A3: C Pointers

General Instructions:

1- Create a C project in eclipse called A3. Download A3.zip file and extract the contents to the A3 project.

2- Rename: “A3_template.txt” to “A3.c”. Enter your credentials on top of the file.

3- Use the file “A3_test.c” to test your solution. Compare your results to those shown in A3_output.txt.

4- Make sure that your solution does not generate any error or warnings. There will be -1 deduction for each error and -0.2 for each warning.

5- When you are done, you need to submit only your “A3.c” file. Do not export the project or upload a .zip file.

General Hints:

1- Before solving the problems, read through the entire assignment and start with the tasks that are easier for you. The questions are not ordered based on difficulty.

2- It is very normal to have strange outputs when implementing the functions. This is the result of poor handling of pointers. As you write the function, test your code line by line. This ensures that you know exactly what is happening inside your function

3- Use the Debugger whenever you can't find the issue with your code. This can be done by selecting: Run → Debug, which will take you to the Debugging perspective. From there you can use the "Step Into" and "Step Over" as necessary.

4- Make sure to handle the error cases first. Otherwise, your functions will not work.

Task1: Find Next Element (2 pts)

The objective of this question is test your understanding of the following points:

1- How to traverse (i.e., loop through) an array that is passed as a pointer to a function.

2- Create and return a pointer

3- Assign an array item address to the pointer

Implement the function *find_next*, which searches an array of integers for a given value, and update a given pointer based on the location of the value. The function prototype is:

```
int* find_next(int* array, const int size, int value){
      //your code here
      return NULL;
}
```

The function receives three parameters:

```
* Parameters
 *       array: a pointer to an array of integers (int*)
 *       size: size of an array (const int)
 *       value: some value to search for in the array (int)
```

The function returns a pointer to an integer, which is one of the elements in the array.

```
* returns:
 *       ptr: pointer to some element in the array (int*)
```

The function searches for the location of "value" in the array. If there are multiple occurrences, the first occurrence is considered. There are four scenarios:

```
 *   1- if value is found at the last element --> ptr should point to last element
 *   2- if value is found at any other location --> ptr should point to next element
 *   3- if value is not found --> ptr should point to the first element
 *   4- if the array pointer is NULL or the size is invalid --> ptr point to NULL
```

Overall, the function searches for "value" and sets the pointer to the next element in the array (this is point 2 above). The other three scenarios are for handling edge cases.

Testing the function will produce:

```
------------------------------
Start: Testing find_next:
```

```
Case 1:
find_next(NULL, 9, 5) --> NULL
Case 2:
find_next([], 0, 10) --> NULL
Case 3:
find_next([10,5,15,10,2,4], 6, 5) --> 15
Case 4:
find_next([10,5,15,10,2,4,16,4,8], 9, 4) --> 16
Case 5:
find_next([10,5], 2, 11) --> 10
Case 6:
find_next([10,5,15,10,2,4,16,4,8,18,3,9], 12, 13) --> 10
Case 7:
find_next([10,5,15,10,2], 5, 2) --> 2
Case 8:
find_next([10,5,15,10,2,4,16,4,8,18,3,9], 12, 9) --> 9

End: Testing find_next
------------------------------
```

Task2: Increment Element (2 pts)

The objective of this question is test your understanding of the following points:

1- Use void pointers to pass different data types to the function

2- Cast a void pointer to a specific type pointer

3- Deference a void pointer after casting it to a specific data type

Implement the function *increment_element*, which increments an item identified by a given index. The function prototype is:

```
void increment_element(void* array, const int size, int indx, char mode){
      //your code here
      return;
}
```

The function receives three parameters:

```
 * Parameters:
 *       array: a pointer to an array of unknown data type (void*)
 *       size: size of the array (const int)
 *       indx: array index of the element to increment (int)
 *       char: array mode which controls the data type of the array.
```

The function does not return any value

As you notice, the array pointer is of type (void*), which means it can be any data type. This offers much flexibility, but requires some careful handling of the pointer.

