COMSC-165 Lab 13
Ordered Sets as Linked Lists

Modify the program you wrote in lab 12, which tracks your college course history. The modifications to **History.cpp** will include sorting of the course histories, so that they are printed out in order. You will add new objects to the linked list by sorting them into the correct position in the list, instead of using a simple stack as in the lab 12 version.

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 **lab13** folder provided.

LAB 13: Working With Sorting [`History.cpp`]
Create a console application named **History**. Do NOT submit until you complete the last step.

STEP 1 of 2: History.cpp
This lab does not contain step-by-step instructions for rewriting the **History.cpp** program from the previous lab -- instead it presents a specification which you are to follow. Read the entire specification before doing any coding or planning. Once you fully understand the specification, begin by planning the different parts of the program that you know you will need. Start coding *after* you have the program as planned out as you can get it.

SPECIFICATION

1. Modify **History.cpp** (ref: lab 12) as described below. The *only difference* between the new program and the version from lab 12 is that the new code inserts new linked list nodes in their proper order in the linked list, while the lab 12 version inserted new nodes at the front of the list (i.e., using the *stack* model). The new program requires a way to compare two nodes, so that the order can be established. A comparison function is provided for you in exercise 2 below, but for exercise 1, just use a simple **`if`** statement instead of a function.

2. As new objects are created and initialized, add them by inserting them into their proper place in the linked list. This will replace the stack insertion code from lab 12. Use the code presented in the lecture for building a sorted linked list. You will need a basis for comparing two courses, to determine sort order. The next step explains how to do this.

3. Compare courses based on the *units* data element. This will be replaced in exercise 2 below, but this is an easy way to compare because it does not require a function.

*Hint: Precede the insertion code with a traversing for-loop, to identify the insertion location with pointers **`prev`** and **`p`** -- insert the new object (pointed to by **`c`**) between the objects pointed to by **`prev`** and **`p`**.*

Compile and run your code. Enter at least 5 courses to test it.

STEP 2 of 2: History.cpp Final
Modify **History.cpp** from exercise 1, so that it sorts based on the *term* data element. Replace the units comparison and use the comparison function given below to compare courses and determine which comes first and which comes second.

As required in lab 12, express *term* as a 6-character string, with a 2-character semester code (FA=fall, SU=summer, and SP=spring), followed by the year (e.g, 1999, 2000, etc.). Sort chronologically by *term*. In case of a tie when sorting the list by *term*, sort by *Course Designation*. Comparing *terms* is not as easy as comparing strings with **`strcmp()`**, because they are not alphabetical. SP comes first, then SU, then FA. Here is a function to compare two objects (assuming that the `struct` is named **`course`**) -- it returns -1 if the first object in the parameter list precedes the second, 1 if the second precedes the first, and 0 if they tie:

```
int courseCmp(const course* a, const course* b)
{
  // validate the length of the strings
  if ((strlen(a->term) != 6) || (strlen(b->term) != 6)) // ...in cstring
    return myStricmp(a->desig, b->desig);

  // handle ties here
  if (myStricmp(a->term, b->term) == 0)
    return myStricmp(a->desig, b->desig);

  // compare the years
  int yearA = atoi(a->term + 2); // atoi is found...
  int yearB = atoi(b->term + 2); // ...in cstdlib
  if (yearA < yearB)
    return -1; // termA comes first
  if (yearA > yearB)
    return 1; // termB comes first

  // compare semesters in case of same year
  if (myStrnicmp(a->term, "SP", 2) == 0)
    return -1; // termA comes first
  if (myStrnicmp(a->term, "SU", 2) == 0)
    return myStrnicmp(b->term, "SP", 2) ? -1 : 1;
  return 1;
}
```

You will have to modify the above code to match the name of your **`struct`** and its data elements. See the note at the end of this writeup for the myStricmp and myStrnicmp, which do what the Microsoft stricmp and strnicmp functions do. Compile and run your code. Enter at least 5 courses to test it.

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

*Note: **History.cpp** will be modified and improved further in lab 14 the term project.*

LOOK AHEAD EXERCISE: Persistence
Do this, if you wish to get a head-start on one of the next lab assignment, before it becomes available online. Modify **History.cpp** so that it saves its objects to a disk file named **History.txt** or **History.dat**

(depending on whether you choose to use a text or binary file). Include a *header record* in the file, containing the number of records that are stored. The header record can simply be an **`int`**. When the program starts, attempt to open the data file. If successful, read the contents of the file and populate the linked list before prompting the user to enter more data. When your program ends, rewrite the data file.

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

Here are the `myStricmp` and `myStrnicmp` functions, use these:

```
int myStricmp(const char* x, const char* y)
{
  int f, l;
  do
  {
    if (((f = (unsigned char)(*(x++))) >= 'A') && (f <= 'Z')) f -= 'A' - 'a';
    if (((l = (unsigned char)(*(y++))) >= 'A') && (l <= 'Z')) l -= 'A' - 'a';
  } while (f && (f == l));
  return(f - l);
}

int myStrnicmp(const char* x, const char* y, int count)
{
  int f, l;
  do
  {
    if (((f = (unsigned char)(*(x++))) >= 'A') && (f <= 'Z')) f -= 'A' - 'a';
    if (((l = (unsigned char)(*(y++))) >= 'A') && (l <= 'Z')) l -= 'A' - 'a';
  } while (--count && f && (f == l));
  return (f - l);
}
```

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.