16: 332: 503 PROGRAMMING FINANCE

Account Management System

Project Description

Name: Sai Laxman Jagarlamudi
NetID: Sj1018

Project Overview: Account Management System

- The Account Management System is an advanced software solution designed for managing financial portfolios, encompassing both stock portfolio accounts and bank accounts. The system uses an inheritance structure with an abstract base class Account, which is extended by two derived classes: StockAccount and BankAccount.

Key Features and Functionalities

- Bank Account Management: Allows users to view, deposit, and withdraw funds, along with transaction history tracking.
- Stock Account Management: Enables users to buy and sell stocks, view their stock portfolio, and track stock transaction history.
- Data Visualization: Provides graphical representations of stock portfolio performance over time.

Technology Stack & Setup

- Programming Language: C++17
- Compiler: g++
- IDE: Visual Studio Code
- Libraries/Frameworks: Standard Template Library (STL), R for data visualization

Architecture and Design

- **Singleton Design Pattern**: Implemented in the BankAccount class to ensure a single instance across the application.
- **Strategy Design Pattern:** Used in the Sort class for flexible sorting strategies of stock data.
- **Doubly Linked List**: Utilized for efficient data manipulation in stock account management.

Modules and Class Descriptions

- **Account**: Base class for handling common account features.
- **BankAccount**: Derived from Account, manages bank-related transactions, implementing the Singleton pattern.
- **StockAccount**: Extends Account, manages stock transactions
- **Node**: Supports the doubly linked list structure in stock account management.
- **Sort**: Implements sorting algorithms (bubble sort, selection sort) for stock data organization.

Implementation Details

Account Class

1. Header File (Account_sailaxman.h):
    - Class Definition: Defines an Account class.
    - Private Member: cashBalance - A static double variable to hold the cash balance.
    - Public Constructor: Account() - Initializes the account.
    - Pure Virtual Destructor: ~Account() - A pure virtual destructor, making this a base class for derived classes.

- Public Methods: getCashBalance() const - A getter method to retrieve the current cash balance. setCashBalance(double balance) - A setter method to update the cash balance.

2. C++ Source File (Account_sailaxman.cpp):
   - Implementation of Account Class:
   - Static Member Initialization: Initializes the cashBalance with a value of 10000. Constructor Implementation: Provides the implementation for the Account constructor.
   - Destructor: Defines the pure virtual destructor for the class.
   - Getter and Setter Implementation: Implements the methods to get and set the cashBalance.

Bank Account Class

(BankAccount_sailaxman.h, BankAccount_sailaxman.cpp)

- static **BankAccount* getInstance()**; Purpose: This function is a static member of the BankAccount class. It ensures that there is only one instance of the BankAccount class throughout the program (**Singleton pattern)**. If the instance doesn't exist, it creates it; if it does exist, it returns the existing instance. (more details about design pattern below)
- void **loadAccountBalance()**; Purpose: This function loads the account balance from a persistent storage (like a file: portfolio_history.txt file here). It's responsible for initializing the account's balance when the program starts.
- void **deposit(**double amount); Purpose: This function handles deposit transactions. It increases the account's balance by the specified amount.
- void **withdraw**(double amount); Purpose: This function manages withdrawal transactions. It decreases the account's balance by the specified amount, assuming there are sufficient funds.
- **BankAccount()**; Purpose: This is the constructor for the BankAccount class. It's private in the context of the Singleton pattern to prevent direct instantiation of the class from outside.
- virtual ~**BankAccount()**; Purpose: The destructor for the BankAccount class. It cleans up when an instance of the class is destroyed.

- void **addTransaction**(const std::string& type, double amount); Purpose: This function adds a transaction record to the account's transaction history. It records the type of transaction (like "Deposit" or "Withdrawal").

Transaction History Format (bank_transaction_history.txt): Each line in the file follows this format: [Transaction Type: e.g., Withdrawal, Deposit] [Amount: The value of the transaction] [Date: The date of the transaction] [Balance After Transaction]

Portfolio History Format(portfolio_history.txt): Each line in the file follows [Cash Balance] [Total Portfolio Value] [Timestamp (Seconds)] [Date]. This file is used by both Bank Account Class, as well as Stock Account class

Singleton Design Pattern Implementation of Bank Account

Static Instance Member:

- static BankAccount* instance;
- A static member variable instance is used to hold the single instance of the BankAccount class. Being static ensures that there is only one instance across all objects of the class.

Private Constructor:

- BankAccount();
- The constructor is made private to prevent instantiation of the BankAccount class from outside the class. This ensures control over the creation of the instance, adhering to the singleton pattern.

