Missouri State University

CSC232 Final Project

BEAR Bank Software

Vitor Freitas, Hung Nguyen, Huy Huynh

CSC 232 - Data Structures

Dr. Razib Iqbal

December 5, 2020

**Table of Contents**

**1.Project Information**

1.1 Project Name

1.2 Team Members Information

1.3 Project GitHub Repository

1.4 Instructions to Interact with Code

**2.Description of Features**

2.1 Core Features

2.2 Implementation

**3.Data Structures Information**

3.1 Structures

3.2 Classes

**4.Data Structures Table**

**5.Document Body**

5.1 Justification of Data Structures Used

5.2 Bonus Points

**1.Project Information**

1.1 - Project Name

* CSC232 Final Project - BEAR Bank: A console-based application that represents a bank software that allows the interaction of different user types with multiple account types.

1.2 - Team Members Information

* Vitor Freitas (vit1905@live.missouristate.edu)
* Hung Nguyen (hung249@live.missouristate.edu)
* Huy Huynh (huy0312@live.missouristate.edu)
* Missouri State University - Fall 2020 - CSC 232

1.3 - Project GitHub Repository

* <https://github.com/vitorslaibi/CSC232_Final_Project>
* This repository included all files to run the program, history of updates to the code, and different branches used during the development of the software

1.4 - Instructions to Interact with Code

* The program will be submitted in a .zip file that will include all files that are necessary to compile and run the code. The main.cpp file will drive the entire software and is the one that should be executed.
* To compile and run the program, follow those instructions:
* Type the following command, and press enter: g++ path-to-file/main.cpp -o main (It tells the c++ compiler (g++) to take the file main.cpp and make an output file called main.exe)
* -Type ./main.exe and press enter. This will execute the code.
* This is a console-based application, so the console will show the correspondent menu of options and the software user should enter the desired input and press enter to navigate through and use the different functionalities of the program.
* IMPORTANT: After the program is first run, there will already be an existing system administrator, and its’ encrypted information is saved in the file ‘A0.txt’. To login as a system administrator and use the admin menu, please insert the following credentials:
  + ID: A0
  + password:123

**2. Description of Features**

2.1 - Core Features

* This program represents a bank software that uses classes and data structures to define and store the multiple account types that are available and allow interaction from different types of users. The application will also contain security features that will give privacy to account holders by using data encryption, which will not allow administrators or bank officials to retrieve customers’ private information, such as their password.
* The first kind of user is system administrators (SystemAdmin.h), who are allowed by the program to create, modify and delete account types, enable or disable login profiles, retrieve user IDs, and change account passwords. When the admin closes an account type, the accounts in that type will still be active and generate interest, but new accounts cannot be created until the account is enabled again. There is one main drawback, which is when the admin disables an account type, and enables it again with new information (interest rate, safe level, penalty, fees), the old accounts information will be changed too. The software can also be utilized by bank officials (BankOfficial.h). These users can open or close customers’ accounts, deposit and withdraw money from them and search accounts using different account holder’s information such as name of phone number. The application can also be used by the bank account holders, who can have multiple accounts and are able to interact and see the information related to them. The software also permits holders to retrieve their login dates and transaction story for each account type.
* As of now, customers can open three types of accounts and each of them will hold data about the account holder and account information, such as balance, interest rate, opening and closing day, history of transactions, among others(BankAcc.h). The initial three available account types extend the parent BankAcc.h class and are: Checking account(Chk.h) and savings account(SavAcc.h), which have all the functionalities above with the difference being that the checking has no interest or monthly fees while the saving has a fixed interest rate set by bank officials. Lastly, the certificate of deposits (CD.h) is an account type with fixed-rate and fixed-term, Its balance will increase according to a set interest rate and the customer can withdraw the money after the defined time range, unless there is an early cancelation. The application also includes five spare classes(named ClassU.h, ClassV.h, ClassX.h, ClassY.h, ClassZ.h) that will be activated if the system administrator chooses the option to create a class
* After the application is compiled and ran, the console will ask the user whether he/she wishes to log in as a system admin, bank official, or account holder and ask for their login credentials based on the choice. If successfully logged in, each user type will be provided a menu that will display all their possible interactions with the software, as discussed above. Users are allowed to perform as many operations as they wish and login/logoff of different accounts without having to exit the program and all account and user information will be stored in files with the ability to retrieve when the program restarts.

