DIMP: Dynamic Inventory Maintenance Program

This assignment will modify and extend SIMP. All of the specified functionality of SIMP will be retained. In order to make the program more useful, you will add the ability to save the inventory database to file, and to load a saved inventory database file from disk. When DIMP is invoked from the command line as:

```
DIMP <InitialInventoryFileName> <ActionsFileName>
```

it will behave precisely as SIMP, aside from a few new commands that may appear in the actions file. When invoked as:

```
DIMP -L <DataBaseFileName> <ActionsFileName>
```

DIMP will first read the specified inventory database file and create the same sort of simple in-memory database structure as SIMP, and then process a second input file specifying actions to perform on the reconstructed inventory data.

Any invocation of DIMP with fewer than two command line parameters should result in a useful error message. Invocations with more than three command line parameters may be handled at your discretion.

As with SIMP, when all the specified actions have been processed, DIMP will exit.

Input file descriptions and samples:

The initial inventory file and actions file have the same syntax as for SIMP (aside from 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 inventory database file 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 will almost 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 SIMP specification, or one of the commands described below. As before, commands are case-sensitive and take a fixed number of arguments. The 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 an inventory 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 list.

```
load <DataBaseFileName>
```

> This causes the reading of the named inventory database file and the creation of a new database list in memory. If the named file does not exist, an appropriate useful error message should be displayed. Otherwise, any previous inventory data stored in memory should be properly deallocated 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.

Output description and sample:

Output data resulting from a `dump` command must be written to a file named `dbase.list` — use of any other output file name **will** annoy the person evaluating your program and you will be charged points for that annoyance. The format of `dump` output should be the same as for SIMP, except that the title should say "Dynamic" instead of "Static".

Dynamic List Management:

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

Note this crudely mimics the behavior of a C++ vector object. You are specifically forbidden to use any C++ vector objects in this program. 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. All the discussions in class about formatting, structure, and commenting your code should be followed. 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 specifies any constraints or assumptions a user should be aware of (such as preset file names, value ranges, etc.).
- Each source file (`.h` and `.cpp`) must start with a comment block identifying the programmer.
- 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 identifier.
- Precede every major block of your code with a comment explaining its purpose.
- Precede every function definition 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 source files should be named descriptively. Generally, each `cpp` file should have an accompanying header file. For reference, my solution involves 10 `cpp` files.
- Use named constants instead of variables where appropriate.
- Use `double` variables for all dollar amounts.
- 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()`. For reference, my solution uses about thirty-one.
- `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 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.

Testing:

At minimum, you should test your program on **all** the posted input/output examples given along with this specification. The same program that generated those input/output examples will generate the input/output files used to test your program. 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:

You must submit a hardcopy interim design document, in the form of a modular structure chart, no later than the end of class on Friday September 17. This structure chart must indicate your current design plans for DIMP. It is expected that your final design will differ from the interim design. Nevertheless, your interim design should be relatively complete. The structure chart **must** be turned in at class on the specified day. Structure charts drawn by hand will not be accepted. The drawing tools in MS Word are entirely adequate for producing structure charts.

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 `inventory.data` and `actions.data`, and the corresponding final dump `dbase.list`, and the corresponding saved database file, named `dimp.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 EAGS. 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 program WinZip is very easy to use and freely available from the University 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 will delay evaluation and result in a loss of points.

Submitting your project archive:

You will submit your source code electronically, as described here. `DIMP` will be subjected to runtime testing by the GTAs, 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 `DIMP` to the EAGS.

Instructions for submitting your project archive are available in the *Student Guide* at the EAGS Homepage:

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

Read the instructions carefully.

Note well: your last submission will be graded.

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 in the main source file:

```
//    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 Automated Grader.
```

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