Programming Assignment 3, TCSS 333A, Winter 2021

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 that compiles and you wrote on your own (no copying or help outside of CSS mentors or TLC or instructor allowed)
- ✓ submit your files through Canvas exactly as instructed (naming, compatibility, etc.)
- ✓ submit your assignment by the due date

PROBLEM STATEMENT

Create a program that implements a string calculator (addition and multiplication of two strings) using John Napier's location arithmetic according to the set of rules described below. The program will take two strings and an operator provided via command-line arguments and produce and print a resulting string as output.

John Napier's location arithmetic

John Napier (discoverer of logarithms who lived at the turn of XVI/XVII centuries) used letters to denote specific powers of 2, where:

$a = 2^0$, $b = 2^1$, $c = 2^2$, $d = 2^3$, …, $z = 2^{25}$

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 summation of strings/letters in 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*.

A multiplication using Napier's notation means that every character in one string has to be multiplied by all the characters in the second string, resulting in a concatenated string. For example, if the two original values were *bcdfgh* and *acd*, then the resulting string should contain *bcdfghdefhijefgijk* since

a x bcdfgh = bcdfgh

c x bcdfgh = defhij

d x bcdfgh = efgijk

IMPLEMENTATION

Input

The user will enter the strings and an operator as command-line arguments and the input will be of the form

`S ○ S`

where `S` represents a string containing letters, and ○ represents an operator. To stay consistent with command-line argument logic, spaces will used as delimiters in user's input (i.e. blanks between strings and operator). 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 a 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

In our case, although the usage of the binary representation is a tempting proposition, to practice arrays and strings we will use a slightly different approach. In order to produce a concatenated and reduced string, your program is to dynamically allocate an array of 26 counters (1 array element constitutes one letter counter) and the letter counts

for both strings that are being added 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
a	b	c	d	e	f	…..	…..	…..	…..	…..	z

Note that characters are integer types in C and therefore could be used to calculate indexes in a straight-forward way, e.g. 'a' is 97, so an index for any char is its ASCII value - 97. 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 the letter z. For the case described above, the normalized values should be:

0	0	0	0	1	0	0	0	0	0	0	1
a	b	c	d	e	f	…..	…..	…..	…..	…..	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 needed, i.e. exact fit. For the case described above the new string should be of size 2 (plus one char for null) and contain the values *ez*

Multiplication

For multiplication, use the array of counters as well to produce the result, except remember that the logic of what to update is slightly more complex because the first string needs to be multiplied by every char in the second string. However, multiplying a single letter by another single letter is a simple process because "all letters represent powers of 2, and therefore multiplying digits is the same as adding their exponents. This can also be thought of as finding the index of one digit in the alphabet (**a** = 0, **b** = 1, ...) and incrementing the other digit by that amount in terms of the alphabet (**b** + 2 => **d**)." (Wikipedia)

Once you have the result of the multiplication, use the same logic (and dynamic allocations) as in the addition algorithm provided in the prior section to normalize the values and create the normalized final string, e.g. the *bcdfgh* and *acd* will result in *bcddeefffgghhiijjk* which can be simplified to *bcekl*

SAMPLE RUNS

```
---
./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
---
```

ADDITIONAL SPECIFICATIONS

- 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 must use functions and if it declares any arrays or strings, they must be dynamically allocated (i.e. the only exception are command-line arguments)
- You are NOT allowed to use global variables but global constants are ok
- **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 – this is an integral part of this assignment
- 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

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. The flow of parameters is independent of the function return statement. In fact you could think of out/inout flow and the return statement as two alternative mechanisms for a function to communicate the results to the calling block.

Some Examples

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

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

// Swaps parameter values
// pre:   n1 and n2 point to valid locations
// 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.

GRADING

Remember, the programming assignments are graded as pass / no pass only and 85 points (out of 100 are needed to pass). If you do not pass this assignment, you can turn it in again with assignment 4 or 5.
Invalid input handling is worth 15 points
Proper addition with reduction is worth 15 points
Proper multiplication with reduction is worth 15 points
Clean memory report is worth 20 points
Proper decomposition into functions and function comments is worth 20 points
Proper coding style, meaningful identifiers, etc. is worth 15 points

PROGRAM SUBMISSION

On or before the due date for assignment 3, use the link posted in *Canvas* next to *Programming Assignment 3* to submit your C code. Make sure you know how to do that before the due date since late assignments will not be graded until the next grading period. Valid program format: a single file named *pr3.c*