The objectives of this project are (1) to practice implementation and use of a concrete “Landmark” class, where each instance stores a name and a geographic location, and (2) to learn dynamic storage allocation and insertion, search, traversal, deletion, etc. with the doubly linked list data structure. The application idea is to help a world traveller plan an itinerary by inputting, printing, rearranging, deleting, etc. a sequence of Landmarks to visit. Since each Landmark stores its longitude and latitude coordinates, graphical output might show the itinerary drawn upon a world map. (I will assign that eventually if I can get a suitable Mercator projection map in Postscript ...) For simplicity, the world traveler will have to know the coordinates because the software has no atlas to look them up.

At this point, it would be very wise to read the specification below. After it, read the section on how to do the project. Then make sure you are familiar enough with the course contents featured in Main and Savitch’s DSO Chapters 2, 4 and 5 required for the data structure implementation. All the Chapter 5 operations or certain variants of them will be needed for this project, except for list copying. Study the pseudo-code and sample implementations as necessary. The essentials will be covered in the upcoming lectures and in Lab 3. When you begin draft design, creating some C++ files (RCS use required!) and the build script, and continue to complete the project, keep the this assignment sheet handy so you can consult it for details. (Maybe also use it to write notes and a “to-do” list.)

1 Specification of Commands, other inputs and actions

This software maintains for you (the user) a main list of Landmarks. The software has a simple and primitive text-based user interface: You are prompted to type commands and data input, and the software responds by printing more prompts and/or results.

We will evaluate most of the functionality of your project work in the order of the description of the commands below.

For each command, what you type or a general syntactic description of it is given first in Italic Font Type. Under some commands is given the exact text(s) or form of messages your program must print to report some specified result or when certain user error conditions occur. What the computer should print, such as the Command> prompt, is given in Fixed Font Type. Each command and message(s) are followed by an explanation, printed in Roman Font Type.

Each command and message text is available in the file below on itsunix
"acsi310/Proj2/strings.txt"
so you can copy and paste them into your program instead of typing them. For full credit, messages must be printed exactly as shown.

Unlike those in a practical “user manual” type specifications, the explanations include information about how each command should be implemented because learning these implementation techniques is a project objective.

First implementation information item: There must be 3 separate modules: a “user interface” module for inputting commands and displaying some or all of the results, a separate module for a
sequence of Landmarks implemented by C++ class LSequence, and a third separate module for the C++ class Landmark, where each instance represents one Landmark.

Since class LSequence implements doubly-linked lists, each instance must have two pointer-to-node variables head_ptr and tail_ptr to quickly locate the first and the last node of the list. (You can use alternative names for them if you want.)

For the user interface module: Program with strcmp (covered in Ch. 4) to distinguish cases 1-6. For cases 7-11 you can get started by coding int Pos; cin >> Pos; to read each position number; later (for a little more credit) you can add more subtle code using cin.peek to detect and recover from badly formatted or missing integers on the command line. Finally, you can use cin.getline() to read the search strings of cases 12-13.

1.1 Commands and other input operations one by one

1. Command><misspelled or unimplemented command>
<misspelled or unimplemented command}>
ERR: I don’t know that command.

If the white-space terminated token of characters at the beginning of the line does not match any of the commands defined below, or it matches a command you didn’t implement, reprint the command and then print the indicated error message. (A very small amount of credit will be awarded for “flushing away” the remainder of the bad command line after the first token.) Continue to input and process lines, except when the last input caused the program to exit.

Input of the command must be done with code like the following, or, if you want try for full error recovery, the code using cin.getline() which appears later.

```
#include <iostream>
#include <cstdlib> //supplies exit()  
using package std;
...
const int INBUFSIZE = <your figure out a suitable number!!>
char inbuf[INBUFSIZE];
...

cin.width(INBUFSIZE);  
cin >> inbuf;
//First whitespace is skipped. Then characters are successively read and  
//copied into the array until either whitespace is encountered, end-of-file  
//is encountered, or INBUFSIZE-1 characters have been copied. Finally, the  
//copied string is terminated by the NULL character ‘\0’. The terminating  
//character REMAINS in the input stream.
```

Warning: One of the grading tests will make your program read an excessively long command line. If it crashes instead of printing the error message and going on to the next command, you will lose credit for this case!
2. **Command>** *quit*

   Happy Travelling! Bon Voyage!

   The quit command makes the editor exit. Errors are impossible!

