CSc 352 (Fall 2018): Assignment 5

Due Date: 11:59PM Sun, Oct 14

The purpose of this assignment is to continue to work with string, linked lists, and memory allocation.

General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza. This includes protecting against buffer overflows whenever you read strings.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

Execution	Exit Status
Normal, no problems	**0**
Error or problem encountered	**1**

3. Under *bash* you can check the exit status of a command or program ***cmd*** by typing the command "**echo $?**" immediately after the execution of ***cmd***. A program can exit with status *n* by executing "`exit(n)`" anywhere in the program, or by having `main()` execute the statement "`return(n)`".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the `diff` command to compare your output to that of the example executable.
5. To get full points your code should compile without warnings or errors when the `-Wall` flag is set in `gcc`
6. Anytime you input a string you must protect against a buffer overflow. Review slides 82 – 87 of the basic_C deck.
7. You must check the return values to system calls that might fail due to not being able to allocate memory. (e.g. Check that malloc/calloc don't return NULL) getline() is an exception to this rule.
8. **A new grading criterion this assignment will be that your code must run without errors using valgrind. NOTE you do not need to free memory. Valgrind might report things called memory leaks. Don't worry about this. What you need is to see the line that includes: "ERROR SUMMARY: 0 errors"**

Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

`/home/cs352/fall18`

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assg#/prob#`** subdirectory of this directory. They will have the same name as the assigned program with "ex" added to the start and the capitalization changed to maintain camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don't terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

Makefiles

You will be required to include a Makefile with each program. Running the command:

`make` ***`progName`***

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc command in your Makefile must include the **`-Wall`** flag. Other than that, the command may have any flags you desire.

Submission Instructions

Your solutions are to be turned in on the host **lectura.cs.arizona.edu**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **assg#**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **prob#** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions. For this assignment the directory structure should look like:

```
assg5
   prob1
      median2.c
      Makefile
   prob2
      wordCount.c
      Makefile
```

To submit your solutions, go to the directory containing your assg5 directory and use the following command:

turnin cs352f18-assg5 assg5

Problems

prob1: median2

Write a C program in a file called **median2.c** and a Makefile which creates the executable **median2** that calculates the median of a list of numbers as described below:

- **Input:**

  The input will consist of a nonempty sequence of integers A_1 A_2 ... A_N ($N > 0$) which are read from **stdin**. The integers are separated by whitespace and you don't know how many integers are going to be input.

- **Program Behavior:**

  Your program should read in the input values, compute its median, and print this to **stdout**. The procedure for computing medians is described here.

- **Output:**

  Since the median value may not be an integer, print out the median as a floating point value using the following statement:

  ```
  printf("%.1f\n", median)
  ```

  Note the ".1" which tells printf to print a single decimal place.

- **Restrictions:**

  The point of this problem is to get experience using `malloc()` or `calloc()` to create `linked lists`. Therefore you may NOT use an array where you allocate some large amount space for and for which you perhaps reallocate more space as necessary. Instead you must use a linked list to hold exactly as many integers as are input.

- **Errors:**

  If no integers are input this is an error. In this case your program should print an error message to **stderr** and exit with a status of 1 without printing anything to **stdout**. If at least one integer is input followed by something is input that can't be interpreted as an integer, your program should print an error message to **stderr** and then calculate and print the median of the integers that were input. Note your program's exit status should still reflect that an error was seen.

- **Hints:**

  This is like the median problem from A4 except that now we don't know how many integers are going to be input. You may reuse your code from that program, altering it to use the new data structure. Please remember the new name of this program is **median2** If you leave off the "2" the grading script won't find your program.

- **Makefile**:

  In addition to your source files, you should submit a make file named **Makefile** that supports at least the following functionality:

  make median2
  Compiles the C source code to create an executable named `median2`. The compiler options used should include `-Wall`.

prob2: wordCount

Write a C program **wordCount**, to print a frequency count of all the words coming from input from stdin. The list of words and their counts will be listed in alphabetical (lexigraphical) order as specified below:

- **Input:**

  A sequence of "words" from **stdin**.

- **Definition of words**

  Words are strings that are separated by white space. Therefore, you can (and should) use scanf to read in each string as a word. However, we must do some further processing. Imagine the input includes the line: "Cats rained down on other cats inlucding the purple cats." Notice that the following *words* appear in the input "Cats", "cats", and "cats." We would like to count them all as the same word, "cats". To do this we will process every string separated by white space as follows:

  1. Convert all the uppercase letters to lowercase.
  2. Strip all the nonalphabetic symbols from the front of the string.
  3. Strip all the nonalphabetic symbols from the end of the string.

  Following this process "cats!" becomes "cats", "**WARNING**" becomes "warning", "Planet9" becomes "planet". This is not a perfect solution. For example, "you're" stays "you're", but we don't want the program to be too complex. Notice that this procedure can leave you with an empty string if the string has no alphabetic characters. For example, "99" and "&" both become "". Your program should ignore such strings. They are NOT considered to be errors in the input, nor should they be included in the word count.

- **Output:**

  After all the input has been read, your program should print out the list of words together with the number of times each appeared. Each word and count should be on its own line. Use the following print statement:

  ```
  printf("%s %d\n", word, num)
  ```

  where **`word`** is the word and **`num`** is the number of times it appeared in the input. These words should be printed in alphabetical (lexigraphical) order.

For example, the input below:

I am the eggman. You are the eggman. I am the Walrus.
contains 12 words and its output should be:
am 2
are 1
eggman 2
i 2
the 3
walrus 1
you 1

- **Assumptions:**

  You can assume that each of the words read from the input is at most 128 characters.

- **Error Conditions:**

  none

- **Makefile:**

  In addition to your source files, you should submit a make file named **Makefile** that supports at least the following functionality:

  make wordCount
  Compiles the C source code to create an executable named **`wordCount`**. The compiler options used should include **`-Wall`**.

- **Restrictions:**

  The purpose of this assignment is to get you used to using structs and pointers. You may therefore NOT allocate a bunch of memory for a big array and then resize it as necessary. Instead you should create something like a linked list that uses exactly the amount of memory needed to keep track of the data. That said, you do have freedom in implementation. You may choose to store all the words and then calculate the counts and order, or you may choose to keep track of those things as you process the input.

- **Example:** Suppose the input consists of the following lines:

```
It was the best of times,
it was the worst of times,
```

Then the output should be

```
best 1
it 2
of 2
the 2
times 2
was 2
worst 1
```