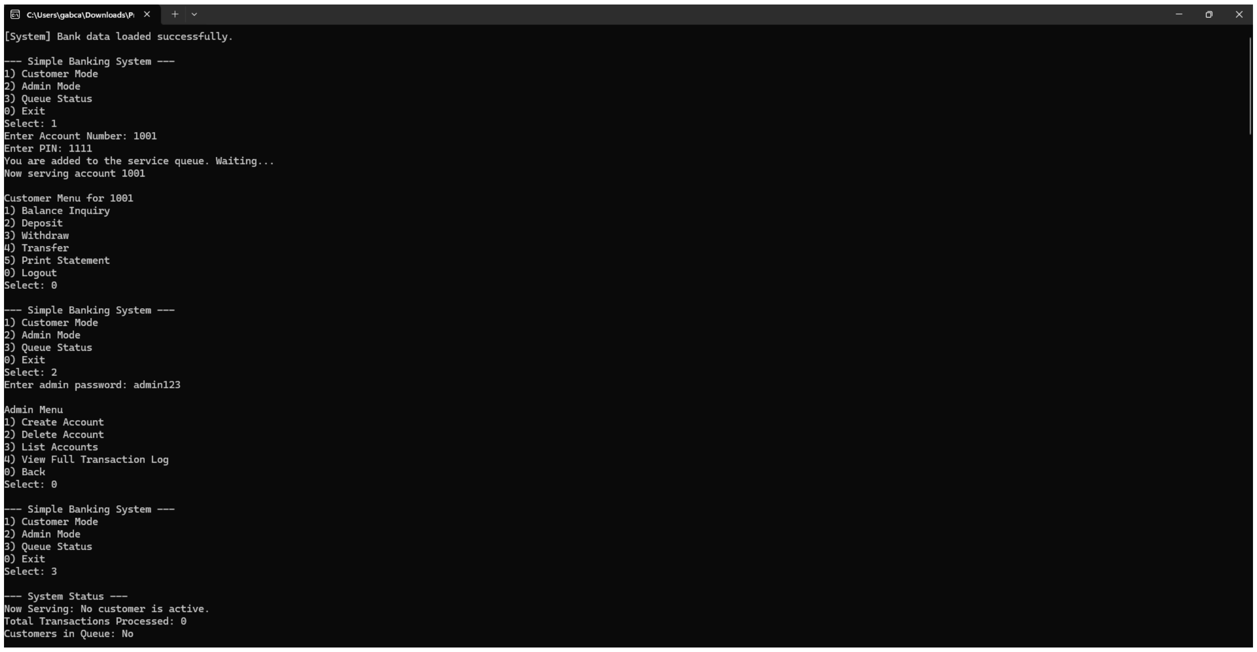
```cpp
#include <iostream>
#include <string>
#include <limits>
#include <fstream>
#include "Bank.h"

using namespace std;

void showMainMenu() {
    cout << "\n--- Simple Banking System ---\n";
    cout << "1) Customer Mode\n";
    cout << "2) Admin Mode\n";
    cout << "3) Queue Status\n";
    cout << "0) Exit\n";
    cout << "Select: ";
}

// helper to read an int
int getIntegerInput() {
    int value;
    while (!(cin >> value)) {
        cout << "Invalid input. Please enter a whole number: ";
        cin.clear(); // Clear the error flag
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard bad input
    }
    return value;
}

// helper to read a double
double getDoubleInput() {
    double value;
    while (!(cin >> value)) {
        cout << "Invalid input. Please enter a number: ";
        cin.clear(); // Clear the error flag
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard bad input
    }
    return value;
}

int main() {
    Bank bank;

    bool running = true;
    while (running) {
        showMainMenu();
        int choice = getIntegerInput();

        if (choice == 0) {
            running = false;
            continue;
        }

        if (choice == 1) {
            cout << "Enter Account Number: "; int acc = getIntegerInput();
            cout << "Enter PIN: "; int pin = getIntegerInput();
            if (!bank.authenticateCustomer(acc, pin)) {
                cout << "Authentication failed. Invalid Account Number or PIN.\n";
                continue;
            }

            cout << "You are added to the service queue. Waiting...\n";
            bank.enqueueCustomer(acc);
            int served = bank.serveNext();
            if (served == -1) { cout << "No customers to serve.\n"; continue; }
            cout << "Now serving account " << served << "\n";
            bank.setServingStatus(served);
            bool custLoop = true;
            while (custLoop) {
                cout << "\nCustomer Menu for " << served << "\n";
                cout << "1) Balance Inquiry\n";
                cout << "2) Deposit\n";
                cout << "3) Withdraw\n";
                cout << "4) Transfer\n";
                cout << "5) Print Statement\n";
                cout << "0) Logout\n";
                cout << "Select: ";
                int op = getIntegerInput();
                try {
                    switch (op) {
                        case 0:
                            custLoop = false;
                            bank.setServingStatus(-1); // Clear serving status on logout
                            break;
                        case 1: cout << "Balance: " << bank.getBalance(served) << "\n"; break;
                        case 2: {
                            cout << "Amount to deposit: "; double amt = getDoubleInput();
                            if (bank.deposit(served, amt)) cout << "Deposit successful\n";
                            break;
                        }
                        case 3: {
                            cout << "Amount to withdraw: "; double amt = getDoubleInput();
                            if (bank.withdraw(served, amt)) cout << "Withdrawal successful\n";
                            break;
                        }
                        case 4: {
                            cout << "Destination Account: "; int to = getIntegerInput();
                            cout << "Amount to transfer: "; double amt = getDoubleInput();
                            try {
                                if (bank.transfer(served, to, amt)) cout << "Transfer
successful.\n";
                            } catch (const std::runtime_error& e) {
                                cout << "Error: " << e.what() << "\n";
                            }
                            break;
                        }
                        case 5: bank.printAccountStatement(served); break;
                        default: cout << "Unknown option\n"; break;
                    }
                } catch (const exception& ex) {
                    cout << "Error: " << ex.what() << "\n";
                }
            }
        } else if (choice == 2) {
            cout << "Enter admin password: "; string pass; cin >> pass;
            if (!bank.authenticateAdmin(pass)) { cout << "Admin auth failed\n"; continue; }
            bool adminLoop = true;
            while (adminLoop) {
                cout << "\nAdmin Menu\n";
                cout << "1) Create Account\n";
                cout << "2) Delete Account\n";
                cout << "3) List Accounts\n";
                cout << "4) View Full Transaction Log\n";
                cout << "0) Back\n";
                cout << "Select: "; int op; cin >> op;
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); // consume newline
                switch (op) {
                    case 0: adminLoop = false; break;
                    case 1: {
                        cout << "Name: "; string name; getline(cin, name);
                        cout << "PIN (numeric): "; int pin = getIntegerInput();
                        cout << "Initial deposit: "; double d = getDoubleInput();
                        int newAcc = bank.createAccount(name, pin, d);
                        cout << "Created account " << newAcc << "\n";
```



```
                        break;
                    }
                    case 2: {
                        cout << "Account number to delete: ";
                        int acc = getIntegerInput();

                        cout << "Are you sure you want to delete account " << acc << "? (y/n):
";
                        char confirm;
                        cin >> confirm;
                        if (confirm != 'y' && confirm != 'Y') {
                            cout << "Deletion cancelled.\n";
                            break;
                        }

                        cout << "Please re-enter admin password to confirm deletion: ";
                        string confirmPass;
                        cin >> confirmPass;
                        if (bank.authenticateAdmin(confirmPass)) {
                            bank.deleteAccount(acc);
                            cout << "Account " << acc << " successfully marked as deleted.\n";
                        } else {
                            cout << "Password incorrect. Deletion failed.\n";
                        }
                        break;
                    }
                    case 3: bank.printAllAccounts(); break;
                    case 4: {
                        cout << "\n--- Full Transaction Log ---\n";
                        ifstream log_file("transaction_log.txt");
                        if (log_file.is_open()) {
                            string log_line;
                            while (getline(log_file, log_line)) {
                                cout << log_line << "\n";
                            }
                        } else {
                            cout << "Transaction log is empty or could not be opened.\n";
                        }
                        cout << "--- End of Log ---\n";
                        break;
                    }
                    default: cout << "Unknown option\n"; break;
                }
            }
        } else if (choice == 3) {
            cout << "\n--- System Status ---\n";
            int serving = bank.getServingAcc();
            if (serving != -1) {
                cout << "Now Serving: Account " << serving << "\n";
            } else {
                cout << "Now Serving: No customer is active.\n";
            }
            cout << "Total Transactions Processed: " << bank.getTransactionCount() << "\n";
            cout << "Customers in Queue: " << (bank.hasQueuedCustomers() ? "Yes" : "No") <<
"\n";
        } else {
            cout << "Unknown main menu choice\n";
        }
    }

    cout << "Exiting.\n";
    return 0;
}
```

Figure 1. Main.cpp Code

Figure 1 shows how the program implements a menu-driven banking system that supports customer transactions, admin account management, and queue-based service handling with robust input validation and file logging. The showMainMenu function displays a clean interface with three modes plus exit, while getIntegerInput and getDoubleInput ensure type-safe numeric entry using cin.clear() and cin.ignore() to discard invalid input. In Customer Mode, users authenticate via account number and PIN, join a service queue with enqueueCustomer, and access operations like balance inquiry, deposit, withdrawal, transfer, and statement printing which are all supported by try-catch blocks to handle exceptions or errors. The Admin Mode requires password authentication and enables account creation (with getline for full names), secure deletion (with confirmation and re-authentication), account listing, and full transaction log viewing from transaction_log.txt. The Queue Status option reveals the currently served account, total transactions, and queue presence, promoting operational transparency. This design effectively combines object-oriented encapsulation (via Bank class), error resilience, security checks, and persistent logging, delivering a scalable foundation for real-world banking terminals.