Static Access Method:

- static BankAccount* getInstance();
- Provides a global access point to the singleton instance. This method creates the instance if it doesn't already exist and returns it. If the instance already exists, it simply returns the existing instance. This method is crucial for accessing the singleton instance.

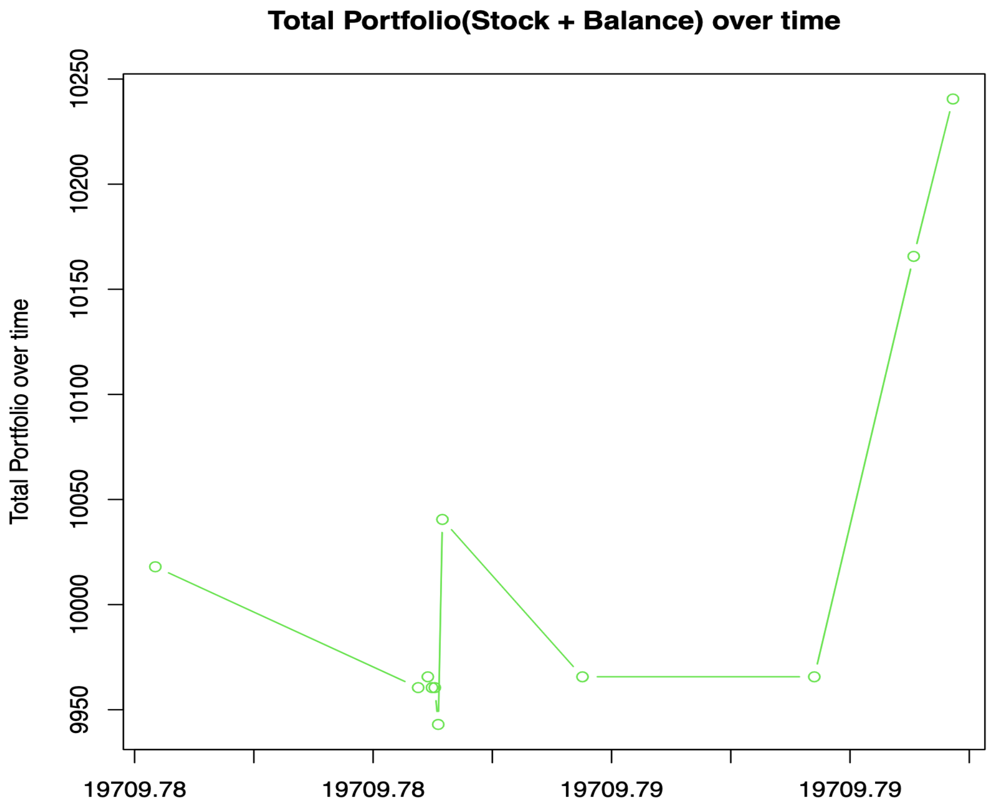
Deleted Copy Constructor and Assignment Operator:

- BankAccount(BankAccount const&) = delete;

- BankAccount& operator=(BankAccount const&) = delete;
- Both the copy constructor and assignment operator are deleted to prevent copying of the singleton instance. This is an important feature to maintain the singleton property, ensuring that no additional copies of the object can be created through copying or assignment.

Stock Account Class

- **StockAccount()**; Constructor:
  - ❖ Initializes Head to nullptr and sz (size) to 0. Head points to the start of a doubly linked list (DLL) that will store stock information.
  - ❖ Sorting Strategy: Sets the sorting strategy for the stock list to Selection Sort using sortContext.setSortStrategy(new SortSelection());.
  - ❖ Reading Transaction History: Opens and reads from "stock_transaction_history.txt" to Uses an unordered_map<string, int> stockCounts to keep track of the net quantity of each stock symbol based on buy and sell transactions and creates Doubly Linked List of Nodes from this map.
- ~**StockAccount()**; Destructor: Cleans up resources when an instance of StockAccount is destroyed as well as stores portfolio
- void **storePortfolio()**; Purpose: calculates the current total stock/shares values+ cashBalance with timestamps and saves it to portfolio_history.txt
- void **viewTransactionHistory()**; Purpose: Displays the transaction history of the stock account, showing details of all buy and sell operations performed.
- void **plotGraph()**; Purpose: Plots 2 graphs using R which creates Rplots.pdf which has i.e Executes an R script to plot portfolio graphs(**plotPortfolioGraphs.R**)
  1. The total portfolio value (stocks value + balance) over time.
  2. The change in total portfolio value(stocks value + balance) over time.



