Programming Assignment #3: Listy String

COP 3502, Spring 2023

Due: Sunday, March 12, *before* 11:59 PM

Abstract

In this programming assignment, you will use linked lists to represent strings. You will implement functions that manipulate these linked lists to transmute the strings they represent. In doing so, you will master the craft of linked list manipulation!

By completing this assignment, you will also gain experience with file I/O in C and processing command line arguments. You might find it useful to refer to the notes on file I/O in Webcourses as you work on this assignment.

Important note: In your assignments, you can use any code I've posted in Webcourses, as long as you leave a comment saying where that code came from. Of course, you cannot use code posted by other professors, and you should never incorporate or refer to code from online resources or from other individuals.

Deliverables

ListyString.c

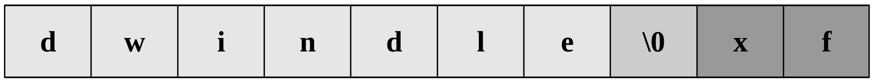
Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Overview

1.1. Array Representation of Strings in C

We have seen that strings in C are simply `char` arrays that use the null terminator (the character '`\0`') to mark the end of a string. For example, the word "dwindle" is represented as follows:

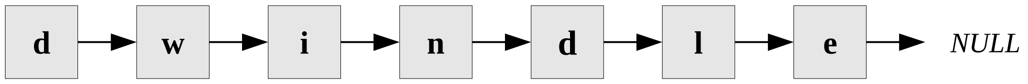


Notice the unused portion of the array may contain garbage data.

1.2. A Linked List Representation of Strings

In this assignment, we will use linked lists to represent strings. Each node will contain a single character of the string. The null terminator ('\0') will not be afforded its own node in the linked list. Instead, we will know that we have reached the end of a string when we encounter a NULL pointer.

For example, the word "dwindle" is represented as follows:



The bulk of this assignment will involve writing functions to transmute these so-called "listy strings."

1.3. ListyString and ListyNode Structs (*ListyString.h*)

For your linked lists, you must use the structs we have specified in *ListyString.h* without any modifications. You **must** #include this header file from *ListyString.c* like so:

```
#include "ListyString.h"
```

The node struct you will use for your linked lists is defined in *ListyString.h* as follows:

```
typedef struct ListyNode
{
   char data;                  // Each node holds a single character.
   struct ListyNode *next;     // Pointer to next node in linked list.
} ListyNode;
```

Additionally, there is a *ListyString* struct that you will use to store the head of each linked list string, along with the length of that list:

```
typedef struct ListyString
{
   struct ListyNode *head;     // Pointer to head of string's linked list.
   int length;                 // Length of this string / linked list.
} ListyString;
```

2. Input Files

In this assignment, you will have to open and process an input file. When we run your program, we will use a command line argument to specify the name of the input file that your program will read. We will always provide a *single* command line argument after the name of the executable file when running your program at the command line. For example:

```
./a.out input01.txt
```

The first line of the input file will always be a single string that contains at least 1 character and no more than 1023 characters, and no spaces. The first thing you should do when processing an input file is to read in that string and convert it to a ListyString (i.e., a linked list string). That will become your working string, and you will manipulate it according to the remaining commands in the input file.

Each of the remaining lines in the file will correspond to one of the following string manipulation commands, which you will apply to your working string in order to achieve the desired output for your program:

Command	**Description**
`@ key str`	In your working string, replace all instances of *key* with *str*.
`> str`	Concatenate *str* to the end of your working string.
`< str`	Prepend *str* to the beginning of your working string.
`~`	Reverse the working string.
`* str`	Censor the working string by replacing every character in the working string that occurs in *str* with an asterisk (‘*’). This process must be O(k + n), where *k* is the length of *str* and *n* is the length of the working string.
`- key`	Delete all instances of *key* (if any) from your working string.
`& str`	Interleave *str* into your working string. See the function description for *stringWeave()* for more details.
`# key`	Print the number of times *key* occurs in the working string (followed by ‘\n’).
`?`	Print the number of characters in the working string (followed by ‘\n’).
`!`	Print the working string (followed by ‘\n’).

