Basics Of Programming 2

Course ID: BMEVIIIAA03

Student: Dimitrija Krstev

Neptun code: KORHHK

Lab Group ID: CS16D

Lab Instructor: Vaitkus Márton

Project Title: Banking System

Type of Project: Final Project

Submission Date: 13th May 2025

1. Introduction

Managing bank accounts, transactions, and user data is a fundamental task in financial software systems. In real-world banking applications, features such as secure account creation, balance tracking, and data persistence are essential. This project aims to implement a simplified version of such a system using C++, focusing on the practical application of object-oriented programming principles, file handling, and user input validation. The key challenge addressed by this project is designing a system that can dynamically store and manage multiple types of bank accounts (checking and savings), while ensuring safety through proper error handling and input validation.

Terminology

- Bank Account - a class representing the essential properties of a user's account
- Checking Account - type of bank account that includes transaction fees and overdraft limit
- Savings Account - type of account that includes an interest rate on the balance.

---

2. Program Interface and Execution

When the program starts the user will be met with the terminal menu of the banking system. Here the user has 4 options:

1. Create Account
2. Login
3. Save Accounts Data
4. Exit

He selects the option from the Menu by typing the number before the option into the terminal. When selecting the first option (creation of an account) the user is first prompted to enter the account holder full name, this will in turn be used for logging in. Next the user is asked to enter the password which is also used when logging in. Afterwards the user gets a randomly generated ID for the account. After this the user can check what type of account he/she wants to have by typing in either checking or savings. If the user selects checking the programs asks for a transaction fee (deducted from balance for each transaction) and an overdraft limit. This is specific for the checking account. Whereas for the savings account the user is asked for an interest

rate (specific to savings accounts). After filling this the user inputs the balance in the account. This is followed by being returned to the main menu above.

Now the user can Login by selecting option 2. The user logs in with the full account holder name and the password. If there is a successful login the user is met with a new menu.

1. Deposit Money
2. Withdraw Money
3. Display Balance
4. Display Account Details
5. Check Overdraft Limit and Transaction fee (only for checking accounts)
6. Check Interest Rate (only for savings accounts)
7. Delete Account

10. Logout

Once again the user selects which option is wanted by typing the number of the choice into the terminal. All of these options are pretty self-explanatory.

The Option 3. from the main menu Save Accounts Data saves all of the accounts and their data into a .txt file which can be found in the directory of the program.

---

3. Input and Output

This section describes the format and behavior of user inputs, system outputs, and file-based data storage used by the banking system program. All user inputs are entered via the terminal in response to prompts.

- *Example: Account Creation Input:*

Enter Account holder full name: John Doe
Enter Password: 12345
Generated Account ID: xxxxxx (this is randomly generated by the program)
Enter type of Account (checking or savings): checking
Enter Transaction Fee: 5
Enter Overdraft Limit: 100
Enter Balance: 1000

- If the user wants to create a savings account, he will be asked to enter the Interest Rate instead of Transaction Fee and Overdraft Limit.

Enter Interest Rate: 2.5

Terminal Output Format:

3.1 When and if the account is successfully created the user gets a message like this in the terminal:

Creating Account…

Account Created!

3.2 When logged in the user can see a second menu with options that affect the account and can receive outputs like:

- If money is deposited via the first option, the program returns the current balance
- If money is withdrawn via the second option, the program returns either the current balance or if the funds are not sufficient it gives back an error
- Option 3 Display Account Balance outputs the current balance and past transactions that have been made
- Option 4 Display Account Details outputs the account holder name, ID number, type of account and current balance
- Option 5 and 6 show the Overdraft Limit and transaction fee (only for checking accounts) and Interest Rate (only for savings accounts)
- Option 7 Deletes the Account from memory

3.3 In the Main Menu if the user selects the third option Save Accounts Data the program makes a accounts.txt file in which all the accounts created in the session will be saved in the following format

AccountType: checking
ID: 123456
Name: John Smith
Password: examplepass
Balance: 10000
TransactionFee: 2
OverdraftLimit: 500

AccountType: savings
ID: 654321
Name: Alice Doe
Password: secure123
Balance: 15000
InterestRate: 3

3.4 Input Validation

- All numerical inputs (e.g. balance, fees) must be non-negative numbers.
- Invalid or mixed inputs (e.g. letters in numeric fields) are caught using try-catch blocks.
- The input stream is cleared on failure using cin.clear() and cin.ignore().

---

4. Program Structure

4.1 **File and Module Overview** - The source code is organized into several .cpp and .h files, separating implementation from declarations for better modularity and maintainability.

- main.cpp: Contains the entry point of the program and the main menu loop.
- Menu.h / Menu.cpp: Declares and defines the menu-related functions that manage user interaction.
- BankAccount.h / BankAccount.cpp: Declares and defines the base BankAccount class used for both checking and savings accounts.
- CheckingAccount.h / CheckingAccount.cpp: Contains the definition of the CheckingAccount class derived from BankAccount.
- SavingsAccount.h / SavingsAccount.cpp: Contains the definition of the SavingsAccount class derived from BankAccount.
- Vector.h / Vector.cpp: Implements a custom dynamic array to manage bank accounts with manual memory management.

