|  |  |
| --- | --- |
|  | |
|  | |
|  | |
|  | |
|  | |
|  |

Advanced Programming

Assessment 1 - Report

January 27th, 2022

Mathews Joy

Department of Computer Science

**Table of content**

Contents Page

Introduction ....……………………………………………………………………………….. 2

Basic design of the application (Implemented Functionality) ……………………….………. 2

Implemented functionality and deviations/additions from brief ………………….. 2

Objected oriented concepts (OOP) ………………………………………………... 2

Advanced features …………………………………………………………………. 3

Stretch task ………………………………………………………………………… 3

Testing and Evaluation ………………………………..…………...………………………… 3

Testing of application ……………………………………………………………… 3

Evaluation of solution time/space complexity ……………………………….……. 5

Conclusion …………………………………………………………………………………… 5

**Introduction**

This report outlines the creation of a simple console-based banking application in C++ following a written brief, this is in order to demonstrate an understanding of programming techniques such as low-level memory management, input/output, logical & mathematical concepts, and multi-paradigm development. Discussion on all implemented/not implemented functionality, how the solution was tested for integrity and security and a critical analysis (time/space complexity) of algorithms used is covered within this report.

# Implemented functionality

Implemented functionality and deviations/additions from brief

For this assignment all commands stated in the brief including an implementation of the stretch task has been provided. The program has tried to be kept as abstracted as possible, this can be shown in the decision to create an additional class called ‘AccountManager’ which stores all opened accounts, and provides function to be called from main for commands such as deposit, withdraw, transfer etc. That way the main file ‘LincBank.cpp’ can be kept as minimal as possible and only requiring code that is related to its functionality. This level of abstraction has been kept consistent among all my files/classes, allowing a higher overall cohesiveness and room has been left for easy scalability due to loose coupling between components.

There were no deviations from the brief, however a lot of further additional attributes/functions and classes. Some of the main additions were the inclusion of a ‘Money’ class, which would be the data type used to handle the storage of money, e.g. Money was used as the datatype to store balance in the Account class. The justification for using a Money class instead of a double/float for storing money is due to the inconsistencies and lose of precision when doing arithmetic with primitive types. This money class includes overloading of arithmetic operators such as + and -, so it becomes easier to do operations such as depositing and withdrawing. Another major addition to the design of the banking application was the use of custom exceptions, such as ‘IncorrectAmountofArguments’ and ‘IncorrectArgumentValue’. These custom exceptions where included, as there were no specific exceptions built into C++ that met the need of exceptions that needed raising during the handling of user input. Last but not least, smart pointers were used throughout the app where necessary (unique and shared pointers). The primary intention of using smart pointers is they can alleviate many of the problems with 'raw' pointers, mainly smart pointers will free the memory if an exception occurs and forgetting to delete the object then leaking memory.

Object oriented concepts (OOP)

During the process of the implementation of the application, the inclusion of key object-oriented concepts were made sure to be used effectively.

A vector of unique pointers of type ‘Account’ was used in order to store all opened accounts of a customer, by using a pointer of the base class as the main data type it allowed late binding so that any account types of current, isa, savings could be created (through polymorphism) and however many, this could not have been achieved via early binding and would limit the scalability of the application. Encapsulation rules were strictly stuck by using appropriate access modifiers. All class attributes that shouldn’t need direct access were kept private, protected attributes/methods were used to allow derived classes to have direct access to important data. Furthermore getters were given to appropriate attributes and setters where generally avoided in my class implementation to allow optimal object data security. Abstraction was achieved through the use of interfaces such as ‘InterestEarning’ and abstract class ‘Account’, allowing to hide the implementation. Finally use of inheritance can be seen through the hierarchy of the type of bank accounts, a base class of ‘Account’ was used to declare all common functionality/attributes of a bank account and further derived classes such as ‘Current’ were used to provide furthermore specific operations.

Advanced features

Many advanced features have been made use off, in order to improve the overall efficiency/reusability or maintainability of code.

Operator overloading can be seen in the ‘Transaction’ class where comparison operators (>,<,!=,==) where overloaded, making it much easier and readable to compare transactions by their amount. Lambdas can be observed in ‘AccountManager’ class, in the ‘doesCurrentISAExist’ function. 2 Lambdas were made use of in order to have a simple statement to check how many current and ISA accounts existed in a vector of accounts (through use of built-in function std::count\_if). However lambda usage was kept to a minimum due to their lack of readability. Templates have been implemented for my stretch task, a red black tree was made in order to store all transaction made in an account,. To allow this tree data structure to be a generic construct and be used for other cases, it was made as a template, thus providing better reusability of code.

Stretch task

For the stretch task it was required that all transaction history should be searchable by their value (transaction amount). It was justified to use a self-balancing binary search tree that automatically sorts after an insert and allows fast retrieval of data. The decision to implement a Red-Black tree (RBT) was made due to its self-balancing nature so there is no need to run a sorting algorithm every time we want to search for a transaction. An insert function, search function and re-balance functions have been implemented (along with helper functions), however the deletion function was not included as it does not make sense to ever delete a transaction from history. A major problem faced was binary search trees can’t store duplicates and this was a critical issue as the transaction amount was used as the ‘key’ for the tree nodes (transaction will usually have duplicate amounts). In order to overcome this problem, a vector of transactions was made as an additional tree node attribute, this meant when inserting into the tree if a node with the same value as the inserted transaction was found, instead of creating a new node, the new transaction was added to the vector of the existing node.