Important note: In the table above, *key* is always a single character, and *str* is a string. Both *key* and *str* are guaranteed to contain alphanumeric characters only (A-Z, a-z, and 0-9). Not counting the need for a null terminator (‘\0’), *str* can range from 1 to 1023 characters (inclusively). So, with the null terminator, you might need up to 1024 characters to store *str* as a char array when reading from the input file.

Another important note: If one of the above commands modifies your working string, you should also ensure that the *length* member of that ListyString struct gets updated.

For more concrete examples of how these commands work, see the attached input/output files and check out the function descriptions below in Section 3, "Function Requirements." For a refresher on how to process command line arguments in C, please refer to the PDF for Program #1 (Numeronym).

3. Function Requirements

In the source file you submit, *ListyString.c*, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

Super Important Note: The input file specification in Section 2, "Input Files," gives certain restrictions on the strings you'll have to process from those input files. Namely, strings in the input file are limited to 1023 characters (not including the null sentinel) and are always alphanumeric strings with no spaces or other non-alphanumeric characters. Those restrictions are designed to make your *processInputFile()* function more manageable, and **only** apply when reading from an input file. Those restrictions do **not** apply when we call your functions in unit testing. For example, we could pass the string "Hello, world!" to your *createListyString()* function when we call it manually during unit testing.

```
int main(int argc, char **argv);
```

Description: You have to write a *main()* function for this program. It should only do the following three things: (1) capture the name of an input file (passed as a command line argument), (2) call the *processInputFile()* function (passing it the name of the input file to be processed), and (3) return zero.

Returns: Your *main()* function must return zero (0).

```
int processInputFile(char *filename);
```

Description: Read and process the input file (whose name is specified by the string *filename*) according to the description above in Section 2, "Input Files." To perform the string manipulations described in that section, you should call the corresponding required functions listed below. In the event that a bad filename is passed to this function (i.e., the specified file does not exist), this function should simply return 1 without producing any output.

Output: This function should only produce output if the input file has "#", "?", and/or "!" commands. For details, see Section 2 ("Input Files"), or refer to the input/output files included with this assignment. Note that this function should not produce any output if the input file does not exist.

Special Considerations: You should only pass through the input file once. Do not open the input file multiple times, read its contents multiple times, or use the *rewind()* function (or any similar functions) to move backward in the file.

Returns: If the specified input file does not exist, return 1. Otherwise, return 0.

```
ListyString *createListyString(char *str);
```

Description: Convert *str* to a ListyString by first dynamically allocating a new ListyString struct, and then converting *str* to a linked list string whose head node will be stored inside that ListyString struct. Be sure to update the *length* member of your new ListyString, as well.

Special Considerations: *str* may contain any number of characters, and it may contain non-alphanumeric characters. If *str* is NULL or an empty string (“”), simply return a new ListyString whose *head* is initialized to NULL and whose *length* is initialized to zero.

Runtime Requirement: This should be an O(*k*) function, where *k* is the length of *str*.

Returns: A pointer to the new ListyString. Ideally, this function would return NULL if any calls to *malloc()* failed, but I do not intend to test your code in an environment where *malloc()* would fail, so you are not required to check whether *malloc()* returns NULL.

```
ListyString *destroyListyString(ListyString *listy);
```

Description: Free any dynamically allocated memory associated with the ListyString and return NULL. Be sure to avoid segmentation faults in the event that *listy* or *listy*→*head* are NULL.

Returns: NULL.

```
ListyString *cloneListyString(ListyString *listy);
```

Description: Using dynamic memory allocation, create and return a new copy of *listy*. Note that you should create an entirely new copy of the linked list contained within *listy*. (That is, you should not just set your new ListyString’s head pointer equal to *listy*→*head*.) The exception here is that if *listy*→*head* is equal to NULL, you should indeed create a new ListyStruct whose *head* member is initialized to NULL and whose *length* member is initialized to zero. If *listy* is NULL, this function should simply return NULL.

Runtime Requirement: The runtime of this function must be no worse than O(*n*), where *n* is the length of the ListyString.

Returns: A pointer to the new ListyString. If the *listy* pointer passed to this function is NULL, simply return NULL.

```
ListyString *listyCat(ListyString *listy, char *str);
```

Description: Concatenate *str* to the end of the linked list string inside *listy*. If *str* is either NULL or the empty string (“”), then *listy* should remain unchanged. Be sure to update the *length* member of *listy* as appropriate.