3. **Command>** *help*

   First, print 1-3 lines that explain that this program maintains a sequence of landmarks.

   Then, for each command you have implemented, print 1-3 lines to tell people how to use the feature.

   (You may implement this with a file if you know how, so long as you submit the file in the same directory where the executable program file proj2 will be built.)

4. **Command>** *newfirst*

   Name: *New York City, USA*

   Longitude (-180 to +180 degrees): -73.98
   Latitude (-90 to +90 degrees): 40.77

   (1) New York City, 73.98 degrees West, 40.77 degrees North.

   **Command>** *newfirst*

   Name: *Cape Town, South Africa*

   Longitude (-180 to +180 degrees): 18.37
   Latitude (-90 to +90 degrees): -35.92

   (1) Cape Town, South Africa, 18.37 degrees East, 35.92 degrees South.

   The *newfirst* command makes the computer prompt for the input of the name and coordinates in the form exactly like the examples above. First the user must type the landmark name on one line of 80 characters or fewer, including the newline character. Next, the computer prompts for the longitude and latitude exactly as shown, and, for each, separately, the user must respond with a positive or negative number expressed in decimal, within the indicated ranges. Negative signifies West longitude and South latitude. Because of the modernistic style of simplicity, angle measurements will be expressed using decimal fractions of degrees, rather than the traditional degrees, minutes, and seconds.

   It would be OK for this project if the program refuses to go further until the input of a Landmark is complete. (So Landmark detail input can be done by the Landmark module. For simplicity, that module need not report input aborts or failures to its caller.) However, the following kind of error detection and recovery is required:

   **Command>** *newfirst*

   Name: *Timbuktu, Mali*

   Longitude (-180 to +180 degrees): -3.07
   Latitude (-90 to +90 degrees): 98.6

   Coordinate input out of range. Try again.

   Latitude (-90 to +90 degrees): 16.75

   (1) Timbuktu, Mali, 3.07 degrees West, 16.75 degrees North.

   After receiving the complete input of a Landmark, the computer responds by printing first the position of the new Landmark in the sequence, followed immediately by the name and
coordinates. Note that the sign of each coordinate indicates whether the computer should print “East” or “West”, or “South” or “North”. By convention, 0 degrees is considered “East” or “North” according to which coordinate it represents.

The user, of course, must spell the command name `newfirst` exactly as shown, and the your program must print the computer prompts (Name:, Longitude(-180 to +180 degrees):, etc. and the landmark report exactly as shown, except of course, for the data that the user had supplied.

Input of the Landmark name must be read using the following code: (For simplicity, don’t bother removing any leading or trailing whitespace from it.)

```cpp
#include <iostream>
#include <cstdlib> //supplies exit()
using namespace std;
...
const int INBUFSIZE = 80;
char inbuf[INBUFSIZE];

if( !cin.getline(inbuf, INBUFSIZE) )
{
    cout << "ERR:Line input failed(too long?end of file?)..exiting." << endl;
    exit( 0 );
}
```

The call to `cin.getline()` reads an input line up to `INBUFSIZE` characters long including the terminating newline character. The line includes any leading or trailing whitespace characters (spaces or tabs here). The `cin.getline()` call fails if the input is too long. This call stores the inputted characters into the given array except that it replaces the terminating newline with the NULL char value. That replacement turns the inputted characters into a C-string. Ask yourself and become sure of the answer: What is the maximum length of a line this software will handle?

These rules apply even to the empty input line. Users will type that in by pressing the enter key only. It’s important to understand the empty C-string: It’s a NON-EMPTY array of the one character ‘\0’.

If it is the empty line or a non-empty line not beginning with a colon, the program must determine its length and add 1 to the length in order to determine how long an array of char to dynamically allocate. (Why? What can happen if 1 isn’t added?) After dynamically allocating (using new[] of course..) the array, the line must be copied into the array. The resulting new C-string will eventually become part of the data structure within instance of class `Landmark`. The input buffer `inbuf` above is NOT part of `Landmark`'s data structure!

**TIP:** It would be really smart to design, implement and test a concrete class `Landmark` for most of the above functionality all by itself. See DSO pages 74-81 because a `Landmark` is essentially a fancy point with a name!
5. **Command** `printforward`
Prints all the Landmarks, one per output line, preceded by the position of each Landmark in the sequence, in forward order. For example, assuming just New York and Capetown were input exactly as above, the result of the `printforward` command would be:

1. Timbuktu, Mali, 3.07 degrees West, 16.75 degrees North.
2. Cape Town, South Africa, 18.37 degrees East, 35.92 degrees South.
3. New York City, 73.98 degrees West, 40.77 degrees North.