4.2 **The Bank Account class** serves as the abstract base class for both CheckingAccount and SavingsAccount. It defines the shared properties and behaviors of all account types, such as depositing money, withdrawing funds, displaying balances, and managing transaction history.

It cannot be instantiated directly due to the pure virtual function saveToFile().

Important Members are:

- std::string accountHolderName
- std::string accountIDNumber
- std::string typeOfAcc
- double balance
- std::vector<std::string> transactionHistory
- std::string userPassword

Core Methods:

- virtual void saveToFile() const = 0 -- Pure virtual function to write account data to a file. Must be implemented by derived classes.
- void deposit(double amount) -- Adds the specified amount to the balance and records a transaction.
- virtual void withdraw(double amount) -- Subtracts from the balance if funds are sufficient. Can be overridden for overdraft/fee behavior.
- void displayBalance() const -- Displays the current balance to the user.
- virtual void displayAccDetails() const -- Shows general account details like ID, name, type, and balance.
- void showTransactions() const -- Displays the full transaction history recorded by addTransaction().

Getters:

- std::string getAccountHolderName() const -- Returns the account holder's full name.
- std::string getAccountIDNumber() const -- Returns the unique 6-digit account ID.
- double getBalance() const -- Returns the current balance.
- std::string getTypeOfAccount() const -- Returns the account type as a string ("checking" or "savings").
- std::string getPassword() const -- Returns the account password (used for login).

Virtual Getters (for Polymorphism):

These return default or dummy values in the base class but are overridden in derived classes:

- virtual double getOverdraftLimit() const;
- virtual double getTransactionFee() const;
- virtual double getInterestRate() const;

These allow the program to call account-type-specific data through a BankAccount* pointer.

4.3 **The Checking Account class** is a concrete class derived from BankAccount. It adds features specific to checking accounts, including overdraft limits and transaction fees. It overrides relevant methods to implement customized behavior for withdrawals and account detail display.

Data Members:

- double overdraftLimit - The maximum negative balance the account is allowed to reach.
- double transactionFee - A fee deducted from the balance on every withdrawal.

Overridden Methods:

- void withdraw(double amount) override -- Allows withdrawals that may cause the balance to go negative up to the overdraftLimit. Also deducts the transactionFee for each withdrawal. If the resulting balance exceeds the overdraft limit, the transaction is blocked.
- void displayAccDetails() const override -- Displays full account details, including the overdraft limit and transaction fee in addition to inherited fields (ID, name, balance, etc.).
- double getOverdraftLimit() const override -- Returns the overdraft limit for polymorphic access via a BankAccount*.
- double getTransactionFee() const override -- Returns the transaction fee for polymorphic access.

- void saveToFile() const override -- Writes all relevant data fields to a file, formatted for easy parsing and distinction between account types.

4.4 **The Savings Account class** is a concrete subclass of BankAccount, designed to represent accounts that earn interest over time. It introduces an interestRate member and includes functionality to apply interest to the balance. It overrides virtual methods to customize behavior specific to savings accounts.

Data Members:

- double interestRate – The annual interest rate (in percentage) applied to the account balance.

Additional Method:

- void addMonthlyInterest();

Applies one month's worth of interest to the balance

Overridden Methods:

- void displayAccDetails() const override -- Displays account details including account holder name, ID, balance, and the interest rate.
- double getInterestRate() const override -- Returns the interest rate, allowing polymorphic access via a BankAccount* pointer.
- void saveToFile() const override - Outputs all relevant data to the file in the expected format, including the interest rate and account type.

4.5 **The LoginMenu class** is responsible for managing the user interface after login. It provides options for interacting with the selected account, such as depositing, withdrawing, displaying account details, and accessing account-type-specific information. This class ensures that once a user is authenticated, they can safely perform actions on their account. It operates on a single BankAccount* object passed during construction, allowing polymorphic behavior depending on whether the user is using a checking or savings account.

Data Members:

- BankAccount* account – A pointer to the currently logged-in user's account. This allows the login menu to interact with the account using polymorphism.

Methods:

- void showLoginOptions() -- Displays the secondary menu presented after successful login.
- void runLogin() -- Starts the interactive login session loop. It:
  - Displays the login options
  - Processes user input
  - Calls the appropriate BankAccount methods depending on input
  - Detects account type using polymorphic checks to show relevant options
- bool shouldDeleteAccount() const -- Returns true if the user requested deletion.

4.6 **The Menu class** is the main controller of the banking system. It handles the main menu interface, user input for creating and logging into accounts, and manages the lifecycle of all accounts created during the program session. It acts as the entry point for the entire application logic and coordinates actions between the user and the system's core classes.

Data Members:

- Vector accounts – A custom dynamic array that stores all created BankAccount* objects during the current run of the program.
- BankAccount* currentUser – A pointer to the account of the currently logged-in user. Used for login tracking and passing control to LoginMenu.

