

Data Encapsulation and Management:**Stock Portfolio Database**

For this project you will modify and extend your implementation for Project 3 to take advantage of arrays and structures to improve the organization of the data and to support additional operations. You will also explore searching and sorting arrays of structures.

There will be two input files. The first will contain data about a list of stocks. The second will be a script file containing database queries and actions that your program must interpret and carry out.

On startup, your program will read the stock data file and organize the given data in an array of structures. To do this, you must design and use an appropriate `struct` type. Your program will then read the script file and service the queries or carry out the actions, writing descriptive output to a report file.

Database queries and actions:

Your program must be able to service the queries and actions specified below. In general, each query or action will require performing a search for a stock matching some criterion and then reporting or modifying some values for that stock.

show<tab><stock name>

Search the database for a stock with the given name. If a match is found, log the stock name, number of shares, buy price and current price, and the total value of all the held shares to the report file, along with the index at which the stock was found. If no match is found, log the error message "Not found" to the report file.

net change<tab><stock name>

Search the database for a stock with the given name. If a match is found, log the change in the total value of the held shares since the stock was bought. If no match is found, log the error message "Not found" to the report file.

buy<tab><stock name><tab><shares>

Search the database for a stock with the given name. If a match is found, increase the number of shares held by the specified amount and log the message "Bought for" followed by the total amount paid for the shares that were bought, assuming the current price was paid. If no match is found, log the error message "Not found" to the report file.

sell<tab><stock name><tab><shares>

Search the database for a stock with the given name. If a match is found, decrease the number of shares held by the specified amount and log the message "Sold for" followed by the total value of the sold shares. If no match is found, log the error message "Not found" to the report file. If the number of shares held is insufficient, log the error message "Not enough shares".

quote<tab><stock name><tab><price>

Search the database for a stock with the given name. If a match is found, change the current price to the specified amount and log the message "<stock name > updated to" followed by the new current price. If no match is found, log the error message "Not found" to the report file.

sort by<tab>[**name** | **value**]

Sort the database into ascending order, either with respect to the stock name or the total value of the held shares stocks. You must use the bubble sort algorithm when sorting by value and the selection sort algorithm when sorting by name. In either case, log the total number of swaps performed during the sort to the report file.

total value<tab>

Log the total value of all held shares of all stocks to the report file.

display<tab>

Log a list of all stock information from the database. The display must include the index number for each stock, followed by the stock name, number of shares, opening and current prices, and the total value of the shares held. See the sample log file for formatting details.

exit<tab>

Stop processing the script file and log the message "Exiting script processing".

Sample input data files:

The stock data file will be named `PortfolioDB.txt`. There is no customer information. There is a two-line header, followed by a list of data lines. For each stock, you will be given the name, number of shares held, the price per share at which the stock was purchased, and the price of a share. These values are separated by single tab characters.

Stock	Shares	Bought	Current
Microsft	467	45.67	44.62
IBM	985	67.90	71.01
Lucent	413	1.60	4.23
DellCptr	811	24.32	21.44
AT&T	1017	9.31	6.12
CSX	404	33.66	41.32

The number of lines of stock data that may be supplied will vary, but there will never be data for more than 100 stocks; your program must terminate properly when the input stream fails. There will be no sentinel line. The input values are guaranteed to be syntactically and logically correct.

The script file will be named `Queries.txt`. There is no limit on the number of queries that may be supplied, but you should design your program to terminate correctly if it encounters an `exit` command or an input failure occurs.

```

show      Microsft
net change      Microsft
sell      Microsft      100
show      Microsft
total value
buy       Microsft      200
show      Microsft
show      Citi
quote    DellCptr      22.50
show      DellCptr
sort by   name
display
sort by   value
display
exit

```

A sample log file corresponding to these input files is given at the end of this specification.

Design and implementation:

Note: the comments on design are more than just advice; the comments on implementation are merely (good) advice.

The first thing you need to design is a `struct` type that will represent a single stock. It's up to you to determine what data members the type should have, but all of the data relevant to a single stock must be stored within a single `struct` variable. In order to aid in detecting errors, you should initialize the entire data array to hold recognizable, dummy values.

You should practice good procedural decomposition in your design. Reading the stock data from the file would be a good candidate for a function. Managing the reading and interpretation of the queries and actions from the script would be

another good job for a function. The details of handling each particular query or action could be delegated to a particular "handler" function. Of course, some of the handlers might also use helper functions, to perform tasks such as sorting or searching. In addition, there are the usual opportunities to have functions to handle mundane details, such as writing the file header. For reference, my solution uses 21 functions in addition to `main()`.

You should take care to pass parameters intelligently. In particular, do not pass the array of stock data by reference unless the logic of the function requires it. Do not pass unnecessary parameters. Avoid implementing a function whose body is only one statement unless the function is called from more than one place.

As always, you should practice incremental development. Begin by implementing your structured type and the logic needed to read the stock data and store it in the array of structured variables. Write output code to display the full list of stock data and verify that your input logic is correct.

Next implement a function to read the commands from the script file, and identify each command. Knowing which command is given, you can then select what actions to take. Initially, just log a message that identifies the command, but does not attempt to carry it out. Once you're sure you are reading and classifying the commands correctly, add handlers for each command.

