

Fundamental Concepts: array of structures, string objects, searching and sorting

The point of this assignment is to validate your understanding of the basic concepts presented in CS 1044. If you have much difficulty implementing the following specification correctly, that may be good evidence that you are probably not ready to attempt CS 2574. Feel free to consult the online notes for CS 1044 as a reference.

Follow the given specification exactly — this assignment will be scored using an automated grading system and deviations will generally be penalized heavily.

SIMP: Static Inventory Maintenance Program

The Mall-Wart Mercantile Co needs to track its current inventory. The program will read an input file that will specify the initial inventory (see inventory file description below). The initial inventory will consist of a number of different items; for each item, a product number, a text description, the cost of one unit and the price of one unit will be specified. You should `typedef` an appropriate structure variable to store all the information about a particular item.

The program will first read and store the initial inventory data, in a sort of simple in-memory database structure, and then process a second input file specifying actions to take on the inventory data. The supported actions include adding an item to the database, deleting an item from the database, sorting the database on various keys, and dumping a display of the database to file.

When all the specified actions have been processed, the program will exit.

Input file descriptions and samples:

This program requires the use of two input files. The first contains the initial inventory list and will be named `inventory.data`. The second contains a list of actions to be performed on the inventory database, and will be named `actions.data`. Note that, due to the automated testing process, use of incorrect input file names will usually result in a score of zero.

A newline character will terminate each input line, including the last. You may assume that all of the input values will be syntactically correct, and that they will be given in the specified order.

Initial Inventory File

The first line of the input file specifies column labels. Each remaining line of this section of the input file will contain the following data:

- `ItemNumber` (SKU), a positive integer in the range 1000 to 9999, followed by a single tab character.
- `ItemDescription`, a character string no more than 25 characters long, followed by a single tab character and zero or more spaces.
- `UnitPrice` of this item, a positive decimal value, followed by a tab and zero or more spaces.
- `UnitCost` of one unit of this item, a positive decimal value, followed by a newline.

There will never be two different items with the same SKU in the database at the same time. Each of the other fields may be duplicated within the database. There will be no more than 50 items in the inventory database at any time.

You are **required** to use a statically-allocated array of structures to store the inventory information. Use of pointers and/or dynamic memory allocation is expressly forbidden in this assignment.

Database Actions File

Each line of the actions file will contain one of the commands described below. Commands are case-sensitive and take a fixed number of arguments. The command names will be valid and each command will include the correct number of arguments. Command arguments will be tab-delimited.

```
add <ItemNumber> <ItemDescription> <UnitPrice> <UnitCost>
```

This causes the insertion of a new inventory record into the database list. Insertion should place the new record in the proper location with respect to the current sort ordering of the list. The initial inventory list will be given in arbitrary order, and you must sort it by item number before further processing. If an add instruction specifies the SKU number of an item that's already in the list, the list will not be modified. In the event of a tie, on any sort key value other than the SKU, the new record should be placed before any other records with the same key value.

```
delete <ItemNumber>
```

This causes the deletion of the inventory record for the indicated item from the database list. If a delete instruction specifies the number of an item that's not in the list, the list will not be modified.

```
sort <FieldSpecifier>
```

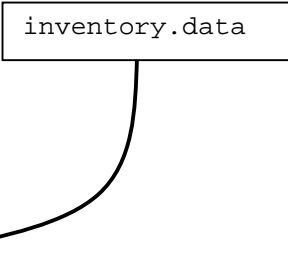
This causes the inventory list to be sorted into ascending order by the specified field. The FieldSpecifier must be one of: SKU, Description, Price or Cost. If a sort instruction specifies an invalid FieldSpecifier, the list will not be modified. You should use the selection sort algorithm.

```
dump
```

This causes the inventory list to be printed to an output file named `dbase.list`. Printing should be in the physical order of the list.

There is no guaranteed limit on the number of transactions. Example input files are shown below:

| SKU | Description | Price | Cost |
|------|---------------------------|-------|-------|
| 3396 | Large Framing Square | 10.05 | 5.77 |
| 3678 | 5.25" diskette | 15.44 | 14.59 |
| 4191 | 25-pack, Manila Envelopes | 19.83 | 18.83 |
| 2256 | 11th Century Anasazi Pot | 9.57 | 7.33 |
| 2954 | The Little LISPer | 12.97 | 9.49 |
| 2201 | #8-32 Knurled Nut | 22.75 | 18.13 |
| 2509 | Small Bevel Square | 3.35 | 2.48 |
| 2775 | 1/4" Chisel | 16.08 | 14.68 |
| 3149 | 10' RS-232 Cable | 3.58 | 1.26 |
| 3802 | #12 Auger Bit | 22.83 | 19.84 |
| 4075 | Box #4 Pan Head Screws | 17.62 | 15.27 |

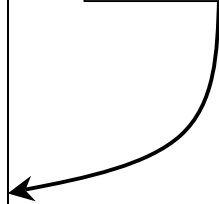


Note that the alignment of the unit numbers in the initial inventory file may not be perfect, because the combination of tabs and spaces may not align the numbers correctly. This is a good example of why we suggest that you use spaces (not tabs) to align your output. Here, we use the tab character because it makes it somewhat easier for you to parse the item descriptions.

```

add 2201 #8-32 Knurled Nut 22.75 18.13
delete 1679
delete 2954
sort Quantity
add 5564 Blueberry iMac system 16.48 16.45
delete 3149
sort Quantity
sort Description
add 5629 20 lb Chrome Steel Sinker 12.39 9.90
sort Quantity
add 9223 Necronomicon 17.18 16.19
delete 4191
add 2091 WDC31600 Hard Disk 14.92 12.05
sort Description
sort Description
delete 5564
add 9912 Java: How to Program 16.86 12.36
add 6598 5 lb 3.5" AOL Diskettes 12.11 8.24
dump
    
```

actions.data



Output description and sample:

