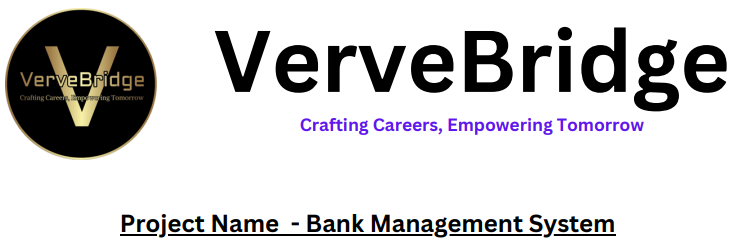
****

**A PROJECT REPORT**

**Submitted By :- SURAJ KUMAR AGRAHARI**

**Date :- 10-08-2024**

**TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| **S.No** | **TOPIC** | **PAGE NO** |
| 1 | Introduction | 1-2 |
| 2 | Objective | 3-4 |
| 3 | Tools/Environment | 5 |
| 4 | Report on Bank management system | 6-9 |
| 5 | Program code | 10-22 |
| 6 | Limitations of the Project | 23-24 |
| 7 | Future Applications of the Project | 24-27 |
| 8 | Bibliography |  |

**INTRODUCTION**

**Introduction to the Bank Management System:**

The Bank Management System (BMS) is a simple yet effective application designed to manage basic banking operations. This system provides functionalities to create new bank accounts, authenticate users, and perform transactions like deposits and withdrawals. It also allows users to view their account details and transaction history, and close their accounts when no longer needed.

**Key Features**

1. **Account Creation** : Users can create a new bank account by providing personal details, choosing an account type (saving or current), setting a password, and making an initial deposit.

2. **Authentication** : To ensure security, users must authenticate themselves by entering their password before performing any operations on their account.

3. **Deposit and Withdrawal** : Users can deposit money into their account or withdraw funds. The system updates the balance accordingly and records each transaction.

4. **Account Information** : Users can view their account details, including their balance and transaction history. This helps in keeping track of financial activities.

5. **Account Closure** : If a user wishes to close their account, the system will delete all associated data files and terminate the account.

**System Components**

* BankAccount Class : This class handles all operations related to individual bank accounts. It includes methods for creating an account, authenticating users, performing transactions, and saving/loading account data from files.
* File Operations : Account information and transaction history are stored in text files, which allows for persistent data storage even after the application is closed.
* Main Function : The main function provides a user-friendly menu for interacting with the system. Users can choose different options to manage their bank accounts.

This introduction gives a broad overview of what the Bank Management System is designed to do and outlines its core features.

**OBJECTIVES**

The Bank Management System (BMS) is designed to meet several key objectives that focus on providing users with efficient and secure management of their bank accounts. Here are the main objectives:

1. Efficient Account Management

* Objective : Enable users to easily create, manage, and close their bank accounts.
* Details : Users can open new accounts with personal details and an initial deposit, modify their account details, and close their accounts when needed.

2. Secure Authentication

* Objective : Ensure that only authorized users can access their accounts and

perform transactions.

* Details : Implement a secure password-based authentication mechanism to verify user identity before allowing access to account functionalities.

3. Transaction Handling

* Objective : Facilitate financial transactions including deposits and withdrawals.
* Details : Users should be able to deposit money into their accounts, withdraw funds, and receive immediate feedback on their updated balance. Each transaction should be recorded for future reference.

4. Account and Transaction Record Keeping

* Objective : Maintain a record of account details and transaction history for users.
* Details : Store account information and transaction history in files to allow users to view their account status and past transactions at any time.

5. User-Friendly Interface

* Objective : Provide an intuitive and easy-to-navigate interface for users to interact with the system.
* Details : Design a simple text-based menu that allows users to select options and perform operations with minimal confusion.

6. Data Persistence

* Objective : Ensure that account data and transaction history are preserved across sessions.
* Details : Use file-based storage to save account information and transaction history, enabling users to access their data even after closing and reopening the application.

