CSc 352 (Fall 2023): Assignment 4

Due Date: 11:59PM Saturday, Sep 29

The purpose of this assignment is to continue to work with strings and arrays, get some practice working with char variables and ASCII encodings, and to continue working with dynamic memory allocation via **`malloc()`** and/or **`calloc()`**. One of the problems requires you to process the input a line at a time. Note that you cannot use **`scanf()`** to read in the input in this case (why not?). Instead, you can use the function **`getline()`**.

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 70 – 75 of the basic C deck.
7. Any time you call a system function like malloc() or calloc() that might fail, you must check the return value to ensure that it succeeded.

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/fall23`

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.

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
      count2.c
   prob2
      strmath.c
```

To submit your solutions, go to the directory containing your assg4 directory and use the following command:
turnin cs352f23-assg4 assg4

Problems

prob1: count2

Write a C program, in a file **count2.c**, that reads in a sequence of integers and counts the number of times each integer value appears in the sequence, as specified below. (This is almost exactly the same as the **count** problem from the previous assignment, the only difference being that you are not told, ahead of time, how many integers will appear on the input.)

- **Input:** The input will consist of a sequence of integers read from `stdin`:

  A_1 A_2 ... A_N

- **Output:** Your program should print out the number of times each distinct integer value appears in the input sequence. These should be printed out in ascending order of the integer value using the statement

  ```
  printf("%d %d\n", input_value, count);
  ```

  where `input_value` is a number occurring in the input and `count` is the number of times that value occurs in the input sequence.

  Each distinct value in the input sequence should be printed out exactly once along with the count of the number of times it occurs.

- **Restrictions:** You are not allowed to just create a big array and resize it when needed. Instead **you must store the input in a linked list**. The memory for this list should be dynamically allocated as the input is read in. Solutions that don't follow this restriction will receive a 0.

- **Error Cases:** A non-integer value in the input stream should cause the program to print an error message to stderr and exit immediately with an exit code of 1. There will be nothing written to stdout in this case.

- **Example:** Input:

  ```
  12   31   -5   31   12   7   12
  ```

  Output:

  ```
  -5 1
  7 1
  12 3
  31 2
  ```

The input is only shown on one line for brevity. The input may actually have any amount of white space (including newlines) separating the numbers.

prob2: strmath

Numerical data types in C are typically no more than 32 or 64 bits wide, and so cannot accommodate integer values that are too big to fit into that many bits. This problem explores one way to get around this by doing arithmetic explicitly on a string representation of numbers.

Write a C program, in a file **strmath.c**, that behaves as follows:

- **Input:** Three strings, one per line, read from **stdin**:
  - The first string, *op*, is one of **`add`** or **`sub`**.
  - The second and third strings, *str1* and *str2* respectively, are sequences of decimal digits.

  There is no limit on how long *str1* and *str2* may be (beyond physical limits resulting from the amount of memory on your computer). They may also contain (arbitrarily many) leading zeros.

- **Output:** The result of the arithmetic operation indicated, printed out on **stdout** using the statement

  ```
  printf("%s\n", str)
  ```

  where **`str`** is the result of the operation.

  Note: The string output by your program should not have any extraneous leading zeros or whitespace. For example:

  OK: Examples of allowable output numbers
  ```
  1234
  -73
  0
  ```
  Not OK: Not allowable output numbers
  ```
  000
  01234
  0007
  ```

- **Behavior:** Your program should perform the operation indicated by the input string *op* on the strings *str1* and *str2*, treating their rightmost digits to be the least significant digit:
  - If *op* is **`add`**, compute the sum of the second and third strings, i.e., *str1 + str2*.
  - If the operation is **`sub`** compute the difference between the second and third strings, i.e., *str1 – str2*.

- **Error Conditions:** Not enough input strings. A line containing only whitespace. First string is not one of "**`add`**", "**`sub`**". Second and third strings contain non-numeric characters.

  In case of an error, give an appropriate error message and exit with the appropriate error code.

- **Comments:**
  - Since the input strings *str1* and *str2* can only contain decimal digits, they can only denote non-negative values.
  - Note that there is no *a priori* bound on the length of the second and third input strings. Because of this, you cannot allocate memory for them ahead of time. Instead, use **`getline()`** to read them in (watch out for the newline character **`'\n'`** at the end).
  - Use the size of the input strings to determine how much memory you need to allocate for the result. You don't need to be exact, but you don't want to be wasteful either. For example, if str1 = "58" and str2 = "76", and the operation is add, then allocating 10 bytes for the result is too much, since clearly in this case you only need 4 (the result is "134" and remember you also need a space for the '\0'). However, you don't need to do the calculation before allocating the memory. In other words if str1 = "22" and str2 = "34" then the result only needs 3 bytes, but you can still allocate 4 since it is possible for two strings of length 2 when added might need 4 bytes for storage. Ask yourself, given that str1 is lenth n1 and str2 is length n2 and the operation is add, what is the maximum possible length of the result? (and similarly for sub)
- **Restrictions:** Space for the result string must be dynamically allocated via **`malloc`** or **`calloc`**.
- **Examples:**
  - *Input:*

    ```
    add
    11112222333344445555666677778888999 9
    22223333444455556666777788899990000
    ```

    Output:

    ```
    333355557778000022224444666688879999
    ```

- *Input:*

  ```
  sub
  10
  89
  ```

  Output:

  ```
  -79
  ```

- *Input:*

  ```
  sub
  12345678901234567890123456789012345678901234567890123456789012345
  67890123456789012345678901234567890
  23456789012345678901234567890123456789012345678901234567890123456
  78901234567890123456789012345678901
  ```

  Output:

  ```
  -11111110111111111011111111101111111110111111111011111111101111111
  11101111111110111111111011111111011
  ```

- *Input:*

  ```
  add 12 34
  ```

  Output: (sent to stderr, not stdout)

  ```
  Error: 1st line not equal to 'add' or
  ```