Stephen Kenyon

Programming Finance C++

Final Project

The Account Manager system that I have built consists of the following .cpp source files:

Account.cpp, BankAccount.cpp, FileReadAdapter.cpp, Fuzzy.cpp, Main.cpp, Position.cpp, StockAccount.cpp. In addition, header files exist for Account.h, BankAccount.h, FileReadAdapter.h, Fuzzy.h, Position.h, and StockAccount.h.

Additional necessary files include Result\_1.txt and Result\_2.txt, which contain price and symbol information for a select grouping of stocks.

The account management system is driven through main.cpp. Main mainly consists of a menu system, calling external classes to perform data operations as needed. The menu system consists of two levels, one upper level and one lower level, with the lower level being split between a menu for the Stock Account and a separate menu for the Bank Account. For all three distinct menus, a case switch is used to select individual cases.

In main, the user can interact with a number of features. First, in the Stock Account menu, they can

1. Display the price for a stock symbol (given that it exists in one of the two Result.txt files)
2. Display the current portfolio, provided that the portfolio has stock positions and is not empty
3. Buy shares, either as new position, or add additional shares to an existing position (the same stock)
4. Sell shares (either liquidate or sell of some quantity for a given position)
5. View a graph of the portfolio value
6. View transaction history
7. Experimental Features (blow up the data from a single stock, in this case hard-coded to MSFT, to provide for simulating higher volatility

The Bank Account portion has a much simpler feature set:

1. View account balance
2. Deposit money
3. Withdraw money
4. Print out (transaction) history

The back-end of the system is handled mainly through the class hierarchy of Account, StockAccount, and BankAccount. Both StockAccount and BankAccount are child classes of Account. In order to facilitate static data sharing of the account cash balance, Account contains a private static double CASH\_BALANCE, which both StockAccount and BankAccount can view and modify, without needing to ping each other to make sure their values match. Account has setters and getters for CASH\_BALANCE, along with file i/o methods. When initialized, if Account cannot find account\_value.txt, it creates it, and sets the CASH\_BALANCE to 10,000.00. After successfully exiting the main program, Account contains a method saveAccountFile(), which allows for overwriting this account\_value.txt, in order to store the account value and access it on reboot of the program. Account accomplishes this by it’s constructor searching for the account\_value.txt file and then initializing CASH\_BALANCE based on the value in this file. Lastly, Account contains the method getDate(), which returns a string formatted as MM/DD/YYYY.

As mentioned previously, BankAccount inherits from Account. It’s constructor initializes BankAccount by searching for the bank\_transaction\_history.txt file. If it cannot find it, then it creates this file. This file is later used for saving the transaction history of the bank account. File operations to this file append instead of overwriting. BankAccount contains methods for both depositing and withdrawing, and modify the cash balance and account history when doing so. Lastly, BankAccount contains a method for printing the bank account history, by reading the values out from the transaction file and printing them to the screen.

StockAccount is more complicated than BankAccount or Account. It operates as a doubly linked list, and is really just the linked list itself. I variously refer to it as the portfolio. Also, separate for conventional nomenclature, I refer to the nodes of this doubly linked list as “positions”. Hence, we have a portfolio that consists of positions, implemented as a doubly linked list. To facilitate the linked list, StockAccount is a friend class with Position. Position contains private data for next and previous pointers, to facilitate movement along the doubly linked list, along with private data values for the stock symbol (as a string), the stock quantity (as int), and the stock value, (as double). On construction, it merely sets the next and previous pointers to null.

All methods for the portfolio operations are otherwise contained within StockAccount.cpp. This includes the graphing connection MATLAB as well. On construction, StockAccount initiliazes to an empty list if it cannot find portfolio.txt. This portfolio.txt file is then created. If instead the program were to load a second time after having successfully exited and saved, then StockAccount would generate its positions through the constructor by reading this portfolio file and adding each value to the start of the list, checking if sorted, and then bubble sorting the list. All of the sort operations performed on StockAccount use bubbleSort, and have the positions arranged by total value, from greatest to least. StockAccount also contains an explicit destructor for deleting positions from the list.

New positions are added to the stock account using addToStart, which adds a new position at the start of the doubly linked list. The list is later sorted to ensure that the newly added item is in the correct position. StockAccount can print out its entire list with printStockAccount. Likewise, methods for removing elements from either the start or end of the list exist as well.

Sorting operations use bubbleSortPortfolio, which as previously mentioned, relies on bubble sort. It accomplishes this by swapping adjacent nodes if the next node has a stock value greater than the current node. Nodes that surround the swapped nodes then have their next and previous pointers adjusted to point accordingly. The swap helper function, swapPositions, relies on the second position being swapped to be to the right of the current position being swapped. It also requires that the positions be adjacent to one another.

StockAccount also has a method to check whether or not a stock position exists as a symbol in the portfolio already.

File i/o methods consist of readFromPortfolioFile (to initialize the list on startup if possible), readFromRandomFile, which is used to generate a random price given a stock symbol from one of two Results.txt files, createPortfolioFile, saveToPortfolioFile, createPortfolioTransactionHistory (which is used for the transaction history), saveToPortfolioTransactionHistory, createPortfolioValueFile (which is only used for saving the entire value of the portfolio on exit), and savePortfolioValueFile.

Methods for both buying and selling shares exist. In order to link the buying and selling of shares to the bank account, modifications to the cash balance and to the bank account transaction history are handled inside of main when they occur.

The MATLAB connection is containerized entirely within graphPortfolioValue, although includes are listed at the top of StockAccount.cpp. This method does have one limitation, however, in that in only plots the first 100 points of data for the portfolio value.

Remaining methods include printTransactionHistory, which reads from portfolio\_transaction\_history.txt, getPortfolioValue, which sums up the current portfolio value across the positions, and getSize, or get size of the doubly linked list. This counts the number of positions.

The remaining two files/class definitions are used solely for the “Experimental” features in the Stock Account menu in main. The goal was to be able to better simulate the volatility in the fluctuation of a stock price. One interesting way in doing that is to constrain the model slightly, so that the stock price consists of generated values entirely from a single value of a single stock. Then, I could apply a function to perform a linear transformation of these values, given that they exist in an array, to show a growing or decreasing stock value over time. For the sake of this program, however, values taken from these linearly transformed values remain random, but from the new array containing those values as opposed to one of the values from the text files. I implemented this using two design patterns. First, I used an adaptor, mostly contained in FileReadAdapter.cpp, which allows for interfacing with StockAccount’s readFromRandomFile, which outputs a double, into an array which can be operated over. Next, I developed to methods to operate over this array. One provides some random noise that gets added with increasing size towards the right end of the array. This method is called TransformTowardsRight. A similar method that instead operates from the right to the left is also provided, called TransformTowardsLeft. Both of these exist in the Fuzzy.cpp file, which acts as a strategy pattern, both classes inherit from a single Transform class, and the user is able to at runtime (in main) select which of these two strategies they would like to implement.

Included in my submission are the usual source and header files, along with screenshots. However, in this instance, I duplicated my c++ files. One zip folder contains files for a linux version of the code, which has the entire MATLAB portion commented out in order to work properly. This linux zip is contained within the main instance of the code, and are clearly labeled otherwise. The main version of the code is clearly labeled FinalProject\_Kenyon.zip