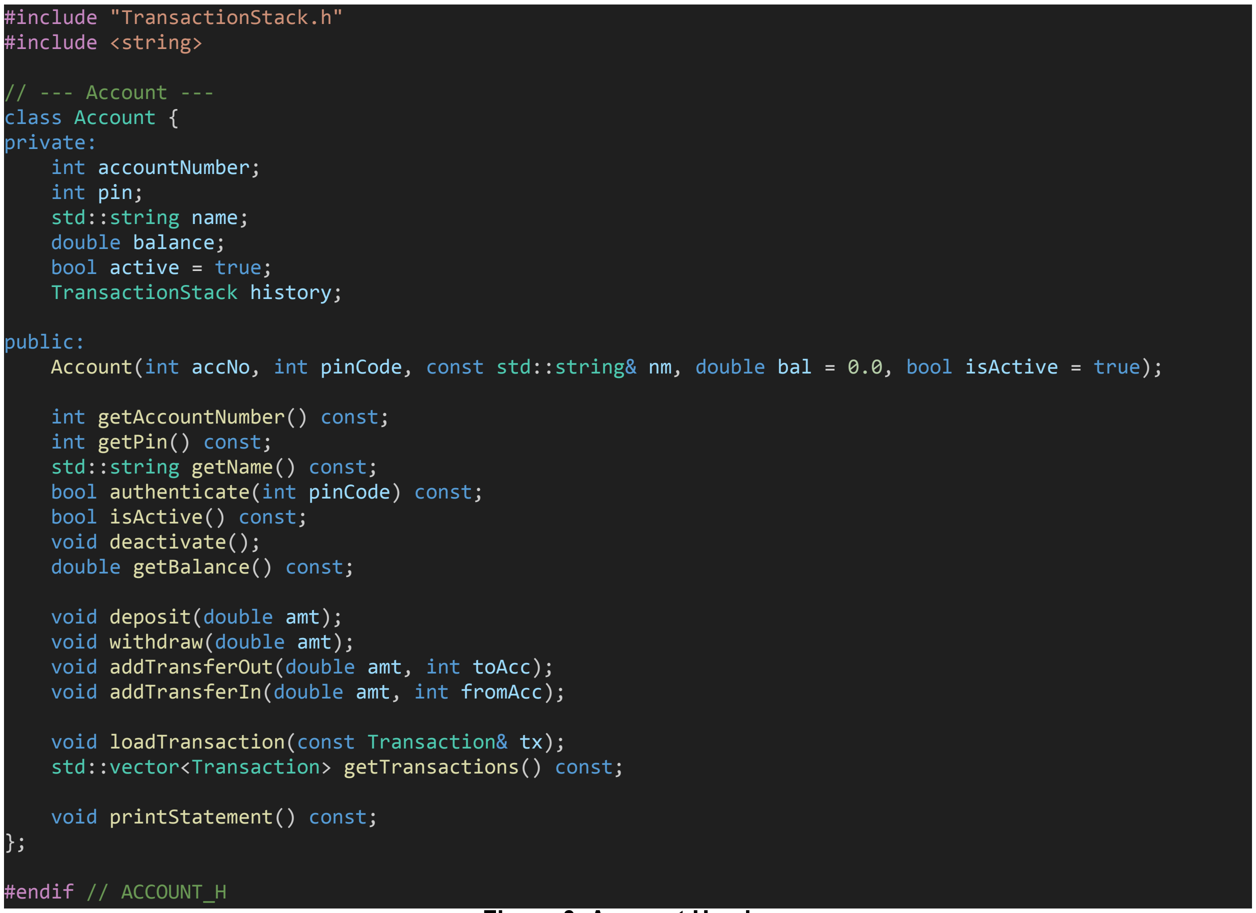


```
[System] Bank data loaded successfully.

--- Simple Banking System ---
1) Customer Mode
2) Admin Mode
3) Queue Status
0) Exit
Select: 1
Enter Account Number: 1001
Enter PIN: 1111
You are added to the service queue. Waiting...
Now serving account 1001

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 0

--- Simple Banking System ---
1) Customer Mode
2) Admin Mode
3) Queue Status
0) Exit
Select: 2
Enter admin password: admin123

Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 0

--- Simple Banking System ---
1) Customer Mode
2) Admin Mode
3) Queue Status
0) Exit
Select: 3

--- System Status ---
Now Serving: No customer is active.
Total Transactions Processed: 0
Customers in Queue: No
```

Figure 2. Main.cpp Code Output

This part of the code sets up the primary menu where the user can choose between Customer Mode, Admin Mode, Queue Status, and Exit and keeps asking for the user's input. Moreover, the file also contains some helper functions, such as getIntegerInput() and getDoubleInput(), which contribute to the input validation process making it more robust by erasing error flags and getting rid of bad input.

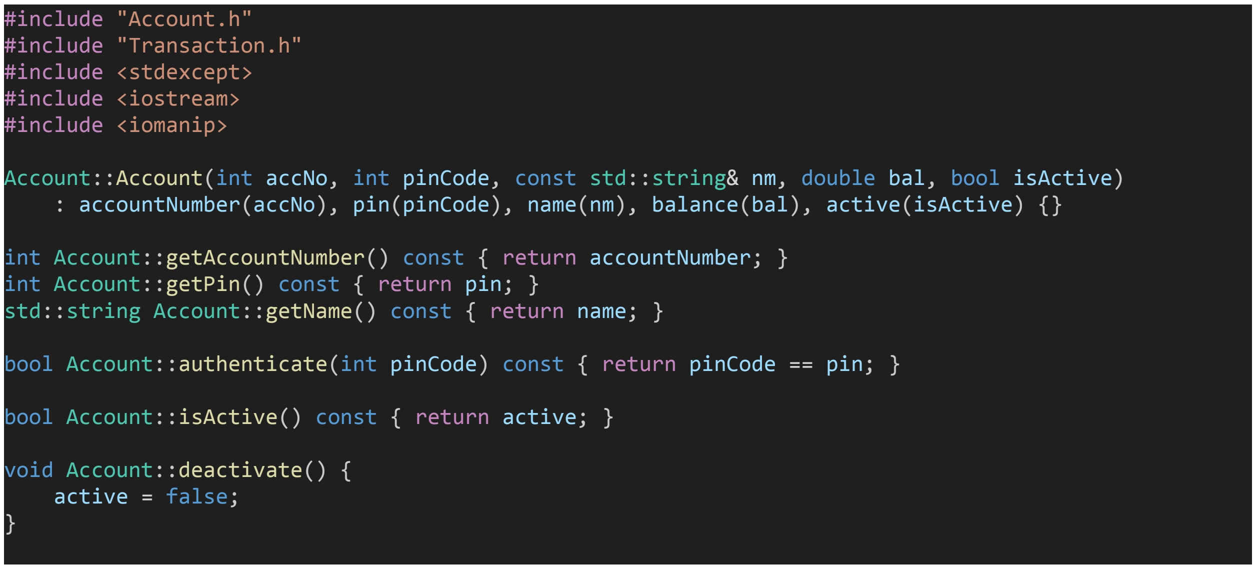
```
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <stdexcept>
#include <iostream>
#include <iomanip>
#include <vector>
```



```cpp
#include "TransactionStack.h"
#include <string>

// --- Account ---
class Account {
private:
    int accountNumber;
    int pin;
    std::string name;
    double balance;
    bool active = true;
    TransactionStack history;

public:
    Account(int accNo, int pinCode, const std::string& nm, double bal = 0.0, bool isActive = true);

    int getAccountNumber() const;
    int getPin() const;
    std::string getName() const;
    bool authenticate(int pinCode) const;
    bool isActive() const;
    void deactivate();
    double getBalance() const;

    void deposit(double amt);
    void withdraw(double amt);
    void addTransferOut(double amt, int toAcc);
    void addTransferIn(double amt, int fromAcc);

    void loadTransaction(const Transaction& tx);
    std::vector<Transaction> getTransactions() const;

    void printStatement() const;
};

#endif // ACCOUNT_H
```

Figure 3. Account Header

The header file Account.h introduces the Account class that serves as a representation of a bank account and consists of the account number, PIN, name, balance, and a TransactionStack for history as its members.