Example figure showing portfolio value over time(in seconds from 1970, Jan 1st)

- void **displayPortfolio()**; Purpose: Displays the current stock portfolio, showing the stocks owned, quantities, and their current values.
- void **updateStockPrices()**; Purpose: Updates the prices of stocks in the portfolio, by randomly fetching values from Result files(I have created 2 more Result files to mimic stock movement)
- void **buyShares(string, int, double)**; Purpose: Handles the purchase of stock shares. It takes the stock symbol, the number of shares to buy, and the price per share as arguments and as well as saves transaction to stock_transaction_history.txt, withdraws amount from the cash balance.
- void **sellShares(string, int, double)**; Purpose: Manages the selling of stock shares. It requires the stock symbol, the number of shares to sell, and the price per share as well as saves transaction to stock_transaction_history.txt, withdraws amount from the cash balance.

- double **getStockPrice(const string&)**; Purpose: randomly fetches the current price of a stock identified by its symbol from one of the result files.
- void **saveTransaction**(const string& type, const string& stockName, int numOfShares, double currAmtPerShare); Purpose: Records a stock transaction (buy or sell) in the transaction history (stock_transaction_history.txt), including details like the type of transaction, stock name, number of shares, and the amount per share.

Node & Sort:

- **Node**(string name, int n, double pps); A constructor for the Node class that initializes a node with a stock name (name), number of shares (n), and price per share (pps).
- static void **swap**(Node*, Node*); A static member function that swaps two nodes in a linked list.
- virtual Node* **sort**(Node* Begin); A virtual function declared in the base SortImplementation class. It's designed to be overridden in derived classes to implement specific sorting algorithms. This function takes a pointer to the beginning of a linked list (Node *Begin) and returns a pointer to the sorted linked list.

Strategy Design Pattern for Sorting

- **SortContext()**; A constructor for the SortContext class. This class is part of the Strategy pattern, allowing the sorting strategy to be set at runtime.
- **~SortContext()**; A destructor for the SortContext class. It ensures proper cleanup of resources associated with the SortContext object.
- void **setSortStrategy**(SortImplementation* newStrategy); A member function of the SortContext class that sets the sorting strategy. It takes a pointer to a SortImplementation object (newStrategy), allowing the sorting behavior to be dynamically changed.
- Node* **executeSort(**Node* Begin); A member function of the SortContext class that executes the sorting operation. It takes a pointer to the beginning of a linked list and

returns a pointer to the sorted list. This function delegates the sorting task to the current sorting strategy set in SortContext.

The SortContext class allows the client to set different sorting strategies at runtime, using setSortStrategy function. Each sorting strategy is a derived class of SortImplementation and overrides the sort function to provide the (selection or bubble) algorithm's logic. This design allows for easy extension and modification of sorting behavior without changing the client code, adhering to the open/closed principle of software engineering.

SortSelection's sort Method

- Functionality: Implements the selection sort algorithm for a linked list of Node objects.
- Process:
    - Iterates through the linked list to find the largest node based on the product of numOfShares and pricePerShare.
    - Swaps the found largest node with the current node in the iteration.
    - Continues the process until the entire list is sorted in descending order of the value (numOfShares * pricePerShare).
- In-Place Sorting: The sorting is performed in-place by swapping nodes within the linked list without using any additional data structures. This is achieved through the Node::swap(curr, largest) function.

SortBubble's sort Method

- Functionality: Implements the bubble sort algorithm for a linked list of Node objects.
- Process:
    - Repeatedly steps through the list, compares adjacent nodes, and swaps them if they are in the wrong order.
    - The pass through the list is repeated until no swaps are needed, indicating that the list is sorted.
    - The sorting is done in descending order based on the value (numOfShares * pricePerShare).

- In-Place Sorting: Similar to the selection sort, bubble sort is also done in-place. Nodes are swapped within the list using Node::swap(curr, curr->next) without any extra space or data structures.

Conclusion:

In conclusion, the Account Management System is a well-optimized, memory-efficient software application, designed following the best practices of Object-Oriented Programming (OOP). Key aspects include:

- **Memory Management**: Careful attention has been paid to prevent memory leaks, especially in dynamic data structures like the custom doubly linked list in the StockAccount class.
- **Use of Virtual Destructors**: Virtual destructors in the base and derived classes ensure proper resource cleanup, avoiding memory issues and enhancing stability.
- **Adherence to OOP Principles**: The system effectively utilizes OOP concepts such as encapsulation, inheritance, and polymorphism, leading to a codebase that is both reusable and extendable.