## CS 1430 - Program 5

## Due Date: Friday, November 21, 2014 at 10:00 PM Grace Date: Tuesday, November 25, 2014 at 10:00 PM 25 Points

## Program Description

This program keeps track of the transactions of a Day Trader (DT). The DT constantly buys and sells stocks to try to make a profit. When the DT buys a stock, they hope to be able to sell it at a higher price and make a profit. The DT can also sell a stock they don’t have (sell short) with the hopes of buying it back at a lower price to make a profit. The DT program will maintain a list of stocks. Whenever a stock is bought or sold, it is added to the list if it isn’t already in the list; otherwise, the data for the stock in the list is updated. If the number of shares owned or sold short for a particular stock in the list becomes zero, it is removed from the list. The list is a fixed size, so there are a maximum number of stocks that a DT can be “juggling” at one time. The DT pays a $5 transaction fee for each stock that the DT buys or sells, i.e., for each transaction. For simplicity, we will assume that the price of a stock will never be zero. Thus, if the value of a stock is zero, then the number of shares must be zero.

## Input and Output Description

You will read and process commands from the standard input, quitting if the Q command is entered. Commands consist of a single character, possibly followed by parameters.

The commands you are to implement are of the form:

T name price shares

D

V <value type>

Q

1. T – Transaction. Name and price are the name and price of the stock. Shares are the number of shares, where a positive number means buy and a negative number means sell. If the stock isn’t already in the list and the list isn’t full, the stock is added to the end of the list. If the stock isn’t in the list and the list is full, an error message is printed. If the stock is in the list, it is updated and the profit is updated, as specified below. If the number of shares for a stock gets to 0, remove it from the list, leaving the rest of the list in the same relative order. A $5 transaction fee is charged for each successful transaction. A simple completion message is printed for cases where this is no error message printed.
2. D – Display the stocks in the list, one per line, in the format specified below.
3. V – Value. Display the value specified by value type. If the value type is a P, then print the current value of the profit. (Other types will be added in the next program.)
4. Q – Quit command.

You also need to check for and handle bad commands. For a bad command, a message should be displayed, the rest of the input line should be ignored, and the program should continue to the next line of input. For a T command, you can assume that the rest of the data on the line are okay. For the V command, you must handle bad options: a message should be displayed, the rest of the input line should be ignored, and the program should continue to the next line of input.

**See the sample output for the exact wording of all the messages.**

## Requirements

There are many places below where you can lose many points, possibly all points down to 0 even if your program runs with no differences (0 is the lowest you can get on a program). If are tempted to do one of the things not allowed, see your instructor. It means you do not understand a key concept of Object-Oriented Programming. When you think you have completed your program, go over the list below one more time, since the point loss for not following the requirements is **VERY SEVERVE**!

You will work in your group. You must follow the Group Requirements. As a reminder, you are required to have each person's estimate of what percent each member contributed to the completion of the program in the top header block. The percentages must add up to one hundred. See GroupRequirements.txt for an example. Expect to **lose up to 3 points** on the program if this is not done. Students not doing their share can expect their grade lowered. If there are significant differences in what is reported (estimated), the instructor can call in the group to discuss.

To get minimal credit for the assignment, your solution must completely work (0 Differences) on test 1. Besides the normal requirements such as follow the programming ground rules and use good, natural decomposition, you must adhere to the additional requirements below.

Often when people are first introduced to OO, their tendency is to make methods that are nothing more than getters or setters for the data. Such tendencies result in “bad OO”. Good classes are behaviorally rich, providing good functionality independent of the particular way the data are stored. This will be covered in much more depth in the next courses. For now, to force you to use “good” OO and get you used to using a class as specified, you are required to use the declarations below. There are substantial point loss penalties if you don’t.

**1. You must use the following constants (and you must declare more constants).**

const int NOT\_FOUND = -1;

const int MAX\_LINE\_LENGTH = 120;

const int MAX\_STOCKS = 6;

const float TRANSACTION\_FEE = 5.0;

The MAX\_STOCKS and NOT\_FOUND constants should only be used in the Portfolio class.

