Lab 8: Strings...

Due date: Friday, November 21, 11:59pm.

Lab Assignment

Assignment Preparation

Lab type. This is a **pair programming lab**. You will work with your partner from Lab 5. If your Lab 5 partner is unavailable, consult the instructor for a new partner assignment. You are required to perform all tasks using pair programming (i.e., with both partners at the computer). Absolutely no code sharing between different pairs is allowed.

Purpose. The lab allows you to manipulate strings, both as arrays of `chars` and using functions from the `strings.h` library.

Programming Style. All submitted C programs must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties. Also note the the Lab 2 requirement for the content of the header comment in each file you submit applies **to each assignment** (lab, programming assignment, homework) in this course.

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

Test-driven development. The assignment in this lab will involve a new software development process called test driven development. For this assignment you **will not** be provided with any tests (except the examples in this document) from the instructor. Your goal will be to develop all necessary test cases to test your program. In addition, your work will be organized around **first developing the tests** and then writing code. More instructions are given below.

Test-Driven Development

Test-Driven Development (TDD) is an emerging software development paradigm that, when properly used, may significantly simplify and speed up software development efforts. The key idea behind TDD is that test cases are developed alongside the software, and, in fact, they are developed **in advance of the software development**.

The test-driven development of software works as follows:

Step 1. A small number of test cases is generated.

Step 2. Code is written to make these test cases pass.

Step 3. Go to **Step 1.**

Informally, test-driven development proceeds in the following manner. You start with no code (or a stub of a `main()` function). First thing you do is develop a simple test case. Your (stub) code fails the test. You improve your code to pass the one test you have. Then you create a new test case. If your code succeeds on it, you create another. If your code fails, you study which part of the software spec that has not been developed is responsible for failing the test case, and you add more code, to implement that part of the software spec until all your current test cases pass. You keep on adding test cases for as long as you believe that there are untested portions of the software spec. Any time your code fails a test case, you add new code, or modify existing code.

TDD in the Lab

This lab assignment shall be performed using test-driven development. To make your testing easier, you can include the following `checkTest()` function in your `transform.h` lab:

```
void checkTest(const char in[], const char out[], const char correct[]) {

  printf("Test: %s   Result: %s  Expected: %s ",in,out, correct);
  if (strcmp(out,correct)) {
     printf("FAILED\n");
  }
  else {
     printf("PASSED\n");
  }
}
```

This function takes as input three string parameters: the `in` string is the input string to the function that is being tested; the `out` string is the output string constructed by the function being tested; `correct` string is the expected **correct** output produced by the function.

The use of this function is explained in the following code fragment:

```
char out[100];

invert("foot",out);
checkTest("foot",out,"toof");
```

This code fragment declares a string `out`, runs an function `invert` on the input `"foot"` and then checks to see if the test has succeeded or failed. The correct expected output (contents of the `out` string) for this function call is `"toof"` (see the description of function `invert()` below).

When this code fragment is run using the instructor's version of hte `transform.h` library, the output is

```
Test: foot   Result: toof  Expected: toof PASSED
```

Making test cases independent. There are three ways in which you can make all your tests independent on each other:

1. Use a separate `.c` program for each test. This is inconvenient, because you have to compile and run each test many times during the development process. Shell scripts may be used to alleviate that problem, but they are not part of the course, so there is no expectation that you use them.

2. Use a single `.c` file for all tests (or for all tests for a single function), but use a separate variable for each test case. This solution allows you to use a single program for all test cases (or a single program for all test cases for a single function). But you will have a lot of variables in your program, each of which is used only once.

3. Use a single `.c` file for all test/all tests for a single function **and use a single string variable for all tests**, but **clear the contents of this variable** after each test and `checkTest()`. To clear the contents of a string, the instructor has implemented a function

   ```
   void clear(char str[]);
   ```

   This function, essentially, puts a NULL character (character with decimal code 0) into every element of the string `str` array.

The following is a fragment of one of the instructor's test programs for the lab:

```
char out[MAX_STR];

 clear(out);
 invert("foot",out);
 checkTest("foot",out,"toof");

 clear(out);
```

```
text2slang("before", out);
checkTest("before",out,"be4");
```

Notice, that in this example, we run `clear()` before each test. This is done to ensure consistency of the results.

