Homework 3, Fall 2017

OBJECTIVE

The objective of this assignment is to give you more practice with using functions, strings, pointers, and dynamic data.

ASSIGNMENT SUBMISSION

To get credit for this assignment, you must

- ✓ write a program in C using functions and dynamic memory allocations (80%)
- ✓ provide function interface comments as instructed below (10%)
- ✓ write a test plan (10%)
- ✓ submit your files through Canvas exactly as instructed (naming, compatibility, etc.)
- ✓ submit your assignment on time

PROBLEM STATEMENT

For this project, you are to write a string calculator that supports addition and multiplication of two strings using John Napier's location arithmetic. The program will take two strings and an operator as input and produce a resulting string as output.

https://en.wikipedia.org/wiki/Location_arithmetic

John Napier used letters to denote specific powers of 2, where:
a = 2^0, b = 2^1, c = 2^2, d = 2^3, …, z = 2^{25}
We will extend this notation to include uppercase letters where A = 2^{26} and Z = 2^{51}
In Napier's notation, the order of letters does not matter, and digits can be repeated, e.g.

abc = cba = bca
abbc = acc = ad

and the value of a number is simply the sum of its digits, e.g.

abdgkl = $2^0 + 2^1 + 2^3 + 2^6 + 2^{10} + 2^{11}$ = 1 + 2 + 8 + 64 + 1024 + 2048 = 3147

The user will enter the strings and an operator as command line arguments and the input will be of the form `S o S`, where `S` represents a string containing letters, and o represents an operator. Spaces are used as delimiters in user's input. Your program is to make sure that the valid number of arguments is entered and that the string and operator contents are valid as well. For strings representing numbers, a legal character is an uppercase or lowercase letter. For operators, a legal character is either + or x. When an invalid input occurs, the program is to echo print the input and report an error, including the type of an error that occurred (e.g. invalid input – operator missing).

If grammatically correct input is entered, the program is to calculate the string result that uses Napier's letters (NOT actual decimal values), as described below.

Addition

An addition using Napier's notation simply means a concatenation of two strings and then the addition of all the letters (powers of 2), with the reduction of multiple letter occurrences, e.g. instead of *aac*, your program should generate a string containing the shorter version of the same number, namely *bc*.
In order to do this efficiently, your program is to dynamically allocate an array of 52 counters (1 array element constitutes one letter counter) and the letter counts for both strings are to be accumulated into that array, e.g. if one string contains *aaab* and the other contains *zbda*, then the array of counters should look as follows:

4	2	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
a	b	c	d	e	f	…..	…..	…..	…..	…..	z	…..	…..	…..	…..	…..	Z

Once you accumulate all the letter occurrences from both strings, you should adjust the counter array values so that the result does not contain repeated characters other than uppercase Z. For the case described above, the normalized values should be:

0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
a	b	c	d	e	f	…..	…..	…..	…..	…..	z	…..	…..	…..	…..	…..	Z

Based on the array of counters, your program is to create a dynamically allocated string that contains the result of the addition operation, of the length exactly as long as needed. For the case described above the new string should be of size 2 (plus one char for null) and contain the value *ez*

<u>Multiplication</u>
A multiplication using Napier's notation means that every character one string has to be multiplied by all the characters in the second string resulting in a concatenated string that subsequently is processed in the same fashion as in the addition algorithm.
For multiplication, first dynamically create a string big enough to contain the resulting concatenated string. The result will contain the letters of the first string (multiplicand) multiplied by the first letter of the second string (multiplier), concatenated with the multiplicand string multiplied by the second letter of the multiplier, and so on.
For example, if the two original values are *bcdfgh* and *acd*, then the resulting string should contain *bcdfghdefhijefgijk* since
a x bcdfgh = bcdfgh
c x bcdfgh = defhij
d x bcdfgh = efgijk

Once you have the result of the multiplication as a concatenated string, use the same logic (and dynamic allocations) as in the addition algorithm to normalize the values and create the normalized final string, e.g. the original *bcddeefffgghhiijjk = bcekl*

These are some sample runs of the program:

```
---
./a.out abc + d
abc + d => abcd
---
./a.out acd x bcdfgh
acd x bcdfgh => bcekl
---
./a.out aaab + zbda
aaab + zbda => ez
---
./a.out abc +
invalid number of arguments
---
./a.out abc / d
invalid operator
---
./a.out 123.45 + abc
invalid operand
---
```

