COMSC-165 Lab 12
Stacks as Linked Lists

Write a database program to track your college course history. The program, called **History.cpp**, will use a newly designed `struct` to store relevant information about courses that you have taken, are taking now, or plan to take. It will prompt the user (you) to type this information, and will store it in objects of the `struct` that are created with the `new` keyword. It will use a stack style (LIFO) linked list to manage these objects. After each course is entered via the keyboard, the full history should be printed, and you should be prompted to either add another course or to quit.

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 **lab12** folder provided.

The full credit version of the program does *not* include *persistence* (i.e., the ability to save the input and recall it the next time the program runs). The look ahead exercise version adds this feature. There is also no attempt to sort the entries or to maintain them in any order other than the order in which they are entered -- this is the subject of the next lab session.

LAB 12: A Database Program [`History.cpp`]

Create a console application named **History**. This lab does not contain step-by-step instructions for writing the **History.cpp** program -- 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. Create a `struct` named **course**, with these four elements:
   (1) course designation (for example, "COMSC-165") -- use any course designation format that you wish
   (2) term (for example: "FA2010", "SP2009", "SU2011") -- use these 6-character designations as shown!
   (3) units (for example: 4)
   (4) grade (for example: 'A',..., or 'X' if not completed yet)

NOTE: Make sure that you can accommodate up to 10 characters in the ***course designation****, such as "COMSC-155h" (without spaces). You may assume that no more than 6 characters will ever be entered by the user for* ***term*** *(without spaces). You may want to include some instructions or examples for the user to follow.*

2. Prepare your `struct` to be used in a *linked list*, by adding a `next` pointer.

3. Create a start pointer for the linked list, and initialize it to `0` so that the list is initially empty.

4. Write a *function* to print the list of courses to `cout`. Use `setw()` so that the output looks well-presented,

organized, and easy-to-read. Use `cout.setf(ios::left, ios::adjustfield);` and `cout.setf(ios::right, ios::adjustfield);` for left- and right-justification. Or use the `left` and `right` manipulators -- your choice. Print the courses in tabular fashion, with column headings, so that each course appears on a separate line, ***for example***:

```
COURSE      TERM    UNITS  GRADE
----------  ------  -----  -----
comsc-110   fa2011    4      a
comsc-165   sp2012    4      a
comsc-266   su2012    4      a
comsc-155h  fa2012    4      a
comsc-260   sp2013    4      a
bus-90      sp2013    4      a
```

Hint: The function should be a ***`void`*** *function, and should take the start pointer as the only parameter in the parameter list.*

5. Write a loop to print the current list and to prompt the user to add another course. Allow the user to quit, in which case you should print the history one more time, and exit the program.

6. If the user elects to add one more course, create a new object with the **`new`** keyword. Otherwise, break from the loop.

7. Read the input data from the keyboard for the new object, using the **`cin.getline`** function or **`cin >>`**. Read each input item with its own separate user prompt (for example: Enter the term (for example: SU2011)...) and its own **`cin`** statement. Be sure to issue **`cin.ignore(1000, 10);`** *before* **`cin.getline`**, *only* if the previously executed **`cin`** statement was a **`cin >>`** statement -- this processes the ENTER key symbol in the input buffer, which **`cin >>`** does not process.

As an alternative, per your choice, your **`cin >>`** statements can follow other **`cin >>`** statements, without **`.ignore`**'s, so that all input for one course can be done on a single line, space-separated. But be sure to include a prompt that explains to the user what to do.

8. Add the newly created object to the *front* of the linked list (using stack insertion).

9. Before the program terminates, traverse the linked list and deallocate the memory used for each of the objects (because they were instantiated with the **`new`** keyword).

10. Do *not* use global variables.

Here is the algorithm:

```
// declare an empty simple linked list
// start a loop
  // print the list
  // prompt user to ADD a node or QUIT
  // if QUIT, break out of loop
  // STEP 1: create new node
  // STEP 2: fill node with data from user (cin)
  // STEP 3+4: insert at start of list (stack method)
// end of loop
```

```
  // deallocate list memory
```

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

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.