COMSC-165 Lab 11
Linked List Concepts

In this lab session you will apply what you've learned about pointers, by creating *linked lists* of objects. You will work with the **`tod`** `struct` that was the subject of the lab 8. You will add (1) a "link", or pointer to the `stuct` definition, and (2) a separate *start pointer*, and use them to manage a linked list.

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 **lab11** folder provided.

LAB 11: Introduction To Linked Lists `[ sLList.cpp ]`
Create a console application named **sLList**. Do NOT submit until you complete the last step.

STEP 1 of 2: A Simple Linked List
Create a project named **sLList.cpp**, containing 5 objects of a modified **`tod`** structure, creates a linked list with them, and prints them in *chronological* order. Borrow the **`coutTod()`** function from Lab 8 or Lab 9b, in order to print the object. If you elected to do the "optional bonus step" in Lab 8, you may use the "member function" version of this function, if you wish to do so. Use the following modified structure definition, which includes abbreviated names for hour, minute, and second:

```
struct tod
{
  int h; // hour, 0-23
  int m; // minute, 0-59
  int s; // second, 0-59
  char d[32]; // description
  tod* next; // link
};
```

Declare and initialize 5 objects of the above structure in `main()`, but do *not* do so in an array -- create separate objects, including these two:

```
tod noon = {12, 0, 0, "noon"};
tod midnight = {0, 0, 0, "midnight"};
```

The three additional objects should be for *lunchtime*, *suppertime*, and *bedtime*, initialized to values of your choosing.

Create a *start pointer* in `main()`, as an object of **`tod*`**. Initialize it to the memory address of **`midnight`**.

This starts the list with the midnight object. E.g.,

```
tod* start = 0; // created an empty linked list
start = &midnight;
```

Now complete the list by setting the `next` member of each struct object to its proper value. E.g.:

```
midnight.next = &noon; // noon follows midnight
```

Repeat with statements similar to the above statement until every one of the 5 struct objects has its `next` member initialized. Remember to set the last object's `next` member to *zero*, to mark it as the *end of the list*. E.g.:

```
bedtime.next = 0;
```

The initialized start pointer and the 5 objects' `next` members constitute a *chronologically* ordered linked list. Print the list in chronological order using a for-loop to traverse the linked list and sent each object individually to a `coutTod` function.

STEP 2 of 2: Linked List Maintenance
Modify **sLList.cpp** by adding code that inserts and deletes objects of the **`tod`** structure. Every time an object is inserted or deleted, reprint the modified list. Make these modifications:

1. **Insert breakfast.** Before the closing brace of the **`main()`** program, add a new section of code. In this new code, declare an object called "breakfast", and initialize its hour, minute, second, and descr to data values of your choosing. Insert it into the linked list by first setting the **`next`** value of the *new* object to the `next` value of the preceding object (presumably midnight), and then resetting the **`next`** value of the *preceding* object to the memory address of the new object. *Be sure to do these in the right order!* Use a **`cout`** to print a label saying that breakfast was added to the list, and reprint the list.

2. **Insert class start.** After the code added in step 1 above, add another new section of code. In this new code, declare an object called "class starts", and initialize its hour, minute, second, and descr to data values of your choosing. Insert it into the linked list, as above. Use a **`cout`** to print a label saying that the new object was added to the list, and reprint the list.

3. **Insert class end.** After the code added in step 2 above, add another new section of code. In this new code, declare an object called "class ends", and initialize its hour, minute, second, and descr to data values of your choosing. Insert it into the linked list, as above. Use a **`cout`** to print a label saying that the new object was added to the list, and reprint the list.

4. **Delete mealtimes.** After the code added in step 3 above, add another new section of code. In this new code, delete all mealtimes (i.e., breakfast, lunchtime, and suppertime) from the linked list. You will not delete the actual objects, but you will need to set the **`next`** values of their preceding objects to their **`next`** values. Use a **`cout`** to print a label saying that mealtimes were removed from the list, and reprint the list.

5. **Reinsert mealtimes.** After the code added in step 4 above, add another new section of code. In that new code, reinsert all mealtimes back into the linked list. Note that their objects already exist, so you just have to reset the appropriate **`next`** values. Use `cout` to print a label saying that mealtimes were reinserted, and reprint the list.

When inserting or deleting more than one node, do so ONE AT A TIME. If you insert or remove more than one node at at time, you're not doing this right. If you use other than TWO statements to insert or reinsert

ONE node, you're not doing this right. If you use other than ONE statement to remove ONE node, you're not doing this right.

*Post **sLList.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.