Note and explain to yourself why Cape Town has position 2 and is printed second.
If sequence (of Landmarks) is empty, nothing is printed.

6. **Command** `newlast`
The `newlast` command works exactly the same as `newfirst` except the new Landmark is put at the end of the sequence instead of at the beginning.

7. **Command** `print <position number>`
ERR: Position `<position number>` is not in your sequence.
ERR: Missing or badly formatted position number.

Command `print 2`
Cape Town, South Africa, 18.37 degrees East, 35.92 degrees South.
Print the data about the one Landmark currently in the given position within the sequence, or the relevant one of the above error messages.
If you cannot implement detection and recovery from the second kind of input errors soon, please move on and go back to that later. The problem is that after the user types a line that begins with `print`, the input buffer used internally by `cin` will contain either a correct decimal integer numeral before the newline, nothing but perhaps whitespace and then the newline, or erroneous “junk” and then the newline. To handle all cases of error, you will have to solve a programming problem that involves the details about `iostream` input explained on pages 756-759 of DSO.

8. **Command** `delete <position number>`
ERR: Position `<position number>` is not in your sequence.
ERR: Missing or badly formatted position number.

If there is no error, print and then delete the Landmark in the indicated position. Naturally, the position of each after the deleted one will then become one step less.

Command `delete 2`
Cape Town, South Africa, 18.37 degrees East, 35.92 degrees South.
DELETED!

Command `printforward`
Timbuktu, Mali, 3.07 degrees West, 16.75 degrees North.
New York City, 73.98 degrees West, 40.77 degrees North.
Notice how New York moved to 2nd place because we deleted Cape Town.

9. **Command** `swapfirst <position number>`
ERR: Position `<position number>` is not in your sequence.
ERR: Trying to swap a thing with itself?? Waste of time!
ERR: Missing or badly formatted position number.

The linked list node at the given position number should be swapped with the node at the beginning (position 1). The second error occurs when the given position number is 1 and the sequence is not empty.

This and the next two operations must be implemented by modifying pointer variables within existing list nodes and possibly the head_ptr and tail_ptr pointers.

10. **Command>** `swaplast <position number>`
    ERR: Position <position number> is not in your sequence.
    ERR: Trying to swap a thing with itself?? Waste of time!
    ERR: Missing or badly formatted position number.

    The linked list node at the given position number should be swapped with the node at the end (position equals length of the sequence `n`). The second error occurs when the given position number is `n` and the sequence is not empty.

11. **Command>** `swap <position number 1> <position number 2>`
    ERR: Position <position number> is not in your sequence.
    ERR: Trying to swap a thing with itself?? Waste of time!
    ERR: Missing or badly formatted position number.

    The two linked list nodes indicated by the given sequence numbers should be repositioned so they reverse their positions. This should be implemented by changing all the values of the relevant pointer variables. They include variables within the swapped nodes, within their neighbors, and sometimes the the head_ptr and/or tail_ptr variables.

    Complete recovery for the last kind of error requires detecting missing or badly formatted numerals in either or both of the two command argument positions. See remarks under command 7.

12. **Command>** `findfirst <Complete name of a landmark>`
    ERR: Line input failed (too long? end of file?) .. exiting.
    ERR: Empty search string?
    ERR: Given landmark not found.

    If there are no errors, print the first Landmark and its position number (formatted as before) for which the name is exactly equal to the substring given on the command line after the whitespace after `findfirst`

    Since the complete name of a landmark has an unpredictable number of whitespaces, punctuation marks, etc, you should invoke code like that under command 4 after the `findfirst` or `findbest` (below) command name is recognized.

13. **Command>** `findbest <Search string>`
    ERR: Line input failed (too long? end of file?) .. exiting.
    ERR: Empty search string?
If there are no errors, print the first Landmark and its position number among those landmarks whose names have the greatest length prefix in common with the the given <Search string>. You will have to write your own code to determine the length of the longest common prefix of two C-strings.

