Managing an Array of Structures

For this program, you will set up a very simple database of geographical information and then respond to queries and update commands on that database. The data will consist of city names, state names, and city populations. Your program will maintain a single array of `struct` variables that represent the attributes of a city (e.g., city name, state name, and population).

The Problem: Simple City DB

There will be two input files this time, one containing the initial city data and one containing a script of commands to be processed. Your program will write all its output to a log file.

Here are some guarantees. The number of cities in the database will never exceed 100. The data will never include two cities with the same name (not even if they are in different states). For formatting purposes, you may assume that city names will never be longer than 20 characters and state names will never be longer than 15 characters.

Note: the use of an array of `struct` variables is absolutely required for this assignment. Students who do not use that approach will be assigned a project score of zero.

The city data file:

The first input file, named `"CityData.txt"`, will consist of lines of data that conform to the format specification:

```
<city name><tab><state name><tab><population><newline>
```

The number of data lines is unknown (although it will never exceed 100), so your program must read until input failure at the end of the file. A sample city data file is given later in this specification.

The commands file:

The commands file, named `"Script.txt"`, will consist of single-line commands. Each of the command lines will begin with a command word, followed immediately by a single tab character, and then by one or two tab-separated values, depending upon the particular command word. The commands will be syntactically correct.

Here is a description of each of the commands your program must recognize and process:

`city<tab><city name><newline>`

> If the given city name is found in the city data array, print the data for that city (see output sample for formatting). If not, print a message* indicating the city was not found. Note that the specified city name cannot occur more than once, so your search should stop if a match is found.

`state<tab><state name><newline>`

> For each city that is in the given state, print the data for that city. If no such cities exist in the database, print an appropriate message* indicating that no such cities were found. Note that the given state name may occur many times, for different city names, so your search should not stop at the first match.

`update<tab><city name><tab><population><newline>`

> If the given city name is found in the city data array, reset the population value for that city to the value given in the command and print the updated city data. If the city name is not found, print a message* indicating that.

`range<tab><lower bound><tab><upper bound><newline>`

> For each city whose population lies between the given bounds (inclusive), print the data for that city. If no matches are found, print a message* indicating that.

`sort<tab><name | population><newline>`

The list of city data should be sorted into ascending order on the specified field, using the bubble sort algorithm from the course notes.

`exit<tab><newline>`

Stop processing the commands file immediately. This produces only a confirming message*.

* Messages must conform to the sample output.

The number of command lines is unknown; your program must read until an `exit` command is found or an input failure occurs. A sample commands file is given later in this specification.

The suggestions that were made for handling script-driven input in project 8 still apply here. Since this is a more complicated program, it is essential that you design your handling of the commands carefully.

The log file:

The output file must be named `"dbLog.txt"`. A sample log file is given later in this specification. See the sample for details of formatting.

As usual, the output begins with a header identifying the programmer and the assignment. For each command, print a comment describing the command to the log file. Immediately after that, print the specified output from the processing of that command. Whenever you print a city record you must also print the array index at which it occurs (as shown in the sample log file).

Delimiting lines, as shown, must be printed to separate successive commands.

Each of the delimiting lines and the command echo lines will be assigned zero points when your output is graded, so it is not absolutely necessary that you follow the sample log file exactly. However, if you omit the delimiting lines and/or the command echo lines, the Curator will become confused and compare the wrong lines when grading. It is important that your phrasing for all the other output lines matches that given in the sample output, both here and on the website.

Function requirements for this program:

You must make good use of functions in your implementation. There are many opportunities to do so in this project. For instance, you might use a function to initialize the city data array to hold dummy values, a function to read the initial city data into the array, a separate function to carry out each command, a function to determine what the command is, a function to print the data for a city given its index, etc.

Your design must include at least seven user-defined functions, not counting `main()`. For reference, my solution uses ten user-defined functions. It is also important that you choose the appropriate parameter-passing mechanisms, especially when you pass the array to a function. Pass parameters by reference only when it is logically necessary; otherwise, pass parameters by value or by constant reference.

Implementation suggestions:

Before reading the input files, initialize your data arrays to store `"No Name City"` for the city names, `"California"` for the state names, and –1 for the populations. (Yes, there's a joke there, but it would ruin it to explain.)

For a program of this size (about 300 lines of C++ code for my solution, not counting any documentation), it is essential that you practice incremental implementation. That is, don't attempt to write the entire program at once, even though you may have a complete design. Here is a suggested order of implementation:

- Declare the data array and initialize it to hold the specified dummy values. Print it out to verify your work.
- Read the initial city data into the data array. Print out the city data to verify your work.
- Implement reading of the commands file. Initially, don't worry about actually carrying out any of the commands, just find the command word, and any parameters, and echo them to the log file to be sure that you're reading them correctly. Also verify that you're stopping on the exit command.
- Add a function (or two) to handle the `city` command. Verify that you're producing correct results in all the logical cases (first city in list, last city in list, city that's not in the list, …).
- One by one, add functions to handle each of the other commands. Check your results for each command thoroughly before proceeding to the next.
- Add handling of the `sort` command. Be careful that you use a bubble sort, as specified, rather than another sorting algorithm. Also be careful that you pass the sort functions (yes, plural) the correct parameters.

