BANKING SYSTEM

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

LOHITH (AP23110010951)

JATHIN (AP23110010943)

SAI KUMAR (AP23110010954)

GOPI (AP23110010994)

SAI SRAVAN (AP23110010956)



**SRM University–AP**

**Neerukonda, Mangalagiri, Guntur Andhra Pradesh – 522 240 [November, 2023]**

**Certificate**

Date: November-2024

This is to certify that the work present in this Project entitled “BANKING SYSTEM” has been carried out by group-5 under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology in School of Engineering and Sciences.

**Supervisor**

Prof. / Dr. Aurobindo Behera.

Assistance Professor.

Department of Computer Science.

Acknowledgements

We would like to express our gratitude Dr. Aurobindo Behera for assisting us with the research at every step. Without the advice and assistance of our adviser, this study effort would not be feasible. The suggestions, feedbacks, and encouragement from our advisors were extremely helpful in assisting us in finishing this research.

**Abstract**

This report details the development of a bank account management system using C++ object-oriented programming (OOP) concepts. The program simulates essential banking functionalities, including account creation, deposits, withdrawals, and balance display, all structured using OOP principles for modular and secure data handling. This application emphasizes encapsulation, data abstraction, and user accessibility while ensuring secure operations and efficient management of customer information and account balances.

**Introduction**

The banking system program serves as a simple model for managing multiple customer accounts and transactions, demonstrating the practical applications of C++ OOP concepts. It involves three core classes: Account, Customer, and Bank. Each class encapsulates specific properties and behaviors related to a banking environment. The Bank class manages customer creation and account setup, while Customer and Account classes handle individual details and transactions. The application allows for secure deposits, withdrawals, and balance inquiries, showcasing C++'s utility in real-world applications.

**1.1 Core Class**

The primary classes within the program are:

1. **Account Class**: Manages individual account details, including account number and balance. It provides methods to deposit and withdraw funds and ensures that transactions are securely processed with accurate balance tracking.
2. **Customer Class**: Maintains customer-related data, including customer name and ID. It also associates multiple accounts with a single customer, allowing for extended functionality in managing customer-specific accounts.
3. **Bank Class**: Acts as the main interface, coordinating between customers and their accounts. It initializes accounts and handles customer registration, functioning as the hub for all banking operations.

**1.2 Key Features**

The program highlights several key features:

* **Encapsulation**: Each class has clearly defined private data members, which are only accessible through public methods. This ensures data integrity and security.
* **Data Abstraction**: By providing simplified interfaces (deposit, withdraw, display info), the program hides underlying complexities, allowing users to interact without needing technical details of implementation.
* **Modularity**: By dividing the program into classes, it supports independent functionality within each component, making the codebase easier to understand, modify, and expand.

**1.3 Security Measures**

Security measures within the program include:

* **Private Access Specifiers**:

Account balance and customer data are private members, restricting access solely to member functions, which prevents unauthorized manipulation of data.

* **Balance Validation in Withdrawals**:

The program checks for sufficient balance before processing withdrawals, preventing negative balances and overdrafts.

* **Input Validation**:

Deposits and withdrawals are validated to ensure only positive amounts are processed. This guards against incorrect data entries and ensures the integrity of account transactions.

**2. Methodology**

The methodology involves structuring the program in stages:

1. **Class Design**: Each class (Account, Customer, and Bank) was designed to encapsulate its own set of related properties and methods, following OOP principles for clear responsibility division.
2. **Function Implementation**: Core functionalities like deposit, withdraw, and displayAccountInfo were implemented to handle transactions securely and ensure accurate balance updates.
3. **Menu-Driven Interface**: A loop-based menu allows users to perform transactions interactively. This enables dynamic responses and continuous operations until the user opts to exit.
4. **Testing and Validation**: Each method was tested with various inputs to ensure accurate results, proper error handling, and adherence to the program's secure banking requirements.

**2.1 Modular Design**

The program's modular design divides functionality across three main classes: Account, Customer, and Bank. Each class encapsulates distinct features, enabling organized code, easier debugging, and the potential for future modifications. This design not only promotes code readability but also aligns with key principles of Object-Oriented Programming (OOP) like encapsulation and abstraction, which protect data and simplify program structure.

**Account Class**

The Account class represents a single bank account and handles account-specific operations such as deposits and withdrawals. Key features of this class include:

* **Private Data Members**:

The accountNumber and balance are private, protecting sensitive account information from unauthorized access or modification outside the class.

* **Parameterized Constructor**:

Initializes accountNumber and balance when an Account object is created, ensuring each account has a unique identity and starting balance.

