CCDB: Class Computer Database

Fundamental Concepts: data class, dynamic array of objects

The point of this assignment is to give you beginning programming experience with programmer-defined classes and dynamic memory. This project will require you to apply what you have learned about classes and dynamic memory in CS 1704. This assignment will modify and extend SCDB. All of the specified functionality of SCDB will be retained. In order to make the program more useful, two other action commands will be implemented. The data structure that holds the in-memory computer system database will also be changed; see the section on Dynamic Array Management below for details. CCDB will first load the initial computer system data file as before, creating an in-memory computer system database array, and then read and process actions from the `Actions.txt` script file. As with SCDB, when all the specified actions have been processed, CCDB will exit.

Interim Design:

You will produce an interim design for CCDB, and represent that design in a modular structure chart. The structure chart must indicate your design plans for CCDB. It is expected that your final design will differ from the interim design. Nevertheless, your interim design should be relatively complete. If the interim design is incomplete or if the differences between your interim design and your final code are excessive, you will be penalized. That means that you should take the production of the interim design seriously, but **not** that you should avoid changes that would improve your final implementation of CCDB. You must submit this interim design, to the Curator System, no later than midnight Wednesday, Sept. 26, (i.e. prior to Sept. 27). Submit a MS Word.doc file. Do **NOT** compress, (zip), the interim design submission! See the course web site for examples and restrictions on the interim design and file submission.

Simple Computer System Class & Objects

You are **required** to change your computer system struct type into a straightforward data class type. This will require the creation of constructor, reporter (get), mutator (set) and perhaps other member function(s). Under no circumstances is the class to contain any dynamic memory allocation. The computer system class will be the **only** allowed programmer-defined class in the program. Since the class will be a simple data class, containing no dynamic memory, no destructor will be required. In addition, no overloading of the assignment operator will be required. Member functions should be documented the same as non-member functions. See the course notes for a description of how the class member functions are to be depicted on the structure chart design.

Dynamic Array Management

You are **required** to use a dynamically-allocated array of computer system class objects. Use of any STL templates for the dynamic memory allocation or any other purpose is expressly forbidden in this assignment. The dynamic array of computer system objects must **not** be implemented as a class itself. As before, if the script actions file contains add commands that would overflow the array capability the commands will be ignored. The array of computer system class objects must be allocated with a size to store 125% of the number of computer system records in the `CompSys.txt` input file, truncated to an integer.

File Descriptions:

The initial computer system data file and the script actions file will have precisely the same syntax as per SCDB, aside from some new actions.

Each line of the actions file will contain one of the commands described in the SCDB specification, or one of the new commands described below. As before, commands are case-sensitive and take a fixed number of tab-delimited arguments. The command names will be valid, and each command will include the correct number and type of arguments.

`find <tab> <model>`

This command results in a search being performed to locate the computer system object with the corresponding model. If located, the command will display the `<index>` of the object in the array followed by the `<manufacturer>` and `<price>` member fields of the object. The command output must be in the following format made to the output file, `dbase.txt`:

```
Find:     <index>     <model>     <manufacturer>    <price>
```

If a corresponding model object cannot be located the output of the command must be the following, where `<model>` is replaced by the model name that was to be found:

```
Find:     <model>     *MISSING*
```

`update <tab> <model> <tab> <price>`

If the given model name is found in the model names array, reset the price value for that model to the value given in the command and print the updated model data. The command output must be in the following format made to the output file, `dbase.txt`:

```
Update:   <index>     <model>     <manufacturer>    <price>
```

If a corresponding model object cannot be located the output of the command must be the following, where `<model>` is replaced by the model name that was to be found:

```
Update:   <model>     *MISSING*
```

For both input files, 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.

Updated sample input files for CCDB will be posted on the course website soon. When they are available, an announcement will be posted on the course Web site.

Input File Descriptions and Samples:

Initial Computer system File

The format of this file is unchanged from SCDB. A sample `CompSys.txt` input file is shown below.

#4explorermicro.com systems

#Model	Manufacturer	Price	Processor	Speed	RAM (MB)	Drive (MB)	Video (MB)	Cache (K)	CD	Sound	Case
MT-359AD	4explorermicro.com	717	Pentium III	1000	64	20000	8	512K	50X	3D Soundpro	ATX
MT-359AF	4explorermicro.com	567	Pentium III	1000	128	20000	32	None	50X	3D Soundpro	ATX
MT-359AL	4explorermicro.com	777	Pentium III	1000	64	40000	32	None	50X	3D Soundpro	ATX
MT-359AT	4explorermicro.com	1469	Pentium III	1000	64	60000	32	None	50X	3D Soundpro	ATX
MT-359AN	4explorermicro.com	1699	Pentium III	1000	128	40000	32	None	50X	3D Soundpro	ATX
MT-359AV	4explorermicro.com	1899	Pentium III	1000	128	60000	32	None	50X	3D Soundpro	ATX

Database Actions File

There is no guaranteed limit on the number of actions. The changes to this file have been discussed previously, (see the File Descriptions section above). A small sample `Actions.txt` input file is shown below.

```
#4explorermicro.com systems: Actions.txt
#COM    ARGS
sort    model
#non-existent delete
del     MT-359AE
#delete first record
del     MT-359AD
#add record back
add     MT-359AD       4explorermicro.com     727     Pentium III    1000    64      20000   8       512K
        50X     3D Soundpro     ATX
#delete last record
del     MT-359AV
#add record back
add     MT-359AV       4explorermicro.com     1999    Pentium III    1000    128     60000   32      None
        50X     3D Soundpro     ATX
sort    price
#delete middle record
del     MT-359AL
#add record back
add     MT-359AL       4explorermicro.com     797     Pentium III    1000    64      40000   32      None
        50X     3D Soundpro     ATX
#attempt to add existing record
add     MT-359AL       4explorermicro.com     999     Pentium III    1000    64      40000   32      None
        50X     3D Soundpro     ATX
sort    model
dump
#non-existent update
update  MT-359AE       9999
# update next to last record
update  MT-359AT       1445
#non-existent find
find    MT-359AE
#find next to last record
find    MT-359AT
#Script End
```