Methods:

- void showOptions() -- Displays the main menu options to the user
- void run() -- Starts the main loop of the banking system. It:

- Displays the menu continuously until the user chooses to exit.
- Processes user input for each option.
- Handles:

    - Account creation (with appropriate prompts)
    - Login (searches for matching username and password in accounts)
    - Saving all accounts to a file (accounts.txt)
    - Clean program termination

4.7 **The Vector class** is a custom dynamic array implementation designed to store pointers to BankAccount objects. It provides basic functionality similar to std::vector, including dynamic resizing, element access, and memory management. It enables the system to store accounts of any derived type (CheckingAccount or SavingsAccount) through base-class pointers.

Data Members:

- BankAccount** data – A pointer to a dynamically allocated array of BankAccount*.
- size_t capacity – The current maximum number of elements the array can hold without resizing.
- size_t size – The number of elements currently stored in the vector.

Methods:

- void resize() -- Doubles the vector's capacity when it reaches its limit. Allocates a new array, copies existing pointers, deletes the old array, and updates internal pointers.

- Vector() -- Constructor that initializes the array with a default capacity (e.g., 5), and sets size to 0.
- ~Vector() -- Destructor that calls clear() to deallocate all stored account objects, then deallocates the array itself.

- void push_back(BankAccount* acc) -- Adds a new BankAccount* to the array. Automatically calls resize() if the current capacity is full.
- BankAccount* operator[](size_t index) -- Returns the account pointer at the specified index. Throws an exception if the index is out of bounds.
- size_t getSize() const -- Returns the number of elements currently in the vector.
- void clear() -- Deletes all dynamically allocated BankAccount objects and resets size to 0.
- BankAccount** begin() -- Returns a raw pointer to the start of the array (for iteration or file saving).
- BankAccount** end() -- Returns a raw pointer to one past the last valid element (data + size).
- void removeAt(size_t index) -- This method deletes the account at the given index, frees its memory, shifts all subsequent elements, and updates the size of the vector. It ensures proper memory cleanup and prevents leaks.

---

5. Testing and Verification

To make sure the Banking System works correctly and doesn't crash or behave unexpectedly, I tested every part of the program manually. The goal was to check that the program does what it's supposed to (verification) and to catch any errors or incorrect behavior (testing).

5.1 Goals of Testing

- Make sure users can create accounts, log in, and manage their money.
- Check that incorrect input is handled properly.
- Confirm that checking and savings accounts work as expected.
- Ensure that saved data is written correctly to a file.
- Avoid memory leaks or crashes.

5.2 Test Setup

- **Device Used**: MacBook Pro with Apple M4 Pro chip
- **Operating System**: macOS 15.4.1
- **Compiler/IDE**: Xcode (latest version, with C++17 support enabled)

5.3 Input Testing

- Entering letters instead of numbers (e.g., “abc” for balance) - Shows error message
- Entering negative numbers - Rejected with error
- Mixed input (e.g., “12abc”) - Rejected

The program uses try-catch, cin.clear(), and cin.ignore() to clean up bad input and prompt again.

5.4 File Output

When saving, the accounts.txt file was created correctly, and the contents matched the accounts created during the session. The format was checked manually and matched expectations.

5.6 Memory Testing

- All accounts are stored using BankAccount* in a dynamic array.
- Memory was cleaned up using the clear() function in the Vector class.
- No memory leaks or crashes occurred during use - Account deletion tested by logging in, deleting the account, and attempting to log in again — confirmed account is removed and memory is freed.

---

6. Improvements and Extensions

6.1 Areas for Improvement:

- Security - Passwords stored in plain text - Implement password hashing (e.g., SHA-256)
- Data Persistence - Only saves to file manually - Add automatic file loading on startup and saving on exit
- UI/UX - Terminal-based only - Create a GUI version using a framework like Qt or SDL
- Transactions - Only description logs - Add timestamps

6.2 Planned Features that were not implemented:

- **Search by Account ID**: Users can only log in using name and password; no direct ID-based access.

---

7. Difficulties Encountered

While working on this project, I faced a few challenges, especially in the beginning.

The two main difficulties were:

- **Manual memory management**: I had some trouble implementing my own Vector class and managing memory manually using raw pointers. Making sure that the accounts were properly stored, resized, and deleted without memory leaks took time to understand and get right.

- **Inheritance and virtual functions**: Since this was one of my first times using inheritance in C++, it was a bit confusing at first to use BankAccount* pointers to access CheckingAccount and SavingsAccount objects. I also had to learn how virtual functions work and how overriding them lets me call the correct version of a method depending on the account type.

---

8. Conclusion

This project gave me the opportunity to apply object-oriented programming principles in C++ by building a complete banking system from scratch. Throughout the development, I implemented account management features, worked with inheritance and polymorphism, and handled file operations and input validation. The program meets its

core objectives: users can create checking or savings accounts, perform transactions, and save account data to a file. While there are still features that could be added or improved, the project serves as a solid foundation for future development.

---

9. References

1. Stack Overflow – https://stackoverflow.com

Consulted for input validation techniques and exception handling.

2. GeeksforGeeks – https://www.geeksforgeeks.org

Referenced for examples related to object-oriented programming in C++.