* **Public Member Functions**:
  + deposit(double amount): Adds a specified amount to the account balance if it is positive, ensuring valid deposits. Outputs a message confirming the deposit.
  + withdraw(double amount): Deducts the specified amount from the balance if sufficient funds are available, preventing overdrafts. Provides feedback to the user in case of insufficient funds.
  + displayAccountInfo(): Outputs the account number and current balance, allowing the user to check account details at any time.

**Customer Class**

The Customer class manages information related to individual customers, allowing for multiple accounts to be associated with a single customer. This structure facilitates the handling of multiple bank accounts for each customer in a cohesive manner.

* **Private Data Members**:

The name and customerID uniquely identify each customer, while accounts (a vector of Account objects) stores all accounts associated with the customer.

* **Parameterized Constructor**:

Sets the customer’s name and assigns a unique customerID upon instantiation, ensuring each customer is individually identifiable.

* **Public Member Functions**:
  + addAccount(const Account& account): Adds a new Account object to the accounts vector, allowing customers to open multiple accounts.
  + getAccount(int accountNumber): Searches for an account by its account number and returns a pointer to the account if found, facilitating access to specific accounts.
  + displayCustomerInfo(): Outputs the customer’s details, including each associated account and its current balance, allowing users to view their complete account portfolio.

**Bank Class**

The Bank class is the main controller, handling the creation of customers and accounts, and providing the primary interface for user interaction. This class brings together the functionality of both Account and Customer, managing data flow and overseeing transaction operations.

* **Private Data Members**:

The bank Name provides a unique identifier for the bank, while customers (a vector of Customer objects) stores customer data. nextAccountNumber and nextCustomerID are used to assign unique identifiers for accounts and customers.

* **Public Member Functions**:
  + addCustomer(const string& name): Registers a new customer with the bank by creating a Customer object and assigning a unique customerID.
  + openAccount(Customer& customer, double initialBalance): Opens a new account for an existing customer, assigning a unique accountNumber and initializing the account with the specified balance.
  + getCustomer(int customerID): Retrieves a customer by their ID, enabling access to specific customer data for operations like transactions and information display.

**Main Program Workflow**

* In the main function, these methods come together to simulate basic banking operations. The displayMenu function provides users with options to deposit, withdraw, and view account information.

3. Discussion

3.1 INPUT OF THE CODE

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class Account {

private:

int accountNumber;

double balance;

public:

Account(int accNum, double initialBalance)

: accountNumber(accNum), balance(initialBalance) {}

int getAccountNumber() const { return accountNumber; }

double getBalance() const { return balance; }

void deposit(double amount) {

if (amount > 0) {

balance += amount;

cout << "Deposited $" << amount << " to account " << accountNumber << endl;

cout << "Current balance: $" << balance << endl; // Display balance after deposit

} else {

cout << "Invalid deposit amount.\n";

}

}

bool withdraw(double amount) {

if (amount <= balance) {

balance -= amount;

cout << "Withdrew $" << amount << " from account " << accountNumber << endl;

cout << "Current balance: $" << balance << endl; // Display balance after withdrawal

return true;

} else {

cout << "Insufficient funds for withdrawal.\n";

return false;

}

}

void displayAccountInfo() const {

cout << "Account Number: " << accountNumber << ", Balance: $" << balance << endl;

}

};

class Customer {

private:

string name;

int customerID;

vector<Account> accounts;

public:

Customer(string name, int customerID)

: name(name), customerID(customerID) {}

string getName() const { return name; }

int getCustomerID() const { return customerID; }

void addAccount(const Account& account) {

accounts.push\_back(account);

cout << "Account created for " << name << " with Account Number: " << account.getAccountNumber() << endl;

}

Account\* getAccount(int accountNumber) {

for (auto& account : accounts) {

if (account.getAccountNumber() == accountNumber) {

return &account;

}

}

return nullptr;

}

void displayCustomerInfo() const {

cout << "Customer ID: " << customerID << ", Name: " << name << endl;

for (const auto& account : accounts) {

account.displayAccountInfo();

}

}

};

class Bank {

private:

string bankName;

vector<Customer> customers;

int nextAccountNumber;

int nextCustomerID;

public:

Bank(string name)

: bankName(name), nextAccountNumber(1001), nextCustomerID(1) {}

Customer\* addCustomer(const string& name) {

customers.emplace\_back(name, nextCustomerID++);

cout << "Customer " << name << " added with Customer ID: " << nextCustomerID - 1 << endl;

return &customers.back();

}

Account\* openAccount(Customer& customer, double initialBalance) {

int accountNumber = nextAccountNumber++;

customer.addAccount(Account(accountNumber, initialBalance));

return customer.getAccount(accountNumber);