2.2 - Implementation

* The implementation of this software is based on interaction with classes, data structures, and object-oriented programming. The application is based on eleven classes, where nine of them represent the different bank account types and the other two represent user types (system admin and bank official). For each class, there are two files, one with the class prototype and another with the method definitions.
* The class BankAcc.h is a virtual parent class and contains base functions and account information that is shared by all of its children classes (different account types). Besides the account holders’ info, this class has as its members the opening and closing date of the account, date of each login, balance, interest rate, monthly fees, etc. The class methods include virtual functions to deposit, withdraw and close the account, getters and setters, functions to save and load account data and transaction history, apply interest rate and fees, and finally, there are two functions to encrypt and decrypt the customer’s private information, such as their password, so that administrators and officials cannot easily retrieve it.
* All other account types derive from BankAcc.h and have small additions and extra methods. The ChkAcc.h and SavAcc.h classes are similar, containing deposit, withdraw, and close account functions that override the virtual function of their parent class. The only difference is that the savings account contains a daily interest rate while the checking account does not. The software also supports a certificate of deposit account type(CD.h) with a fixed term and rate. Besides the override functions, the CD contains a method to calculate the interest on the initial balance based on the number of days passed. If the user decides to close the account before the determined term (183 days), no interest will be paid and a penalty will be accessed based on the number of days remaining.
* The other account types included in the application (U, V, X, Y, Z) are spare classes and can be “created”, deleted or modified if the system administrator wishes to. Each class contains a static variable that indicates whether or not the class can be used by bank customers. This variable is initiated as false for these spare classes at the beginning of the program.
* The last two classes SystemAdmin.h and BankOfficial.h, represent the remaining user types. The administrator class contains the user info and opening date as its private members, which can also be saved and loaded from files. Using the class methods system admin are able to enable/disable login profiles or create and delete account types by changing the value of the ‘active’ static variable that is present in every accounting class. The class also has a method that changes accounts’ password and updates the respective file, without allowing the admin to see the password in plain text. The official class contains the same private member as the admin one with the addition of a variable that indicates if the official login profile is enabled. For each account type, there are methods to deposit and withdraw money, open/close such accounts and save the official info to the closed account files. The search functions allow the class to show the customers their account info based on their name, ID, or phone number.
* Driver files: After the program is run, information will be loaded from ‘preview.txt’. This file has information about each account and user type, such as the number of open accounts, current interest rate, safe level, and penalty for prohibited transactions. Those values are then assigned to its respective class’ variables. Then the code will initialize multiple vectors, one for each account type. For each existing account, a class instance will be called and the correspondent data will be loaded to the class from the .txt files; this class will then be added to their respective account type vector.
* The login menu will be displayed and the user has to choose its type before entering the login credentials. The ID and password have to match with the file information to login. If the login is successful, each user type will be able to perform their allowed actions discussed above using class methods and file comparison. If the user chooses option 4 in the main menu to exit the program, all new and updated information about accounts, accounts types, and login profiles will be saved in the respective text files and the program will terminate

**3.Data Structures Information**

3.1 - Structures

* The main data structure used to develop this application were vectors, used to save, access, manipulate, and load user and account data. In the main.cpp file, ten vectors were initialized, to store the classes from the three existing account types, the five spare account types, and the two of the user types. Vectors are structures that can dynamically store elements of the same type and can change size. In our application, they store all created class instances at the start of the program and after new accounts are created. Since all elements are inserted at the end of the vector, the complexity can be kept constant at O(1). The cost of accessing elements is also constant at O(1). Throughout the driver files, vectors are used to compare class data with user input(ID, password, search functions) and access classes’ setters, getters, and methods

3.2 - Classes

* The application also uses classes to store data and perform actions according to what the different types of users wish to do. The software is based on eleven classes: The parent class BankAcc.h with members and methods that are shared by all account types; eight children classes that derive from BAnkAcc.hand represent the different account types; and two classes that represent the user types bank official and system administrator. Each bank and user account have their own class instance with related information.
* All four pillars of OOP are achieved in the application:
  + Inheritance: All account types are derived from a parent(BankAcc.h)
  + Encapsulation: In each class, all members are private and setters and getters are used to modify and access such members
  + Polymorphism: Each subclass has its own behavior while sharing the functionalities of the parent class
  + Abstraction: Only necessary features of the object were shown outside of the classes(on the main function), hiding the implementation.

**4.Data Structures Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Classes | Description | Vectors | Main usage | Special Characteristic |
| SystemAdmin.h  Each class object is stored as an element in the vector  To use one of the class objects, we use adminList[i] to refer to the admin object at the ith index. | Store information of the system administrator.  Due to the design of the program, there is only 1 system administrator | vector<SystemAdmin> adminList | Quickly access each class object to use the functions of those classes.  For Example:  When a customer wants to make a deposit, the following command is used:  chkList[i].deposit(amount) | Because of the initial design of the program. Currently there is only one element of SystemAdmin in the vector.  To add another SystemAdmin object, it has to be manually write in the code and push to the adminList Vector  The SystemAdmin class will mostly control the other classes. (add, delete, modify the BankOfficial or the BankAcc classes)  For example: if the admin wants to disable a bank official account, the following command is used:  adminList[i]. disableOfficialAcc(account) |
| BankOfficial.h  Each class object is stored as an element in the vector  To use one of the class objects, we use officList[i] to refer to the bank official object at the ith index. | officList | vector<BankOfficial> officList | The BankOfficial class will mostly add, delete or modify the BankAcc classes.  For example, if a Bank Official wants to close a U account, the following command is used:  officList[i].UAcc(account) |
| ChkAcc.h  Each class object is stored as an element in the vector  To use one of the class objects, we use chkList[i] to refer to the checking account object at the ith index. | chkList | vector<ChkAcc> chkList | These classes are basically the same. But they can have different class member values.  For example, the ChkAcc class will not have an interest rate or monthly fee. While the SavAcc or the CD can have. |
| SavAcc.h  Each class object is stored as an element in the vector  To use one of the class objects, we use savList[i] to refer to the saving account object at the ith index. | savList | vector<SavAcc>savList |
| CD.h  Each class object is stored as an element in the vector  To use one of the class objects, we use cdList[i] to refer to the CD account object at the ith index. | cdList | vector<CD> cdList |
| ClassU.h, ClassV.h, ClassX.h, ClassY.h, ClassZ.h  Each class object is stored as an element in the vector  To use one of the class objects, we use uList[i] to refer to the U account object at the ith index.  The same applies to the v, x, y ,z account | uList  vList  xList  yList  zList | vector<ClassU> uList  vector<ClassV> vList  vector<ClassX>xList  vector<ClassY> yList  vector<ClassZ> zList |
|
|
|
|