**2. You must use and finish the following for the Stock class.**

Copy/paste the Stock class below to your program and write the bodies for the methods to satisfy the comment block descriptions. You are not allowed to add any more methods/data, change the prototype or description of any method or change a private member to be public.

**You will lose 12 points** if you do any of these things or if you try to manipulate the data related to a stock (name, price, shares) in any other way other than using this class. As an example of a wrong way and right way to use this class, consider the T command. If you read the name, price, and shares into variables in a function below main, that is the wrong way. The right way to read in this data is something like:

Stock inputStock;

inputStock.Read();

In all the method comment block descriptions, the stock stored in the private data (name, price, shares) is referred to as "**this stock**". It should be noted that all of the methods are easy to write except Transaction. That one has a few tricky pieces. We will use the weighted average to keep track of each stock’s price over several transactions. As an example of how a weighted average is calculated, supposed we have two items, A and B, where A\_quant is the number of A items at a cost of A\_cost and B\_quant is the number of B items at a cost of B\_cost. The weighted average of their costs is:  
 (A\_cost \* A\_quant + B\_cost \* B\_quant) / (A\_quant + B\_quant)

For absolute value, include <cmath> and use **abs**. Since Transaction is tricky, we gave you the first couple of lines.

Here are some examples of how the **Transaction** method of the Stock class works. **Be sure to read the Transaction method comment block below before working through this table.** Assume S is the stock, U is the update. Work though these examples by hand! Each line is considered to be a **separate example**. Several of these are in the sample input / output. Note that these examples do not include the $5 transaction fee. **The Stock Transaction method should not handle the transaction fee: -2 if it does.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| S.price  Before | S.shares  Before | U.price | U.shares | S.price  After | S.shares  After | Profit |
| 10 | 100 | 20 | 40  (Buy) | 12.857 = (10\*100 + 20\*40)/(100+40) | 140 = (100 + 40) | 0 |
| 10 | 100 | 20 | -40  (Sell) | 10 since abs(100) > abs(-40) | 60 = (100 + (-40)) | 400 = abs(-40) \* (20-10) |
| 10 | 100 | 20 | -150  (Sell) | 20 since abs(-150) > abs(100) | -50 = (100 +  (-150)) | 1000 = abs(100)\*  (20-10) |
| 10 | 100 | 5 | -100  (Sell) | Either 5 or 10, depending on your IF tests | 0 | -500 = abs(±100) \*  (5 – 10) |
| 10 | -100 | 20 | -50  (Sell) | 13.333 = (10\*  (-100) + 20\*(-50)) / ((-100) + (-50)) | -150 =  (-100 + (-50)) | 0 |
| 10 | -100 | 20 | 50  (Buy) | 10 since abs(-100) > abs(50) | -50 =  (-100 + 50) | -500 =  - abs(50) \*  (20 – 10) |
| 10 | -100 | 20 | 150  (Buy) | 20 since abs(150) > abs(-100) | 50 =  (-100 + 150) | -1000 =  - abs(-100) \*  (20 – 10) |
| 10 | -100 | 5 | 150  (Buy) | 5 since abs(150) > abs(-100) | 50 =  (-100 + 150) | 500 =  - abs(-100) \*  (5 – 10) |
| 10 | -100 | 5 | 100  (Buy) | Either 5 or 10, depending on your IF tests | 0 | 500 =  - abs(±100) \*  (5 – 10) |

class Stock

{

private:

string name; // Name of the stock

float price; // Price of the stock

int shares; // Number of shares: neg means sold, pos means bought

public:

//-------------------------------------------------------------------

// This method reads the name, price, and number of shares for this

// stock from the standard input. It assumes the data are

// separated by white space and in the correct format. It also

// assumes name is a contiguous sequence of non-white characters.

// A positive number for shares means the stock was bought, a

// negative number means the stock was sold.

// params: (none)

//-------------------------------------------------------------------

void Read()

{

}

//-------------------------------------------------------------------

// This method returns the value of this stock: shares times price.

// A positive number means the stock was bought, a negative number

// means the stock was sold.