}

Customer\* getCustomer(int customerID) {

for (auto& customer : customers) {

if (customer.getCustomerID() == customerID) {

return &customer;

}

}

cout << "Customer ID not found.\n";

return nullptr;

}

};

// Function to print a box around menu options

void displayMenu() {

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout << "\* MENU \*\n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout << "\* 1. Deposit \*\n";

cout << "\* 2. Withdraw \*\n";

cout << "\* 3. Display Account Info \*\n";

cout << "\* 4. Exit \*\n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

}

int main() {

Bank bank("OpenAI Bank");

// Read customer name

string name;

cout << "Enter customer name: ";

getline(cin, name);

// Add customer to the bank

Customer\* customer = bank.addCustomer(name);

// Read initial balance for the new account

double initialBalance;

cout << "Enter initial balance for the new account: ";

cin >> initialBalance;

// Open an account for the customer with the initial balance

Account\* account = bank.openAccount(\*customer, initialBalance);

// Perform a few transactions with user input

int choice;

do {

displayMenu(); // Display the menu with a box around it

cout << "Choose an option: ";

cin >> choice;

switch (choice) {

case 1: { // Deposit

double amount;

cout << "Enter amount to deposit: ";

cin >> amount;

account->deposit(amount);

break;

}

case 2: { // Withdraw

double amount;

cout << "Enter amount to withdraw: ";

cin >> amount;

account->withdraw(amount);

break;

}

case 3: // Display Account Information

customer->displayCustomerInfo();

break;

case 4: // Exit

cout << "Exiting...\n";

break;

default:

cout << "Invalid option. Please try again.\n";

break;

}

} while (choice != 4);

return 0;

}

3.2. Output of code

Enter customer name: alice

Customer alice added with Customer ID: 1

Enter initial balance for the new account: 1000

Account created for alice with Account Number: 1001

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* MENU \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 1. Deposit \*

\* 2. Withdraw \*

\* 3. Display Account Info \*

\* 4. Exit \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Choose an option: 1

Enter amount to deposit: 200

Deposited $200 to account 1001

Current balance: $1200

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* MENU \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 1. Deposit \*

\* 2. Withdraw \*

\* 3. Display Account Info \*

\* 4. Exit \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Choose an option: 2

Enter amount to withdraw: 1500

Insufficient funds for withdrawal.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* MENU \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 1. Deposit \*

\* 2. Withdraw \*

\* 3. Display Account Info \*

\* 4. Exit \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Choose an option: 3

Customer ID: 1, Name: alice

Account Number: 1001, Balance: $1200

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* MENU \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 1. Deposit \*

\* 2. Withdraw \*

\* 3. Display Account Info \*

\* 4. Exit \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Choose an option: 4

Exiting...

**4. Concluding Remarks**

This banking system program in C++ demonstrates object-oriented principles such as encapsulation, modularity, and class interaction to create a streamlined and secure user experience. By structuring the code around Bank, Customer, and Account classes, the program is organized, making it easier to add features and maintain. The program also provides clear and user-friendly interactions for deposits, withdrawals, and account information display, improving usability.

Furthermore, the code incorporates basic error handling, which prevents invalid transactions (like overdrawing an account) and ensures that only valid inputs are processed. While this implementation is suitable for a basic simulation, it also opens the door for potential future enhancements, such as adding transaction histories, implementing authentication, and connecting to a database for real-world applications.

In summary, this project illustrates a practical approach to designing a simple, secure, and modular banking system, emphasizing object-oriented concepts and highlighting the potential for expansion in real-world applications.

**References**

1. ***The C++ Programming Language* by Bjarne Stroustrup**  
   This book, written by the creator of C++, is an authoritative resource on C++ language features and design principles, covering core language concepts, advanced features, and best practices.
2. ***Effective C++* by Scott Meyers**  
   A well-regarded book that provides practical advice on writing efficient, reliable, and maintainable C++ code. This book is especially useful for learning about object-oriented programming best practices and effective class design.
3. ***C++ Primer* by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo**  
   This introductory yet comprehensive book on C++ covers both fundamental and advanced concepts, including object-oriented principles like encapsulation and inheritance. It is widely used for learning C++ with a focus on clarity and depth.
4. ***Object-Oriented Programming in C++* by Robert Lafore**  
   This book provides a detailed exploration of object-oriented programming concepts in C++, including encapsulation, inheritance, and polymorphism, which were central to the design of this banking application.
5. ***C++ How to Program* by Paul Deitel and Harvey Deitel**  
   Known for its practical approach to teaching C++ through real-world examples and exercises, this book covers essential C++ programming constructs, object-oriented design, and modular programming, all of which informed the development of this program.