7. Error Handling and Validation

* Objective : Implement robust error handling and input validation to maintain system stability and prevent invalid operations.
* Details : Check for invalid input values, handle file operation errors gracefully, and provide clear messages to guide users.

8. Account Closure

* Objective : Allow users to close their accounts and delete associated data when they no longer need the account.
* Details : Provide a secure way to close accounts and ensure that all related files are properly deleted to maintain privacy and data integrity.

**TOOLS AND ENVIORNMENT**

**HARDWARE REQUIREMENTS**

**Processor:** Minimum Pentium IV 2.4 GHZ

**RAM:** At Least 100 MB

**Disk Space:** At Least 500 MB

# **SOFTWARE REQUIREMENTS**

**Operating System:** Windows,IOS,LINUX,Etc.

**Code Compiler :** Visual Code Studio / Dev C++/ Turbo C++/Etc.

**Report on Bank Management System**

1. **Overview** : The provided C++ program is designed to manage bank accounts, allowing users to create accounts, deposit and withdraw money, display account details, and close accounts. It tracks transactions associated with each account and stores them in separate files. The program operates through a console-based user interface.

2. **Functionalities** :

The program provides the following functionalities:

1. Create a New Account : Collects user information such as name, address, account type, password, and initial deposit. Saves this information to a file named after the account holder.

2. Select an Account : Allows users to select an account from a list of accounts for further operations.

3. Deposit Money : Allows the authenticated user to deposit money into the selected account and records the transaction.

4. Withdraw Money : Allows the authenticated user to withdraw money from the selected account if sufficient funds are available and records the transaction.

5. Display Account : Displays the account details including transaction history.

6. Close Account : Deletes the account data and transaction history files and exits the program.

3. **Code Structure and Design :**

Class Design

- `BankAccount` class handles all account-related operations.

Data Members:

- `string name`: Account holder's name.

- `string address`: Account holder's address.

- `char accountType`: Type of account ('s' for saving, 'c' for current).

- `int balance`: Current balance.

- `string password`: Account password.

- `vector<string> transactions`: Stores transaction history.

Member Functions :

- `void loadAccountData()`: Loads account data from a file.

- `void saveAccountData()`: Saves account data to a file.

- `void addTransaction(const string&)`: Adds a transaction to the transaction history.

- `void deleteAccountFiles()`: Deletes account and transaction history files.

- `void createAccount()`: Collects data and creates a new account.

- `bool authenticate() const`: Authenticates the user by comparing the entered password with the stored one.

- `void depositMoney()`: Allows the user to deposit money.

- `void withdrawMoney()`: Allows the user to withdraw money.

- `void displayAccount() const`: Displays account details and transaction history.

- `void closeAccount()`: Closes the account and deletes related files.

Main Function

- Manages the menu-driven interface.

- Handles user inputs and calls appropriate methods on `BankAccount` objects.

- Uses a vector to store multiple `BankAccount` instances.

4. **Code Quality and Improvements :**

Strengths :

1. Modularity : The program is modular with clear separation of concerns in the `BankAccount` class.

2. Transaction Tracking : Maintains a history of transactions, which is crucial for a banking application.

3. File Handling : Uses files to persist account and transaction data.

Areas for Improvement :

1. String Handling : Replace fixed-size character arrays with `std::string` to avoid buffer overflow and improve readability.

2. Security : Passwords are stored and compared in plaintext. Implement hashing algorithms (e.g., bcrypt) for secure password handling.

3. Error Handling : Improve error handling for file operations and user inputs.

4. User Experience : Enhance user experience by providing more detailed error messages and instructions.

5. Graceful Termination : Avoid using `exit(0)` abruptly; instead, allow for a more graceful exit.

6. File Management : Ensure proper handling of file I/O errors and check if files exist before attempting to delete them.

5. **Conclusion :**

The C++ bank account management program provides a functional and simple console-based system for managing bank accounts. It effectively handles account creation, balance management, and transaction tracking. However, there are several areas for improvement, particularly concerning security, error handling, and modern C++ practices. Adopting these improvements will enhance the robustness, security, and maintainability of the application.

