L30: Pointers Lab

Note: This work must be completed at the end of the lab on __________.

Name: ________________________________ ID: ________________________________

Goals:

1. Be able to use pointers in C.
2. Be able to test code in C.

Part A: Robust Input

- Write a C program, called *l30a.c*, that prompts the user for a valid string and a filter string. If the user enters a string that contains only characters present in the filter string, your program must print out the string. If the entered string contains one or more characters not present in the filter string, your program must print out an error message.
- For example, if the input string is "COMP" and the filter is "ABCDEFGHIJKLMNOPQRSTUVWXYZ", the string is printed out. However, if the filter string is "0123456789", an error message is printed out instead, because none of the letters in "COMP" appears in the filter string.
- The end of the input is signalled by a newline ('\n').
- You can make no assumptions about the size of the input or filter strings; they must be allocated dynamically.
- Use the *getc(stdin)* function. Do not use any of the built-in C string functions for this part or the next part.

Part B: Functions

- Download the files 30.c 30.h, 30test.c, 30test1.txt, and 30test2.txt from D2L. Do not make any modifications to these files, except the bodies of the 3 functions in 30.c.
- Log into vmadmin and execute the following command:
```
sudo apt-get install ruby
```
- Log back into vmdev, cd into the directory containing the 30* files you downloaded, and execute the following command:
```
git clone https://github.com/ThrowTheSwitch/Unity
```
- Now execute the command:
```
cp Unity/src/unity* .
```
- If this fails, make sure you have copied the above correctly. Note the trailing space and period in the command above!

- Migrate your code from Part A into 30.c; you may have to make changes to your code to make it fit the structure of 30.c.
- To test your code, execute the following three commands, one at a time:

```
ruby Unity/auto/generate_test_runner.rb 30test.c
gcc 30.c 30test.c 30test_Runner.c unity.c -o 30test
./30test
```

- Fix every error caught by the tests. Initially, the output will be as follows:

```
30test.c:13:test_doublearraysize_null:PASS
30test.c:21:test_doublearraysize_empty:PASS
30test.c:32:test_doublearraysize_a:FAIL: Expected 2 Was 1
30test.c:41:test_doublearraysize_alphabet:FAIL: Expected 52 Was 26
30test.c:47:test_stringlength_null:PASS
30test.c:51:test_stringlength_empty:PASS
30test.c:56:test_stringlength_a:FAIL: Expected 1 Was 0
30test.c:60:test_stringlength_alphabet:FAIL: Expected 26 Was 0
30test.c:67:test_getfilteredstring_comp:FAIL: Expected 'MATHEMATICS' Was NULL
30test.c:75:test_getfilteredstring_comp183:FAIL: Expected 'Found invalid character'
Was NULL

-----------------------
10 Tests 6 Failures 0 Ignored
FAIL
```

- The above indicates, for example, that the test on line 13 of 30test.c passed, but the test on line 32 failed because the return value was 1 instead of 2. Make sure that your code ends up passing all the tests.

Deliverables:

- Submit a printout of your 30.c file attached to this lab sheet.
- After submitting the printout, call your instructor to demonstrate your working tests; additional tests may be supplied at that time.
- You will not be able to make any changes once you have submitted the printout, so be sure to test your work thoroughly!

Rating	Correctness/Efficiency	Documentation	Structure/Complexity
******* perfect**	- passes all tests	- well-documented, allowing another programmer to use all functions based on the header comments alone	- well-engineered, consisting of a modular collection of simple, single-purpose functions
	- code review reveals no faults	- responsibilities of all functions are described well, without giving implementation details	- constants are used whenever appropriate
	- efficient (given the requirements)	- all parameters, return values, and side effects are explained	- globals are not used, unless unavoidable
	- no redundant operations	- comments within all functions are helpful, without being distracting, making it easy to follow along	- all constants and variables are named appropriately
			- no layout abnormalities (eg, missing or improper indentation)
			- easily used and reused
			- no undesirable side effects, such as debug output
****** good**	- passes nearly all tests	- occasionally, there are comments that are not complete, helpful, and/or true	- largely well-engineered, except for a few, minor issues
	- a code review reveals nearly no faults; faults that are found, are minor	- could be improved by slightly reworded, slightly more, or slightly fewer comments	- a few, minor issues with constants, variables, or layout
	- generally efficient, except in a few minor cases		- easily used and reused, except in a few, minor instances
	- at most a few redundancies		- generally no undesirable side effects
***** ok**	- passes half the tests, or more, but the failure rate is too high for a 4-star rating	- in a number of cases, comments are cryptic, false, incomplete, misleading, missing, or redundant	- in need of reengineering due to a number of issues, none, or almost none of which, are major
	- a code review reveals a number of faults, but none, or almost none of them, are major		- on a number of occasions, there are issues with constants, variables, or layout
	- somewhat efficient, but there are a small number of major inefficiencies		- often easily used and reused, but there are a small number of major problems, none of which render the work unusable
	- potentially many redundancies		
**** fail**	- fails more than half the tests	- only little relevant documentation	- a number of major issues, requiring major changes
	- a code review reveals a high number of faults, including a number of major faults	- comments are often cryptic, false, incomplete, misleading, missing, or redundant	- not easily used and reused
	- not efficient, although slow progress is being made		
*** fail**	- fails nearly all the tests	- essentially no legitimate documentation	- only a few functions, many of which are responsible for too many things
	- code review reveals a proliferation of major faults		- essentially not usable or reusable
0 fail	- fails all tests and/or does not run	- no legitimate documentation and/or does not run	- no legitimate code and/or does not run

Correctness/Efficiency: _______ / 5 x 2 = _______ / 10
Documentation: _______ / 5
Structure/Complexity: _______ / 5

Total: _____ / 20 = _____ / 10