We are mainly interested in four data types, which are controlled by the parameter mode:

```
 *  Defined types:
 *       'i': integer
 *       'l': long
 *       'f': float
 *       'd': double
```

Once, the right item has been identified the function should:

1- Print the value (before increment)

2- Increment the value by 1

The function handles four error scenarios:

```
 *     1- array pointer is NULL
 *     2- invalid array size
 *     3- invalid value for indx
 *      4- invalid mode
```

For all four error scenarios, the function should print an error message as outlined in the output.txt file.

Testing the function will produce:

```
------------------------------
Start: Testing increment_element:

Case 1: Error cases:
Error(increment_element): NULL array
Error(increment_element): invalid size
Error(increment_element): invalid indx
Error(increment_element): invalid indx
Error(increment_element): invalid mode

Case 2: array of integers:
Before: [10,20,30,40,50]
Value before increment = 40
After: [10,20,30,41,50]

Case 3: array of longs:
Before: [200, 345, 679,501,449,912]
Value before increment = 449
After: [200,345,679,501,450,912]

Case 4: array of floats:
Before: [20.5, 3.45, 6.79, 5.01]
Value before increment = 3.45
After: [20.50,4.45,6.79,5.01]
```

```
Case 5: array of doubles:
Before: [20.5, 3.45, 6.79, 5.01]
Value before increment = 20.50
After: [21.50,3.45,6.79,5.01]

End: Testing increment_element
------------------------------
```

Task3: Find Average Element (2 pts)

The objective of this question is test your understanding of the following points:

1- Use and dereference a double pointer

2- Process an array that is passed as a pointer

Implement the function *find_avg_element*, which searches a given area and returns a pointer to the element that is closest to the average of the array contents..

The function prototype is:

```
void find_avg_element(int *array, const int size, int** avg_ptr) {
      //your code here
      return;
}
```

The function receives three parameters:

```
 * Parameters:
 *       array: a pointer to an array of integers (int*)
 *       size: size of the array (const int)
 *       ptr:  double pointer to store location of average element
```

The function does not return any value

The function starts by finding the average of the array elements and prints it to the screen up to 1 decimal point. The function then performs the following:

```
 *    2- The average is "ceiled", i.e. rounded to the upper integer value.
 *    3- Search the array for an element that is equal to the average,
 *           and sets avg_pointer to that element
 *     If there are multiple values equal to the average, a pointer to the
 *           first element that equals to average is used
 *     If no value is equal to the average, a pointer to the value
 *           closest to the average is returned.
 *    4- prints an error message if
 *           array pointer is NULL, invalid size or avg_ptr is NULL
```

To perform some of the above tasks, you might find that using some functions defined in the math.h library is useful.

Testing the function will produce:

```
------------------------------
Start: Testing find_avg_element:

Case 1: Error Cases
Error(find_avg_element): NULL pointer
Error(find_avg_element): invalid array size
Error(find_avg_element): invalid pointer

Case 2: {1,2,3,4,5,6,7}
Average = 4.0
The Average Element is 4 which is at index: 3

Case 3: {1,2,3,4,5,6,7,8}
Average = 4.5
The Average Element is 5 which is at index: 4

Case 4: {5,5,5,5,5}
Average = 5.0
The Average Element is 5 which is at index: 0

Case 5: {100,10}
Average = 55.0
The Average Element is 100 which is at index: 0

Case 6: {1,7,9,23,44}
Average = 16.8
The Average Element is 23 which is at index: 3

End: Testing find_avg_element
------------------------------
```

Task4: Print Pointer Array (2 pts)

The objective of this question is test your understanding of the following points:

1- Use array of pointers

2- Assign and dereference pointer elements in a pointer array

3- Access arrays of unknown sizes through some given information

Implement the function *print_ptr_array*, which prints selected information about the given area using a group of pointers, stored as an array. The function has the following prototype:

```
void print_ptr_array(double *array, double *ptrs[]){
   //your code here
   return;
}
```

The function receives two parameters:

```
 * Parameters:
 *       array: a pointer to an array of floats (float*)
 *       ptrs: an array of float pointers (float* [])
```

The function does not return any value.

