Programming Assignment #3: Listy Strings

COP 3502, Fall 2017

Due: Wednesday, October 25, *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.

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 cannot incorporate code from online resources or from other students.

Attachments

ListyString.h, input{01-05}.txt, testcase{06-31}.c, output{01-31}.txt, UnitTest.c, test-all.sh

Deliverables

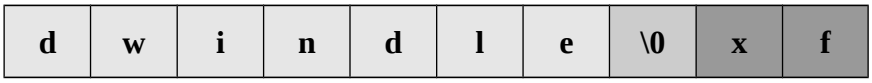
ListyString.c

(***Note!*** Capitalization and spelling of your filename matter!)

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:

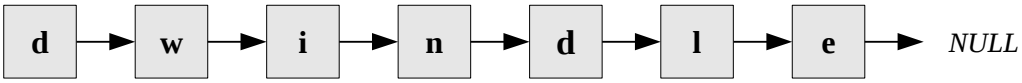


Notice the unused portion of the array may contain garbage data.

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` the 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. Guide to Command Line Arguments

2.1. Capturing Command Line Arguments in a C Program

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* give the name of a valid input file as a command line argument when running your program. For example:

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

It's super easy to get command line arguments (like the string "input01.txt" in this example) into your program. You just have to change the function signature for `main()`. Whereas we have typically seen `main()` defined using `int main(void)`, you will now use the following function signature instead:

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

Within `main()`, `argc` is now an integer representing the number of command line arguments passed to the program (including the name of the executable itself). `argv` is an array of strings that stores all those command line arguments. `argv[0]` stores the name of the program being executed.

2.2. Example: A Program That Prints All Command Line Arguments

For example, here's a simple program that would print out all the command line arguments passed to a program:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int i;
   for (i = 0; i < argc; i++)
      printf("argv[%d]: %s\n", i, argv[i]);
   return 0;
}
```

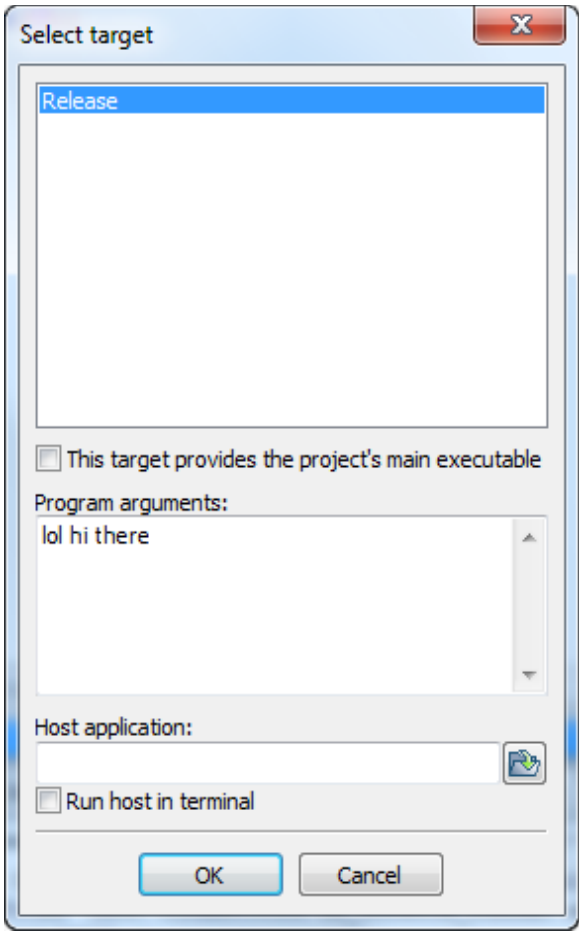
If we compiled that code into an executable file called `a.out` and ran it from the command line by typing `./a.out lol hi there!`, we would see the following output:

```
argv[0]: ./a.out
argv[1]: lol
argv[2]: hi
argv[3]: there!
```

2.3. Passing Command Line Arguments in CodeBlocks

You can set up CodeBlocks to automatically pass command line arguments to your program every time you hit *Build and run (F9)* within the IDE. Simply go to *Project → Set program's arguments...*, and in the box labeled *Program arguments*, type your desired command line arguments.

For example, the following setup passes `lol hi there` as command line arguments to the program when compiled and run within CodeBlocks. (I don't think you need to check off the box next to "*This target provides*...." CodeBlocks seems to take care of that automatically after you hit "OK.")



(Of course, you will want to provide a more reasonable command line argument than `lol hi there`, such as `input01.txt`.)

If you compile your program like this in CodeBlocks and later run your executable from the command line, you will still need to supply fresh arguments at the command line.

At any rate, don't forget to test your code on Eustis before submitting, even if you do most of your development in CodeBlocks. As always, the safest way to test your program is by using the test-all.sh script on Eustis.

3. Input Files

One of the required functions for this program needs to open and process an input file. 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 4, “Function Requirements” (page 6).

The fun continues on the following page!

4. 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 3, "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 3, "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 3 ("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 head 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: An estimate (greater than zero) of the number of hours you spent on this assignment.

5. Test Cases and the test-all.sh Script

We've included multiple test cases with this assignment to show 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. The test-all.sh script is the safest, most sure-fire way to test your code before submitting your assignment. You can run the script on Eustis by placing it in a directory with `ListyString.c`, `ListyString.h`, and all the test case files, and typing:

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

Please note that these test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

6. Compilation and Testing (Linux/Mac Command Line)

To compile at the command line:

```
gcc ListyString.c
```

By default, this will produce an executable file called `a.out`, which you can run by typing, e.g.:

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

If you want to name the executable file something else, use:

```
gcc ListyString.c -o ListyString.exe
```

...and then run the program using, e.g.:

```
./ListyString.exe input01.txt
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./ListyString.exe input01.txt > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
3c3
< riddlee
---
> riddle
seansz@eustis:~$ _
```

7. Deliverables

Submit a single source file, named `ListyString.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "ListyString.h"` in your source code.

Your program must work with the test-all.sh script, and it must be able to compile and run like so:

```
gcc ListyString.c

./a.out input01.txt
```

Be sure to include your name and NID as a comment at the top of your source file.

8. Special Restrictions

You must use linked lists to receive credit for this assignment. Also, please do not use global variables in this program. Doing so may result in a huge loss of points.

9. Grading

The *expected* scoring breakdown for this programming assignment is:

20%	correct output for standard test cases (using input files)
60%	unit testing (see details below)
10%	implementation details (manual inspection of your code)
10%	adequate comments and whitespace; source includes student name and NID

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 primarily on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization, punctuation, or whitespace 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 thoroughly.

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!