**BANK MANAGEMENT SYSTEM C++ CODE**

#include <iostream>

#include <fstream>

#include <cstring>

#include <vector>

#include <cstdlib>

#include <sstream>

using namespace std;

// Class definition for BankAccount

class BankAccount {

char name[100], address[100], accountType; // Basic account information

int balance; // Current balance of the account

char password[50]; // Account password

vector<string> transactions; // To track transactions

// Private methods for account management

void loadAccountData(); // Load account data from file

void saveAccountData(); // Save account data to file

void addTransaction(const string& transaction); // Add a transaction to the record

void deleteAccountFiles(); // Delete account data files

public:

// Constructor initializes balance to 0

BankAccount() : balance(0) {}

// Public methods for interacting with the account

void createAccount(); // Create a new account

bool authenticate(); // Authenticate user with password

void depositMoney(); // Deposit money into the account

void withdrawMoney(); // Withdraw money from the account

void displayAccount(); // Display account details

void closeAccount(); // Close the account and delete files

// Getters and setters for account attributes

const char\* getName() const { return name; }

const char\* getPassword() const { return password; }

void setName(const char\* n) { strcpy(name, n); }

void setPassword(const char\* p) { strcpy(password, p); }

void setAddress(const char\* a) { strcpy(address, a); }

void setAccountType(char at) { accountType = at; }

void setBalance(int bal) { balance = bal; }

int getBalance() const { return balance; }

void setTransactions(const vector<string>& trans) { transactions = trans; }

};

// Load account data from file

void BankAccount::loadAccountData() {

ifstream file(name); // Open file named after the account holder's name

if (file) {

file.getline(name, 100); // Read name

file.getline(address, 100); // Read address

file >> accountType >> balance; // Read account type and balance

file.ignore(); // Ignore the newline after balance

file.getline(password, 50); // Read password

file.close();

} else {

// Initialize with default values if file does not exist

strcpy(name, "");

strcpy(address, "");

accountType = 's'; // Default to saving account

balance = 0;

strcpy(password, "");

}

}

// Save account data to file

void BankAccount::saveAccountData() {

ofstream file(name); // Open file named after the account holder's name

if (file) {

file << name << endl;

file << address << endl;

file << accountType << endl;

file << balance << endl;

file << password << endl;

file.close();

} else {

cerr << "Error: Unable to save account information for " << name << endl;

}

}

// Add a transaction to the transaction history

void BankAccount::addTransaction(const string& transaction) {

transactions.push\_back(transaction); // Store transaction in vector

ofstream transFile(string(name) + "\_transactions.txt", ios::app); // Open transaction file in append mode

if (transFile) {

transFile << transaction << endl;

transFile.close();

} else {

cerr << "Error: Unable to save transaction for " << name << endl;

}

}

// Delete account and transaction files

void BankAccount::deleteAccountFiles() {

if (remove(name) != 0) { // Remove account data file

cerr << "Error deleting account file for " << name << endl;

}

if (remove((string(name) + "\_transactions.txt").c\_str()) != 0) { // Remove transaction file

cerr << "Error deleting transactions file for " << name << endl;

}

}

// Create a new account

void BankAccount::createAccount() {

cout << "Enter your full name: ";

cin.ignore(); // Ignore newline from previous input

cin.getline(name, 100); // Read account holder's name

cout << "Enter your address: ";

cin.getline(address, 100); // Read address

cout << "What type of account do you want to open (s for saving, c for current): ";

cin >> accountType; // Read account type

cout << "Set your account password: ";

cin >> password; // Set account password

cout << "Enter initial deposit amount: ";

cin >> balance; // Set initial deposit amount

addTransaction("Account opened with initial deposit: " + to\_string(balance)); // Record the initial deposit

saveAccountData(); // Save account details to file

cout << "Your account is created successfully!" << endl;