The implementation of the `clear()` function, should you decide to implement it is left up to you. I recommend implementing it, it will make testing your program easier.

The Task

You will create a library `transform.h` of *string transformation functions*. Each string transformation function in the library will take as input two string parameters: an input string and an output string (and, potentially some other parameters). All string transformation functions are `void`, they produce side effects, but "constructing" the output string based on the provided inputs.

There are two possible ways to work with strings: manipulation of individual characters that the string contains (i.e., treating the string as an array of `chars`), and the use of string manipulation functions from the `strings.h` library. For each function you have to design and implement, we specify explicitly, which of the two approaches is applicable. You must follow that apporach.

General Instructions

All your string variables shall be declared as arrays of sizes less than or equal to than some `#define`d constant. (I use `MAX_STR` in my code). This constant needs to be set to a relatively large number (e.g., 100, or 80) in order to pass instructor's tests upon final submission. (my current implementation uses 40, but this may get bumped up).

Be aware that some of your functions may transform input strings into strings of, possibly significantly larger length. My test cases will never produce strings that exceed `MAX_STR`, and neight should your test cases do that.

Function `void invert(const char in[], char out[])`

The first function you have to create is

```
void invert(const char in[] char[] out)
```

This function takes a string `in` as input and constructs a string `out` which is the same string inverted (i.e., spelled backwards). For example, `invert("train",x)` will result in `x` becoming `"niart"`; `invert("rose",x)` will result in `x` becoming `"esor"`, while `invert("tenet",x)` will result in `x` becoming `"tenet"`.

Implementation note: For this function, treat your strings as arrays of `char` elements. Do not use any `string.h` functionality except for `strlen()` (if needed).

Function text2slang(const char in[], char out[])

The second function is

`void text2slang(const char in[], char out[])`

This function takes as input a string and based on its contents creates an `out` string. The `out` string is an attempted "hacker slang" rewriting of the `in` string. The input characters will include all lowercase letters of the Latin alphabet, space and any puncuation and digits (but no upper-case letters). The rewriting will follow the rules stated below:

1. All letters `'l'` (*"el"*) are replaced with the digit `'1'` (one).
2. All letters `'s'` are replaced with the letter `'z'`.
3. All leters `'o'` (*"ou"*) are replaced with the digit `"0"` (zero).
4. All letters `'u'` are replaced with `"oo"` (double `'o'`).
5. All letters `'x'` are replaced with `"ks"`.
6. All letters `'z'` are replaced with the digit `'3'`.
7. If a letter `'c'` occurs in front of a letter `'r'` it is replaced with the letter `'k'`.
8. The combination of letters `'fore'` is replaced with the digit `'4'`.
9. All other occurrences of a letter `'f'` are replaced with `"ff"`.
10. All occurrences of the letter `'y'` at the end of the input word are replaced with `"ee"`.
11. The combination of letters `"ate"` is replaced with the digit `'8'`.

Examples. `before` is converted to `be4`.
`root` is converted to `r00t`.
`force` is converted to `fforce`.
`lucky` is converted to `1oockee`.
`crimson` is converted to `krimz0n`.
`stuff` is converted to `ztuffff`.

Implementation note: For this function, treat your strings as arrays of `char` elements. Do not use any `string.h` functionality except for `strlen()` (if needed).

Function `void shuffle(const char in[], char out[])`

The third function you have to implement is

```
void shuffle(const char in[], char out[]);
```

The function operates as follows. If the length of the input string `in` is *odd*, the function copies the input string into the output string `out` and completes its work.

If the length of the input string `in` is even, your function shall *shuffle* the contents of `in` in the following manner. The string `in` is cut in half, and the `out` string is formed by taking the first character in the first half of the string, following it by the first character of the second half of the string, following by the second character of the first half, and so on.

For example, if the `in` string is `"aaabbb"` the `out` string becomes `ababab`:

```
aaabbb

aaa
bbb

ababab
```

Similarly, `"abc123"` is trnasformed into `"a1b2c3"`.

