Programming Assignment #3: Listy Strings

COP 3502, Spring 2019

Due: Sunday, March 10, *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:

d	**w**	**i**	**n**	**d**	**l**	**e**	**\0**	**x**	**f**

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:

d → **w** → **i** → **n** → **d** → **l** → **e** → *NULL*

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 **<u>must</u>** #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:

```c
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:

```c
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.
`- key`	Delete all instances of *key* (if any) from your working string.
`~`	Reverse the working string.
`?`	Print the number of characters in the working string.
`!`	Print the working string.

Important note: For the first three commands listed in this table, *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" (page 4). For a refresher on how to process command line arguments in C, please refer to the PDF for Program #1 (DupeyDupe).

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.

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 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: 0 (zero).

```
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.

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 should 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.

```
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 should be no worse than $O(n + km)$, where n is the length of the ListyString, 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 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 *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*.

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

Returns: 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);
```

Returns: A reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

4. 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 . YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the dot (".") 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.

5. 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 <u>*exactly*</u> as follows, with no space directly following the "//". This is the <u>*only*</u> 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 <u>*exactly*</u> as follows. This is the <u>*only*</u> 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.*

6. 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 intended 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.)

7. 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 `#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
```

8. 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.

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.

Special Restrictions: You must use linked lists to receive credit for this assignment. Also, please do not use global variables in this program, do not make any system calls (e.g., `system("pause")`), and do not write to any files. Violating any of these restrictions may result in a huge loss of points.

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.

Start early. Work hard. Ask questions. Good luck!