If you get stuck on handling a command, or just run out of time, echo the command to the log file and print a message (like: "Command not implemented"). That way you'll at least have a shot at generating the correct number of output lines.

Documentation and other requirements:

You must meet the following requirements (in addition to designing and implementing a program that merely produces correct output):

- Write a header comment with your identification information, the required pledge statement (below), and a brief description of what the program does.
- Write a comment explaining the purpose of every variable and named constant you use.
- Write comments describing what most of the statements in your program do.
- Use descriptive identifiers for variables and for constants.
- Use named constants instead of "magic numbers" whenever it is appropriate. Note: there <u>are</u> some candidates in this category.
- Use `string` variables to store the command strings. The use of `char` arrays and/or `char` pointers is explicitly banned.
- Design and implement functions, as specified above.
- The first function implemented in your source file must be `main()` – so you must provide appropriate prototypes for each of your functions.
- Each function <u>implementation</u> must be accompanied by a header comment that describes what the function does, the logical purpose of each parameter (including whether it is input, output, or input/output), the pre-conditions that a function call assumes and the post-conditions that are guaranteed, the return value (if any), a list of the functions that call this function, and a list of other functions called by this function (if any). An acceptable sample function header is given on the course website.

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 ten 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* can be found at: `http://courses.cs.vt.edu/curator/`

The submission page can be found at: `http://newcurator.cs.vt.edu:8080/2009FallS01/`

Evaluation:

Your submitted program will be assigned a score based upon the runtime testing performed by the Curator System.

The TAs will also evaluate your submission of this program to see whether you followed the documentation and implementation requirements given in this specification. If you do not, your score will be adjusted (downward) accordingly.

Note that this time the TAs will be conducting a careful examination of your submission for internal documentation; you are expected to follow the requirements given in this specification precisely. Your instructor will specify how the TAs scoring of your submission will be counted.

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.

Handling script-driven computations

There are a number of ways to parse the sort of commands file used in this project. I will describe one good approach (given what we've covered so far) here. Aside from any header lines, each line of the commands file has the form

```
<command string>{<tab><parameter>}*
```

where:

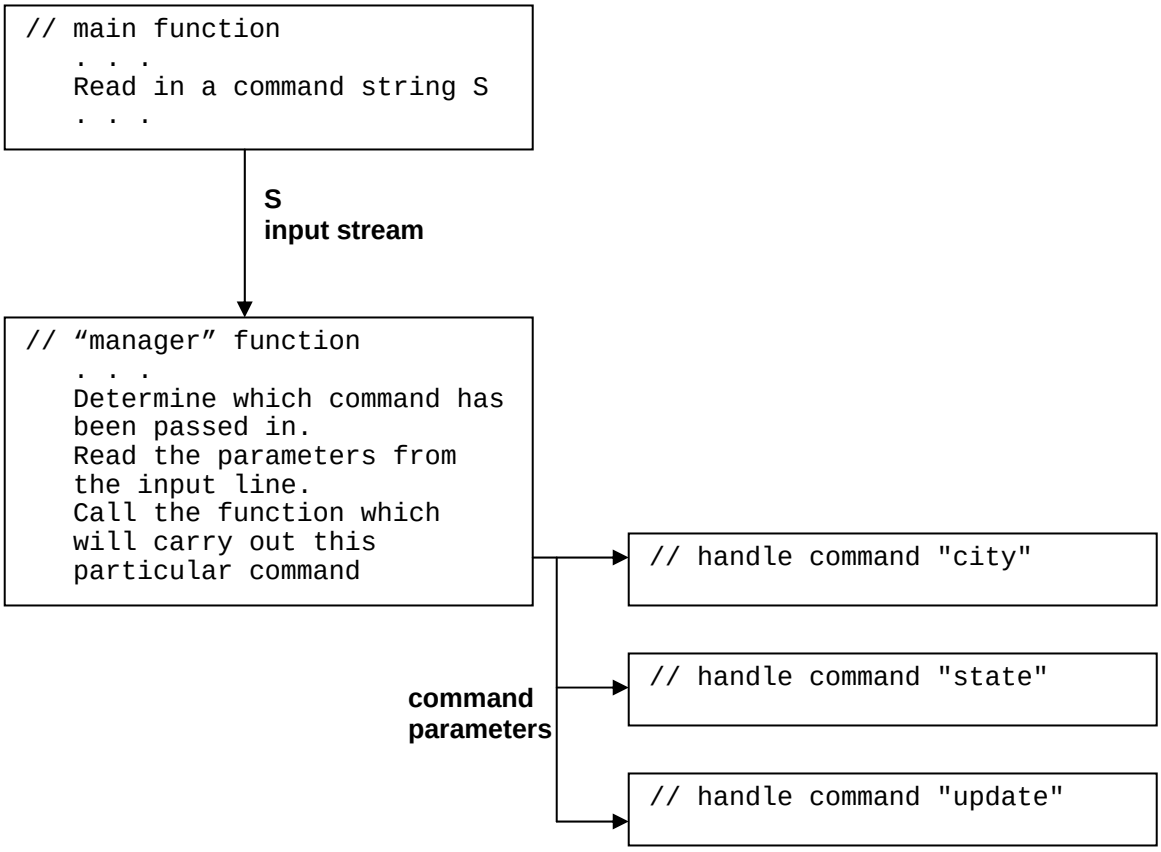
`command string`	denotes any of the specified command words (or phrases)
`parameter`	denotes any valid parameter (index, integer value, etc.) for the given command string

