SUGGESTED PROBLEMS – WEEKS 10 - 12

The following is a last set of questions, on 1D and 2D arrays, pointers, strings, malloc and struct.

#1. Imagine an array of numbers generated randomly, that you'd like to sort into even and odd, in any order. Write a program that generates a random array of numbers:

14 77 8 38 93 19 6 44 10 37

and then creates a second array of the same length in which the even numbers appear first (in any order), followed by the odd:

14 8 38 6 44 10 77 93 19 37

#2. Write a program that generates an array of 10 integers, prints those values to the screen, then asks the user for a number between 1 and 10, and creates and prints a second array that is shifted by that many elements. For example:

the original array: 9 4 7 0 2 6 9 1 5 4
shift how many elements? 3
the shifted array: 1 5 4 9 4 7 0 2 6 9

#3. This is a variation on a question from a previous exam, that asks you to calculate a rolling average of values in an array. What is a rolling average? Consider the following array of numbers:

int A[10] = {8, 50, 74, 59, 31, 73, 45, 79, 24, 10};

A rolling average over 3 values, for example, is the average of every consecutive set of 3 numbers:

(8+50+74)/3.0 = 44.00
(50+74+59)/3.0 = 61.00
(74+59+31)/3.0 = 54.67
...

Write a program that allocates an array of N integers, assigns random integer values and prints them, then asks the user to specify the number of values n to average: $1 < n < N$. Then calculate and print the average values.

How? Note that the number of averages you can calculate is N-n+1 (for the above example, N=10 and n=3, and you can calculate 8 averages). Write a loop to calculate that many averages: in each case it initiates another loop over n values to calculate a sum, and then calculates the average by dividing by n.

Output might look something like this:

array values are: 8 50 74 59 31 73 45 79 24 10
calculate averages over how many values? 3
averages are: 44.00 61.00 54.67 54.33 49.67 65.67 49.33 37.67

#4. An easy one: use pointer notation to write a function that returns the maximum value in an array of length N:

```
int max_value (int *A, int N);
```

#5. Write a function that does basically the same thing, except that the max value resides in the program that calls the function:

```
void max_value (int *A, int N, int *max);
```

#6. Write a program that includes two functions: main ( ) and

```
void min_max (int A[ ], int N, int *min, int *max);
```

In main ( ), allocate an array of N elements and fill it with random values. Then call min_max and pass the array and the size of the array, and two pointers to integers. The function min_max determines the minimum and maximum values in the array and stores those values in the variables min and max in main ( ).

#7. Write a function that reverses an array of length N using pointers:

```
void reverse (int *A, int N);
```

#8. Write a function that accepts pointers to two arrays, A (of length N, an even number) and B (exactly half the length of A). The function adds pairs of values from the first array, and writes each sum into the second array.

```
void sum_pairs (int *A, int N, int *B);
```

#9. Write a program that includes two functions: main ( ) and

```
void sort_three (int *v1, int *v2, int *v3);
```

In main ( ), ask the user to enter three integers. Then call sort_three, with the prototype as shown, which sorts the three integers pointed at by v1, v2, and v3 – i.e. move the smallest value into v1, the middle value into v2, and the largest value into v3. (Note that sort_three doesn't return anything to main). Back in main ( ), print out the three values in order.

#10. Download binary_search.c, which was an earlier Extra Problem. Convert the array syntax in this problem to use pointers. In other words, leave the array declaration as it is at the top of the program:

```
int array[SIZE];   /* declaration in binary_search.c */
```

but replace every other instance of square brackets [ ] with the indirection operator *.

#11. From an old exam, write a function

```
find_and_replace (char *s, char f, char r);
```

that accepts a string and two characters f and r, and searches the string for every instance of f and replaces it with the character in r. Then write the associated main( ) function that asks a user for a string, sends the string to find_and_replace, and finally prints the modified string. For example, input and output might look like this:

```
Enter a string:  good
Replace what char?  d
With what char?  f
Modified string:  goof
```

#12. Write a program that allocates an array of 100 characters, and asks a user for a sentence of less than 100 characters. Assume the user does what he/she is told, and so feel free to use gets ( ) to read the sentence into the array. Then send the string to the function:

```
void capitalize (char *sentence);
```

that capitalizes every word (defined as the first letter of the sentence, and every letter that follows a space). Print the sentence before and after capitalization.