Special Considerations: If *listy* is NULL and *str* is a non-empty string, then this function should create a new ListyString that represents the string *str*. If *listy* is NULL and *str* is NULL, this function should simply return NULL. If *listy* is NULL and *str* is a non-NULL empty string (“”), then this function should return a ListyString whose *head* member has been initialized to NULL and whose *length* member has been initialized to zero.

Runtime Requirement: The runtime of this function must be no worse than O(*n*+*m*), where *n* is the length of *listy* and *m* is the length of *str*.

Returns: If this function caused the creation of a new ListyString, return a pointer to that new ListyString. If one of the special considerations above requires that a NULL pointer be returned, then do so. Otherwise, return *listy*.

```
ListyString *listyPrepend(ListyString *listy, char *str);
```

Description: Prepend *str* to the beginning of the linked list string inside *listy*. If *str* is either NULL or the empty string (“”), then *listy* should remain unchanged. Be sure to update the *length* member of *listy* as appropriate.

Special Considerations: If *listy* is NULL and *str* is a non-empty string, then this function should create a new ListyString that represents the string *str*. If *listy* is NULL and *str* is NULL, this function should simply return NULL. If *listy* is NULL and *str* is a non-NULL empty string (“”), then this function should return a ListyString whose *head* member has been initialized to NULL and whose *length* member has been initialized to zero.

Runtime Requirement: The runtime of this function must be no worse than O(*n*+*m*), where *n* is the length of *listy* and *m* is the length of *str*.

Returns: If this function caused the creation of a new ListyString, return a pointer to that new ListyString. If one of the special considerations above requires that a NULL pointer be returned, then do so. Otherwise, return *listy*.

```
void replaceChar(ListyString *listy, char key, char *str);
```

Description: This function takes a ListyString (*listy*) and replaces all instances of a certain character (*key*) with the specified string (*str*). If *str* is NULL or the empty string (“”), this function simply removes all instances of *key* from the linked list. If *key* does not occur anywhere in the linked list, the list remains unchanged. If *listy* is NULL, or if *listy*→*head* is NULL, simply return.

Important Note: Be sure to update the *length* member of the ListyString as appropriate.

Runtime Requirement: The runtime of this function must be no worse than O(*n* + *km*), where *n* is the length of *listy*, *k* is the number of times *key* occurs in the ListyString, and *m* is the length of *str*.

Returns: Nothing. This is a *void* function.

```
void listyCensor(ListyString *listy, char *badChars);
```

Description: Within *listy*, any character that occurs in the *badChars* string must be replaced with an asterisk (‘*’). For example, if *listy* is `[d]→[w]→[i]→[n]→[d]→[l]→[e]` and *badChars* is “elder”, then <u>**all**</u> instances of ‘e’, ‘l’, ‘d’, and ‘r’ will be replaced with asterisks in the ListyString, and the censored version of *listy* will be `[*]→[w]→[i]→[n]→[*]→[*]→[*]`. Note that the double occurrence of ‘e’ in “elder” has no special impact in this function. Note also that even though ‘d’ only occurred once in “elder”, we censored <u>**both**</u> instances of that character in *listy*.

If *badChars* is either NULL or the empty string (“”), then *listy* should remain unchanged. If *listy* is NULL, or if *listy*→*head* is NULL, you should simply return from this function.

Note that *badChars* may contain non-alphabetic characters. In fact, it could contain any character that we can represent with a *char* in C. Recall that there are 256 distinct possible *char* values in C, with ASCII values 0 through 255. The length of *badChars* may exceed 256, though, because some characters may be repeated (as we saw above in “elder”). *badChars* will of course be terminated with a null sentinel (‘\0’) (unless *badChars* is NULL, in which case it contains no characters – not even a null sentinel). The first null sentinel you encounter in *badChars* should always be considered to mark the end of that string; it should never be considered one of the characters to be censored in *listy*.

Runtime Requirement: The runtime of this function must be no worse than O($k + n$), where k is the length of *badChars* and n is the length of the ListyString. This function should **<u>not</u>** make repeated calls to *replaceChar()*, as that would result in an O(kn) runtime (assuming *replaceChar()* is an O(n) function).