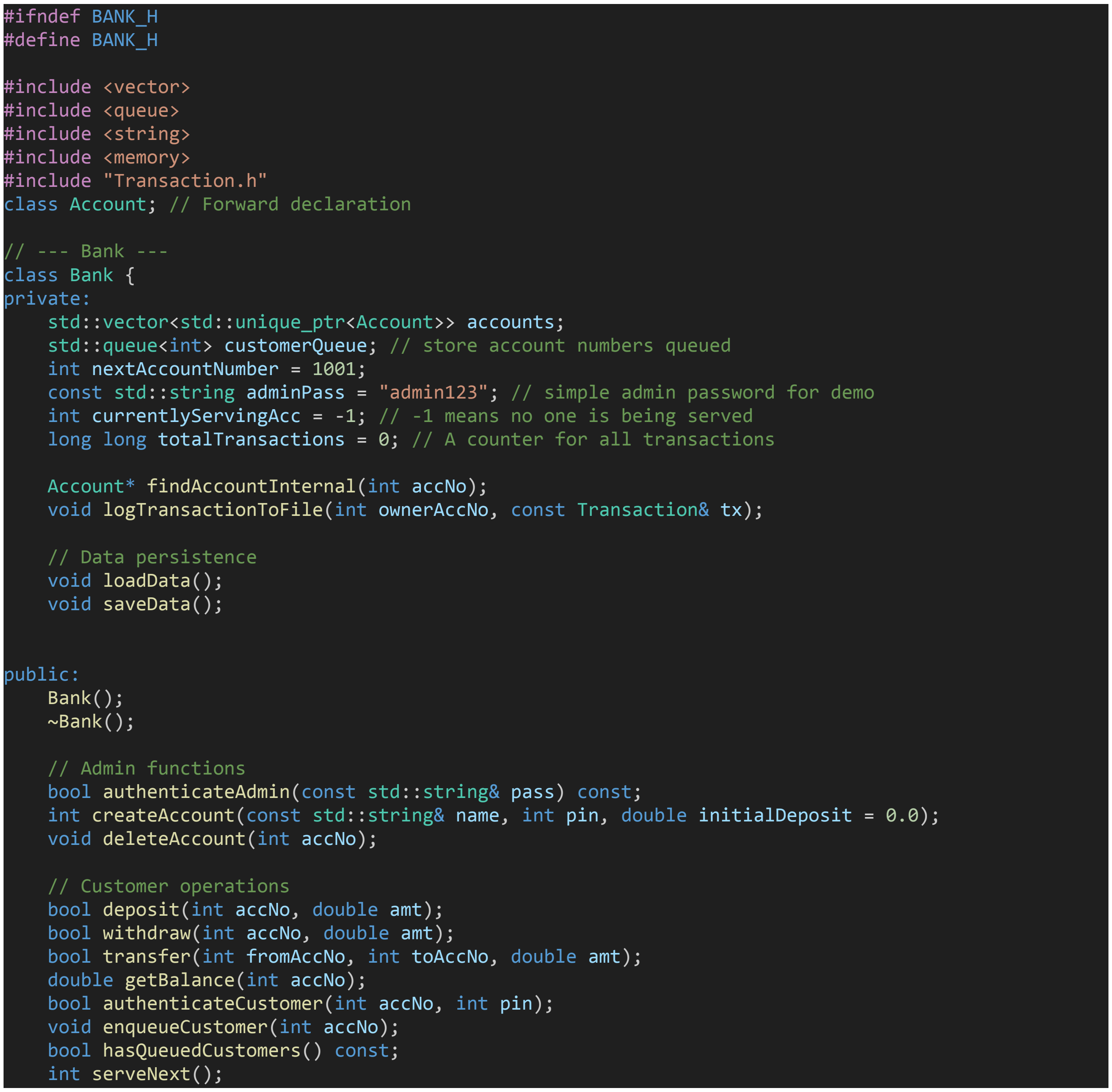


```cpp
#include "Account.h"
#include "Transaction.h"
#include <stdexcept>
#include <iostream>
#include <iomanip>

Account::Account(int accNo, int pinCode, const std::string& nm, double bal, bool isActive)
    : accountNumber(accNo), pin(pinCode), name(nm), balance(bal), active(isActive) {}

int Account::getAccountNumber() const { return accountNumber; }
int Account::getPin() const { return pin; }
std::string Account::getName() const { return name; }

bool Account::authenticate(int pinCode) const { return pinCode == pin; }

bool Account::isActive() const { return active; }

void Account::deactivate() {
    active = false;
}
```

```cpp
double Account::getBalance() const { return balance; }

void Account::deposit(double amt) {
    if (amt <= 0) throw std::runtime_error("Deposit amount must be positive");
    balance += amt;
    Transaction t{TxType::DEPOSIT, amt, -1, accountNumber, currentTimestamp(), "Deposit"};
    history.push(t);
}

void Account::withdraw(double amt) {
    if (amt <= 0) throw std::runtime_error("Withdrawal amount must be positive");
    if (amt > balance) throw std::runtime_error("Insufficient funds");
    balance -= amt;
    Transaction t{TxType::WITHDRAWAL, amt, accountNumber, -1, currentTimestamp(), "Withdrawal"};
    history.push(t);
}

void Account::addTransferOut(double amt, int toAcc) {
    if (amt <= 0) throw std::runtime_error("Transfer amount must be positive");
    if (amt > balance) throw std::runtime_error("Insufficient funds for transfer");
    balance -= amt;
    Transaction t{TxType::TRANSFER, amt, accountNumber, toAcc, currentTimestamp(), "Transfer Out"};
    history.push(t);
}

void Account::addTransferIn(double amt, int fromAcc) {
    if (amt <= 0) throw std::runtime_error("Transfer amount must be positive");
    balance += amt;
    Transaction t{TxType::TRANSFER, amt, fromAcc, accountNumber, currentTimestamp(), "Transfer In"};
    history.push(t);
}

void Account::loadTransaction(const Transaction& tx) {
    history.push(tx);
}

std::vector<Transaction> Account::getTransactions() const {
    return history.toChronologicalVector();
}

void Account::printStatement() const {
    std::cout << "\nStatement for Account " << accountNumber << " (" << name << ")\n";
    std::cout << "Balance: " << std::fixed << std::setprecision(2) << balance << "\n";
    std::cout << "-------------------------------\n";
    auto entries = history.toChronologicalVector();
    for (const auto& e : entries) {
        std::string typeStr;
        switch (e.type) {
            case TxType::DEPOSIT:         typeStr = "DEPOSIT   "; break;
            case TxType::WITHDRAWAL:      typeStr = "WITHDRAW  "; break;
            case TxType::TRANSFER:        typeStr = "TRANSFER  "; break;
            case TxType::ACCOUNT_CREATED: typeStr = "AC_CREATED"; break;
            case TxType::ACCOUNT_DELETED: typeStr = "AC_DELETED"; break;
            default:                      typeStr = "UNKNOWN   "; break;
        }

        std::cout << e.timestamp << " | " << typeStr << " | " << std::setw(8) << std::fixed <<
std::setprecision(2) << e.amount << " ";
        if (e.fromAccount != -1) std::cout << "from: " << e.fromAccount << " ";
        if (e.toAccount != -1) std::cout << "to: " << e.toAccount << " ";
        if (!e.note.empty()) std::cout << "| " << e.note;
        std::cout << "\n";
    }
    std::cout << "-------------------------------\n";
}
```

Figure 4. Account.cpp Code

The Account.cpp file serves as the operational engine for individual accounts, defining the specific procedures for all banking activities. It contains the logic for transactions like deposits and withdrawals, strictly applying validation rules to prevent errors such as overdrafts or invalid amounts. Most importantly, after any successful change to the account's balance, it instantly calls the history.push(t) method, ensuring that a real-time record of the event is immediately saved within the internal data structure to maintain a synchronized and accurate transaction history.



```cpp
#ifndef BANK_H
#define BANK_H

#include <vector>
#include <queue>
#include <string>
#include <memory>
#include "Transaction.h"
class Account; // Forward declaration

// --- Bank ---
class Bank {
private:
    std::vector<std::unique_ptr<Account>> accounts;
    std::queue<int> customerQueue; // store account numbers queued
    int nextAccountNumber = 1001;
    const std::string adminPass = "admin123"; // simple admin password for demo
    int currentlyServingAcc = -1; // -1 means no one is being served
    long long totalTransactions = 0; // A counter for all transactions

    Account* findAccountInternal(int accNo);
    void logTransactionToFile(int ownerAccNo, const Transaction& tx);

    // Data persistence
    void loadData();
    void saveData();


public:
    Bank();
    ~Bank();

    // Admin functions
    bool authenticateAdmin(const std::string& pass) const;
    int createAccount(const std::string& name, int pin, double initialDeposit = 0.0);
    void deleteAccount(int accNo);

    // Customer operations
    bool deposit(int accNo, double amt);
    bool withdraw(int accNo, double amt);
    bool transfer(int fromAccNo, int toAccNo, double amt);
    double getBalance(int accNo);
    bool authenticateCustomer(int accNo, int pin);
    void enqueueCustomer(int accNo);
    bool hasQueuedCustomers() const;
    int serveNext();
    void printAllAccounts() const;
    void printAccountStatement(int accNo);
    void setServingStatus(int accNo);
    int getServingAcc() const;
    long long getTransactionCount() const;
};

#endif // BANK_H
```

Figure 5. Bank Header

Bank.h defines the Bank class, which acts as the central controller for the system. It uses a std::vector of smart pointers to encapsulate all accounts, and a std::queue to simulate the customers waiting in line.