#13. Write a program that asks a user for a string of characters, and reads that string into a character array. You may assume that the string is shorter than the array. Then write a function

```
int no_multiples (char *string);
```

that searches the string for multiple instances of any character, overwrites all but the first instance to a blank, and then returns the number of characters that have been overwritten. For example, if the user were to enter:

```
What a wonderful day!!
```

the function would convert that string to:

```
What   wonderful   y!
```

Notice that W and w are different characters, but that the function overwrote two instances of a, one d, and the second exclamation mark. The function would have returned the number 4.

Hint: you can do this in different ways, although I think in each case you'll want to interpret these characters as integers between 0 and 255. One approach is to loop through the string 255 times, looking for each of the 255 characters in the ASCII table (every character but the space). If you find a first instance, blank out every subsequent one. The other approach is to set up an array of 256 integers that you'll use to keep track of how many instances of each character you find. Initialize that array to zero, and then loop through the string only once, incrementing the appropriate counter for each character in the string. When you come across characters that you've found before, you can blank them out.

#14. This one is related to the last one, but a bit trickier. Write a similar code, but this time shrink the string rather than overwrite the multiple characters. In other words,

```
What a wonderful day!!
```

would this time become

```
What  wonderful dy!
```

#15. Write a code that asks a user for a string of characters, and read that string into a character array. You may assume that the string is shorter than the array. Then write a function

```
void ASCII_order (char *string);
```

that reorders the characters (up to, but not including the '\0') in the order that they appear in the ASCII table.

Hint: How to do this? A couple of ways come to mind. One way is to allocate a second array that's as long as the first, then sweep through the original array 255 times (from 1 to 255; we're not moving the 0'th character '\0'), and each time copy characters from the original to the new array. Once done, you can copy the entire new array back into the original. The other approach doesn't require a second array. Loop the same way, but this time swap characters within the one array to get them in order.

#16. Write a program that asks a user for a sentence, and read that string into a character array. You may assume that the sentence is shorter than the array. Then write a function

```
void noBlanks (char *string);
```

that removes all blanks from the string, and instead capitalizes the first letter of every word. For example:

```
Enter a sentence: What a beautiful    day!
Capitalized: WhatABeautifulDay!
```

#17. Write a program that yields the following output (you can think of this as array gymnastics):

```
 1  2  3  4  5   /* original array */
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25

 1  6 11 16 21   /* after a flip across the TLBR diagonal
 2  7 12 17 22      (TLBR = top left to bottom right) */
 3  8 13 18 23
 4  9 14 19 24
 5 10 15 20 25

25 24 23 22 21   /* after a flip across the BLTR diagonal
```

```
20 19 18 17 16      (BLTR = bottom left to top right) */
15 14 13 12 11
10  9  8  7  6
 5  4  3  2  1

25 20 15 10  5   /* another flip across the TLBR diagonal */
24 19 14  9  4
23 18 13  8  3
22 17 12  7  2
21 16 11  6  1

 1  2  3  4  5   /* a second flip across the BLTR diagonal */
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

Notice that after four flips, the array is back to its original form.

In main ( ) you declare the array, initialize it, and then call the functions TLBR, BLTR, TLBR, and BLTR, that do the flipping, each time printing the array. Note that you should write this code for a square array of any SIZE.

#18. Here's a question on the topic of dynamic memory allocation. Write a program that generates N unique random numbers between values that the user specifies. For example:

How many random numbers N? 7
Limiting values? 6 17
The random values are 8 13 6 15 12 9 7

Note that you'll need to check that the range of values can accommodate N numbers; if not, ask for a new pair of limiting values.

Note: Why do you need dynamic memory allocation for this problem? Because you'll need to keep track of which random numbers you've already printed to the screen, and to do that you'll need an array of N numbers.

#19. And finally, a small question to illustrate the use of structures. Consider a deck of playing cards: there are 13 types: numbers from 2 to 9, plus the jack, queen, king, and ace, and 4 suits: spades, clubs, hearts, and diamonds.

Write a code that declares a structure to represent one playing card:

```
struct Card {
  char suit;
  char type;
};
```

Note that each type and each suit can be represented by single characters (e.g. '1','2',… ,'9','J','Q','K','A' for the types). Allocate an array of 52 cards, and then initialize that array, so that each of the 52 cards is unique. Then write another loop that deals 5 cards to a player, chosen randomly from the deck.