2 How to Do This Project; Some Graded Items!

C-string literals of the exact (and other) strings that your program should print or recognize have been pre-typed for you. I also prepared sample compile and link commands for you to use to start your build.sh file. Copy them from ~acsi310/Proj2/messages.txt or download (same thing??) them from http://www.cs.albany.edu/~sdc/CSI310/Proj/Proj2/messages.txt

For design practice, you must design a class named class LSequece whose public member functions provide the core operations a list of class Landmark instances. This class must implement one sequence of Landmarks using one doubly linked lists of nodes, where each node holds a pointer to a dynamically allocated Landmark instance. The implementation’s data structure consists of private data members of LSequence and dynamically allocated linked list nodes that hold pointers to Landmarks. The private data members must of course include head and tail pointers to the linked list. You might choose to include other private data members for such things as the number of Landmarks, etc.

It is up to you how much of the rest of the program is designed and implemented using classes and other object oriented practices. One choice is to make use of Main and Savitch’s “linked list toolkit”. Another (my preference for CS2 at Albany) is to use a struct for list nodes and let LSequence’s member functions manipulate node fields directly. For example,

```cpp
#include "Landmark.h"
struct node {
    class Landmark *pData;
    node *pNext;
    node *pPrev;
};
```

For simplicity, I think it would be a good idea to provide public functions of class LSequence to print using cout.

It is your design choice about whether class LSequence’s functions detect and/or print messages about user errors described below, or whether the LSequence should enforce preconditions and its caller detect and act on user errors. It is smart to think about what would happen with user errors before you commit much time into the design of LSequence.

The job of (an instance of) class LSequence is to hold the data being edited, carry some out changes to it, and print (at your design option) copies of selected contents. Following this design requirement, class LSequence MUST NOT obtain input from cin or a file, NOR figure out commands by examining input strings! If the user interface language changes, the LSequence class must be reusable without change.

One critical design choice for LSequence is “Will LSequence call new to allocate space for each new Landmark then store the name and coordinates given to it, or (alternatively) will the
caller (probably the user interface module) call \texttt{new} and pass to \texttt{LSequence} the address of the new \texttt{Landmark}?

Another design choice involves which class will call \texttt{delete[]} on pointers to C-strings and call \texttt{delete} on pointers to \texttt{Landmarks} to recycle dynamic memory when a \texttt{Landmark} is deleted.

Part of the project grade will come from your \texttt{LSequence} and \texttt{Landmark} classes’ header files \texttt{LSequence.h} and \texttt{Landmark.h} to document their interfaces in terms of pre and post conditions on their public function members. In addition to this documentation of the public function members, comments to document the data members (where the data structure begins), preferably written in the form of invariants, will also be graded.

I suggest you write the user interface in a “main” module much like the test driver of the previous project and Main/Savitch’s sample programs. This main module, or perhaps classes of your own design if you choose that route, but not of class \texttt{LSequence}, should analyze the user input commands! One main difference is that the user interface module should read in whole lines of input, not just single characters. It could use C-string library function \texttt{strcmp()}, described in DSO Chapter 4, to test if the string given by the user equals a legal command (see below).

1. All the software development practices given on the project 1 assignment must also be followed for project 2 in order to earn full credit. Some details and additional requirements are given below.

2. Submission of the RCS revision history database for non-trivial files, showing your development process, is REQUIRED. It was ok (with points off) to omit it or start it late for the first project, but now, \textbf{PROJECT 2 (and the rest) will earn 0 points WITHOUT IT}. Right now is the time to learn basic RCS use, if you haven’t already.

3. Organization of source files must reflect the software’s design, or points will be taken off. In particular, at the very least, \texttt{LSequence.h}, \texttt{LSequence.cxx}; \texttt{Landmark.h}, \texttt{Landmark.cxx} and the the \texttt{.cxx} file containing the user interface code must be separate files. The file \texttt{LSequence.h} must document your \texttt{LSequence} class with pre and postconditions for all public functions. Similarly for \texttt{Landmark.h} and class \texttt{Landmark}. You may write the definition of \texttt{struct node} either in \texttt{LSequence.h} or in a separate header file.

4. An executable “build” script named \texttt{build.sh} is required for full credit. When run, the build script should compile and link the software to create an executable file named “\texttt{proj2}” in the same directory.

At this point, it would be wise to also write and test a “cleanup” script that automatically deletes any “core”, object or executable program files so that you can use it before submitting the project.

5. Submit a single directory to project name \texttt{Proj2} using \texttt{“turnin-csi310 -c csi310 -p Proj2 Directory ”}. Verify with \texttt{“turnin-csi310 -v -c csi310 -p Proj2”}.

As before, points will be deducted if object, “core” or executable files are submitted.