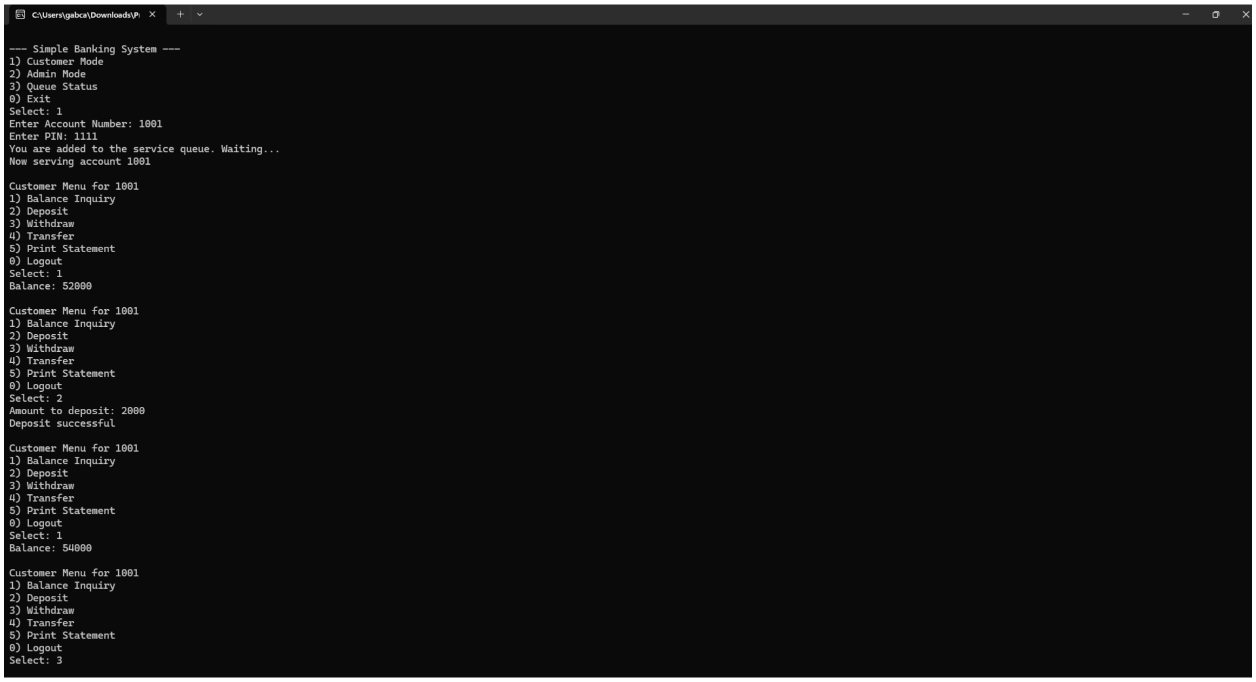
```
#include "Bank.h"
#include "Account.h"
#include <iostream>
#include <iomanip>
#include <stdexcept>
#include <fstream>
#include <sstream>

Bank::Bank() {
    loadData();
}
Bank::~Bank() {
    saveData();
    std::cout << "\n[System] Bank data saved.\n";
}

Account* Bank::findAccountInternal(int accNo) {
    for (auto& a : accounts) {
        if (a->getAccountNumber() == accNo && a->isActive()) {
            return a.get();
        }
    }
    return nullptr;
}

void Bank::logTransactionToFile(int ownerAccNo, const Transaction& tx) {
    std::ofstream log_file("transaction_log.txt", std::ios::app);
    if (!log_file.is_open()) {
        std::cerr << "Error: Could not open transaction_log.txt for writing." << std::endl;
        return;
    }

    log_file << "[" << tx.timestamp << "] ";

    switch (tx.type) {
        case TxType::DEPOSIT:
            log_file << "DEPOSIT: " << tx.amount << " to account " << tx.toAccount << ".";
            break;
        case TxType::WITHDRAWAL:
            log_file << "WITHDRAWAL: " << tx.amount << " from account " << tx.fromAccount << ".";
            break;
        case TxType::TRANSFER:
            log_file << "TRANSFER: " << tx.amount << " from account " << tx.fromAccount << " to " <<
tx.toAccount << ".";
            break;
        case TxType::ACCOUNT_CREATED:
            log_file << "SYSTEM: Account " << tx.toAccount << " created with initial deposit of " <<
tx.amount << ".";
            break;
        case TxType::ACCOUNT_DELETED:
            log_file << "SYSTEM: Account " << tx.fromAccount << " was deactivated.";
            break;
    }
    // This ownerAccNo is needed for loading data back to the correct account object
    log_file << " {owner:" << ownerAccNo << "}\n";
}

bool Bank::authenticateAdmin(const std::string& pass) const { return pass == adminPass; }

int Bank::createAccount(const std::string& name, int pin, double initialDeposit) {
    int accNo = nextAccountNumber++;
    auto acct = std::make_unique<Account>(accNo, pin, name, 0.0, true);

    Transaction t{TxType::ACCOUNT_CREATED, initialDeposit, -1, accNo, currentTimestamp(), "Account
created"};
    acct->loadTransaction(t);
    totalTransactions++;
    logTransactionToFile(accNo, t);

    if (initialDeposit > 0) {
        acct->deposit(initialDeposit);
    }

    accounts.push_back(std::move(acct));
    saveData(); // Immediately save state after creation
    return accNo;
}

void Bank::deleteAccount(int accNo) {
    Account* acc = findAccountInternal(accNo);
    if (!acc) return;

    acc->deactivate();
    Transaction t{TxType::ACCOUNT_DELETED, 0.0, accNo, -1, currentTimestamp(), "Account
deactivated"};
    acc->loadTransaction(t);
    totalTransactions++;
    logTransactionToFile(accNo, t);
    saveData(); // Immediately save state after deletion
}

bool Bank::deposit(int accNo, double amt) {
    Account* a = findAccountInternal(accNo);
    if (!a) return false;
    try {
        a->deposit(amt);
        totalTransactions++;
        logTransactionToFile(accNo, a->getTransactions().back());
        saveData(); // Immediately save state after deposit
        return true;
    } catch (const std::runtime_error& e) {
        std::cout << "Error: " << e.what() << "\n";
        return false;
    }
}

bool Bank::withdraw(int accNo, double amt) {
    Account* a = findAccountInternal(accNo);
    if (!a) return false;
    try {
        a->withdraw(amt);
        totalTransactions++;
        logTransactionToFile(accNo, a->getTransactions().back());
        saveData(); // Immediately save state after withdrawal
    } catch (const std::runtime_error& e) {
        std::cout << "Error: " << e.what() << "\n";
        return false;
    }
    return true;
}

bool Bank::transfer(int fromAccNo, int toAccNo, double amt) {
    if (fromAccNo == toAccNo) {
        throw std::runtime_error("Cannot transfer money to your own account.");
    }
    Account* from = findAccountInternal(fromAccNo);
    Account* to = findAccountInternal(toAccNo);
    if (!from) return false; // Should not happen if called from an authenticated session
    if (!to) {
        throw std::runtime_error("Destination account not found or is inactive.");
    }
    try {
        from->addTransferOut(amt, toAccNo);
        totalTransactions++;
        logTransactionToFile(fromAccNo, from->getTransactions().back());
        to->addTransferIn(amt, fromAccNo);
        totalTransactions++;
        logTransactionToFile(toAccNo, to->getTransactions().back());
        saveData(); // Immediately save state after transfer
    } catch (const std::runtime_error& e) {
        std::cout << "Error: " << e.what() << "\n";
        return false;
    }
    return true;
}

double Bank::getBalance(int accNo) {
    Account* a = findAccountInternal(accNo);
    if (!a) throw std::runtime_error("Account not found");
    return a->getBalance();
}

bool Bank::authenticateCustomer(int accNo, int pin) {
    Account* a = findAccountInternal(accNo);
    if (!a) return false;
    return a->authenticate(pin);
}

void Bank::enqueueCustomer(int accNo) { customerQueue.push(accNo); }
bool Bank::hasQueuedCustomers() const { return !customerQueue.empty(); }
int Bank::serveNext() { if (customerQueue.empty()) return -1; int accNo = customerQueue.front();
customerQueue.pop(); return accNo; }

void Bank::setServingStatus(int accNo) { currentlyServingAcc = accNo; }
int Bank::getServingAcc() const { return currentlyServingAcc; }
long long Bank::getTransactionCount() const { return totalTransactions; }

void Bank::printAllAccounts() const {
    std::cout << "All accounts:\n";
    for (const auto& a : accounts) {
        std::cout << "  Acc#: " << a->getAccountNumber() << " Name: " << a->getName() << " Balance:
" << std::fixed << std::setprecision(2) << a->getBalance() << (a->isActive() ? " [Active]" : "
[Inactive]") << "\n";
    }
}
```

```cpp
void Bank::printAccountStatement(int accNo) {
    Account* a = findAccountInternal(accNo);
    if (!a) { std::cout << "Account not found\n"; return; }
    a->printStatement();
}

void Bank::saveData() {
    std::ofstream bank_file("bank_state.txt");
    if (bank_file.is_open()) {
        bank_file << nextAccountNumber;
    }
    bank_file.close();

    std::ofstream accounts_file("accounts.csv");
    if (!accounts_file.is_open()) {
        std::cerr << "Error: Could not open data files for writing." << std::endl;
        return;
    }

    for (const auto& acc : accounts) {
        accounts_file << acc->getAccountNumber() << "," << acc->getPin() << "," << acc->getBalance()
<< "," << (acc->isActive() ? "1" : "0") << "," << acc->getName() << "\n";
    }
    accounts_file.close();
}

void Bank::loadData() {
    std::ifstream bank_file("bank_state.txt");
    if (bank_file.is_open()) {
        bank_file >> nextAccountNumber;
        bank_file.close();
    }

    std::ifstream accounts_file("accounts.csv");
    std::string line;
    if (accounts_file.is_open()) {
        while (getline(accounts_file, line)) {
            std::stringstream ss(line);
            std::string item, name;
            int accNo, pin;
            double balance;
            bool isActive;

            getline(ss, item, ','); accNo = std::stoi(item);
            getline(ss, item, ','); pin = std::stoi(item);
            getline(ss, item, ','); balance = std::stod(item);
            getline(ss, item, ','); isActive = (item == "1");
            getline(ss, name);

            accounts.push_back(std::make_unique<Account>(accNo, pin, name, balance, isActive));
        }
        accounts_file.close();
    }

    // 3. Load transactions
    std::ifstream transactions_file("transaction_log.txt");
    if (transactions_file.is_open()) {
        while (getline(transactions_file, line)) {
            std::stringstream ss(line);

            // --- Parse new format ---
            std::string timestamp, typeStr, word;
            int ownerAcc = -1, fromAcc = -1, toAcc = -1;
            double amount = 0.0;
            // Extract timestamp: [YYYY-MM-DD HH:MM:SS]
            ss.get();
            getline(ss, timestamp, ']');

            // Extract owner: {owner:1001}
            size_t ownerPos = line.find("{owner:");
            if (ownerPos != std::string::npos) {
                ownerAcc = std::stoi(line.substr(ownerPos + 7));
            }

            ss >> typeStr; // DEPOSIT: or WITHDRAWAL: or TRANSFER: or SYSTEM:

            TxType type;
            std::string note;

            if (typeStr == "DEPOSIT:") {
                type = TxType::DEPOSIT;
                ss >> amount >> word >> word >> toAcc;
                note = "Deposit";
            } else if (typeStr == "WITHDRAWAL:") {
                type = TxType::WITHDRAWAL;
                ss >> amount >> word >> word >> fromAcc;
                note = "Withdrawal";
            } else if (typeStr == "TRANSFER:") {
                type = TxType::TRANSFER;
                ss >> amount >> word >> word >> fromAcc >> word >> toAcc;
                note = (ownerAcc == fromAcc) ? "Transfer Out" : "Transfer In";
            } else if (typeStr == "SYSTEM:") {
                ss >> word; // "Account"
                if (word == "Account") {
                    ss >> fromAcc; // Account number
                    ss >> word; // "created" or "was"
                    if (word == "created") {
                        type = TxType::ACCOUNT_CREATED;
                        toAcc = fromAcc; fromAcc = -1;
                        ss >> word >> word >> word >> word >> amount;
                        note = "Account created";
                    } else { // "was deactivated"
                        type = TxType::ACCOUNT_DELETED;
                        toAcc = -1;
                        note = "Account deactivated";
                    }
                }
            } else {
                continue; // Skip malformed line
            }

            if (ownerAcc == -1) continue; // Cannot process log entry without an owner

            Transaction tx = {type, amount, fromAcc, toAcc, timestamp, note};

            // Find account without checking if active, as we want to load history for all accounts
            for (auto& acc : accounts) {
                if (acc->getAccountNumber() == ownerAcc) {
                    acc->loadTransaction(tx);
                    break;
                }
            }
        }
        transactions_file.close();
    }

    if (!accounts.empty() || nextAccountNumber > 1001) {
        std::cout << "[System] Bank data loaded successfully.\n";
    }
```

```
}
```

