CSc 352 (Fall 2018): Assignment 3

Due Date: 11:59PM Saturday, Sep 22

The purpose of this assignment is to do more work with strings and arrays, and perhaps experiment with using pointers. You should use **scanf** to read in the strings. We've seen examples in class on how to do that.

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:

```
assg2
   prob1
      splitString.c
      Makefile
   prob2
      cipher.c
      Makefile
   prob3
      vowels.c
      Makefile
```

To submit your solutions, go to the directory containing your assg3 directory and use the following command:

turnin   cs352f18-assg3   assg3

Problems

prob1 splitString

This problem is designed to give you experience manipulating a string. I would encourage you to try to use pointers, but you can use indexes into an array if you prefer. Because of what I want you to learn with this program, there are several restrictions on what you can do. Please take note of the **Restrictions** section. **If you violate these restrictions, you will not receive credit for this program.**

You will write a program in a source file called splitString.c and a Makefile which creates an executable called splitString that reads in a sequence of strings using scanf() and looks for the dash character, '-', which it uses as a separator. It then prints the portions of the string separated by dashes on separate lines. For example, if the input is:

```
This-is-an_example-string
```

then the program would output:

```
This
is
an_example
string
```

Your program should use scanf() to read the strings until EOF is reached. (i.e. until no more strings may be read in)

Error conditions: There are no error conditions for this program.

Special cases: Dashes at the start and end of the string should be removed. Likewise, multiple consecutive dashes should be treated as a single dash. Thus if the string input is:

```
--What-----is_it?-
```

then the input should be:

```
What
is_it?
```

with no blank lines.

Assumptions: You may assume no string is longer than 255 characters.

Restrictions:

1. I want you to get practice manipulating strings on your own, **so for this program only you may NOT use any functions from the string.h library. Do not include this library in this program.**

2. Let's pretend you're space limited so your program may only include one char array of size 256. Your program must NOT declare any other arrays. Thus you must accomplish your task by manipulating the array that contains the input. For a hint of how to do this, review question 3 on quiz 2.

prob2: cipher

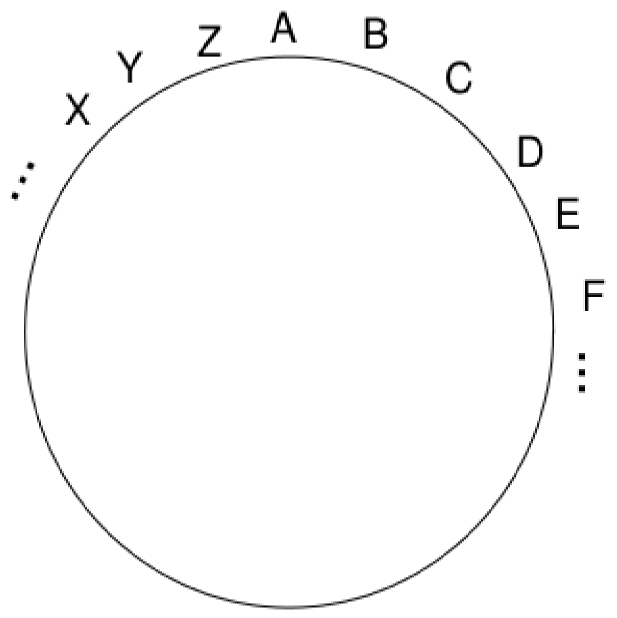
Write a C program, in a file **cipher.c**, and a Makefile that creates an executable called **cipher** that implements a very simple substitution cipher, a described below, on strings it reads in. (This program can be seen as a generalization of the rot-13 cipher in that that it admits rotation by amounts other than just 13.)

