LIMP: Linked Inventory Maintenance Program

This assignment will modify and extend DIMP. All of the specified functionality of DIMP will be retained. There are two major extensions.

First, the underlying data structure will be completely replaced. LIMP will use a linked list of dynamically allocated nodes to store inventory item records. The list may be either singly or doubly linked, whichever you decide is most appropriate for your implementation. In order to receive full credit, the list must be fully encapsulated using a C++ class, and the list nodes must themselves be implemented using a node class. In addition, the inventory record stored in each list node must be encapsulated; but it is acceptable for the inventory records to be implemented as struct variables (as you did for SIMP and DIMP).

Second, in order to make the program even more useful, you will add the ability to update the inventory database to reflect selling and buying units of items, and will now store how many units of each item are in stock. This will require changes to the input and output specifications and the addition of two new actions:

```
sell  <item number>     <units>
```

and

```
buy   <item number>     <units>
```

LIMP will process command-line arguments in exactly the same way as DIMP.

Input file descriptions:

The initial inventory file and actions file have almost the same syntax as for DIMP (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 only difference is that the number of units of each item initially in stock is specified in the initial inventory file and in the `add` commands in the actions file. Sample input files will be posted on the website shortly.

You will have to adjust the format of your saved inventory database file slightly to account for the new data being stored.

Each line of the actions file will contain one of the commands described in the SIMP and DIMP specifications, 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.

```
sell  <item number>     <units>
```

> A `sell` command causes the specified number of units of the specified item to be removed from inventory. If the current inventory does not contain enough units to satisfy the request, then no units are removed from inventory and the sale is not made. If the specified item number is not found in the inventory database, then no action is taken.

```
buy   <item number>     <units>
```

> A `buy` command causes the specified number of units of the specified item to be added to inventory. If the specified item number is not found in the inventory database, then no action is taken.

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 you must now display the number of units currently in inventory. That should be displayed between the item description and the columns for price and cost. Sample output files will also be posted on the course website shortly.

Linked List:

You are **required** to use a linked list to store the inventory information. Your implementation of this list will be examined. In order to receive full credit, you must implement the nodes using a well-designed node class and the linked list itself must be encapsulated using a well-designed list class. In addition, the inventory records must be encapsulated so that the data they store is logically separated from the node pointers.

Note this mimics the behavior of a C++ vector object. You are specifically forbidden to use any C++ vector objects in this program, or any other sort of predefined dynamic list type. You may base your implementation on the linked list implementations shown in the textbook, or those shown in the notes and lectures. You may not use linked list code from any other source. 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 requirements for documentation and coding given in the DIMP specification are still in effect. In addition:

Documentation:

- You must describe the purpose of each of your classes in a header comment that precedes the class declaration.
- You must document each data member and function member of your classes, both in the header file containing the class declaration and in the corresponding source file containing the implementation. The header file does not have to contain full documentation for each function, but the purpose of each function should be described there.

Coding:

- You must separate the interface of each of your classes from its implementation by placing each class declaration (interface) in its own header file and the implementation of that class in a corresponding source file. The name of the class should be used as the name of the header and source files.
- You must protect access to your data by making **all** data members of your classes private.
- When a node is removed from your linked list, you must dispose of it properly by using `delete`.
- When you discard your inventory database list to load an existing one from a file, you must `delete` every node in the old list so that your program does not waste memory.
- In fact, you should avoid memory leaks altogether.

Deliverables:

You must submit an interim design document, to the EAGS, no later than midnight Monday October 18. The required format of this design document will be specified shortly. This design document 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.

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 design document 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 `limp.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 is 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.

Evaluation:

`LIMP` will be subjected to runtime testing by the GTAs, who will also score your implementation for adherence to the specified programming standards. Your score will depend on the quality of your design documentation, the amount of change from your interim to your final design, the quality of your coding and internal documentation, and the correctness and completeness of the output your program produces during runtime testing. The relative weights of these factors may be announced later.

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.

Submitting your project:

You will be allowed to make up to five submissions of `LIMP` 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.