Figure 6. Bank.cpp Code

The Bank.cpp file contains the main control logic for the banking simulator, acting as the system's central control unit. It includes the entire transfer function for moving money between accounts and all administrator functions for account management creation and deletion. Most importantly, it ensures system integrity by writing every transaction to an external file for later auditing, as well as state recoverability through the implementation of loadData() and saveData() methods that read and write all account and queue information, thus preserving the bank's status between program sessions.

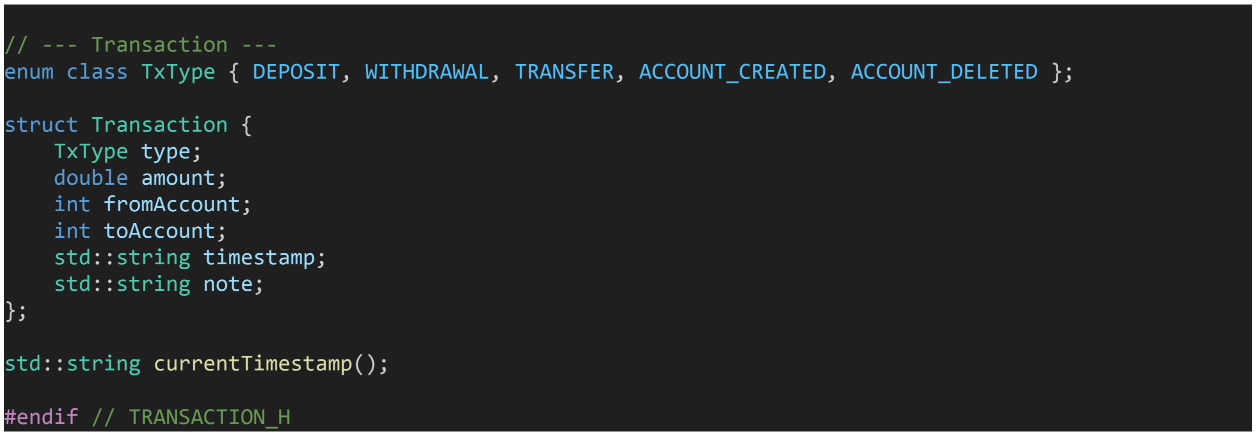


```
--- Simple Banking System ---
1) Customer Mode
2) Admin Mode
3) Queue Status
0) Exit
Select: 1
Enter Account Number: 1001
Enter PIN: 1111
You are added to the service queue. Waiting...
Now serving account 1001

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 1
Balance: 52000

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 2
Amount to deposit: 2000
Deposit successful

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 1
Balance: 54000

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 3
```

Figure 7. Bank Output, Customer Mode

This figure confirms that the system successfully authenticates, queues, and serves Account 1001, transitioning securely to display the interactive Customer Menu.

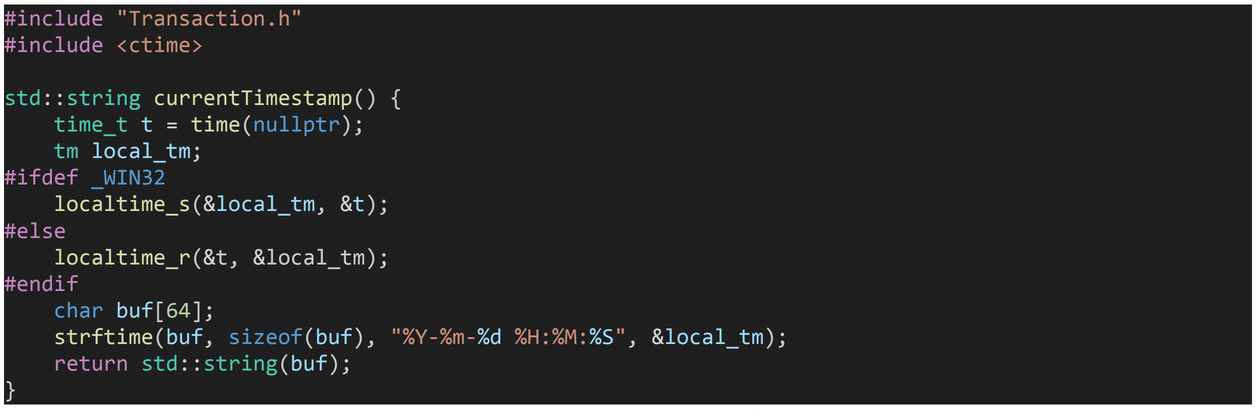
```
#ifndef TRANSACTION_H
#define TRANSACTION_H

#include <ctime>
#include <string>
```



```
// --- Transaction ---
enum class TxType { DEPOSIT, WITHDRAWAL, TRANSFER, ACCOUNT_CREATED, ACCOUNT_DELETED };

struct Transaction {
    TxType type;
    double amount;
    int fromAccount;
    int toAccount;
    std::string timestamp;
    std::string note;
};

std::string currentTimestamp();

#endif // TRANSACTION_H
```

Figure 8. Transaction Header

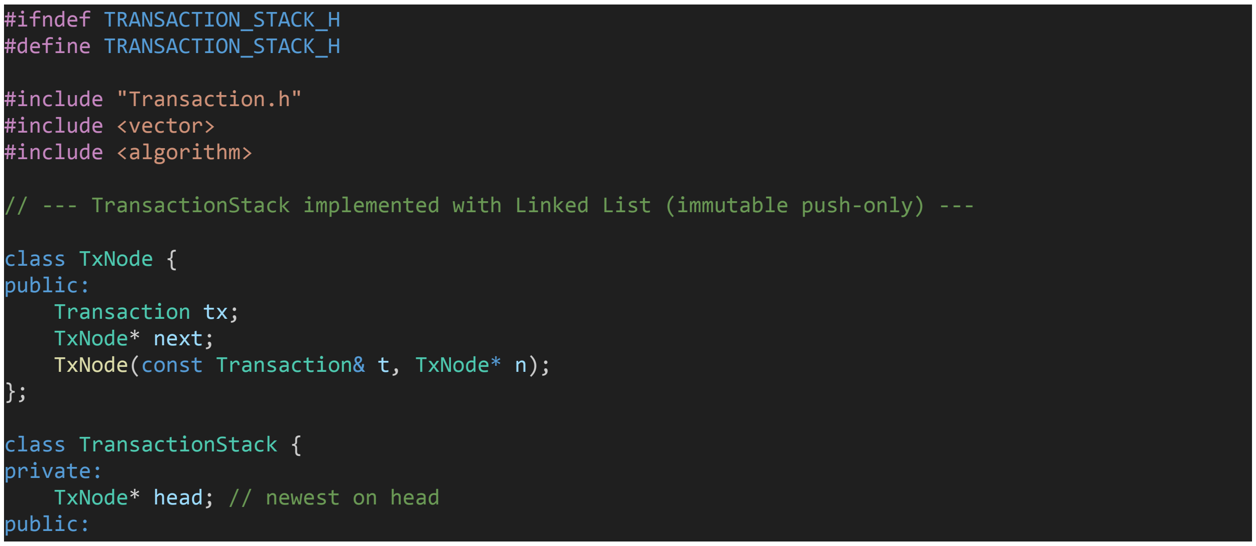
These definitions establish the core Transaction struct and the relevant logic that models a single financial event within the ecosystem. The struct has fields for type (an enum), amount, fromAccount, toAccount, and a description timestamp string.



```
#include "Transaction.h"
#include <ctime>

std::string currentTimestamp() {
    time_t t = time(nullptr);
    tm local_tm;
#ifdef _WIN32
    localtime_s(&local_tm, &t);
#else
    localtime_r(&t, &local_tm);
#endif
    char buf[64];
    strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", &local_tm);
    return std::string(buf);
}
```

