A4: C Strings

General Instructions:

1- Create a C project in eclipse called A4. Download A4.zip file and extract the contents to the A4 project.

2- Rename: “A4_template.txt” to “A4.c”. Enter your credentials on top of the file.

3- Use the file “A4_test.c” to test your solution. Compare your results to those shown in A4_output.txt. Before making your submission, verify that all your outputs match the desired output.

4- When you are done, you need to submit only your “A4.c” file. Do not export the project or upload a .zip file.

Task 1: Updating an Array

Implement the function:

```
void update_array(int** array, const int size, int multiplier)
```

The function receives an array of integers along with its size and an input parameter called `multiplier`.

The array is passed as a double integer pointer, to allow the function to make changes to the array.

The function repeats the array elements based on the `multiplier`.

For instance, if the array is `[10, 20, 30]` and `multiplier` is `2` the output will be:

```
[10, 10, 20, 20, 30, 30].
```

You can see how each element is being repeated twice.

If the `multiplier` is `4`, then the output will be:

```
[10, 10, 10, 10, 20, 20, 20, 20, 30, 30, 30, 30]
```

And so forth.

The function does not return any value.

The array had been created using dynamic memory allocation. Therefore, it should be resized inside the function based on the extra spots controlled by the multiplier.

Since the array is passed as a double pointer, the changes made unto the array inside the function will be preserved when existing the function.

If the value of multiplier is 0 or a negative number, then the function should print an error message and exist without making changes to the array.

If the multiplier equals to 1, then this is valid, but no changes are made to the array.

Assume that the array pointer and the size are always passed as valid values. Therefore, no error checking is required for them.

Running the testing file will produce the following output:

```
--------------Testing update_array --------------

Case 1: size = 6, multiplier = 2
Array Before update:
[ 10 , 20 , 30 , 40 , 50 , 60 ]
Array after update:
[ 10 , 10 , 20 , 20 , 30 , 30 , 40 , 40 , 50 , 50 , 60 , 60 ]

Case 2: size = 4, multiplier = 3
Array Before update:
[ 10 , 20 , 30 , 40 ]
Array after update:
[ 10 , 10 , 10 , 20 , 20 , 20 , 30 , 30 , 30 , 40 , 40 , 40 ]

Case 3: size = 10, multiplier = 1
Array Before update:
[ 10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 , 100 ]
Array after update:
[ 10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 , 100 ]

Case 4: size = 5, multiplier = 0
Array Before update:
[ 10 , 20 , 30 , 40 , 50 ]
Error (update_array): invalid multiplier
Array after update:
[ 10 , 20 , 30 , 40 , 50 ]

--------------End of Testing update_array --------------
```

Task2: Format Cities

Implement the function:

```
void format_cities(char city_array[][MAX], const int size);
```

The two input parameters are defined as below:

1- *cities*: a two dimensional array of characters, representing a list of cities. Each city is a string with maximum MAX-1 characters.

2- *size*: number of cities

For each city in the array, the function does the following:

1- convert the first letter to an upper case. For example: chicago should become "Chicago"

2- If the city name contains a space, e.g. "new york", then the function should remove the space and capitalize the second word, i.e. "new york" becomes: "NewYork".

Since the list of cities is passed by reference, executing the above function will change the cities without the need to make any returns.

In order to achieve the above, it is easier to create another utility function called: `void format_city(char *city)` which takes a single city string and apply the above formatting. Note how the function receives a pointer, not an array.

The *format_cities* function would need to call *format_city* to perform the formatting through looping into the city list.

The function should print an error message if the given list is empty or if the given size is invalid.

Task3: Manipulating Strings

Implement the function:

```
void format_str(const char *in_str, char *out_str)
```

The two input parameters:

3- *in_str*: a pointer to an array of characters (source string)

4- *out_str*: a pointer to an array of characters (destination string)

The function perform the following manipulations on the input string *in_str* and store it at the output string: *out_str*. The following is done in order:

3- ***String Length:*** Finds the length of the input string and print it to the console.

4- ***Middle Caps:*** Copy all of the characterss in *in_str* to *out_str*, but all characters are converted to upper case, except the first and last characters which are in lower case. For instance, *"abcdef"* should become: *"aBCDEf"*. The output should be printed to the console.

5- ***Split:*** The *out_str* is split into two halves, separated by a space character (both are contained in the *out_str*). The right side is either equal to the left side, or greater by one character. For instance: *"abcde"* should become: *"ab cde"* . The output is printed to the console.

6- ***Reverse Left Side***: The left side, i.e. substring before the space, should be reversed and printed to the console. For instance, *"abc def"* should become: *"cba def"*. The output should be printed to the console.

7- ***Reverse Right Side***: The right side, i.e. substring after the space, should be reversed and printed to the console. For instance, *"abc def"* should become: *"abc fed"*. The output should be printed to the console.

Assume that the length of the *in_str* will not exceed 2*MAX. Therefore, the length of the *out_str* will not exceed 2*MAX + 1 (because of inserting the space character).

Task4: Multiples Array (Version 1)

In R5 and Lab5, you learnt how to create, destroy and adjust arrays using dynamic memory allocation. However, all of these arrays were one dimension. Creating two dimensional arrays using dynamic memory allocation is not very different, but could be confusing if not handled properly. In Task 3 and Task 4, you will learn different methods to create 2D arrays using malloc/calloc. Refer to the following link for information:

https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/

Implement the following function:

```
int* get_multiples_array1(int *multiples, const int size);
```

The `multiples` array contain integers, and the number of these integers is specified by the parameter `size`.

The function creates and returns a new array that contains numbers between 0 and 1000, exclusive both ends, which are multiples of the given numbers in the input parameter array.

In row *x* of the output array, the multiples of the number at *multiples[x]* are stored. Each row contains no more than NUM multiples. If there are less than NUM multiples in the range (0,1000), then the rest of the array is filled with 0's.

For example, let the input parameters be:

1- multiples = {11,14,250}
2- size = 3

Then the output array will look like:

```
11  22  33  44  55  66  77  88  99 110
14  28  42  56  70  84  98 112 126 140
250 500 750   0   0   0   0   0   0   0
```

The first row represent the first NUM multiples of the number 11 in the range (0,1000). The second row represent the first NUM multiples of the number 14 in the range (0,1000). The third row represent the first NUM multiples of the number 250 in the range(0,1000). But since there are only 3 multiples, the rest of the array is filled with 0's.

The output array should be created using dynamic memory allocation (malloc/calloc). Use the FIRST METHOD provided in the above link.

The function prints an error message if the input array is NULL or if the value of size is invalid.

Once you have completed the above function, implement another function called:

```c
void print_multiples1(int *array, const int size);
```

The function is designed to print the output array produced by the previous function. The array is printed such that each row appear on a separate line, and each number occupies 3 digit spaces aligned to the right. Elements are separated by a space.
The function prints an error message if the given array is NULL or the given size is invalid.

Task 5: Multiples Array (Version 2)

Implement the two functions:

```c
int** get_multiples_array2(int *multiples, const int size);
void print_multiples2(int **array, const int size);
```

The above two functions are similar to Task 3, with two main differences.

First, the function does not use the first Method in the following link:

https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/

Instead, it uses either method 3 or method 4. It is up to you to pick which one to use.

Second, the get function returns a double pointer, not a single pointer. Also, the print function receives a double pointer, not a single pointer.