CMPUT 201 - Winter 2023
Lab 7
Henry Tang -- University of Alberta

Concepts covered:

- Pointers and arrays
- Strings

Prep before the lab:

- Revise slides and reading material from Week 7 and Week 8.
- Revise and understand all demo programs/hand-written examples used in class.

Exercise overview: This lab consists of 3 parts. Part 1 is worth 2 marks, part 2 is worth 3 marks, and part 3 is worth 2 marks.

General instructions: As with all labs and assignments in this course, your code **must compile on the lab machines without any warnings or errors** using `gcc -Wall -std=c99 ...` Please check the "Labs & Lab Demo Guidelines" page on eClass for further important details for all labs.

Part 1

This exercise is from Ch12, Programming Project 2b, p275/276.

Write a program called `palindrome.c` which has a function `check_palindrome()`. Your `main()` function must read an input string from the user, then make a call to `check_palindrome()` to confirm whether it's a palindrome (the letters in the message are the same from left to right as from right to left).

If it is a palindrome, the function should return `1`; if it is not a palindrome, it should return `0`. Based on the return value of `check_palindrome()`, your `main()` function must print an appropriate output message.

A user input string will have at most 100 characters.

Your `check_palindrome()` function must be case-insensitive (i.e., "A" should be treated as equal to "a") and ignore all characters that are not letters.

Your program MUST use pointers and NOT integer array indices (i.e., a[1], a[5] etc.) to process the character array. You will get a 0 on this part if your program uses integer array indices instead of pointers.

Examples:

```
Enter a message: He lived as a devil, eh?
Palindrome
```

```
Enter a message: Madam, I am Adam.
Not a palindrome
```

Files that need to be on GitHub:

- `part1/palindrome.c`
- `part1/Makefile`

Possible Demo Tasks:

- Show and explain the `check_palindrome()` function in your code to your TA
- Compile your program using `make`
- Your TA will tell you what input to run your program with (multiple runs)
- Run `make clean`

Grading:

2 marks for Part 1

- **0.5 marks** for a correct `Makefile`
- **1.5 marks** for proper functionality of `palindrome` on all test cases (the function must be implemented using pointers).

Part 2

This exercise is based on a combination of Exercise 12 from Ch12 and Programming Project 18 from Ch13.

You are provided a file called `get_extension.c` that has the following function prototype:

```
int get_extension(const char *file_name);
```

where `file_name` is a string containing a file name. Your task is to implement this function as follows: The function processes the passed file name to find the file extension and returns the index of this file extension from a lookup array called `extensions`. The function must declare `extensions` as a **static** array of pointers to strings that stores the following file extensions in this exact order:

```
.txt
.out
.bkp
.dot
.tx
```

For example, if the input file name is `memo.txt`, the function will return 0. If the file name is `memo.tx.dot`, the function will return 3. If the file name doesn't have an extension (or the extension is not in the array), the function will return -1. Keep the function as simple as possible by having it use functions from the string library such as `strlen`, `strncpy`, `strcmp`, etc.

To test your `get_extension()` function, the provided program's `main()` function takes the file name as a program argument and passes it to `get_extension()`. The `main()` function then prints the value returned by `get_extension()` to `stdout`.

You can safely assume that the program will always be provided with an argument. You do not need to do any argument validation for this exercise. You can assume that the file name will have at most 20 characters, including the extension.

Files that need to be on GitHub:

- `part2/get_extension.c`
- `part2/Makefile`

Possible Demo Tasks:

- Show and explain the `get_extension()` function in your code to your TA
- Compile your program using `make`
- Your TA will tell you what input to run your program with (multiple runs)
- Run `make clean`

Grading:

3 marks for Part 2

- **0.5 marks** for a correct `Makefile`
- **2.5 marks** for proper functionality of `get_extension` on all test cases.

Part 3

This is based on Ch12, exercise 17.

Recall how 2D arrays are represented in memory. You are provided a C file called `sum-2d-array.c` that contains the function `sum_two_dimensional_array` shown below. You are also provided a Makefile that compiles the program. Rewrite the given function to use pointer arithmetic instead of array subscripting. In other words, eliminate the variables `i` and `j` and all uses of the `[]` operator.

Use a single loop instead of nested loops and do not change the function signature. You will get a 0 on this part if your program uses integer array indices instead of pointers, or if it uses a nested loop.

```
int sum_two_dimensional_array(int a[][LEN], int n){

    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        for (j = 0; j < LEN; j++)
            sum += a[i][j];

    return sum;
}
```

The following is the expected output of the program. This output should remain the same after you re-write the `sum_two_dimensional_array` function.

```
The sum of all elements of the array is: 120
```

Files that need to be on GitHub:

- `part3/sum-2d-array.c` that contains the modified version of the program after re-writing the loop
- `part3/Makefile`

Possible Demo Tasks:

- Show and explain the `sum_two_dimensional_array()` function in your code to your TA
- Compile your program using `make`
- Your TA will tell you what input to run your program with (multiple runs)
- Run `make clean`

Grading:

2 marks for Part 3

- **0.5 marks** for a correct `Makefile`
- **1.5 marks** for proper functionality of `sum-2d-array` (the function must be implemented using pointers and a single loop).