Figure 9. Transaction.cpp Code

The figure shows the currentTimestamp() function that returns a formatted string to represent the time of a transaction for accurate recording.



```
#ifndef TRANSACTION_STACK_H
#define TRANSACTION_STACK_H

#include "Transaction.h"
#include <vector>
#include <algorithm>

// --- TransactionStack implemented with Linked List (immutable push-only) ---

class TxNode {
public:
    Transaction tx;
    TxNode* next;
    TxNode(const Transaction& t, TxNode* n);
};

class TransactionStack {
private:
    TxNode* head; // newest on head
public:
    TransactionStack();
    ~TransactionStack();

    void push(const Transaction& t);
    std::vector<Transaction> toChronologicalVector() const;
};

#endif // TRANSACTION_STACK_H
```

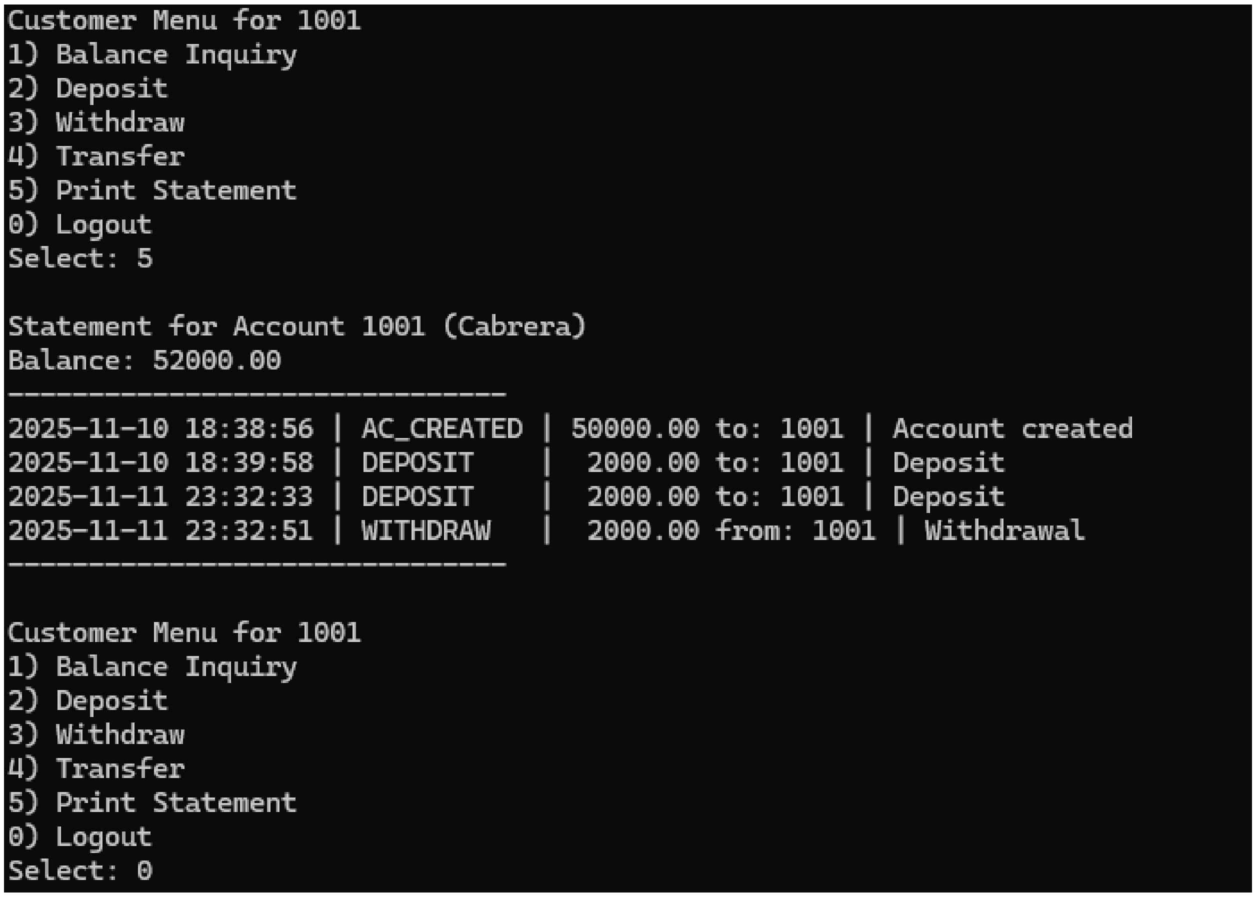
Figure 10. TransactionStack Header

The TransactionStack class is shown in the figure. It uses a singly linked list to maintain the transaction history for an account. Because of this choice of data structure, the push operation (adding a new transaction) can be performed in constant time O(1).

```
#include "TransactionStack.h"
#include <algorithm>

TxNode::TxNode(const Transaction& t, TxNode* n) : tx(t), next(n) {}

TransactionStack::TransactionStack() : head(nullptr) {}

TransactionStack::~TransactionStack() {
    TxNode* cur = head;
    while (cur) {
        TxNode* n = cur->next;
        delete cur;
        cur = n;
    }
}

void TransactionStack::push(const Transaction& t) {
    head = new TxNode(t, head);
}

std::vector<Transaction> TransactionStack::toChronologicalVector() const {
    std::vector<Transaction> out;
    TxNode* cur = head;
    while (cur) {
        out.push_back(cur->tx);
        cur = cur->next;
    }
    std::reverse(out.begin(), out.end());
    return out;
}
```

Figure 11. TransactionStack.cpp Code

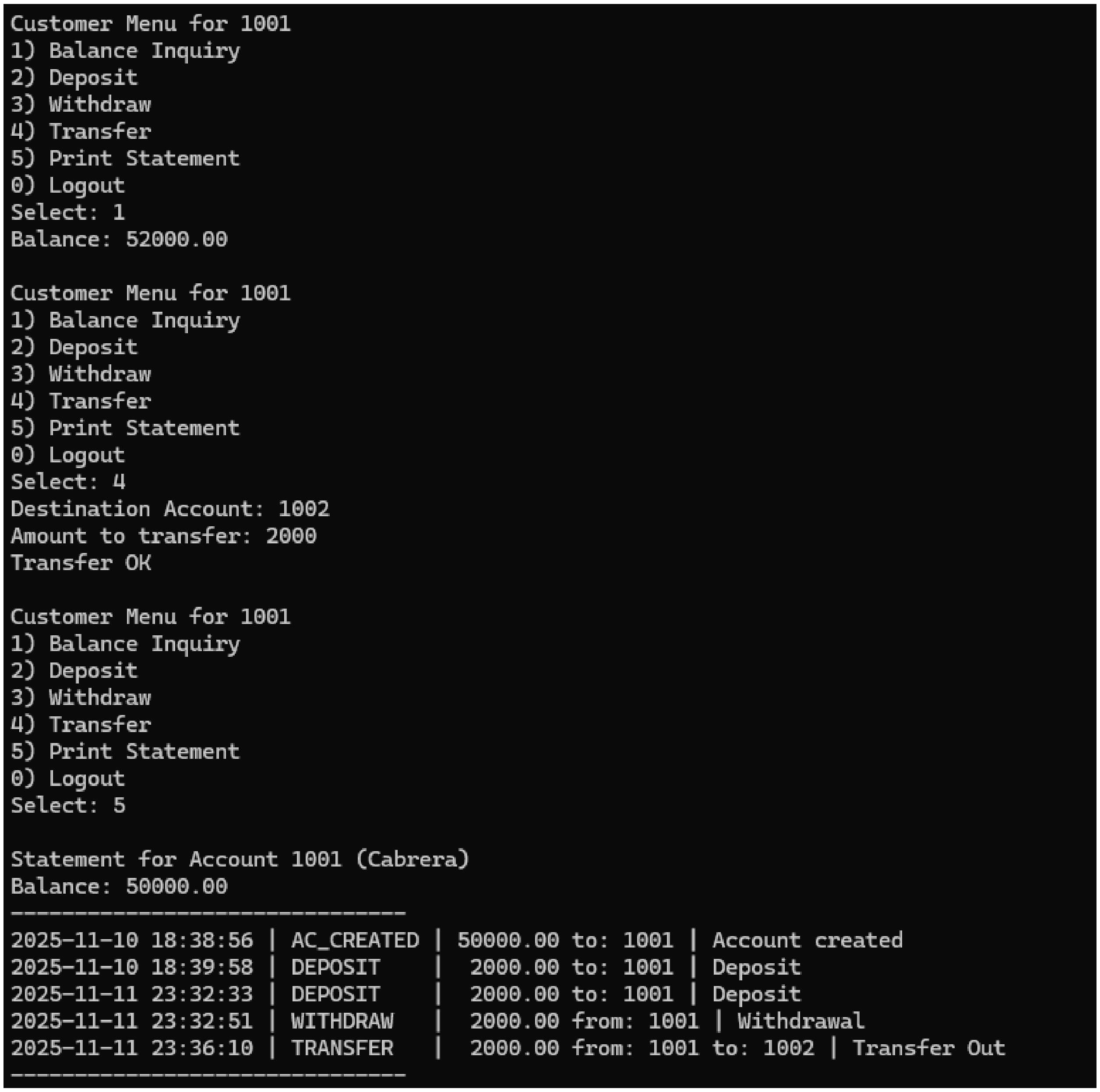
This figure shows the TransactionStack, which works like a stack data structure following the Last In, First Out (LIFO) concept, meaning the most recent transaction added is the first one accessed. Each transaction is stored inside a TxNode, which contains the transaction data and a pointer to the next node, linking them together. The push function adds new transactions to the top of the stack, linking it to the previous nodes. The destructor (~TransactionStack) deletes all nodes to ensure proper memory cleanup. Overall, this figure demonstrates how transactions are efficiently stored and managed within the stack structure.



```
Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 5

Statement for Account 1001 (Cabrera)
Balance: 52000.00
-------------------------------
2025-11-10 18:38:56 | AC_CREATED | 50000.00 to: 1001 | Account created
2025-11-10 18:39:58 | DEPOSIT    |  2000.00 to: 1001 | Deposit
2025-11-11 23:32:33 | DEPOSIT    |  2000.00 to: 1001 | Deposit
2025-11-11 23:32:51 | WITHDRAW   |  2000.00 from: 1001 | Withdrawal
-------------------------------

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 0
```

