CSc 352 (Fall 2018): Assignment 4

Due Date: 11:59PM Saturday, Oct 6

The purpose of this assignment is get experience using the functions malloc or calloc and getline. It also will give you further practice with pointers and parsing strings.

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.
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. For this assignment what that means is you should never have a statement like `scanf("%s", str)` but instead use something like `scanf("%16s", str)` where the number (in this case 16) is one less than the size of the char array `str`. (In this case `str` would be declared by: `char str[17];`)

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:

```
assg4
   prob1
      sumLine.c
      Makefile
   prob2
      median.c
      Makefile
   prob3
      shuffle.c
      Makefile
```

To submit your solutions, go to the directory containing your assg4directory and use the following command:

turnin cs352f18-assg4 assg4

Problems

prob1 sumLine

Write a C program, in a file **sumLine.c**, and a Makefile that creates an executable called **sumLine** which reads in input a line at a time, parses out the integers, and prints out the sum as specified below.

- **Input:**

  A sequence of lines to be read in from `stdin`, each line consisting of a non-empty sequence of non-negative decimal integers, with adjacent numbers separated by whitespace. (Thus, leading negative signs are not allowed.)

- **Output:**

  For each line read in, the sum of the numbers in that line, printed to `stdout` using the following statement:

  ```
  printf("%d\n", sum)
  ```

  where `sum` is the number being printed out.

- **Error Conditions:**

  Non-positive integer or non-whitespace values in the input and empty lines (lines containing only whitespace) are both errors. In each case, print an error message to `stderr`, skip the culprit line (i.e., don't print any value for it), and continue processing. Use the exit status of your program to indicate whether any errors were encountered during processing.

- **Suggested Approach:**

  Reading in the input numbers using `scanf` won't work. Instead, read in each line using `getline()`. Iterate over the line thus read in using `sscanf()` to read the numbers one after another. (Read the man page on sscanf)

  Note that in order for this to work, once a number is read from the string, the string that is passed into `sscanf()` the next time should be the "rest of the string" so that you don't read the same number over and over again. You can do this by using a pointer to keep track of where you are in the string and moving this pointer past each number that is read.

- **Restrictions:**

  One of the goals of this assignment is to teach you to move a pointer through a string

while parsing. Therefore, you are NOT allowed to use a library function like strtok() to process the input. See the functionsList document on Piazza for which functions are allowed.

- **Assumptions:**

  You may assume no integer on the input line is to long to fit into a variable of type int. Likewise, you may assume the sum is not larger than an int.

- **Makefile:**

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

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

  Note: the 'L' in **sumLine** is uppercase. All the other letters are lowercase.

- **Example:**

  Suppose the input consists of the following lines:

```
12  128   23   97
   34   23   19835   257   87
176
982   83
```

  Then the output should be

```
260
20236
176
1065
```

If the input was coming from the keyboard, the output would be printed after each line was read in. In other words, you should get a line, process it, print the output, and then get the next line. This problem is to introduce you to using **getline** and processing a string it returns. You do not need to save more than one line at a time in memory.

prob2: median

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

- **Input:**

  The input will consist of a nonempty sequence of integers N A_1 A_2 ... A_N ($N > 0$). Note the first input is a positive integer giving the number of integers to follow.

- **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()`. Therefore you MUST use one of these functions to allocate the memory needed for the array of integers to receive credit. The first input will tell you how much memory you need to allocate. Once again you MUST use malloc() or calloc().

- **Errors:**

  The following are error conditions:
  - The first input is not a positive integer.
  - N (the first value input) integers cannot be read after the first input. This can happen because EOF is reached or because the input contains a non-numeric character.

  If an error is detected, your program should give an appropriate error message and exit with status **1**. Note that extra input is not considered an error and can be ignored. (In other words, use scanf() to read the N integers A_1 … A_N)

- **Hints:**

  Casting in C works the same as in Java. If you do arithmetic on two integers, the result will be an integer. If you cast one to a float, the other will be upconverted to a float and the result will be a floating point value.

- **Makefile**:

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

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

prob3: shuffle

Background and Definitions

This problem simulates a technique, known as riffle shuffle, for mixing up the cards in a card deck. For this problem, we define a shuffle as follows.

Definition: Given two sequences

A_1 A_2 A_3 ...
B_1 B_2 B_3 ...
we take the first element of the first sequence, then the first element of the second sequence, then the second element of the first sequence, then the second element of the second sequence, and so on, picking alternately from the two sequences:
A_1 B_1 A_2 B_2 A_3 B_3 ...
If one of the sequences runs out of elements, we continue picking from the other sequence. Thus, if the two input sequences ars
(Seq1) `12 3 19 212 7`
(Seq2) `712 93`
then the shuffle of these sequences is
`12 712 3 93 19 212 7`
As a degenerate case, if one sequence is empty, the shuffle is just the other sequence. If both sequences are empty, then their shuffle is also empty.

The Problem

Write a C program in a file **`shuffle.c`** and a **`Makefile`** that creates an executable called **`shuffle`** that behaves as described below.

- **Input:**

  The input will come from **stdin** and consist of zero or more lines, read from **`stdin`**, where each line consists of a sequence of integers separated by whitespace:

  A_1 A_2 ... A_n
  B_1 B_2 ... B_m

  where $m, n \geq 0$. The two lists of integers need not be of the same length, i.e., m and n may be different in the lists A and B shown above.

  It is OK for the input to contain more than two lines. Additional lines, if any, should simply be ignored. There is no assumed maximum length for the lines of input and your program should work for any size.

- **Program Behavior:**
    - If the input contains at least two lines: your program should print out the shuffle of the input sequences on the first two lines, starting with the first sequence.
    - If the input contains only one line: the output should be just that single line.
    - If the input contains 0 lines (i.e., is empty), the output should be empty (i.e., nothing should be printed out).
- **Output:**

  Print the shuffle to **stdout**, one value per line, using the statement

  ```
  printf("%d\n", val)
  ```
- **Error conditions:**

  It is an error for the first two lines of input to contain anything other than integers separated by whitespace.
- **Hints:**

  There is a lot of similarity between this program and sumLine so you might want to reuse some of your code. One large difference is that this program allows negative numbers so your check for a valid line will be more complex here. Remember input is illegal if it contains something that is not an integer and all integers must be separated by white space so, for example the line:

  ```
  342-234 234
  ```

  is illegal because there is no space between 342 and -234.
- **Makefile:**

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

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