- **Program behavior:** Your program will read in, from **stdin**, a decimal integer *N* (which may be positive, zero, or negative) followed by zero or more alphanumeric strings (see the C library function `isalnum()`). Each alphanumeric string will be rotated by the amount *N* and the result ***`S`*** printed to **stdout** using the statement

  ```
  printf("%s\n", S);
  ```

  Rotations should be done as follows:

  - Only letters are rotated; digits remain unchanged.
  - Rotation preserves case: rotating an upper-case letter produces an upper-case letter, and similarly with lower-case letters.
  - The result of rotating a letter can be specified by arranging all the upper-case letters in a clockwise circle, as shown here, and analogously for lower-case letters:

    A positive value of *N* means that letters are rotated *clockwise* by *N* positions in this circle, e.g., if *N* = 3 then **`A`** becomes **`D`**, **`c`** becomes **`f`**, **`Y`** becomes **`B`**, etc. A negative value of *N* means that letters are rotated *counter-clockwise* by *N* positions in this circle, e.g., if *N* = –2 then **`A`** becomes **`Y`**, **`c`** becomes **`a`**, **`Y`** becomes **`W`**, etc. If *N* = 0, no rotation takes place.

Notice that the magnitude of *N* may be larger than 26. In this case, you just "go around the circle" as many times as necessary. Thus, the behavior for *N* = 30 is the same as that for *N* = 4.

- **Input format:** The first item read from **stdin** is an integer value (the value may be positive, zero, or negative). The upper and lower bounds on the possible values of this value are determined by the largest and smallest values that can be taken on by a **signed int**.

  This is followed by a sequence of zero or more alphanumeric strings. You may assume that each such string is at most 64 characters long.

- **Error conditions:** It is an error the first input to the program is not a number. In this case, your program should give an error message and exit with exit code **`1`**.

  It is an error for any of the remaining input strings to contain any non-alphanumeric characters. In this case, your program should give an appropriate error message, ignore the offending string (and so not produce any output), and continue processing any remaining input. If such an error occurs, the exit code for the program when it eventually exits should be **`1`**.

- **Example:**

  Suppose that the input is

  ```
  2   abC42   8xZ5e   w
  ```

  The output should be

  ```
  cdE42
  8zB5g
  y
  ```

  Suppose that the input is

  ```
  -2   abC42   8xZ5e   w
  ```

  The output should be

  ```
  yzA42
  8vX5c
  u
  ```

prob3: vowels

Write a program, in a file **vowels.c** and a Makefile that creates an executable **vowels** that behaves as specified below.

- **Definitions:**
    - For the purposes of this assignment, a *word* is any sequence of upper- and lower-case letters. For example, **uncle** and **clever** are words, but **abc$!^def** and **cl3v3r** are not. This definition implies that a word can be read in using `scanf("%s" ...)`, but requires additional checking to ensure that it is in fact a valid word.
    - Vowels are the characters: a, A, e, E, i, I, o, O, u, U
    - The vowels in a word are said to occur *in order* in a word *w* if and only if the following holds: for any two vowels v_1 and v_2 occurring in *w*, if v_1 comes before v_2 in the English alphabet, then every position in which v_1 occurs in *w* comes before every position in which v_2 occurs.
- **Behavior:** Your program should repeatedly read in words from the input until no more words can be read in. For each word read in, your program should determine whether or not the vowels that occur in that word occur in order. Note that this does not require all of the vowels to occur in a word; however, those vowels that *do* occur in the word should occur in order. Your program should print out the value **1** on **stdout** if the word contains the vowels in order, and **0** if not, as described under "**Output**" below.

  You should not distinguish between upper and lower case letters for this problem. Thus, the word **abstEmiOus** should be considered to have the vowels in order. The simplest way to deal with this is to convert all of the letters in the input word to the same case, either upper or lower, before processing it further: see the library functions `toupper()` and `tolower()`.

  If an input string is not a word according to the definition above, your program should give an appropriate error message, discard that string, and continue processing.

  Your program's exit status should be **0** if no errors were encountered during processing, **1** if any error was encountered.
- **Output:** For each word, the value `val` indicating whether or not it contains vowels in order should be printed using the statement

  ```
  printf("%d\n", val)
  ```
- **Assumptions:** You may assume that each input word is at most 64 characters long.

- **Error cases:** Input string contains one or more non-letter characters.

- **Example**: Given the input

  ```
  a
  antecEdent
  Cookiemonster
  bailout
  haha
  ultra-orthodox
  ```

  The output generated should be

  ```
  1
  1
  0
  1
  1
  error
  ```