Figure 12. Transaction Output

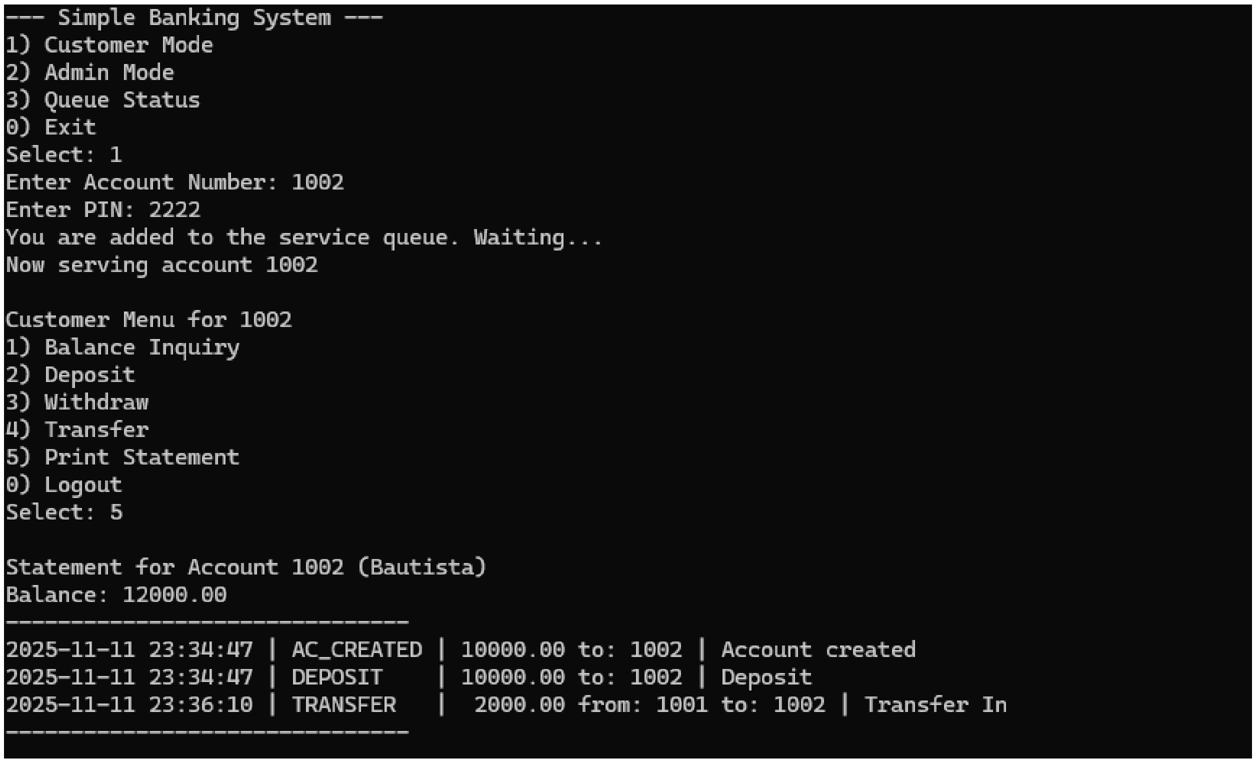
This figure shows the printed statement of a certain account after some financial activities. It should show each transaction event, including Deposits and Withdrawals, their timestamps, and the final Balance calculated. This output is to demonstrate that the TransactionStack retrieves and prints the transaction history in the correct chronological order.



```
Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 1
Balance: 52000.00

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 4
Destination Account: 1002
Amount to transfer: 2000
Transfer OK

Customer Menu for 1001
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 5

Statement for Account 1001 (Cabrera)
Balance: 50000.00
-------------------------------
2025-11-10 18:38:56 | AC_CREATED | 50000.00 to: 1001 | Account created
2025-11-10 18:39:58 | DEPOSIT    |  2000.00 to: 1001 | Deposit
2025-11-11 23:32:33 | DEPOSIT    |  2000.00 to: 1001 | Deposit
2025-11-11 23:32:51 | WITHDRAW   |  2000.00 from: 1001 | Withdrawal
2025-11-11 23:36:10 | TRANSFER   |  2000.00 from: 1001 to: 1002 | Transfer Out
-------------------------------
```

Figure 13. Account Transfer Output, Transfer Out Side

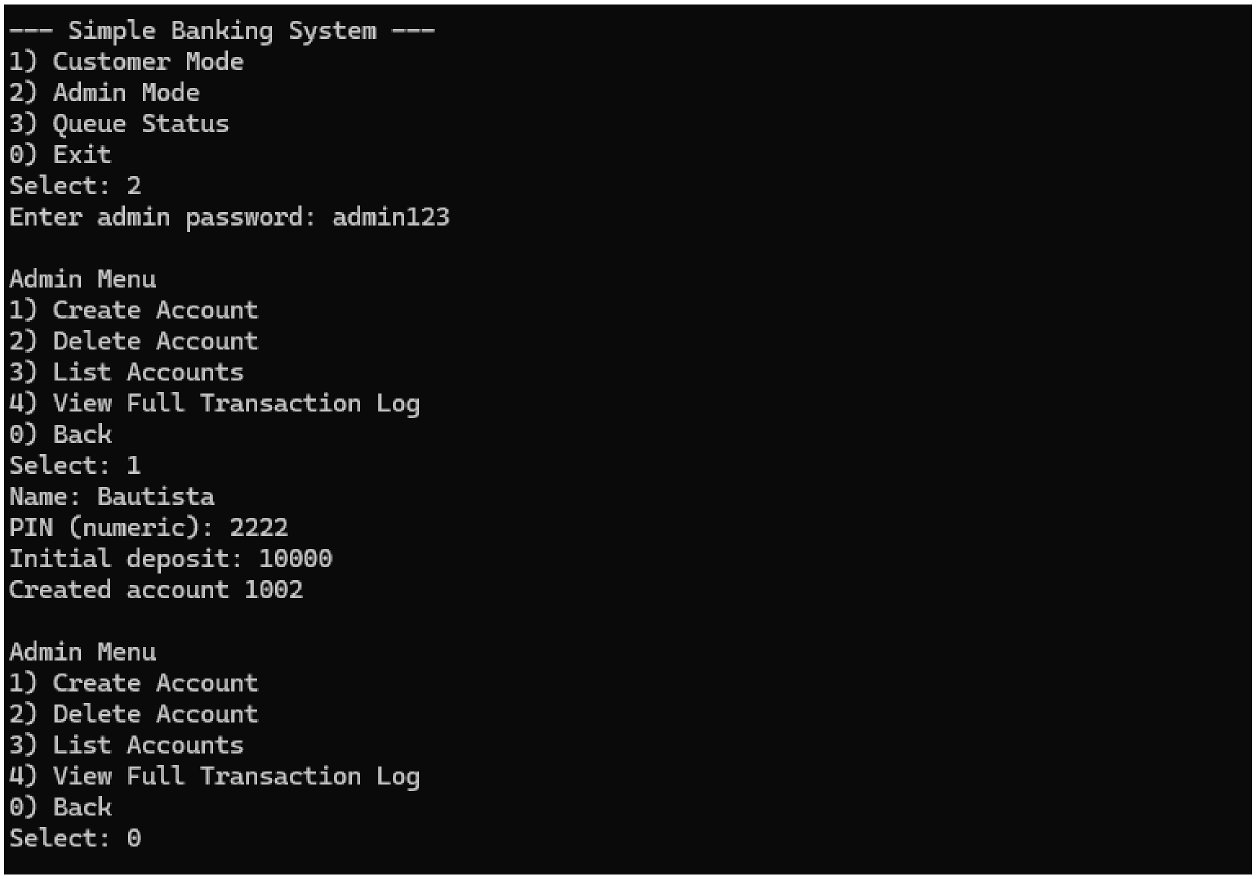
The figure shows the customer menu and transaction log showing a successful transfer of 2000.00 from Account 1001 to Account 1002. The top panel displays the interactive CLI menu during a transfer operation, while the bottom panel shows the resulting statement with a complete, timestamped audit trail.



```
--- Simple Banking System ---
1) Customer Mode
2) Admin Mode
3) Queue Status
0) Exit
Select: 1
Enter Account Number: 1002
Enter PIN: 2222
You are added to the service queue. Waiting...
Now serving account 1002

Customer Menu for 1002
1) Balance Inquiry
2) Deposit
3) Withdraw
4) Transfer
5) Print Statement
0) Logout
Select: 5

Statement for Account 1002 (Bautista)
Balance: 12000.00
-------------------------------
2025-11-11 23:34:47 | AC_CREATED | 10000.00 to: 1002 | Account created
2025-11-11 23:34:47 | DEPOSIT    | 10000.00 to: 1002 | Deposit
2025-11-11 23:36:10 | TRANSFER   |  2000.00 from: 1001 to: 1002 | Transfer In
-------------------------------
```