Hint: Highlight and/or copy and paste the following block of text into a text editor for a hint on how to meet the runtime restriction on this function. Challenge yourself to work on this one without peeking at the hint, though!

Returns: Nothing. This is a *void* function.

```
void reverseListyString(ListyString *listy);
```

Description: Reverse the linked list contained within *listy*. Be careful to guard against segfaults in the cases where *listy* is NULL or *listy*→*head* is NULL.

Runtime Consideration: Ideally, this function should be O(n), where n is the length of the ListyString. Note that if you repeatedly remove the tail of *listy*’s linked list and insert it at the tail of a new linked list using a slow tail insertion function, that could devolve into an O(n^2) approach to solving this problem.

Returns: Nothing. This is a *void* function.

```
ListyString *stringWeave(ListyString *listy, char *str);
```

Description: Interleave the characters of *str* into *listy* in such a way that each subsequent character from *str* appears after each subsequent character from *listy*. Be sure to update the *length* member of *listy* as appropriate.

For example, if *listy* is `[d]→[w]→[i]→[n]→[d]→[l]→[e]` and *str* is “elder”, this function should modify *listy* to become `[d]→[e]→[w]→[l]→[i]→[d]→[n]→[e]→[d]→[r]→[l]→[e]`. Notice that we start inserting characters from *str* **<u>after</u>** the first character in *listy*. Once we run out of characters in *str*, there is no impact on the rest of the ListyString. If we were instead to run out of characters in the ListyString before running out of characters in *str*, we would simply tack the remaining characters from *str* onto the end of the ListyString. For example, if *listy* is `[x]→[y]→[z]` and *str* is “elder”, *listy* becomes `[x]→[e]→[y]→[l]→[z]→[d]→[e]→[r]`.

Special Considerations: If *str* is NULL, then *listy* should remain unchanged, and you should simply return *listy* from this function. If *listy* is NULL and *str* is a non-NULL, non-empty string, then this function should create a new ListyString that represents the string *str*. If *listy* is NULL and *str* is a non-NULL empty string (“”), then this function should create a new ListyString whose *head* member has been initialized to NULL and whose *length* member has been initialized to zero. If *listy* and *str* are non-NULL but *listy*→*head* is NULL, you should simply update *listy* to contain *str*.

Special Consideration Related to Runtime: Do not destroy nodes or change the character within any of the original nodes in the ListyString. If you find yourself copying characters from one node to another in the ListyString, you are probably producing an unnecessarily slow runtime for this function. The goal here is to create new nodes containing the characters from *str* and to link them up to the existing nodes in *listy* without moving *listy*’s nodes to new locations in memory or changing the contents of any of *listy*’s original nodes.

Runtime Requirement: The runtime of this function must be no worse than O(n + k), where *n* is the length of *listy* and *k* is the length of *str*. To achieve this, you must avoid repeatedly calling slow insertion functions or repeatedly looping through *listy* and/or *str*. You should loop through *listy* and *str* exactly once each.

Returns: If this function caused the creation of a new ListyString, return a pointer to that new ListyString. Otherwise, return *listy*.

```
ListyString *listyWeave(ListyString *listy1, ListyString *listy2);
```

Description: This function is similar to *stringWeave()*, except that it operates on two ListyString arguments. Interleave the characters of *listy2* into *listy1* in such a way that each subsequent character from *listy2* appears after each subsequent character from *listy1*. Be sure to update the *length* member of *listy* as appropriate. Note that this function ultimately modifies *listy1*, but *listy2* remains unchanged, in the same way that *stringWeave()* modifies its listy argument but leaves the str argument unchanged.

Refer to *stringWeave()* for information about the expected behavior of this function, such as how to deal with NULL pointers and empty strings and what to do if one ListyString passed to this function is longer than the other. Keep in mind that a non-NULL ListyString pointer with a NULL *head* pointer (and a *length* of zero) is equivalent to a non-NULL empty string (“”). If you require further clarification on the expected behaviors of this function, be sure to refer to the test cases included with this assignment.