// params: (none)

//-------------------------------------------------------------------

float Value() const

{

}

//-------------------------------------------------------------------

// This method prints the data for this stock in the format:

// name (width of 10 characters)

// shares (width of 6 characters)

// price (width of 8 characters with 2 decimal places)

// value (width of 10 characters with 2 decimal places)

// followed by a newline.

// params: (none)

//-------------------------------------------------------------------

void Print() const

{

}

//-------------------------------------------------------------------

// This method compares the name of this stock to the name of

// compareStock. It returns a negative number if the name of this

// stock is lexicographically (dictionary order) less than the name

// of compareStock, a positive number if it is greater, and 0 if

// they are equal.

// params: (in)

//-------------------------------------------------------------------

int CompareByName( const Stock & compareStock ) const

{

}

//-------------------------------------------------------------------

// This method updates this stock with the data from updStock and

// returns any resulting profit.

//

// If the number of shares for this stock and updStock have the same

// sign (both positive or both negative), then there is no profit

// (just buying or selling more) and the price of this stock

// is updated to be the weighted average of the prices.

//

// Otherwise, profit is the minimum of the absolute values of the

// shares times the difference between the prices of updStock and

// this stock. If the number of shares of this stock is negative,

// the sign of the profit is reversed. The price of this stock is

// updated to be the price of whichever of this stock or updStock

// has the larger absolute value of shares.

//

// In all cases, the number of shares for this stock is updated by

// adding in the number of shares for updStock.

//

// Note that a negative value for the profit indicates a loss.

// params: (in)

//-------------------------------------------------------------------

float Transaction( const Stock & updStock )

{

float profit = 0.0;

if ( shares < 0 && updStock.shares < 0 ||

shares > 0 && updStock.shares > 0 )

price = (shares \* price + updStock.shares \* updStock.price) /

(shares + updStock.shares );

else if (

}

}; // Stock

**3. You must use and finish the following for the Portfolio class.**

Copy/paste the Portfolio class below to your program and finish as specified, removing the commentary comments when finished and assuring all method comment blocks follow the programming ground rules. The additional rules you **MUST** adhere to when completing this class are:

1. You cannot add any more data and you can’t make the private data or private methods public. **-10 if you do**.
2. No method of Portfolio is allowed to read anything from standard input: **-5 if they do**.
3. The only method allowed to print anything to standard output in Portfolio is the Print method. It must print the stocks in the list, one per line. **-5 for printing in the Portfolio class other than the Print method**.
4. No method of Portfolio can return numStocks or stocks or any array element from stocks. Furthermore, no method in Portfolio can take an array index as a parameter except the private method Remove. **-5 if they do**.
5. You can have a method that returns the Profit.
6. You cannot do any array related computations or have any array code in main or any of the functions below main. **-5 if you do**

class Portfolio

{

private:

float profit;

int numStocks;

Stock stocks[MAX\_STOCKS];

// Finish the bodies of the private methods.

// **Remove the comments and replace them**

// **with good “what” comments** and do NOT

// reference how they will be used.

// That is for your benefit.

// **Make sure to remove this comment and**

**// all other such commentary.**

// Returns the index in stocks where stock with

// the same name as findStock.name is found;

// returns NOT\_FOUND otherwise. But you can’t

// access findStock.name! So

// what Stock method do you have to use?

int Find( const Stock & findStock ) const

{

}

// Remove item from array at the specified index.

// Shoves the ones below it up one.

// You'll use this to delete a stock

void Remove( int index )

{

}

public:

// Make a default (no parameters) constructor.

// What should it set the data to?

// Here is a good method to have. It can’t print anything,

// but it can do a bunch of heavy lifting for the T command.

// It will return false if updStock isn’t in the list and

// the list is full. Otherwise, it will behave as indicated

// by the T command above and return true. It will make

// use of the Transaction method in Stock as well as the

// private Find and Remove methods. It will also update

// the profit. Be sure to charge the $5 for each Transaction.

// All this work and it can easily be written in under 20

// lines!

bool Transaction(const Stock & updStock)

{

}

// Make other good methods (such as Print)

// subject to the rules above.