}

// Authenticate user by checking password

bool BankAccount::authenticate() {

char enteredPassword[50];

cout << "Enter your account password to authenticate: ";

cin >> enteredPassword; // Read entered password

if (strcmp(password, enteredPassword) == 0) {

return true; // Password matches

} else {

cout << "Authentication failed. Incorrect password." << endl;

return false; // Password does not match

}

}

// Deposit money into the account

void BankAccount::depositMoney() {

int depositAmount;

cout << "Enter the amount you want to deposit: ";

cin >> depositAmount; // Read deposit amount

if (depositAmount < 0) {

cout << "Invalid amount. Please enter a positive value." << endl;

return;

}

balance += depositAmount; // Update balance

addTransaction("Deposited: " + to\_string(depositAmount)); // Record deposit

saveAccountData(); // Save updated details

cout << "Your total balance is now: " << balance << endl;

}

// Withdraw money from the account

void BankAccount::withdrawMoney() {

int withdrawAmount;

cout << "Enter the amount you want to withdraw: ";

cin >> withdrawAmount; // Read withdrawal amount

if (withdrawAmount < 0) {

cout << "Invalid amount. Please enter a positive value." << endl;

return;

}

if (withdrawAmount > balance) {

cout << "Insufficient funds. Your current balance is: " << balance << endl;

return;

}

balance -= withdrawAmount; // Update balance

addTransaction("Withdrew: " + to\_string(withdrawAmount)); // Record withdrawal

saveAccountData(); // Save updated details

cout << "Your remaining balance is: " << balance << endl;

}

// Display account details and transaction history

void BankAccount::displayAccount() {

cout << "Name: " << name << endl;

cout << "Address: " << address << endl;

cout << "Account Type: " << (accountType == 's' ? "Saving" : "Current") << endl;

cout << "Balance: " << balance << endl;

cout << "Transaction History:" << endl;

ifstream transFile(string(name) + "\_transactions.txt"); // Open transaction file

if (transFile) {

string transaction;

while (getline(transFile, transaction)) {

cout << transaction << endl; // Display each transaction

}

transFile.close();

} else {

cout << "No transactions found." << endl;

}

}

// Close the account and delete files

void BankAccount::closeAccount() {

char confirm;

cout << "Are you sure you want to close your account? (y/n): ";

cin >> confirm; // Read confirmation

if (confirm == 'y' || confirm == 'Y') {

deleteAccountFiles(); // Delete account and transaction files

cout << "Account closed successfully." << endl;

exit(0); // Terminate the program since the account is closed

} else {

cout << "Account closure canceled." << endl;

}

}

// Main function to interact with user

int main() {

vector<BankAccount> accounts; // List of accounts

int choice;

char continueChoice;

int selectedAccountIndex = -1; // Index of selected account

do {

// System clear command is commented out for portability

// system("cls");

cout << "01) Create a new account" << endl;

cout << "02) Select an account" << endl;

cout << "03) Deposit money" << endl;

cout << "04) Withdraw money" << endl;

cout << "05) Display account" << endl;

cout << "06) Close account" << endl;

cout << "07) Exit" << endl;

cout << "Please select an option: ";

cin >> choice; // Read user choice

switch (choice) {

case 1: {

BankAccount newAccount;

newAccount.createAccount(); // Create a new account

accounts.push\_back(newAccount); // Add to list of accounts

break;

}

case 2: {

cout << "Select an account by index (0 to " << accounts.size() - 1 << "): ";

cin >> selectedAccountIndex; // Select an account

if (selectedAccountIndex < 0 || selectedAccountIndex >= accounts.size()) {

cout << "Invalid index. Please try again." << endl;

selectedAccountIndex = -1;

}

break;

}

case 3:

if (selectedAccountIndex >= 0 && selectedAccountIndex < accounts.size()) {

if (accounts[selectedAccountIndex].authenticate()) {

accounts[selectedAccountIndex].depositMoney(); // Deposit money into selected account

