CARS: Class Automobile Record System

Fundamental Concepts: data class, dynamic array of objects

The *object* 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. This assignment will modify and extend SARS. All of the specified functionality of SARS will be retained. In order to extend the program, two new action commands will be implemented. The data structure that holds the in-memory automobile database will also be changed; see the section on Dynamic Array Management below for details. CARS will first load the initial automobile data file as before, creating an in-memory automobile database array, and then read and process actions from the `Actions.txt` script file. As with SARS, when all the specified actions have been processed, CARS will exit.

Interim Design:

You will produce an interim design for CARS, and represent that design in a modular structure chart. The structure chart must indicate your design plans for CARS. 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 CARS. You must submit this interim design, to the Curator System, no later than midnight Monday, Feb. 4, (i.e. prior to 2/5). 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 Automobile Class & Objects

You are **required** to change your automobile 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 automobile class will be the **only** allowed programmer-defined class in the program, (with one possible exception – see output description section). Since the class will be a simple data class, containing no dynamic memory, no destructor or copy constructor 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 automobile 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 automobile 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 automobile class objects must be allocated with a size to store 125% of the number of automobile records in the `Cars.txt` input file, truncated to an integer.

File Descriptions:

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

Each line of the actions file will contain one of the commands described in the SARS 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> <VIN>`

This command results in a search being performed to locate the automobile object with the corresponding VIN. If located, the command will display the `<index>` of the object in the array followed by the `<dealer>` 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>     <dealer>    <price>
```

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

```
Find:     <VIN> *MISSING*
```

`update <tab> <VIN> <tab> <price>`

If the given `VIN` is found in the automobile array, reset the price value for that `VIN` 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>     <VIN>       <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:   <VIN> *MISSING*
```

`dealer <tab> <dealer name>`

Search the automobile database array for all records with the given `dealer name`. For all located records, count the number of entries and determine the average sale price. The command output must be in the following format made to the output file, `dbase.txt`:

```
Dealer:   <number of cars>  <dealer name>     <average price>
```

If no records for the dealer can be located the output of the command must be the following, where `<dealer>` is replaced by the `dealer name` that was to be found:

```
Dealer:   <dealer>    *MISSING*
```

Updated sample input files for CARS 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 Automobile File

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

```
#Automobile Data
#VIN                 Dealer       Manufacturer Model           Year   Price
1FH2D8C19I2129562    Shelob MM    Lincoln      Continental     1987   5750
1MMD4C629H1142630    Shelob MM    Mercury      Mystique        1986   3400
14B2D6C1921102113    Pooka GM     Buick        LeSabre         2002   22250
1CC4D4CC9Y0325751    Shelob MM    Chrysler     PT Cruiser      2000   24500
1S4WD4449V4268422    Pooka GM     Subaru       Outback         1997   16750
1BD2D8C89Y1285738    Pooka GM     Dodge        Intrepid        2000   19990
```

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.

```
#Class Automobile Record System: Actions.txt
#non-existent delete
del       1BD2D8C89Y1285739
#delete first record
del       1FH2D8C19I2129562
#add record back
add       1FH2D8C19I2129562   Shelob MM     Lincoln       Continental  1987   5755
#delete last record
del       1BD2D8C89Y1285738
#add record back
add       1BD2D8C89Y1285738   Pooka GM      Dodge         Intrepid     2000   19999
sort
#delete middle record
del       1MMD4C629H1142630
#add record back
add       1MMD4C629H1142630   Shelob MM     Mercury       Mystique     1986   3300
#attempt to add existing record
add       1MMD4C629H1142630   Shelob MM     Mercury       Mystique     1986   4300
sort
dump
#non-existent update
update    1BD2D8C89Y1285739   99999
# update next to last record
update    1MMD4C629H1142630   3800
#non-existent find
find      1BD2D8C89Y1285739
#find next to last record
find      1MMD4C629H1142630
#non-existent dealer
dealer    Hud's Son
#find next to last record
dealer    Shelob MM
#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 "`CARSlog.txt`", will record the processing of the actions file as before. For the above `Actions.txt` input file the corresponding `CARSlog.txt` would be:

```
Programmer:  Dwight Barnette
Class Automobile Records System Log
_____________________________________________________________________________
### COMMAND            ARG             ACTIVITY      COMMENT
001 del          1BD2D8C89Y1285739:  FAILURE       Not Found
002 del          1FH2D8C19I2129562:  SUCCESS
003 add          1FH2D8C19I2129562:  SUCCESS
004 del          1BD2D8C89Y1285738:  SUCCESS
005 add          1BD2D8C89Y1285738:  SUCCESS
006 sort                         :  SUCCESS
007 del          1MMD4C629H1142630:  SUCCESS
008 add          1MMD4C629H1142630:  SUCCESS
009 add          1MMD4C629H1142630:  FAILURE       Present
010 sort                         :  SUCCESS
011 dump                         :  SUCCESS
012 update       1BD2D8C89Y1285739:  FAILURE       Not Found
013 update       1MMD4C629H1142630:  SUCCESS
014 find         1BD2D8C89Y1285739:  FAILURE       Not Found
015 find         1MMD4C629H1142630:  SUCCESS
016 dealer       Hud's Son       :  FAILURE       Not Found
017 dealer       Shelob MM       :  SUCCESS
```

Optionally the log file stream may be encapsulated in a class. This is not a requirement and with the automobile calss constitutes the only allowed classes in the program.

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 Automobile Records System

_____________________________________________________________________________
IDX VIN               Dealer       Manufacturer Model           Year    Price
000 14B2D6C1921102113 Pooka GM     Buick        LeSabre         2002     22250.00
001 1BD2D8C89Y1285738 Pooka GM     Dodge        Intrepid        2000     19999.00
002 1CC4D4CC9Y0325751 Shelob MM    Chrysler     PT Cruiser      2000     24500.00
003 1FH2D8C19I2129562 Shelob MM    Lincoln      Continental     1987      5755.00
004 1MMD4C629H1142630 Shelob MM    Mercury      Mystique        1986      3400.00
005 1S4WD4449V4268422 Pooka GM     Subaru       Outback         1997     16750.00
[007] END
_____________________________________________________________________________
Update:            1BD2D8C89Y1285739   *MISSING*
Update:     004    1MMD4C629H1142630    3800
Find:              1BD2D8C89Y1285739   *MISSING*
Find:       004    Shelob MM            3800
Dealer:            Hud's Son           *MISSING*
Dealer:     003    Shelob MM           11351.67
```

The `dump` command will be modified slightly. Each automobile record output will be numbered starting at zero, (i.e., the index of the record in the array). At the end of the automobile 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 Automobile Records System`" 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 car data echoed from the current automobile 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()` to check for error conditions and verify function pre- and post-conditions whenever possible.
- 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 automobile 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 automobile 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 automobile 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 `Cars.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 car data into the array. Print out the car 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.
- Add a function (or two) to handle the `dealer` 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 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. CARS 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 CARS 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 Curator 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.