DCDB: Dynamic Computer Database

Fundamental Concepts: separate compilation, elementary operator overloading, dynamic resizable array class of objects

The focus of this programming assignment is to extend your previous work with programmer-defined classes and dynamic memory. This project requires the modification and extension of CCDC. All of the previous functionality of SCDB and CCDC will be retained, unless explicitly removed or modified in this specification. Separate compilation is required for this assignment. Each user-defined class must have its own source code and header files. Other compilation units should be created that reflect the modular decomposition design of the system. Related system sub-sections should be grouped into separate compilation units.

To make the program more memory resource efficient, the dynamically allocated array must be implemented in a class. It must have the ability to dynamically resize itself during execution as the number of computer records to be stored grows and shrinks. Additionally, you will add the ability to save the in-memory computer system database to a file on disk, and to load a computer system database that was previously saved by your program; see the section on the Dynamic Array Class below for details. The `sort` action command will no longer be used. A **selection** sort on `model` could be performed after reading the initial computer system data; all subsequent commands will maintain the ascending `model` ordering. (Alternatively ordered insertion can always be used to maintain the model ordering.

DCDB will normally be invoked from the command-line, and the names of the input files will be specified on the command-line, as:

```
DCDB <InitialComputerSystemDataFileName> <DatabaseActionsFileName>
```

DCDB will first load the initial computer system data file, creating an in-memory database structure, and then read and process actions from the database actions file. As with SCDB and CCDB, when all the specified actions have been processed, DCDB will exit. (For ease of testing command line arguments can be specified form within the MS VC++ IDE. From the Project menu select the Settings… option and in the dialog window select the Debug tab and enter the file names in the Program arguments: field.)

Interim Design:

You will produce an interim design for DCDB, and represent that design in a modular structure chart. The structure chart must indicate your design plans for DCDB. 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 DCDB. You must submit this interim design, to the Curator System, no later than midnight Monday, Oct. 15, (i.e. prior to Oct. 16). 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.

File Descriptions:

The initial computer system data file and the script actions file will have precisely the same syntax as per SCDB/CCDB, aside from some new actions. Note that use of hard-coded names for these files will annoy the person evaluating your program, and you **will** be charged points for that annoyance.

The format of the Computer system database file, created by the save command, is not specified. Part of your assignment is to design a sensible layout for this file. Note that this means that programs from two different students may certainly have incompatible database file formats, and so will not be able to load database files created by another program. The only restrictions imposed on your database file design are that you should not waste too much space and that the file must be an ASCII text file.

Each line of the actions file will contain one of the commands described in the SCDB/CCDB 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 of arguments.

`save <DatabaseFileName>`

This causes the creation of a Computer system database file on disk, in the current directory. The default extension for the file, "`.cdb`" should be automatically appended to the name. As stated above, the format of this file is up to you, subject to light restrictions. Saving does not clear the current in-memory database.

`load <DatabaseFileName>`

This causes the reading of the named, (previously saved) Computer system database file, and the creation of a new in-memory database holding the information from that file. The default extension for the file, "`.cdb`" should be automatically appended to the name for opening. Any previous in-memory database should be properly deallocated, (but not automatically saved to disk), before the file is read. If the named file does not exist, the following error message must be written to the dump file, "`dbase.txt`" out:

`***Fatal Error***: <`*`DatabaseFileName`*`> does not exist!`

The database filename specified in the load command must be substituted for <*DatabaseFileName*> in the above message. At this point, your program should gracefully shutdown. If the named file does exist, any previous in-memory database should be properly de-allocated, (but not automatically saved to disk), before the file is read.

Note that if you do not properly implement the `save` command, there will be absolutely no way to test your implementation of the `load` command. 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 DCDB will be posted on the course website soon. When they are available, an announcement will be posted on the course Web site.

Dynamic Array Class of Computer System Objects

In this program you are **required** to convert your array database of computer system objects into a class. Be aware that a correct implementation will result in no file input or output, (I/O), being performed by any of the array class member functions. (Note: this separation of the I/O from a class also applies to the Computer system class.) The array class will contain constructors, (copy constructor), reporters (get, search, etc.), mutators (set, remove, grow, shrink, sort, etc.), an assignment overload operator and destructor member function(s). The array class should be viewed and implemented as a container class that is completely unaware of the type of object being stored in it. To this end the Computer system class should overload equality and inequality operators as needed by the array class code.

Dynamic Array Management

You will initially allocate an array capable of holding exactly 5 computer system record objects. If the list outgrows the current array size, you will dynamically enlarge the array to hold exactly 5 additional computer system record objects. If the number of unused array locations grows to 10, you will dynamically shrink the array to hold 5 fewer records (allowing some slack space for future growth).

