COMSC-165 Lab 14
Queues as Linked Lists

Modify the program you wrote in lab 12, and improved in lab 13, which tracks your college course history. The modifications to **History.cpp** will include *re*sorting of the course histories, so that they are printed out in different orders. You will exercise a common technique for resorting a linked list. You will also add persistence to the program, so that your course history can be saved on disk.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab14** folder provided.

LAB 14: Working With Resorting `[ History.cpp ]`
Create a console application named **History**. Do NOT submit until you complete the last step.

STEP 1 of 3: Using A Queue
Change **History.cpp** from the previous lab so that it inserts new nodes using the *queue* method. Include any corrections that you may need, so that there are no memory leaks, fatal logic errors, or user-interface problems. Make sure that the program prints the full list of courses *after* the user chooses to enter no more courses.

Here are some *program improvements* to make, if you have not already done so:

1. Include a code block or function to *add a node* to the *end of the list*. You may either use an end pointer, or not, as you choose.

2. Include a void function to *print* the full list, passing the start pointer in the parameter list.

3. Include a code block or function to *deallocate* the list. It is not necessary to set the start (and end) pointer to zero, but you may do so, as you choose.

Compile and run. Verify that the program works.

STEP 2 of 3: Add Persistence
Change **History.cpp** so that it saves its records to a disk file. When the program starts, and the start pointer is initialized to zero, ask whether the user wants to recall the course history from the disk. If so, read the disk file. When the user is finished entering records and before they are printed to the screen, save the records to the disk.

Here is the function that you can use to read the file. You will have to modify the code to match your **`struct`** name. You can rename the database file name, but be sure it is stored in the working folder.

```
  void restoreList(course*& start)
  {
    fstream fin;
    fin.open("courses.dat", ios::binary|ios::in);
    if (!fin)
      return;

    // read the number of objects from the disk file
    int nRecs;
    fin.read((char*)&nRecs, sizeof(int));

    // read the objects from the disk file
    for (int i = 0; i < nRecs; i++)
    {
      course* t = new course;
      fin.read((char*)t, sizeof(course));
      ... // insert t at the end of the list -- you figure out this part!
    }
    fin.close();
  }
```

Study the above code until you understand the process. Supply the needed prototype and includes.

Write the corresponding function (`saveList`) to write the nodes to the disk file, and call these functions in the correct places in `main()`. Note that this function will have to *count the number of nodes* and write that to the file *before* it writes the nodes to the file.

Compile and run. Verify that the program works. If you have problems, you may need to delete the **.dat** file manually each time before running your program, while you debug.

STEP 3 of 3: Resorting
Modify **History.cpp** so that it sorts the records and reprints the sorted list. Include *four* different sorting orders. *After* the data is entered and saved to disk, and *before* the node memory is deallocated, do this:

1. Print the heading "By unit...", sort the list by units (low to high), reprint the list, and pause for the user to press **Enter** to continue.

2. Print the heading "By grade...", sort the list by case *independent* grade (starting with 'A'), reprint the list, and pause for the user to press **Enter** to continue.

3. Print the heading "By description...", sort the list alphabetically by case *independent* description, reprint the list, and pause for the user to press **Enter** to continue.

4. Print the heading "By term...", sort the list by term, using the compare function from lab 13, reprint the list, and pause for the user to press **Enter** to continue.

5. Write four code blocks or functions that you can call to sort the lists. If you choose to use functions, make your calls like this:

```
  sortByUnit(start);
  ...
  sortByGrade(start);
  ...
  sortByDescription(start);
```

```
...
sortByTerm(start);
```

Note that each sort starts with the list as sorted by the call above it. Inside each of the functions, declare a new start pointer, set to zero. Then traverse the list pointed to by the old start pointer, and perform an insertion sort into the list pointed to by the new start pointer. As each node is added to the new list, delete it from the old. The modified start pointer gets returned through the parameter list, since it's passed by reference. You may use either nested-for-loop sorting or qsort or insertion sort. For example, here's an implementation using an "insertion sort" function:

```
void sortByUnit(course*& start)
{
  course* newStart = 0; // a new empty list
  while (start)
  {
    course* t = start; // get node from old list
    start = start->next; // remove node from old list

    // find insertion point in new list
    ...you figure out this part!

    // insert "t" between "prev" and "p"
    ...you figure out this part!
  }
  start = newStart;
}
```

Study the comments in the above code, which explain the process. Write your code block or function for *one* of the four sorts. Once you verify that it works, copy/paste/markup your code as needed, for each of the other three sorts.

When you sort by term, use the compare function provided in lab 13, which uses description as its only tie breaker. For the other 3, you will need to write your own compare functions or code blocks, and *decide on your own tie-breakers*. Each function may none, or as many as three tie breakers -- you choose how many and what order.

Compile and run your code. Enter at least 5 courses to test it. Post the **.cpp** file to the COMSC server, *without* your **.dat** file.

*Post **History.cpp** to the COMSC server.*

*Note: **History.cpp** will be modified and improved further in the term project.*

NOTE: The data file location is the "working folder". Do not use any other file location specification.

GRADING POLICY

Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.
9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.

8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.