}

} else {

cout << "No account selected. Please select an account first." << endl;

}

break;

case 4:

if (selectedAccountIndex >= 0 && selectedAccountIndex < accounts.size()) {

if (accounts[selectedAccountIndex].authenticate()) {

accounts[selectedAccountIndex].withdrawMoney(); // Withdraw money from selected account

}

} else {

cout << "No account selected. Please select an account first." << endl;

}

break;

case 5:

if (selectedAccountIndex >= 0 && selectedAccountIndex < accounts.size()) {

if (accounts[selectedAccountIndex].authenticate()) {

accounts[selectedAccountIndex].displayAccount(); // Display selected account details

}

} else {

cout << "No account selected. Please select an account first." << endl;

}

break;

case 6:

if (selectedAccountIndex >= 0 && selectedAccountIndex < accounts.size()) {

accounts[selectedAccountIndex].closeAccount(); // Close selected account

accounts.erase(accounts.begin() + selectedAccountIndex); // Remove from list

selectedAccountIndex = -1; // Reset selected account index

} else {

cout << "No account selected. Please select an account first." << endl;

}

break;

case 7:

cout << "Exiting..." << endl;

break;

default:

cout << "Invalid choice, please try again." << endl;

}

if (choice != 7) {

cout << "\nDo you want to continue? (y/n): ";

cin >> continueChoice; // Ask if user wants to continue

} else {

continueChoice = 'n'; // Exit the loop if user chooses to exit

}

} while (continueChoice == 'y' || continueChoice == 'Y'); // Repeat until user chooses to exit

return 0;

}

**Limitations of the Project**

While the Bank Management System (BMS) provides fundamental functionalities for managing bank accounts, there are several limitations to be aware of:

#### 1. ****Basic Security Measures****

* **Limitation**: The system uses simple password-based authentication without advanced security measures.
* **Details**: Passwords are stored and compared as plain text, which is insecure. There is no support for encryption or multi-factor authentication.

#### 2. ****File-Based Storage****

* **Limitation**: Data is stored in text files, which can be prone to corruption and are not optimized for concurrent access.
* **Details**: This method may lead to data loss or inconsistencies, especially if multiple instances of the program are run simultaneously or if there are file system issues.

#### 3. ****Limited Error Handling****

* **Limitation**: Error handling is minimal and does not cover all potential failure points.
* **Details**: The system may not handle unexpected errors gracefully, such as file read/write errors or invalid input formats, which could lead to program crashes or incorrect data handling.

#### 4. ****No User Interface Beyond CLI****

* **Limitation**: The system operates through a text-based command-line interface (CLI).
* **Details**: There is no graphical user interface (GUI), which can be less user-friendly and might not be suitable for all users.

#### 5. ****No Support for Multiple Users****

* **Limitation**: The system does not support simultaneous multi-user access.
* **Details**: Each instance of the program operates independently, which means it cannot handle concurrent transactions or interactions from multiple users effectively.

#### 6. ****Limited Account Types and Features****

* **Limitation**: The system only supports basic savings and current account types with limited features.
* **Details**: There are no options for more complex account types or additional banking services such as loans, interest calculations, or investment management.

#### 7. ****No Data Backup or Recovery****

* **Limitation**: There is no built-in mechanism for data backup or recovery.
* **Details**: If account or transaction files are lost or corrupted, there is no way to recover the lost data, which could lead to permanent data loss.

#### 8. ****No Real-Time Updates****

* **Limitation**: Transactions and account status are updated only when the user performs an action.
* **Details**: There is no real-time monitoring or notification system for account changes, which might limit the ability to track and manage accounts dynamically.

#### 9. ****Lack of Scalability****

* **Limitation**: The system may not handle a large number of accounts or transactions efficiently.
* **Details**: As the number of accounts and transactions grows, file-based storage and processing may become less efficient, leading to potential performance issues.

#### 10. ****No External Integration****

* **Limitation**: The system does not integrate with external systems or services.