# Testing and evaluation

Testing of the application

The testing of my application was done through 2 forms, the first type used was end to end testing (in conjunction with black box), this type of testing meant I was assessing the functionality and performance of an application under product-like circumstances, through the entire life cycle of developing the application including planning, development, and the end result. This helped saved time by reducing the amount of testing needed at the end, verifies the application logic, and enables testing from a user perspective.

Another method of testing used was white box, this meant during testing the source code was known, this form of testing was especially beneficial as it helped uncover many logic problems e.g. the money class initially wasn’t adding other money objects correctly.

In order testing (one session)

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input Values** | **Expected output** | **Passed?** |
| User opens an ISA | open, 3, 999 | Error message observed: ISA initial balance must be >= £1000. | YES |
| User opens an ISA | open, 3, 1000 | Account created, with message:  ISA Account has been opened with £1000.00 | YES |
| Open current account | open, 1, 500 | Account created, with message: Current Account has been opened with £500.00 | YES |
| Try to open another current account | open, 1, 10 | Error message observed: Sorry you already have an existing Current account. | YES |
| Transfer from current to ISA > balance + overdraft | transfer, 2, 1, 1001 | Error message observed: Withdrawal to [Current] unsuccessful. No available funds in your current account. | YES |
| View accounts | view | Display of all account details , and their transaction history | YES |
| View account by index | view, 1 | Display specific account details (ISA account in current scenario) | YES |
| View an account by a non-existing index | view, 5 | Error message observed: Cannot view account: Invalid Index. | YES |
| Deposit into ISA account | deposit, 500 | Deposit success, with message: Deposit to [ISA] account successful. | YES |
| Project ISA balance by 2 years | project, 2 | Message: Projected balance: £1534.88 | YES |
| Search for a transaction in ISA | search, 500 | Message showing all transactions with 500 as the amount, there should be only 1 in this situation | YES |
| Invalid money amount is entered (withdraw command) | withdraw, 50.555 | Error message observed: Unrecognizable withdrawal amount value. | YES |
| Invalid money amount is entered (deposit command) | deposit, 70.777 | Error message observed: Unrecognizable deposit amount value. | YES |
| Withdraw from current account into overdraft | view, 2 (PASS)  withdraw, 1000 | Success, message displaying: Remaining Overdraft: £0.00, Account Balance: £0.00 | YES |
| Withdraw from current with not enough overdraft | withdraw, 50 | Error message observed: Withdrawal to Current unsuccessful. No available funds in your current account. | YES |
| Deposit into current account money is added to the balance amount not overdraft amount | deposit, 560  view, 2 (PASS) | Deposit success, with message: Deposit to [Current] account successful.  View command shows: Balance £560.00 | YES |
| Open an account with an exceptionally large opening balance | open, 1, 5000000000000000000000000000 | Account isn’t open, message: Unrecognizable deposit amount value. | YES |
| Open a current with negative balance | open, 1, -50 | Error message observed: Unable to open an account with negative opening balance. | YES |
| Open a savings with negative balance | open, 2, -50 | Error message observed: Unable to open an account with negative opening balance. | YES |
| Open a ISA with negative balance | open, 3, -50 | Error message observed: Unable to open an account with negative opening balance. | YES |
| No input is entered |  | Error message observed: You have passed in an incorrect amount of arguments | YES |
| Enter incorrect number of parameters for a command (tested for all commands) showing 1 for simplicity | open, 1 | Error message observed: Make sure you have entered all valid parameters for this option. | YES |
| Random invalid input | ksid, 323 | Error message observed: Invalid command: ksjd | YES |

Evaluation of solution time/space complexity

For the “stretch” task, a red-black tree was constructed as mentioned previously, a red-black tree is a self-balancing tree so their operations are guaranteed to be O(log(n)). The search function for my particular implementation would be O(log(n)) + K, K being duplicate transactions (transactions with the same amount value). Red-black trees offer logarithmic average and worst-case time complexity for insertion, search, and deletion. Rebalancing has an average time complexity of *O(1)* and worst-case complexity of *O(log n)*. Space complexity would be O(n) as it would grow linearly as more transactions are inputted into the tree.

As for the overall program itself, it is a bit trickier to find the time/space complexities. Regarding space complexity we can expect it to be 0(n) as space complexity is equal to input size plus auxiliary space (extra space or temporary space used by an algorithm). 0(n) space is justified as where we are storing the vector of all opened accounts, will grow linearly, as when the user creates a new account a new pointer is allocated to the heap and uses up more memory. Moving onto time complexity, we can find this by identify our worst case. In this solution the worst case can be found in the main ‘LincBank.cpp’ file, For each n (number of times user submits an input) we are iterating over a number of parameters (m). If the number of parameters m are variant increasing the time complexity to O(n\*m), however there is further added complexity due to further function calls through the compare command if statements. The worst case function call is for the ‘search’ command on the ‘AccountManager’, where we also go on to call a binary search tree, search function which takes O(logn). This then gives us an overall time complexity of O(n\*m+logn).

**Conclusion**

A fully operational basic console banking application has been designed, implemented and tested for vulnerabilities. Discussion on the key OOP concepts and advanced features used have been shown and justified for their practical uses. Thorough explanation of the time/space complexities of the application as a whole and the ‘stretch’ task has been evaluated. Further improvements can include the creation of a database to store account data, allowing a stateful application.