Add and test the handlers one by one; test each one thoroughly before moving on to the next one. Some handlers will be more complicated than others. The most critical operation is searching, since that is used in servicing several commands. Be sure to test your search code thoroughly! The most complex commands are probably the sorting commands. You will need two different sort functions, since two different sort algorithms are required. (Yes, I know you could combine those into a single, overly-complex, non-reusable function.)

If you do this well, you will produce a design for handling script-driven processing that you can reuse in later projects (in later courses).

Evaluation:

Everything that you have been told about testing in class applies here. Do not waste submissions to the Curator in testing your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that are provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, based upon the runtime testing performed by the Curator System. We will also be evaluating your submission of this program for documentation style and a few good coding practices. This will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

Read the *Programming Standards* page on the CS 1044 website for general guidelines. You should comment your code in the same manner as the code given for the first two programming assignments. In particular:

- You should have a header comment identifying yourself, and describing what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every function should have a header comment, as described in the notes and the *Programming Standards* page.
- Every major block of code should have a comment describing its purpose.
- Adopt a consistent indentation style and stick to it.

Your implementation must also meet the following requirements:

- Choose descriptive identifiers when you declare a variable or constant. Avoid choosing identifiers that are entirely lower-case.
- Use named constants instead of literal constants when the constant has logical significance.
- Use C++ streams for input and output, not C-style constructs.
- Use C++ string variables to hold character data, not C-style character pointers or arrays.
- Use C++ functions intelligently and appropriately. In particular, you must implement and use at least 4 functions in addition to `main()`. At least one of these functions must be `void`; at least one must return a value via a

return statement, and at least one must use a reference parameter to return a value. It is not acceptable that all function calls are from `main()` to other functions.

- Use the appropriate protocol when passing each parameter to a function. In particular, do not pass any parameter by reference unless it is being used to communicate a changed value from the called function to its caller.
- In a good design, each function is responsible for performing a single, constrained task, or for managing other functions. To promote that, none of the functions in your implementation should contain more than 30 executable statements. Comments and variable and type declarations are not executable; everything else is.

Understand that the list of requirements here is not a complete repetition of the *Programming Standards* page on the course website. It is possible that requirements listed there will be applied, even if they are not listed here.

This project requires implement quite a few features. None of them are terribly difficult to achieve, but there are quite a few. As a general rule, if you do not manage to complete everything, it is better to handle some of the commands completely correctly and to ignore the rest. Of course, that won't produce a good score unless you're handling most of the commands correctly, but it will still be better than having a lot of "almost there" handlers but none or few that are really correct.

Finally... this project serves in some sense as an exit exam for CS 1044. Students who cannot complete this project have very little chance of passing CS 1704. In order to give you the best possible chance of succeeding, I have included a relatively detailed discussion of design and implementation. I have also made the due date for the project as late as possible and still give the TAs time to evaluate your submissions fairly. As a result, there will be a very small window for late submissions. No submissions of this project will be accepted after midnight on the last day of classes.

Submitting your program:

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* and submission link can be found at:

<http://www.cs.vt.edu/curator/>

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
// On my honor:
//
// - I have not discussed the C++ language code in my program with
//   anyone other than my instructor or the teaching assistants
//   assigned to this course.
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - If any C++ language code or documentation used in my program
//   was obtained from another source, such as a text book or course
//   notes, that has been clearly noted with a proper citation in
//   the comments of my program.
//
// - I have not designed this program in such a way as to defeat or
//   interfere with the normal operation of the Curator System.
//
```

```
// <Student Name>
```

Failure to include this pledge in a submission is a violation of the Honor Code.

Sample report:

The log file must be named QueryLog.txt. Here is a sample log file that corresponds to the input data given above:

```
Programmer: Bill McQuain
Portfolio Database

-----
show Microsft
  0:           Microsft    467    45.67    44.62    20837.54
-----
net change Microsft
-490.35
-----
sell 100 of Microsft
Sold for 4462.00
-----
show Microsft
  0:           Microsft    367    45.67    44.62    16375.54
-----
total value
128372.54
-----
Buy 200 of Microsft
Bought for      8924.00
-----
show Microsft
  0:           Microsft    567    45.67    44.62    25299.54
-----
show Citi
Not found
-----
quote DellCptr    22.50
DellCptr updated to 22.50
-----
show DellCptr
  3:           DellCptr    811    24.32    22.50    18247.50
-----
sort by name
Swaps: 5
-----
display
  0:           AT&T      1017     9.31     6.12     6224.04
  1:           CSX       404    33.66    41.32    16693.28
  2:           DellCptr   811    24.32    22.50    18247.50
  3:           IBM       985    67.90    71.01    69944.85
  4:           Lucent    413     1.60     4.23    1746.99
  5:           Microsft  567    45.67    44.62    25299.54
-----
sort by value
Swaps: 5
-----
display
  0:           Lucent    413     1.60     4.23    1746.99
```

1:	AT&T	1017	9.31	6.12	6224.04
2:	CSX	404	33.66	41.32	16693.28
3:	DellCptr	811	24.32	22.50	18247.50
4:	Microsft	567	45.67	44.62	25299.54
5:	IBM	985	67.90	71.01	69944.85

exit

Exiting script processing.
