String Notes

Strings are an important data type in c. Strings are usually represented using character arrays. We typically make constant strings when we print out values using printf (e.g. “%d\n”). String constants are made by enclosing a series of characters by double quotes (“).

How do we make a string constant that contains double quote?

To make a string constant with a double quote inside we precede the double quote with a backslash (\).

How do we make a string constant that contains a backslash (e.g. the string \”)?

To include a backslash in a string we precede it with a black slash. To make the string “\”” (quotes for clarity), we would use the following string “\\\””.

There are other special characters, that you have hopefully already used (e.g. “\n”).

When reading in strings and characters, sometimes you can accidentally read in white space. To skip over whitespace use ***“% “*** in a scanf.

Common Functions

Additionally there are many functions for manipulating strings or reading out particular information of strings. To access such functions you should include the string library (<string.h>). Functions that are included are strlen(), strcat(), strcpy(), and strcmp().

Strings in c are typically denoted with the null terminator character (‘\0’). This is why if you have a string of n characters, you need n + 1 character length array to store the string, the last, extra spot in the array is for the terminating character. Without the terminator when you print a string, you may begin looking into parts of your memory that you haven’t initialized (or worse parts that have not been allocated). To find the position of the null terminator you can use the function **strlen()**, which takes in a char array, and returns the first location a null terminator is found. If you over write your null terminator with something, you could see very interesting output when print your string (try it as an exercise).

If you wish to merge to strings together (e.g. “lazy” and “dog”). We can use the function **strcat()**. It takes two strings and “adds” the contents of the second string to the end of the first. However, no extra characters are added, and you may end up writing outside the memory of your first string. ***NOTE strcat() does not extend the memory of the first string***. Be cautious when using strcat(). There are additional, unmentioned functions that exist in string.h that can help with this type of issue.

If you wish to make an exact replica of a second string use the function **strcpy()**. It copies the contents of second string (up to and including the first null terminator) into the first string.

The last function **strcmp()** returns an integer depending on the relationship between two strings passed as parameters. A 0 is returned, if the two given strings are equivalent. If the first passed string is lexicographically less than the second passed string, while a negative value is returned if the second passed string is lexicographically less than the first passed string.

Exercises

Write a method that reads in a list of names, sorts them, and then checks for membership of a second list of names. Use a binary search to speed up the querries, and use an O(n log (n)) sort to quickly sort the original list.

Input	**Output**

`5`
`James`
`John`
`Robert`
`Mary`
`Michael`
`3`
`Helen`
`James`
`Susan`
	`Helen is not in the list`
	`James is in the list`
	`Susan is not in the list`

`1`
`Mark`
`2`
`Mark`
`Mark`
	`Mark is in the list`
	`Mark is in the list`

`8`
`Linda`
`Mary`
`Robert`
`William`
`Linda`
`Barbara`
`Elizabeth`
`Lisa`
`5`
`Joseph`
`Thomas`
`Daniel`
`Jennifer`
`Nancy`
	`Joseph is not on the list`
	`Thomas is not on the list`
	`Daniel is not on the list`
	`Jennifer is not on the list`
	`Nancy is not on the list`

Write a program that takes in a string (of uppercase characters), a number of transformations, **n**, and a number of rules, **r**, and transforms the string a given number of times. The first line of input contains the original string, the number of transformations and the number of rules respectively. Each transformation will turn each uppercase character into a different non-empty string of uppercase letters. Each transformation will be defined on its own line. The transformation will consist of a single character followed by a non-empty string of upper case characters (separated by a single space). For each transformation each letter of the string will become the given string of characters defined by the corresponding transformation rule.

Input	**Output**
`ABBA 1 3`	`BCCB`
`A B`	
`B C`	
`C C`	
`ABC 5 6`	`FABBC`
`A B`	
`B C`	
`C D`	
`D E`	
`E F`	
`F AB`	
`A 4 2`	`ABAABABA`
`A AB`	
`B A`	