This crudely mimics the behavior of a C++ vector object. You are specifically forbidden to use any C++ vector objects or any other any STL templates in this program, or to use a linked list of any type in place of the specified array. Violating that restriction would remove one of the major points of this assignment and will certainly result in a major deduction.

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
#save initial database
save    dcdb0
#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
#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
#save modified database
save    dcdb1
#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
#delete first record
del     MT-359AD
#delete last record
del     MT-359AV
#delete middle record
del     MT-359AL
#load initial database
load    dcdb0
dump
#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 "`dcdblog.txt`", will record the processing of the actions file as before. For the above `Actions.txt` input file the corresponding `dcdblog.txt` would be:

```
Programmer:  Dwight Barnette
Dynamic Computers Database Log
_____________________________________________________________________________
###     COMMAND         ARG             ACTIVITY        COMMENT
001     save            dcdb0           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     del             MT-359AL:       SUCCESS
008     add             MT-359AL:       SUCCESS
009     add             MT-359AL:       FAILURE         Present
010     save            dcdb1           SUCCESS
011     update          MT-359AE:       FAILURE         Not Found
012     update          MT-359AT:       SUCCESS
013     find            MT-359AE:       FAILURE         Not Found
014     find            MT-359AT:       SUCCESS
015     del             MT-359AD:       SUCCESS
016     del             MT-359AV:       SUCCESS
017     del             MT-359AL:       SUCCESS
018     load            dcdb0           SUCCESS
019     dump                            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
Dynamic Computers Database
Update:   MT-359AE   *MISSING*
Update:   004        MT-359AT      4explorermicro.com          1445
Find:     MT-359AE   *MISSING*
Find:     004        MT-359AT      4explorermicro.com          1445
_____________________________________________________________________________
IDX  Model      Manufacturer        Price  Processor  Speed  RAM (MB)  Drive (MB)  Video (MB)  Cache (K)  CD   Sound        Case
000  MT-359AD   4explorermicro.com    717  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    777  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   1899  Pentium III  1000       128       60000          32  None       50X  3D Soundpro  ATX
[010] END
_____________________________________________________________________________
```

The first line of your output must include your name only. The second line must include the title "`Dynamic Computers Database`" only.

The output of dump commands will contain the area data echoed from the current computer system database array, aligned under the appropriate headers. The first and last line of each dump must be a line of underscores. The column field headings should be repeated for each display listing resulting from a dump. However, other lines (programmer, and program title) are not to be repeated. As previously, the `dump` command will output computer records numbered starting at one. 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 records stored in the array. Note well that the `Actions.txt` file is not required to end in a dump or save command and multiple dump commands may exist in the file.

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 **set of separately compiled source files**, with user-defined header files.
- 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.

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. 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.

Deliverables:

Your final project submission must include the following (and absolutely **no** other files):

- all source code (`*.cpp` and `*.h` files) comprising your project
- The MS Visual C++ project files (`.dsp` and `.dsw`). Do **NOT** submit the debug/release project subdirectories or an executable.exe file.
- a revised modular structure chart reflecting the final design of your project, at the time of submission; this must in the same MS Word format as the interim chart.
- one set of input files, named `AreaData.txt` and `Actions.txt,` and the corresponding final dump `dbase.txt`, and the corresponding saved database file(s)`.cdb.`
- a brief ASCII text readme file, named `readme.txt`, with any special execution instructions

Submissions will be archived, but not scored, by the Curator System. You will submit your project as an archive file in a zipped archive containing the items listed above. (The shareware program WinZip is very easy to use and is available from the Computing Services website: `http://www.ucs.vt.edu/`)

Note that omitting files from your archive is a classic error. Once you've created your project archive, copy the file to a new location, unzip it, attempt a build and then test the resulting executable. Submitting an incomplete copy of your project may delay its evaluation and **will** result in a substantial loss of points. In particular, if you omit a source file necessary to compile your program, you **will** be allowed to supply that file; however, we will then apply a **late penalty** corresponding to the date that you have provided a complete copy for evaluation. There will be **no exceptions** to this penalty. **Also note** that including unnecessary files is also a classic error. Visual C++ users: do **not** zip up the **debug** subdirectory!

Submitting your project archive:

You will submit your project archive to the Curator System, as described above. `DCDB` will be subjected to runtime testing by the TAs, who will also score your implementation for adherence to the specified programming standards. Demonstration time slot signup forms will be made available in the CS lab. An announcement will be posted when they are available. Students will only be allowed to schedule and perform one demonstration. TA/student demonstration assignments will be posted on the course Web site. You will be allowed to make up to five submissions of `DCDB` to the Curator. Note well: **your last submission will be graded.** There are no exceptions to this policy! Changes made to code during a demonstration will be heavily penalized.

Pledge:

Each of your project submissions to the Curator system 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.