**FUTURE APPLICATION OF THE PROJECT**

The Bank Management System (BMS) you've created is a foundational tool for managing bank accounts. It can be evolved and extended in several ways to enhance its functionality and usability. Here are some potential future applications and improvements:

### 1. ****Enhanced Security Features****

* **Encryption**: Implement encryption for storing passwords and sensitive account information to improve security.
* **Multi-Factor Authentication (MFA)**: Introduce multi-factor authentication to strengthen the login process and protect user accounts from unauthorized access.
* **Secure Communication**: Use secure communication protocols for data exchange to protect against interception and tampering.

### 2. ****Graphical User Interface (GUI)****

* **Desktop Application**: Develop a GUI-based desktop application using frameworks like Qt or GTK to make the system more user-friendly.
* **Web Application**: Create a web-based interface using technologies like HTML, CSS, and JavaScript to provide access from any device with a web browser.
* **Mobile Application**: Develop mobile apps for Android and iOS to allow users to manage their accounts on the go.

### 3. ****Advanced Banking Features****

* **Interest Calculation**: Add support for calculating and applying interest to savings accounts.
* **Loan Management**: Introduce features for managing loans, including application, approval, repayment, and tracking.
* **Investment Services**: Offer features for managing investments and tracking their performance.

### 4. ****Integration with External Systems****

* **API Integration**: Develop APIs to integrate with other financial systems, third-party services, or banking networks for enhanced functionality.
* **Banking Network Connectivity**: Integrate with existing banking networks to allow for real-time transactions and account management.

### 5. ****Data Management and Analytics****

* **Data Backup and Recovery**: Implement robust backup and recovery solutions to protect against data loss.
* **Analytics and Reporting**: Provide tools for generating reports and analyzing transaction data to help users make informed financial decisions.

### 6. ****Real-Time Updates and Notifications****

* **Push Notifications**: Implement push notifications to alert users about account activities, transaction statuses, or security issues.
* **Real-Time Monitoring**: Introduce real-time monitoring of transactions and account activities for better visibility and management.

### 7. ****Scalability and Performance Enhancements****

* **Database Integration**: Replace file-based storage with a relational database management system (RDBMS) or NoSQL database to handle larger volumes of data efficiently.
* **Performance Optimization**: Optimize the system to handle a high number of concurrent users and transactions without performance degradation.

### 8. ****User Personalization and Customization****

* **User Profiles**: Allow users to customize their profiles with additional information and preferences.
* **Personalized Features**: Offer personalized recommendations and financial insights based on user behavior and account activities.

### 9. ****Regulatory Compliance****

* **Compliance with Standards**: Ensure the system complies with financial regulations and standards, such as GDPR for data protection and PCI-DSS for payment security.
* **Audit Trails**: Implement features for maintaining audit trails and logging user activities for compliance and security purposes.

### 10. ****Educational and Support Resources****

* **User Guides and Tutorials**: Provide comprehensive user guides, tutorials, and FAQs to help users understand and utilize the system effectively.
* **Customer Support**: Implement a support system for users to get assistance with issues or questions related to their accounts.

### 11. ****Artificial Intelligence and Machine Learning****

* **Fraud Detection**: Integrate AI and machine learning algorithms to detect and prevent fraudulent activities.
* **Financial Advice**: Use AI to offer personalized financial advice and budgeting tips based on user data and behavior.

These future applications can significantly enhance the functionality, usability, and security of the Bank Management System, transforming it from a basic tool into a comprehensive and modern banking solution.

**BIBLIOGRAPHY**

**Books**

* Let Us C by Yashavant Kanetkar.
* Let us C++ by Yashavant Kanetkar.
* C in Depth by S.K Srivastava.
* The C++ Programming Language By Bjarne Stroustrup.

**Websites**

* [www.google.com](http://www.google.com)
* [www.youtube.com](http://www.youtube.com)
* [www.w3schools.com](http://www.w3schools.com)
* [www.geeksforgeeks.com](http://www.geeksforgeeks.com)