Figure 14. Account Transfer Output, Transfer In Side

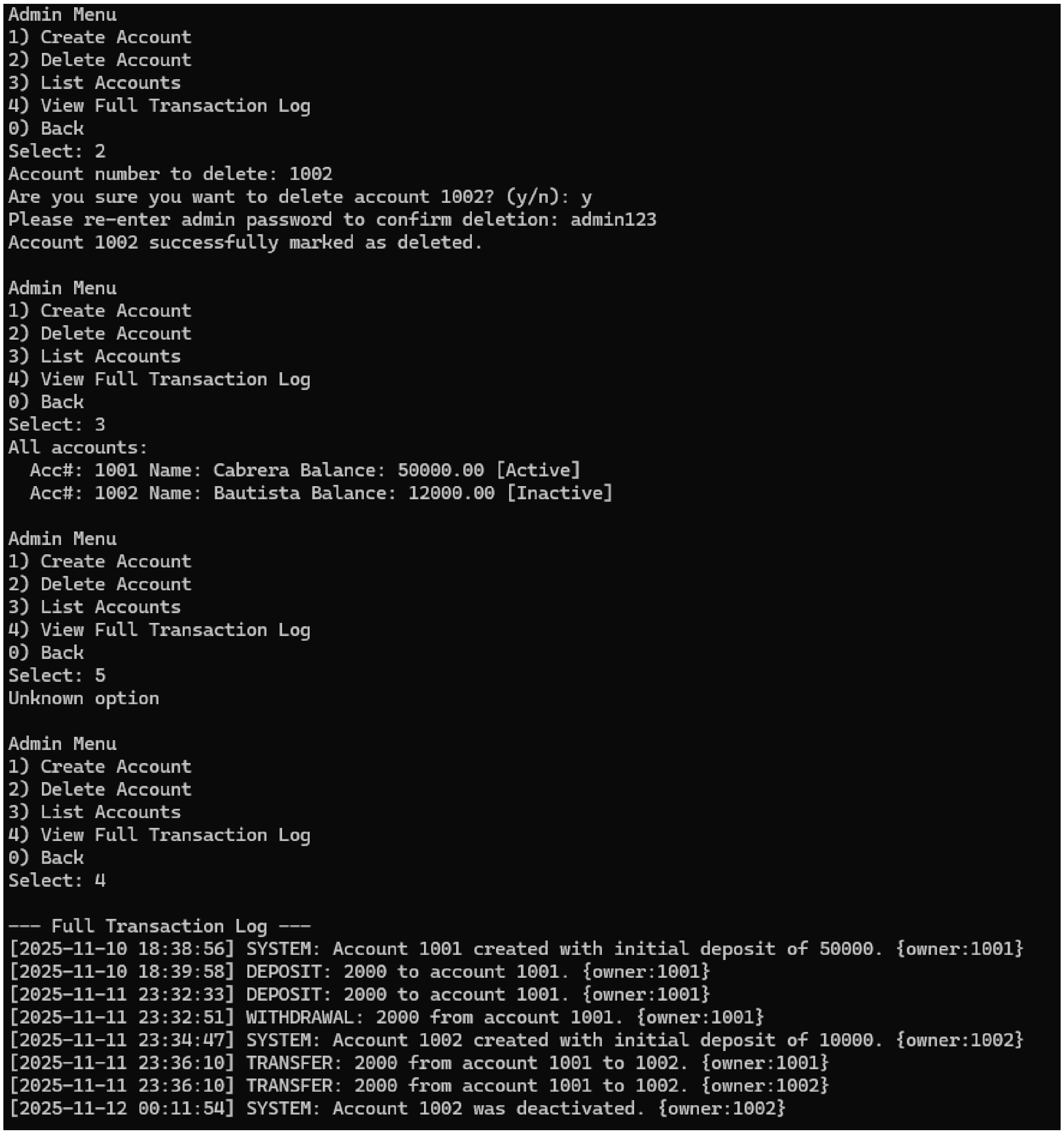
This figure shows the Account Transfer Output for the receiving side of the transaction. It displays the actions performed in the Customer Mode for account number 1002 (Bautista), including account creation, deposit, and a transfer received from another account. The statement section shows a clear record of these activities with timestamps and transaction details. The final balance reflects the updated total after the transfer was received. Overall, this figure confirms that the system correctly records and updates transactions for the receiving account.Figure 14 completes the end-to-end transfer workflow initiated in Figure 13 (Account 1001 to 1002). It illustrates the receiving account's perspective, reinforcing data isolation, transaction scoping, and audit integrity, which are core learning objectives in the Banking System Simulator.



```
--- Simple Banking System ---
1) Customer Mode
2) Admin Mode
3) Queue Status
0) Exit
Select: 2
Enter admin password: admin123

Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 1
Name: Bautista
PIN (numeric): 2222
Initial deposit: 10000
Created account 1002

Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 0
```

Figure 15. Bank Output, Admin Mode Creation of Account

This figure depicts the administrative ability to create a new client within the system. It demonstrates entering the admin password, selecting the Create Account option, and successfully specifying a new client's name, PIN, and initial deposit. The outcome is the system displaying a unique new account number, indicating that a new record has been successfully added to the bank's account vector.



```
Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 2
Account number to delete: 1002
Are you sure you want to delete account 1002? (y/n): y
Please re-enter admin password to confirm deletion: admin123
Account 1002 successfully marked as deleted.

Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 3
All accounts:
  Acc#: 1001 Name: Cabrera Balance: 50000.00 [Active]
  Acc#: 1002 Name: Bautista Balance: 12000.00 [Inactive]

Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 5
Unknown option

Admin Menu
1) Create Account
2) Delete Account
3) List Accounts
4) View Full Transaction Log
0) Back
Select: 4

--- Full Transaction Log ---
[2025-11-10 18:38:56] SYSTEM: Account 1001 created with initial deposit of 50000. {owner:1001}
[2025-11-10 18:39:58] DEPOSIT: 2000 to account 1001. {owner:1001}
[2025-11-11 23:32:33] DEPOSIT: 2000 to account 1001. {owner:1001}
[2025-11-11 23:32:51] WITHDRAWAL: 2000 from account 1001. {owner:1001}
[2025-11-11 23:34:47] SYSTEM: Account 1002 created with initial deposit of 10000. {owner:1002}
[2025-11-11 23:36:10] TRANSFER: 2000 from account 1001 to 1002. {owner:1001}
[2025-11-11 23:36:10] TRANSFER: 2000 from account 1001 to 1002. {owner:1002}
[2025-11-12 00:11:54] SYSTEM: Account 1002 was deactivated. {owner:1002}
```

Figure 16. Bank Output, Admin Mode Deletion of Account and Viewing of Transaction Log

This figure presents the two most important administrative oversight functions: account deletion and log viewing. It first depicts the "soft delete" process for an account, which requires password confirmation and then successfully marks the account as deactivated. Next, it displays the Full Transaction Log to demonstrate that an immutable audit trail of all activities is maintained by the system, including creation and deletion of accounts.

CONCLUSION

The project successfully demonstrated the design and implementation of an efficient and persistent banking system simulation. By leveraging fundamental data structures and modern software engineering principles, the application effectively models core banking operations, including account management, financial transactions, and administrative oversight. The primary objective, which is to create a functional system with a strong emphasis on data integrity and persistence, was successfully achieved.

The architectural design centered on a strategic application of data structures: a std::vector of std::unique_ptrs provided a flexible and memory-safe container for managing Account objects. A std::queue effectively modeled a fair, first-in-first-out customer service line. The custom-built linked list, implemented as a TransactionStack, offered an efficient LIFO mechanism for recording an immutable history of transactions. The integration of these structures within an object-oriented framework allowed for clear separation of concerns and high code cohesion. Furthermore, the disciplined use of modern features, such as using the <memory> library, proved critical in developing a stable application that prevents memory leaks and manages runtime errors gracefully. The file I/O system, utilizing fstream and sstream, ensures that the application state is preserved across sessions, providing the necessary data persistence for a practical simulation.

In conclusion, this project serves as a comprehensive practical study in applying computer science fundamentals to a real-world problem. It validates the efficacy of classic data structures in their intended roles and highlights the value of modern C++ structures and concepts in building software that is not only functional but also safe, robust, and maintainable.

Github Link: ggacabrera-bit/Bank-Simulator: Final Project for Data Structures and Algorithms
Video Demonstration: Project Demo.mkv

REFERENCES

Burns, J. (2023, November 29)*. Three critical challenges for bank audit committees. Bank Director. https://www.bankdirector.com/article/three-critical-challenges-bank-audit-committees/*

Cohen, K. J., & Heames, J. T. (n.d.)*. FDIC bank management simulation. Association for Computing Machinery. https://dl.acm.org/doi/pdf/10.1145/800196.805989*

Derrico, H. (2014, October 28). *BB&T: Training methods for bank tellers. https://hderrico.wordpress.com/2014/10/28/bbt-training-methods-for-bank-tellers/*

Hurix Digital. (2024, October 22). *Business simulation training for superior decision-making in finance. https://www.hurix.com/blogs/how-does-business-simulation-training-drive-superior-outcomes-in-banking-finance/*

IvyPanda. (2023, December 20). *Training methods for bank tellers - 848 words* | *Report example. https://ivypanda.com/essays/training-methods-for-bank-tellers-report/*

Penceo eLearning Provider. *(2024, December 16). Common pitfalls in online retail training and how to fix them. https://www.penceo.com/blog-elearning-platform/common-pitfalls-in-online-retail-training-and-how-to-fix-them*

Shivam. (2025, June 13). *Top skills you'll gain from a banking course after graduation. Medium. https://medium.com/@shivam.g3443/top-skills-youll-gain-from-a-banking-course-after-graduation-f55053c53e41*

Simarch. (n.d.). *Banking simulations, games, learning solutions. https://www2.simarch.com/*

Zambianchi, L., & Genovese, S. (2024). Bridging the gap between theoretical learning and practical application: A qualitative study in the Italian educational context. *Education Sciences, 14(2), 198. https://www.mdpi.com/2227-7102/14/2/198*