Special Considerations: As with *stringWeave()*, you should not destroy nodes in *listy1* or change the contents of any of *listy1*’s original nodes. You should simply be weaving new nodes into *listy1* without moving its existing nodes around in memory or changing the characters those existing nodes contain. Furthermore, you should not make any nodes within *listy1* point to actual nodes contained within *listy2*. Instead, you should create **copies** of *listy2*’s nodes and inject those new copies into *listy1*.

Runtime Requirement: The runtime of this function must be no worse than O(*m* + *n*), where *m* is the length of *listy1* and *n* is the length of *listy2*. For further details, refer to the function description for *stringWeave()*, above.

Returns: If this function caused the creation of a new ListyString, return a pointer to that new ListyString. Otherwise, return *listy*.

```
int listyCmp(ListyString *listy1, ListyString *listy2);
```

Description: Compare the two ListyStrings. Return 0 (zero) if they represent equivalent strings. Otherwise, return any non-zero integer of your choosing. Note that the following are **not** considered equivalent: (1) a NULL ListyString pointer and (2) a non-NULL ListyString pointer in which the *head* member is set to NULL (or, equivalently, the *length* member is set to zero). For the purposes of this particular function, (2) represents an empty string, but (1) does not. Two NULL pointers are considered equivalent, and two empty strings are considered equivalent, but a NULL pointer is not equivalent to an empty string.

Runtime Requirement: The runtime of this function must be no worse than O(n+m), where n is the length of *listy1* and m is the length of *listy2*.

Returns: Zero (0) if the ListyStrings represent equivalent strings; otherwise, return any integer other than zero.

```
int charCount(ListyString *listy, char key);
```

Description: Return the number of times *key* occurs in the ListyString.

Runtime Requirement: The runtime of this function must be O(n), where n is *listy*'s length.

Returns: The number of times *key* occurs in *listy*. If *listy* is NULL, return -1. If *listy* is non-NULL, but *listy*→*head* is NULL, return zero.

```
int listyLength(ListyString *listy);
```

Description: Return the length of the ListyString (i.e., the length of *listy*'s linked list).

Runtime Requirement: The runtime of this function must be O(1).

Returns: The length of the string (i.e., the length of the linked list contained within *listy*). If *listy* is NULL, return -1. If *listy* is non-NULL, but *listy*→*head* is NULL, return zero.

```
void printListyString(ListyString *listy);
```

Description: Print the string stored in *listy*, followed by a newline character, '\n'. If *listy* is NULL, or if *listy* represents an empty string, simply print "(empty string)" (without the quotes), follow by a newline character, '\n'.

Returns: Nothing. This is a void function.

```
double difficultyRating(void);
```

Description: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Description: Return an estimate (greater than zero) of the number of hours you spent on this assignment. Your return value must be a realistic and reasonable estimate.

4. Searching the Test Case Files for Particular Symbols or Function Calls

Given the large number of test cases included with this project, you might be wondering how you can quickly find which input files contain particular symbols or which source files call particular functions. At the command line on Mac or Linux systems (including the WSL command line for those running Windows), there is a command called *grep* that lets you search files for a particular string. Here's the general syntax for *grep*:

```
grep "SEARCH_STRING" filename
```

For example, the following command will find all occurrences of the @ symbol in *input04.txt*:

```
grep "@" input04.txt
```

If you want to search all files that end with *.txt*, simply use `*.txt` in place of the filename, like so:

```
grep "@" *.txt
```

Similarly, if you want to search all files that end with *.c* (which is especially useful if you want to find out which of the *.c* files call some particular function that you're working on) simply use `*.c` in place of the filename, like so:

```
grep "replaceChar" *.c
```

In those cases, the asterisk is what we call a "wildcard," and we're saying, "search files named ANYTHING, as long as the filename has .txt or .c at the end."

Note that *grep* is case sensitive, so searching for "replacechar" is **not** the same as searching for "replaceChar".

Note also that failing to use "double quotes" around your search string might result in weird behaviors in some cases but not others. You can search online for a guide to *grep* if you want to know more, but for now, you're better off just always using the double quotes.

5. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *ListyString.c*, *ListyString.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm isn't too hard, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*ListyString.c*, *ListyString.h*, *test-all.sh*, the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

```
bash test-all.sh
```

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd ListyProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

```
cd listystring\ files
```

```
cd "listystring files"
```

It's probably easiest to just avoid file and folder names with spaces.

