L20: Arrays and Functions Lab

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

Name: _____ ID: _____

Goals:

1. Be able to use arrays and functions in C.

Part A: Robust Input

- Write a C program, called *l20a.c*, that prompts the user for a valid integer and then prints out that integer. If the user enters an invalid character, the program re-prompts the user until a valid integer is entered.
- The end of the integer is signalled by a newline ('\n').
- If the user enters CTRL-D (also known as *EOF*) instead of any input, the result must be interpreted as a 0. One of your test cases must involve pressing CTRL-D right on start of your program. The number 0 should then be printed out.
- Your program must re-prompt for an integer if faced with the following types of issues: Misplaced - sign (e.g., *12-34*), invalid digit (e.g., *12A.*), space in the input (e.g., *12 34*), and more than 9 digits (e.g., *1234567890*; but *-123456789* is valid). All of these problems must cause your program to discard all existing input and prompt the user again for a valid integer.
- Negative integers (e.g., *-1234*) and positive integers must be accepted, as long as they do not exceed the above number of digits.
- Use the *getc(stdin)* function. This function reads exactly one character from *stdin*. Note that *getc(stdin)* returns an int, not a char. It will return the ASCII value of the character entered, or the value *EOF* (also known as *CTRL-D*). You can take advantage of the fact that chars are promoted to integers, i.e., *getc(stdin) == '0'* is true if and only if the user entered the digit 0.
- To discard all existing input, without using operating system-specific or compiler-specific calls, use the following code:

```
char tmp;
do {
    tmp = getc(stdin);
} while (tmp != '\n' && tmp != EOF);
```

- Test your program thoroughly! Use the slides on debugging to come up with a good set of tests.

Part B: Functions

- Write a C program, called *l20b.c*, that moves the code of Part A into a function called *int getInt()*. *int getInt()* should not print out the entered integer, but return it to the calling function instead. There should be no *printf()* in *int getInt()*.

- Use the following to test your function:

```
#include <stdio.h>
...
int main() {
    printf("%d\n", getInt());
    return 0;
}
```

Part C: Arrays

- Write a C program, called *l20c.c*, that prompts the user to enter a 4 x 4 array of integers, prints out the entered array, and then checks to make sure that all rows and columns add up to 10. Don't forget to use constants!

- Reading in the array, printing the array, and checking the array must be done by three separate functions --- *readArray(...)*, *printArray(...)*, and *checkArray(...)* --- and must only be called from *main()*.

- Be sure to use *getInt()* appropriately. You should not use on any other input functions.

- A sample run is below. Please follow this format if there are no problems with the input:

```
Requesting element [0][0]:
Enter an integer:
1
Requesting element [0][1]:
Enter an integer:
2
Requesting element [0][2]:
Enter an integer:
3
Requesting element [0][3]:
Enter an integer:
4
Requesting element [1][0]:
Enter an integer:
2
...
Requesting element [3][3]:
Enter an integer:
3
1       2       3       4
2       3       4       1
3       4       1       2
4       1       2       3
```

- Another sample run is below. Please follow this format if a problem occurs with the input:

```
Requesting element [0][0]:
Enter an integer:
1
...
Requesting element [3][3]:
Enter an integer:
4
1       2       3       4
2       2       2       2
3       3       3       3
4       4       4       4
Row 1 adds up to 8 not 10
Row 2 adds up to 12 not 10
Row 3 adds up to 16 not 10
Column 1 adds up to 11 not 10
Column 2 adds up to 12 not 10
Column 3 adds up to 13 not 10
```

- Test files are posted on D2L. You can try them out using UNIX redirection. For example, lab3c < sample1.txt (everything ok), lab3c < sample2.txt (issues except row 0 and column 0), lab3c < sample3.txt (issues with row 3 and column 3).

Deliverables:

- Submit a printout of Part C attached to this lab sheet.
- Be prepared to demonstrate all 3 Parts during the lab.
- After submitting the printout, call your instructor to demonstrate your work.
- You will not be able to make any changes once you have submitted the printout, so be sure to test your work thoroughly!

Notes:

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