**5.Document Body**

5.1 - Justification of Data Structures Used

* We choose to use vectors, with the created classes as data type, to store and retrieve user and account data throughout the driver code. Vectors can change their size dynamically and the time complexity for insertion, removal and element access is constant (O(1)). Besides that, vector elements can be easily accessed using indices and the search(O(n)) and comparison(O(1)) of elements is also facilitated, which keeps the overall code short and clean. Since all vectors were ordered by account number, we can simply use such number as an index to access the account information(For example: For a checking account with number 0, we can call chkList[accNum] and that would access the first element of the vector, which would contain information about the checking account with accNum = 0).
* Alternatives: For our application, vectors are a better option than arrays due to their variable and unlimited size. The use of a fixed-sized array could lead to troubles since we would need to constantly resize the array as new elements were added. While stacks and queues operations have similar time complexity as vectors, the access of elements that are located in the middle of those structures is complicated and would require unnecessary addition and removal of elements. Trees, such as BST or AVL, are recommended when there is only one key and in this application the various search functions require different types of keys(ID, phone, name). Furthermore, we would need to constantly balance the tree to keep the complexity low. Therefore, we judged that considering overall time complexity, size, and ease of performing operations, the use of vectors would be the better option for this application.
* We decided to use classes as our user-defined type because we judge that it would be the best way to represent each account and user types, since classes can store data and also perform actions using its methods. For example, each instance for the class SystemAdmin.h can store the administrator information and easily perform each of the admins’ permitted actions using the class methods. That way we can keep the code clean and organized, and reduce the number of operations and consequently, the overall time complexity. The same is true for the account type classes. The only alternative UDT that is comparable would be structures(struct), but they could only store the account data, while the user and account actions would have to be defined in outside functions. That would most likely lead to a code that is repetitive, redundant and with a high overall complexity.

5.2 - Bonus Points

* UX/UI Design: In our program, we express our care to our users with our attention to the smallest of details, in order to provide the customers with the best experience. The loading time is next to non-existent. The menu is concise and easy to use with absolutely zero softlock. All of the descriptions are brief, while still on points. We always notify customers with any changes in their accounts, no matter how small. For bank officials and system administrators accounts, it’s easy to get a hold of their abilities and limits. Every time they decide to make a change to the system, they can observe the results without any barriers. Moreover, all of the changes are updated to the .txt files in real time, to minimize any loss of information.
* Memory usage: We use 2 .txt files for every account: one for account information, and one for transaction history and one additional .txt file to keep track of the total number of accounts in any type, as well as class variables. It might take a little more space compared to saving all accounts’ data in one big text file, but it makes the code simpler, while also limiting the damage any errors might cause for the system as a whole. The choice of using vectors, while is not extraordinary in any specific task, is a perfect all-around option. Initially, I think AVL binary search tree might be the best choice, since we have to search regularly in this project; however, BST is only good if there is one type of key, while in this program we need to search by IDs, names, phone numbers, one after another. Therefore, the O(n) when traversing the vector is a completely reasonable solution. Additionally, we even have O(1) while searching with IDs, and fortunately, it’s the main way we search this time. The downside of vectors, which is when adding and deleting elements in random indexes, is not a concern in this case, because when we “delete” an account, we don’t actually delete it but rather make that account unable to generate interest or perform transactions. Moreover, the simple structure of vectors make the code much more friendly with lower-level programmers, and can be used widespreadly.
* Security: The object-oriented data structure used in this project greatly increased its security value, with inheritance, polymorphism, encapsulation and abstraction. New child classes can be generated from the base class and used without any conception of how the base class operates. The 5 spare classes for the system administrator can be easily increased to hundred, or even thousand of spare classes, all with dynamic properties. The class structure itself inherits a lot of characteristics of the Bank class in Project 1, and can be used to develop more advanced classes in the future. The encryption, while is still not secure to the extent of uncrackable, would still pose a decent challenge for anybody who got their hands on the text file. To make up for its low level of security, the encryption method is highly reliable. It never results in special characters that might corrupt the string, rendering it unusable (like \n,\t,\r), and in case we need to manually decode it for whatever reasons, an ASCII table along with the key is enough.