6. Checking the Output of Individual Test Cases

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. This section tells you how to do that.

There are two types of test cases included with this assignment: (1) the test cases where you compile your program and run it with an input filename (such as *input01.txt*) specified as a command line argument, and (2) the test cases where you have to compile one of our source files (*UnitTest06.c* through *UnitTest31.c*) along with your source file (*ListyString.c*) in order to run.

If you want to run one of these test cases individually in order to examine its output outside of the *test-all.sh* script, here's how you do it:

1. Instructions for running your program with an input file specified as a command line argument:

a. Place all the test case files released with this assignment in one folder, along with your *ListyString.c* file.

b. In *ListyString.h*, make sure line 14 is commented out *exactly* as follows, with no space directly following the "//". This is the *only* line of *ListyString.h* that you should ever modify:

```
//#define main __hidden_main__
```

c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program:

```
gcc ListyString.c
```

d. To run your program and redirect the output to *output.txt*, provide one of the input filenames at the command line after `./a.out`, like so:

```
./a.out input01.txt > output.txt
```

e. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/input01-output.txt
```

2. **Instructions for compiling your program with one of the unit test cases:**

a. Place all the test case files released with this assignment in one folder, along with your *ListyString.c* file.

b. In *ListyString.h*, make sure line 14 uncommented and appears *exactly* as follows. This is the *only* line of *ListyString.h* that you should ever modify:

```
#define main __hidden_main__
```

c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program with *UnitTestLauncher.c* and any *one* of the *UnitTestXX.c* files you would like to test:

```
gcc ListyString.c UnitTestLauncher.c UnitTest06.c
```

d. To run your program and redirect the output to *output.txt*, execute the following command:

```
./a.out > output.txt
```

e. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/UnitTest06-output.txt
```

Super Important: *Remember, using the test-all.sh script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.*

7. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in the *ListyString.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ You must use linked lists to receive credit for this assignment.
- ★ In general, if a string is passed to a function and the function description does not ask you to modify that string, then you should not modify it. In all the functions you write, avoid destroying the data pointed to by any pass-by-reference arguments as an unwanted side effect of accomplishing the task at hand.
- ★ Do not write to any files. If your code attempts to open a file in any sort of write mode, we might refuse to compile your program and instead assign a zero for the assignment.
- ★ Please do not use *scanf()* to read input from the keyboard.
- ★ Do not declare new variables part way through a function. All variable declarations should occur at the *top* of a function, and all variables must be declared inside your functions or declared as function parameters.
- ★ Do not use *goto* statements in your code.
- ★ Do not make calls to C's *system()* function.
- ★ Do not write malicious code, including code that attempts to open files it shouldn't be opening, whether for reading or writing. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

8. Style Restrictions (***Super Important!***)

These are the same as in the previous assignment. Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your *#include* statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align

with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be indented with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(int argc, char **argv)* instead of *int main (int argc, char **argv)*. Similarly, use *printf("...")* instead of *printf ("...")*.

★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.

★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do <u>not</u> have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

9. Deliverable (Submitted via Webcourses, not Eustis)

Submit a single source file, named *ListyString.c*, via Webcourses. The source file should contain definitions for all the required functions (listed above). Be sure to include your name and NID as a comment at the top of your source file. Don't forget to *#include "ListyString.h"* in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also compile and run on Eustis like so:

```
gcc ListyString.c
./a.out input01.txt
```

10. Grading

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

75%	Passes test cases with 100% correct output formatting. This portion of the grade may include tests for memory leaks.
10%	Adequate comments and whitespace. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. Please include a header comment with your name and NID (<u>**not**</u> your UCF ID).
10%	Implementation details and adherence to the special restrictions imposed on this assignment. This will likely involve some manual inspection of your code.
5%	Source file is named correctly. Spelling and capitalization count.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

For this program, we will also be unit testing your code. That means we will devise tests that determine not only whether your program's overall output is correct, but also whether each function does exactly what it is required to do. So, for example, if your program produces correct output but your *createListyString()* function is simply a skeleton that returns NULL no matter what parameters you pass to it, your program will fail the unit tests.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements.

Start early. Work hard. Good luck!