Unlike the other tasks, here you neither know the size of the given arrays is unknown. Since there is no way for the function to know the size of the passed array, we need some information about the arrays to help us traverse them. Assume the following:

1- The array and the pointer array are valid pointers, i.e., they are not NULL.
2- The maximum size of the pointer array is MAX
3- The end of the pointer array is marked by finding a pointer that points to NULL
4- All pointers in the pointer array point to elements within the given array. None of them point to a data other than the array elements

Note that the sizes of the array and the pointer array are independent of each other. Also, it is possible that two pointers point to the same element in the array.

Use the pointer array to print the following data from the array:

```
 *    1-     The value of the element it points to
 *    2-     The index of that element in the array
 *    3-     The value of the previous element
 *           If the pointer points to the first element, previous value is 0
```

Inspect the output file to understand the expected printing. Note that all prints are formatted in 13 digit spaces, and all float numbers are formatted in 2 decimals.

Testing the function will produce:

```
-----------------------------
Start: Testing print_ptr_array:

Case 1:
[10.09,-4.18,20.27,-4.36,30.45,40.54,50.63,-4.72,60.81,70.90]
Value        Index        Previous
-4.18        1            10.09
-4.36        3            20.27
-4.72        7            50.63

Case 2:
```

```
[10.09,-4.18,20.27,-4.36,30.45,40.54,50.63,-4.72]
Value        Index        Previous
20.27        2            -4.18
-4.36        3            20.27
30.45        4            -4.36
50.63        6            40.54
30.45        4            -4.36
-4.72        7            50.63

Case 3:
[1.90,2.81,3.72,1.63,5.54,6.45,1.36,7.27,8.18,1.09]
Value        Index        Previous
1.90         0            0.00
1.63         3            3.72
1.36         6            6.45
1.09         9            8.18

Case 4:
[1.90,2.81,3.72,1.63,5.54,6.45,1.36,7.27,8.18,1.09]
Value        Index        Previous

End: Testing print_ptr_array
------------------------------
```

Task5: Special Print Array (2 pts)

The objective of this question is test your understanding of the following points:

1- Perform pointer arithmetic to access array elements

2- Access arrays of unknown sizes through some given information

Implement the function *special_array_print*, which prints an array in a specific format guided by the position of a given pointer. The function prototype is given below:

```
void special_array_print(short* array, short* ptr){
   //your code here
   return;
}
```

The function makes no returns and receives the following two parameters:

```
* Parameters:
 *       array: a pointer to an array of shorts (short*)
 *       ptr: a pointer to an item in the array (short*)
 * returns: No returns
```

Below is a detailed description of the function:

```
*       An array of shorts is given, and you do not know its size
 *      However, you know the following about the array:
```

```
 *           1- the minimum length is 3 items
 *           2- The first element has a value of 1
 *           3- The last element has a value of 1
 *    You are given some pointer to an element in the array.
 *    The pointer points to some item other than the first and last items
 *      Your job is to print all the items in the array in the following format:
 *           [items before pointer]-->[items after pointer]
 *      The printing ignores the first and last items which have the value of 1
 *    The function should print an error message if the array or the pointer is NULL
```

Running the testing function will produce the following output:

```
------------------------------
Start: Testing special_array_print:

Case 1: Error cases:
Error(special_array_print): NULL pointer
Error(special_array_print): invalid pointer

Case 2:
[13, 48, 91]-->[74, 66, 12]

Case 3:
[48]-->[95, 74, 66]

Case 4:
[21, 48]-->[95]

Case 5:
[21, 48, 91]-->[]

Case 6:
[]-->[48, 91, 95]

Case 7:
[]-->[]

End: Testing special_array_print
------------------------------
```