**Name** - Sammed Sunil Admuthe

**NetID** - ssa180

***Project Description -***

**1. Account :**

* virtual double getBalance() const: A pure virtual function to retrieve the current balance of the account. Derived classes implement this function.
* virtual void setBalance(double amount): A pure virtual function to set the balance of the account. Derived classes implement this function.
* virtual ~Account(): A virtual destructor for the base class.

**2. BankAccount :**

* void setBalance(double amount): Sets the balance of the bank account.
* double getBalance() const: Retrieves the current balance of the bank account.
* void deposit(double amount): Deposits a specified amount into the bank account and maintains synchronization of cash balance between BankAccount and StockAccount.
* void withdraw(double amount): Withdraws a specified amount from the bank account and maintains synchronization of cash balance between BankAccount and StockAccount.
* void accountBalance(): Displays the current balance of the bank account.
* void recordBankTransaction(std::string event, double amount): Records a bank transaction event (e.g., deposit or withdrawal) along with the transaction amount.
* void viewBankTransactionHistory(): Displays the transaction history of the bank account.

**3. StockAccount :**

* void setBalance(double amount): Sets the balance of the stock portfolio account.
* double getBalance() const: Retrieves the current balance of the stock portfolio account.
* DoublyLinkedList\* getList(): Retrieves the pointer to the doubly-linked list containing stock portfolio data.
* DoublyLinkedList& getStockPortfolioInfo(const std::string& filename): Reads stock portfolio information from a file.
* static StockData readStockData(std::string filename, std::string symbol): Reads stock data from a file for a specific symbol.
* static double displayPriceOfStock(std::string symbol): Displays the current price of a stock.
* void displayCurrentPortfolio() const: Displays the current stock portfolio.
* void buyShares(const std::string& symbol, int shares, double maxPrice): Buys a specified number of shares of a stock at or below a maximum price.
* void sellShares(const std::string& symbol, int shares, double minPrice): Sells a specified number of shares of a stock at or above a minimum price.
* void recordTransaction(const std::string& type, const std::string& symbol, int shares, double price): Records a stock transaction (buy or sell) into stock\_transaction\_history.txt.
* void recordWithdrawal(double totalCost): Records a withdrawal from the stock portfolio account into bank\_transaction\_history.txt whenever user buys a stock.
* void recordDeposit(double totalCost): Records a deposit into the stock portfolio account into bank\_transaction\_history.txt whenever user sells a stock
* void viewTransactionHistory(): Displays the transaction history of the stock portfolio account.
* Rscript plot.R -> Used to plot a graph of Total Portfolio Value over time.

**4. Design Patterns :**

* **Singleton** :
  + The project uses the Singleton design pattern for the AccountManager class to ensure that there is only one instance of the AccountManager throughout the application's lifecycle.
  + The AccountManager class provides access to both BankAccount and StockAccount instances.
  + The Account Manager helps maintain synchronization of cash balance across StockAccount and BankAccount classes. This is because only once instance of account Manager can be created that manages get and set balance methods.
  + If Singleton pattern was not implemented for the above use case then the user might accidentally create two instances for BankAccount and StockAccount which would lead to synchronization issues whenever the user buys/sells a stock or deposits/withdraws the amount from the Bank Account. In such cases the cash value won't be in sync for two accounts.
  + Thus, this pattern restricts the client to create multiple instances of AccountManager class which would lead to synchronization issues across the accounts when it comes to cash balance

* **Strategy** :
  + The SortingStrategy class is an abstract base class that defines the interface for different sorting strategies that can be applied to a list of stocks. It serves as the foundation for concrete sorting strategies.
  + The ShareValueSortStrategy class is a concrete implementation of the SortingStrategy interface. It sorts stocks in descending order of their total values (shares \* price per share) - uses Selection Sort
  + The SymbolSortStrategy class is another concrete implementation of the SortingStrategy interface. It sorts stocks in ascending order of their symbols (alphabetical order) - uses Bubble Sort
  + These strategies provide flexibility in sorting stocks within the Account Management System and allows to easily switch between sorting criteria as needed. Internally it uses different sort algorithm (Selection and Bubble Sort). (Note the sort doesn't actually change the link because the problem statement says the list should always be sorted by total value. Instead, the program prints the values after performing a temporary sort.)

**5. Text Files Created:**

* **accountLog.txt -** Updated every time buy/sell or deposits/withdrawal are made. Keeps the track of cash balance. This file is also used to initialize balance every time program starts.
* **Stock\_Portfolio\_Info.txt -** Contains the information about current stock(symbol & number of shares) in the portfolio
* **stock\_transaction\_history.txt -** Contains history of stocks bought and sold (event, symbol, number of shares, value per share and time) in the portfolio
* **bank\_transaction\_history.txt -** Contain history of deposits and withdrawals from both accounts (event, amount withdrawn/deposited, date-time, cash balance, identifier (if withdrawal or deposit is from stock account)) in the portfolio. Stock identifier at end of every record is used to distinguish deposits and withdrawals from buying and selling the stock. All other deposits and withdrawals indicate cash inflow and outflow from an application.

**6. Key Points :**

* Cash Balance is synchronized across Stock Account and Bank Account.  
  Logic - initially the constructor checks if there exists an accountLog.txt file. If it doesn’t of if the accountLog.txt doesn’t contain any values then use initial balance i.e. 10000. In other cases, read the latest value i.e last entry in accountLog.txt.
* This ensures that every time when the program starts value is picked from accountLog.txt.
* In case if user need to start the application from scratch i.e reset everything delete all .txt files

**7. Extra :**

* Option to sort portfolio by symbol (alphabetically) or total value of particular stock (descending)
* This functionality implements the strategy design pattern which is discussed above.

**8. Data Structures Used :**

* Doubly LinkedList (implemented from scratch)
* Map
* Vector

**9. Requirements:**

* C++11
* Ggplot library to run R-Script for viewing graph