Implementation note. For this function, you may use the `string.h` functionality and manipulate the strings you use in the function as arrays.

Function `void computerName(const char in[], char out[])`

The computer network of a small company has two subnets `happy.smallcompany.com` and `sane.smallcompany.com`. The network administrators decided on the following computer naming scheme. Computers on the `happy` subnet have their names start with three letters that are lexicographically less than or equal to `"mov"`. Computers on the `sane` network have names with three-letter prefexes being lexicographically greater than `"mov"`. Moreover, the computer names, for simplicity are abbreviated by the first three characters of the name.

You shall implement the function

```
void computerName(const char in[], char out[]);
```

that, given a word in the `in` string, shall put in the `out` string the name of the computer, which is formed from *the first three characters* of the `in` string, followed by a period (`"."` followed by the full name of the proper subnet.

Examples. `in`: `"stuff"` yields `out`: `"stu.sane.smallcompany.com"`. `in`: `"motor"` yields `out`: `"mot.happy.smallcompany.com"`. `in`:`"movie"` yields `out:` : `"mov.happy.smallcompany.com"` (note, while `"movie"` is lexicographically greater than `"mov"`, the function will only compare the first three letters of the word, not the full word, and `"mov"` is equal to `"mov"`).

Implementation note. For this function you may use only the `string.h` functions. You are not allowed to use any string variables in your code as arrays and assign values to individual array elements.

Function `void geoDecode(const char in[], char out[])`

The www.geocaching.com website uses a very simple text encoding scheme to hide hints from immediate observation. This encoding allows people not interested in hints to ignore them, while people who want them may easily decode the hints, even when they are in the field holding a hard copy of the page with a hint.

You will write the function

```
void geoDecode(const char in[], char out[]);
```

that decodes (or encodes, since the process is reversible) a string.

The encoding mechanism employed by the `geocaching.com` web site is simple. The Latin alphabet is separated into the upper (`'a'`–`m`) and lower (`'n'`— `'z'`) halves. A word is encoded by replacing each character in it with the matching character from the other half of the alphabet. The matches are straightforward: the first character of the upper half matches the first charachter of the lower half, and so on. The encoding/decoding table is shown below.

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

For example, `"the"` is encoded as `"gur"`, `"place"` is encoded as `"cynpr"`, and `"computer"` is encoded as `"pbzchgre"`.

The `geoDecode` function will decode (or encode) the `in` string using the encoding/decoding mechanism described here and will place the decoded/encoded string into the `out` variable.

The string may contain lower-case and upper-case letters - both types need to be properly handled (upper-case `'A'` is encoded as upper-case `'N'` and so on). It may also contain spaces, which shall be ignored (i.e., a space is encoded as a space).

Implementation note. Use strings as arrays. My suggestion is to declare and define a function that may be helpful in writing the code for `geoDecode`, but it is not a requirement to have it.

Submission.

Files to submit. Each pair submits one set of files from one account. The following files are mandatory:

`team.txt`, `transform.h`

In addition to that, **you must submit all the tests you have used**. Submit all `.c` programs you used for testing your program. (I recommend having one `.c` file per function being tested, but this is not a hard requirement, feel free to submit all your tests as a single file).

`team.txt` file shall contain the name of the team and the names of all team members in each pair, and the Cal Poly IDs of each. E.g, if I were on the team with Dr. John Bellardo, my `team.txt` file would be

```
Go, Poly!
John Bellardo, bellardo
Alex Dekhtyar, dekhtyar
```

Submit `team.txt` as soon as you form your group.

Files can be submitted one-by-one, or all-at-once.

Submission procedure. You will be using `handin` program to submit your work. The procedure is as follows:

- `ssh` to `vogon` (`vogon.csc.calpoly.edu`).

- Students from **Section 009** shall execute the following submission command:

```
> handin dekhtyar-grader lab08-09  <your files go here>
```

- Students from **Section 011** shall execute the command:

```
> handin dekhtyar-grader lab08-11 <your files go here>
```

`handin` is set to stop accepting submissions 24 hours after the due time.