Program Specs

- Your program is to be contained in a single c file named `pr3.c`
- Your program has to follow basic stylistic features, such as proper indentation (use whitespaces, not tabs), meaningful variable names, etc.
- Your program <u>must</u> use functions and dynamically allocated character and integer arrays

- You are NOT allowed to use global variables
- **Your program must compile in gcc gnu 90 – programs that do NOT compile will receive a grade of 0**
- Run your program through Valgrind to make sure you catch all memory management mistakes
- Your program should include the following comments:
    - Your name at the top
    - Whether you tested your code on the cssgate server or Ubuntu 16.04 LTS Desktop 32-bit
    - Comments explaining your logic
    - Function comments, as specified in the *function comments* section below
    - If your program does not run exactly as shown above, explain at the top how to run your program – you will not receive full credit but at least you will receive some credit rather than none

Command-line arguments
Command-line arguments are discussed in your textbook in chapter 3. However, better examples are provided at:

- http://www.tutorialspoint.com/cprogramming/c_command_line_arguments.htm
- http://www.thegeekstuff.com/2013/01/c-argc-argv/

Note that your program will receive the input as strings.

Function Comments
In case you have not heard of design by contract, the general idea behind design by contract is that the software designer defines precise and verifiable interface specifications for software components. In case of functions, this translates to a function header that specifies the function's purpose, as well as documented preconditions and postconditions for a function. These comments should be written next to function prototypes but in this case, you need to write them by function definitions.

Preconditions focus on arguments/parameters passed to functions – any assumptions that a function does regarding the state of the parameter need to be written as assertions. Postconditions focus on the effect the function holds on the state of computation, so it should describe, as assertions, the value return statement and the state of parameters passed by reference. In addition, write a comment by each parameter to denote a data flow of each parameter: in, out, or in/out. A flow of parameter is *in*, if it is passed by value or as a const reference and a function only uses it for its internal processing. A flow of parameter is *out*, if it is passed by reference and a function only uses it to replace its value. A flow of parameter is *in/out*, if it is passed by reference and a function uses its contents before replacing them with new values.

Some Examples

```
// Prints a header to standard output
// pre:   none
// post: none
void printMessage() {
       puts("some code that prints something");
}

// Prints parameter values to standard output
// pre:   n1 assigned, n2 assigned
// post:  none
void printMessage(/* in */ int n1, /*in*/ int n2) {
       printf("%d %d\n", n1, n2)
}

// Swaps parameter values
// pre:   n1 and n2 assigned
// post:  *n1 == *n2@entry and *n2 == *n1@entry
void swap(/*inout*/ int* n1, /*inout*/ int* n2) {
       int temp = *n1;
       *n1 = *n2;
       *n2 = temp;
}
```

```
// Fills array parameter with data from the user and returns array count of even
// numbers
// pre:   array allocated, size == array length
// post:  array filled with user data, count of even numbers returned
int fillArray( /*out*/ int array[], /*in*/ int size) {
       int i, count = 0;
       printf("enter %d integer values: ", size);
       for (i = 0; i < size; i++) {
              scanf("%d", &array[i]);
              if (!(array[i] % 2))
                     count++;
       }
       return count;
}
```
This last example does not follow good design practices as one should not mix returning via the return statement and returning via function parameters at the same time. If you are only to return one value, use value-return functions. If you need to return multiple items, use pass-by-reference for all of them.

Extra Credit (15%)
Open-ended – extend the program in some interesting fashion. For example, you may look into other Napier's operations such as subtraction or division, you may want to display the resulting strings as strings of binary digits along with their conversions to the decimal system. However, if you decide to compute decimal equivalents, you cannot use any of the primitive types as some of the resulting decimal integers will be larger than the allowed integer limit and you will need to think about representing decimal numbers as strings or integer arrays. Remember that the focus of the assignment is dynamic arrays and strings, so the processing that does NOT include them does NOT qualify for extra credit on this assignment.

Program Submission
On or before the due date, use the link posted in *Canvas* next to *Programming Assignment 3* to submit your test plan and your C code. Make sure you know how to do that before the due date since late assignments will not be accepted. Valid documentation file formats are: pdf, jpg, gif, png. Valid program format: a single file .c. *Keep in mind that the test plan should test **all the corner cases** of your program in addition to the valid inputs.*