DARP: Dynamic Advising Records Program

This assignment will modify and extend SARP. All of the specified functionality of SARP will be retained. In order to make the program more useful, you will add the ability to save the in-memory student database to a file on disk, and to load a student database that was previously saved by your program. The data structure that holds the in-memory database will also be changed; see the section on Dynamic List Management below for details.

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

```
DARP <InitialStudentFileName> <DatabaseActionsFileName>
```

DARP will first load the initial student data file, creating an in-memory database structure, and then read and process actions from the database actions file. As with SARP, when all the specified actions have been processed, DARP will exit.

File descriptions:

The initial student data file and the database actions file will have precisely the same syntax as for SARP, 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.

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

`save <DatabaseFileName>`

> This causes the creation of a student database file on disk, in the current directory. 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 student database file, and the creation of a new in-memory database holding the information from that file. If the named file does not exist, an appropriate useful error message should be displayed to standard output. Otherwise, any previous in-memory database should be properly deallocated (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.

The format of the saved student database file is not specified. Part of this assignment is to design and use a sensible layout for this file. Note that this means that programs from two different students may use incompatible database file formats, and so will not be able to load databases 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.

Recall that while we suggest that you use spaces (not tabs) to align your output (dump) file, we use the tab character to format your input file because it makes it somewhat easier for you to parse the input. Note it may be advantageous if you also use the tab character when you design the format for your saved database file.

Updated sample input files for DARP will be posted on the course website no later than Monday, February 14.

Dynamic list management in DARP:

You are **required** to use a dynamically-allocated array of structures to store the courses information. You will initially allocate an array capable of holding exactly 10 course records. If, at any time, the list outgrows the current array size, you will dynamically enlarge the array to hold exactly 10 additional course records. If the number of unoccupied array locations grows to 20, you will dynamically shrink the array to hold 10 fewer records (allowing some slack space for future growth).

For example, suppose the student data file contains 17 courses. You'd add the first 10 course records to the array, filling it. When the 11th course record was to be added, your program would detect that the array was full, and expand it to dimension 20. After adding the remaining course records from the student data file, you'd have 17 occupied array locations and 3 free ones.

Now suppose that the actions file started with 7 add commands. You'd add the first three new courses, again filling the array. When adding the 21st course record, you'd expand the array to dimension 30, and subsequently store the remaining 4 new course records. Now, suppose the actions file specified 17 drops. You'd process 13 drops, at which point the number of occupied locations would be 11. When you processed the next drop, the number of unoccupied array locations would be 20, so at that point you'd shrink the array to dimension 20.

This crudely mimics the behavior of a C++ vector object. You are specifically forbidden to use any C++ vector objects in this program, or to use a linked list of any sort 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.

Programming standards:

You'll be expected to observe good programming/documentation standards. Some specifics:

Documentation:

- You must include the honor pledge in your program header comment.
- 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 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 collection of source files, appropriately partitioning your functions by logical purpose. The number of resulting source files may vary, but a good decomposition for this project would result in at least five `cpp` files. The source files should be named descriptively. Generally, each `cpp` file should have an accompanying header file.
- Use named constants instead of variables where appropriate.
- Use `double` variables for all decimal numbers, such as the QCA.
- Use a dynamically managed array of structure variables, as described above, 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 implement and use a reasonable number of functions, besides `main()`.
- `main()` must be the only function defined in the main source file.
- You may use globally-scoped function prototypes and you may use globally-scoped constants. You may also make the `typedef` statement for your structured variable type globally-scoped (in fact you must do this).
- You may not use globally-scoped or 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.

Interim design:

You will produce an interim design for DARP, and represent that design in a modular structure chart. The structure chart must indicate your current design plans for DARP. We expect that your final design will differ from the interim design. Nevertheless, your interim design should be relatively complete. If the differences between your interim design and your final design are excessive, you will be penalized. That means that you should take the production of the interim design seriously, **not** that you should avoid changes that would improve your final implementation of DARP. You must submit this interim design, to the Curator System, no later than midnight Monday, February 14.

Testing:

At minimum, you should test your program on **all** the posted input/output examples to be 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.

Deliverables:

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

- all source code (`*.cpp` and `*.h` files) comprising your project
- a revised modular structure chart reflecting the final design of your project, at the time of submission; this must either be in a format that can be viewed in MS Word or be a PDF file.
- one set of input files, named `students.data` and `actions.data`, and the corresponding final dump `dbase.list`, and the corresponding saved database file, named `darp.db`
- a brief ASCII text readme file, named `readme.txt`, with any special execution instructions
- either the MS Visual C++ `.dsp` and `.dsw` files, or a UNIX makefile, as appropriate

Submissions will be archived, but not scored, by the Curator System. You will submit your project as an archive file in one of two formats. For Windows users, submit 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/`) For UNIX users, submit a gzipped tar file containing the items listed above.

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.

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 here. DARP will be subjected to runtime testing by the TAs, who will also score your implementation 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 DARP to the EAGS.

You may use the JSP Server client, as you did for Homework 1 and (possibly) SARP, or you may use either the application or applet version of the Java client. In any case, begin at:

`http://spasm.cs.vt.edu:8080/curator/`

We recommend using the Java application version of the client. If you use either of the Java clients, instructions for submitting your project archive are available in the *Student Guide* at the Curator Project Homepage:

`http://ei.cs.vt.edu/~eags/Curator.html`

Read the instructions carefully.

Note well: your last submission will be graded.

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 of the `cpp` file containing `main()`:

```
//    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.