Output description and sample:

As previously, your program will produce two output files: 1. a log file and 2. output report file. The log file named "`ccdblog.txt`", will record the processing of the actions file as before. For the above `Actions.txt` input file the corresponding `ccdblog.txt` would be:

```
Programmer:  Dwight Barnette
Class Computers Database Log
_____________________________________________________________________________________________
###     COMMAND         ARG             ACTIVITY        COMMENT
001     sort            model:          SUCCESS
002     del             MT-359AE:       FAILURE         Not Found
003     del             MT-359AD:       SUCCESS
004     add             MT-359AD:       SUCCESS
005     del             MT-359AV:       SUCCESS
006     add             MT-359AV:       SUCCESS
007     sort            price:          SUCCESS
008     del             MT-359AL:       SUCCESS
009     add             MT-359AL:       SUCCESS
010     add             MT-359AL:       FAILURE         Present
011     sort            model:          SUCCESS
012     dump                            SUCCESS
013     update          MT-359AE:       FAILURE         Not Found
014     update          MT-359AT:       SUCCESS
015     find            MT-359AE:       FAILURE         Not Found
016     find            MT-359AT:       SUCCESS
```

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

```
Programmer:  Dwight Barnette
Class Computers Database

_____________________________________________________________________________________________________________

IDX  Model       Manufacturer          Price  Processor    Speed  RAM (MB)  Drive (MB)  Video (MB)  Cache (K)  CD   Sound          Case
000  MT-359AD    4explorermicro.com      727  Pentium III   1000        64       20000           8  512K       50X  3D Soundpro    ATX
001  MT-359AF    4explorermicro.com      567  Pentium III   1000       128       20000          32  None       50X  3D Soundpro    ATX
002  MT-359AL    4explorermicro.com      797  Pentium III   1000        64       40000          32  None       50X  3D Soundpro    ATX
003  MT-359AN    4explorermicro.com     1699  Pentium III   1000       128       40000          32  None       50X  3D Soundpro    ATX
004  MT-359AT    4explorermicro.com     1469  Pentium III   1000        64       60000          32  None       50X  3D Soundpro    ATX
005  MT-359AV    4explorermicro.com     1999  Pentium III   1000       128       60000          32  None       50X  3D Soundpro    ATX
[007] END

_____________________________________________________________________________________________________________

Update:  MT-359AE  *MISSING*
Update:  004       MT-359AT        4explorermicro.com        1445
Find:    MT-359AE  *MISSING*
Find:    004       MT-359AT        4explorermicro.com        1445
```

The `dump` command will be modified slightly. Each computer record output will be numbered starting at zero. At the end of the computer system record table listing, the current size of the dynamic array will be output. Note that this is not the same as the number of objects stored in the array. Note well that the `Actions.txt` file is not required to end in a dump command and multiple dump commands may exist in the file. The first line of your output must include your name only. The second line must include the title "`Class Computers Database`" only. The first and last line of each dump must be a line of underscores. The second line of a dump command output will contain the area data echoed from the current computer system database array, aligned under the appropriate headers. The column field headings should be repeated for each display listing resulting from a dump. However, the first two lines (programmer and program title) are not to be repeated.

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 underscore 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.

Programming Standards:

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

Documentation:

- You must include the honor pledge in your program header comment, (see below).
- You must include a header comment that describes what your program does and specifying any constraints or assumptions of which a user should be aware, (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 suggesting the meaning/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 give the name of the function, explain in one sentence what the function does, describe any special algorithm used/implemented, give a list of the other functions that call this function, give a list of the other functions that this function calls, then describe the logical purpose of each parameter (if any), describe the return value (if any), give your name as the author of the function, and state reasonable pre- and post-conditions and invariants.
- Use the assert function to check for error conditions and verify function pre- and post-conditions whenever possible.

- 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 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 decimal numbers.
- Use an array of objects to store the computer system 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()`.
- 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 `class` declaration statement for your computer system class 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. Pointers should be passed by reference, `const` pointer and/or `const` target as appropriate.

Stepwise Refinement:

For medium and large sized programs it is critical that programmers practice incremental implementation. Here is one possible suggested order of implementation:

- Define the computer object data class. Implement and test each of the member functions.
- Declare a static data object array. Print it out to verify the constructor initialization.
- Determine the number of records in the `CompSys.txt` input file.
- Change the static data object array into a dynamic data object array of size 1.25 * number of input records.
- Read the initial model data into the data array. Print out the model data to verify the correct input.
- Implement reading of the actions 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 end of the file correctly.
- Add a function (or two) to handle the `update` command.
- Add a function (or two) to handle the `find` command.
- Modify the `dump` command.

If you get stuck on handling a command, or 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 may receive partial credit.

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. 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. `CCDB` will be subjected to runtime testing by the Curator automated grading system. Your code will also be evaluated by a GTA for adherence to these specification requirements, software engineering principles and good programming practices. Deductions incurred will be applied to your project Curator grade. You will be allowed to make up to five submissions of `CCDB` to the Curator.

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

http://ei.cs.vt.edu/~eags/Curator.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. There will be absolutely **no** exceptions to this policy! If two or more of your submissions are tied for highest, the earliest of those will be graded and also evaluated by the TAs who will assess a deduction for adherence to the specified programming standards. The deduction will be applied to your highest score from the Curator. 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 Curator Server.
```

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