}; // Portfolio

**4. You must have several stand-alone functions besides main.**

Put the prototypes for the stand-alone functions after the Portfolio class and put the functions themselves after main (as always). All methods and functions, including main, must be less 30 lines.

**5. You must use a switch statement.**

You must use a switch statement to process the commands. You will **lose 2 points** if you don't use a switch statement.

Some hints:

1. You can use magic numbers for setw and setprecision. You don’t need to make constants.
2. Put the following early in main to get two decimal places in the float output:

cout << fixed << setprecision(2) << showpoint;

1. #include <cmath>
2. Use the cmath function **abs** to get the absolute value
3. Use setw for the Print method of stock.
4. For the Bad Command case, use the following to read in the rest of the line and toss it away:

cin.ignore( MAX\_LINE\_LENGTH, '\n' );

# **Sample Input and Output**

The following input is in the share as a file that you can use to test your program.

**Sample Input:**

D

T GE 10 100

T Cisco 10 100

T Adobe 10 100

D

V P

T Apple 10 -100

T VMWare 10 -100

T Dell 10 -100

T Deere 100 -10

D

V P

T GE 20 40

V P

T Cisco 20 -40

V P

T Adobe 20 -150

D

V P

V X

T GE 5 -140

V P

D

T Deere 10 -100

D

T Apple 20 50

V P

T VMWare 20 150

V P

T Dell 5 +150

D

V P

D

D

V T

S X

B ABC 10 10

V P

Q

Don't read down here!

**Corresponding Sample Output:**

Recall that prices are stored to fractional decimal places but rounded to two decimal places, so taking the price times the shares may not give exactly equal the value but should always be within a penny.

Transaction completed.

Transaction completed.

Transaction completed.

GE 100 10.00 1000.00

Cisco 100 10.00 1000.00

Adobe 100 10.00 1000.00

Profit so far: $-15.00

Transaction completed.

Transaction completed.

Transaction completed.

Transaction NOT completed. Maximum number of stocks being monitored.

GE 100 10.00 1000.00

Cisco 100 10.00 1000.00

Adobe 100 10.00 1000.00

Apple -100 10.00 -1000.00

VMWare -100 10.00 -1000.00

Dell -100 10.00 -1000.00

Profit so far: $-30.00

Transaction completed.

Profit so far: $-35.00

Transaction completed.

Profit so far: $360.00

Transaction completed.

GE 140 12.86 1800.00

Cisco 60 10.00 600.00

Adobe -50 20.00 -1000.00

Apple -100 10.00 -1000.00

VMWare -100 10.00 -1000.00

Dell -100 10.00 -1000.00

Profit so far: $1355.00

Not a valid value option.

Transaction completed.

Profit so far: $250.00

Cisco 60 10.00 600.00

Adobe -50 20.00 -1000.00

Apple -100 10.00 -1000.00

VMWare -100 10.00 -1000.00

Dell -100 10.00 -1000.00

Transaction completed.

Cisco 60 10.00 600.00

Adobe -50 20.00 -1000.00

Apple -100 10.00 -1000.00

VMWare -100 10.00 -1000.00

Dell -100 10.00 -1000.00

Deere -100 10.00 -1000.00

Transaction completed.

Profit so far: $-260.00

Transaction completed.

Profit so far: $-1265.00

Transaction completed.

Cisco 60 10.00 600.00

Adobe -50 20.00 -1000.00

Apple -50 10.00 -500.00

VMWare 50 20.00 1000.00

Dell 50 5.00 250.00

Deere -100 10.00 -1000.00

Profit so far: $-770.00

Cisco 60 10.00 600.00

Adobe -50 20.00 -1000.00

Apple -50 10.00 -500.00

VMWare 50 20.00 1000.00

Dell 50 5.00 250.00

Deere -100 10.00 -1000.00

Cisco 60 10.00 600.00

Adobe -50 20.00 -1000.00

Apple -50 10.00 -500.00

VMWare 50 20.00 1000.00

Dell 50 5.00 250.00

Deere -100 10.00 -1000.00

Not a valid value option.

Bad command.

Bad command.

Profit so far: $-770.00