The asterisk simply indicates that the term enclosed in the braces may occur zero or more times (actually, one or more times for most of the commands in this project).

Typically, the particular command string that is used will determine the number of parameters. In this project, for example, the `city` and `state` commands each take one parameter, while the `update` command takes two and the `exit` command takes none. So, it is necessary to read the command string and determine what it is before the remainder of the line can be parsed.

Also, the logical significance of the parameters depends upon the particular command string that is used. For example, while `update` and `range` both take two parameters, the first parameter to `update` is city name and the second is a positive integer but both parameters to `range` are positive integers. So, we may need to know the command before we know what types to use to read the parameters, much less how to interpret the parameters after we've read them.

Here is one good organization for processing this sort of command file. Each box represents a function, each arrow a function call. Some (but not all) of the parameters that need to be passed are indicated next to the call arrows.



Determining which command has been passed in requires comparing the string you've just read to the known command strings. Once that has been done, you know how many parameters are expected (and their types), and you know what function to call to handle that command.

The design makes it relatively simple to add support for new commands, and logically isolates the code to read and handle each command in a sensible way.

Sample files:

City data file:

```
La Luz              New Mexico          1625
Providence          Rhode Island        160728
Gadsden             Alabama             42523
Ten Sleep           Wyoming             311
Rifle               Colorado            4636
Knoxville           Tennessee           165121
Jackson             Wyoming             4472
Union               Oregon              1847
Clear Brook         Virginia            150
Red Lodge           Montana             1958
Metairie            Louisiana           149428
Delta               Utah                2998
Clarksville         Tennessee           75494
La Crosse           Wisconsin           51003
Kiowa               Oklahoma            718
Greensboro          North Carolina      183521
Baton Rouge         Louisiana           219531
Ed                  West Virginia       7
Zap                 North Dakota        287
Sturgis             South Dakota        5330
Buffalo             South Dakota        488
Indiana             Pennsylvania        15174
Cortez              Colorado            7284
Shelbyville         Indiana             15336
Little Rock         Arkansas            175795
Bowling Green       Virginia            727
Vernal              Utah                6644
Chamberlain         South Dakota        2347
Clearbrook          Virginia            400
Nags Head           North Carolina      1838
Pecos               Texas               12069
```

Commands file:

```
sort     name
city     La Luz
city     Pecos
city     Chamberlain
sort     population
city     Baton Rouge
city     Clarksville
state    Wyoming
update   Clarksville    75580
update   Zap            321
sort     name
range    100            300
city     Oyster Bay
state    Florida
exit
```

Log file:

```
Programmer:  Bill McQuain
CS 1044 Project 6 Fall 2009
------------------------------------------------------------------------
Sorting by name
------------------------------------------------------------------------
Looking for city named:   La Luz
 17:  La Luz                        New Mexico                1625
------------------------------------------------------------------------
Looking for city named:   Pecos
 21:  Pecos                         Texas                    12069
------------------------------------------------------------------------
Looking for city named:   Chamberlain
  3:  Chamberlain                   South Dakota              2347
------------------------------------------------------------------------
Sorting by population
------------------------------------------------------------------------
Looking for city named:   Baton Rouge
 30:  Baton Rouge                   Louisiana               219531
------------------------------------------------------------------------
Looking for city named:   Clarksville
 24:  Clarksville                   Tennessee                75494
------------------------------------------------------------------------
Looking for cities in:   Wyoming
  3:  Ten Sleep                     Wyoming                    311
 14:  Jackson                       Wyoming                   4472
------------------------------------------------------------------------
Updating population for:   Clarksville
 24:  Clarksville                   Tennessee                75580
------------------------------------------------------------------------
Updating population for:   Zap
  2:  Zap                           North Dakota               321
------------------------------------------------------------------------
Sorting by name
------------------------------------------------------------------------
Looking for populations between 100 and 300:
  5:  Clear Brook                   Virginia                   150
------------------------------------------------------------------------
Looking for city named:   Oyster Bay
Oyster Bay not found
------------------------------------------------------------------------
Looking for cities in:   Florida
No cities in Florida found
------------------------------------------------------------------------
Exit command found
------------------------------------------------------------------------
```