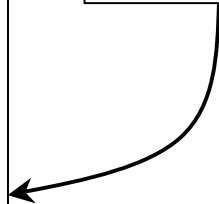
Your program must write its output data to a file named `dbase.list` — use of any other output file name **will** result in a runtime testing score of zero. Here is the correct output file corresponding to the given sample input files:

```

Programmer: Bill McQuain
Static Inventory Management Program

Number Name Price Cost
-----
3802 #12 Auger Bit 22.83 19.84
2201 #8-32 Knurled Nut 22.75 18.13
2775 1/4" Chisel 16.08 14.68
2256 11th Century Anasazi Pot 9.57 7.33
5629 20 lb Chrome Steel Sinker 12.39 9.90
6598 5 lb 3.5" AOL Diskettes 12.11 8.24
3678 5.25" diskette 15.44 14.59
4075 Box #4 Pan Head Screws 17.62 15.27
9912 Java: How to Program 16.86 12.36
3396 Large Framing Square 10.05 5.77
9223 Necronomicon 17.18 16.19
2509 Small Bevel Square 3.35 2.48
2091 WDC31600 Hard Disk 14.92 12.05
-----
    
```

dbase.list



The first line of your output should identify you by name, as shown. The second line should include the title “Static Inventory Maintenance Program” only. The third line should be blank. The fourth and fifth lines should contain the specified column labels and a row of delimiters to mark the top of the table.

Next your output file will contain a table, with a line of output for each inventory item. Each line should contain the item number, and description, the unit price and the unit cost of that item. After the last line of the table, print a line of delimiters.

You are not required to use the exact horizontal spacing shown in the example above, but your output must satisfy the following requirements:

- You must use the specified header and column labels, and print a row of delimiters before and after the table body, as shown.
- You must arrange your output in neatly aligned columns. Use spaces, not tabs to align your output.
- You must use the same ordering of the columns as shown here, and print the dollar amounts with precision two.

Programming Standards:

You'll be expected to observe good programming/documentation standards. All the discussions in class about formatting, structure, and commenting your code should be followed. Some specifics:

Documentation:

- You must include the honor pledge in your program header comment.
- You must include a header comment which describes what your program does and specifies any constraints or assumptions a user should be aware of (such as preset file names, value ranges, etc.).
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose.
- Precede every function you write with a header comment. This should explain in one sentence what the function does, then describe the logical purpose of each parameter (if any), describe the return value (if any), and state reasonable pre- and post-conditions.
- You must use indentation and blank lines to make control structures like loops and if-else statements more readable.

You are also required to conform to the coding requirements specified below.

Coding:

- Implement your solution without any user-defined classes.
- Implement your solution in a single source file, with no user-defined header files. (This restriction is for ease of testing and evaluation.)
- Use named constants instead of variables where appropriate.
- Use `double` variables for all dollar amounts.
- Use an array of structure variables to store the inventory data.
- Use C++ `string` objects, not C-style `char` arrays to store character strings (aside from string literals).
- Declare and make appropriate use of an enumerated type in your program.
- You must make good use of user-defined functions in your design and implementation. To encourage this, the body of `main()` must contain no more than 20 executable statements and the bodies of the other functions you write must each contain no more than 40 executable statements. An executable statement is any statement **other than** a constant or variable declaration, function prototype or comment. Blank lines do not count.
- You must write at least ten functions, besides `main()`. For reference, my solution uses twenty-one.
- The definition of `main()` must be the first function definition in your source file. You may use file-scoped function prototypes and you may use file-scoped constants. You may also make the `typedef` statement for your structured variable type file-scoped (in fact you must do this).
- You may not use file-scoped variables of any kind.
- Function parameters should be passed appropriately. Use pass-by-reference only when the called function needs to modify the parameter. Pass array parameters by constant reference (using `const`) when pass-by-reference is not needed.

Testing:

Obviously, you should be certain that your program produces the output given above when you use the given input files. However, verifying that your program produces correct results on a single test case does not constitute a satisfactory testing regimen.

At minimum, you should test your program on **all** the posted input/output examples given along with this specification. The same program that will be used to test your solution generated those input/output examples. You could make up and try additional input files as well; of course, you'll have to determine by hand what the correct output would be.

Submitting your solution:

You will submit your source code electronically, as described here. SIMP will be subjected to runtime testing by the Enhanced Automated Grading System (EAGS) and also scored by the GTAs for adherence to the specified programming standards. The relative weights of the two scores will be announced. You will be allowed to make up to five submissions of SIMP to the EAGS.

Instructions for submitting your program are available in the *Student Guide* at the EAGS Homepage:

<http://ei.cs.vt.edu/~eags/EAGS.html>

Read the instructions carefully.

Note well: your submission that receives the highest score will be graded for adherence to these requirements, whether it is your last submission or not. If two or more of your submissions are tied for highest, the earliest of those will be graded. Therefore: implement and comment your C++ source code with these requirements in mind from the beginning rather than planning to clean up and add comments later.

Pledge:

Each of your project submissions to the EAGS 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 EAGS Server.
```

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