C Programming, Exercises for the second week

Notice: Remember that you can find information about a standard C library function by writing `man 3 function_name` in the terminal, or by going to the address:

http://linux.die.net/man/3:function_name

In this week exercises we learn to handle arrays and pointers. As a prerequisite for a successful programming you should know the various list data structures and their implementation principles.

1. A program has the following definitions and statements:

   int i, j=25;
   int *pi, *pj=&j;
   *pj=j+5;
   i= *pj+5;
   pi=pj;
   *pi=i+j;

   Test experimentally and explain the results:

   a) After the definitions, what values are represented by i, j, &i, &j?

   b) After the definitions, what values are represented by pi, pj, *pi, *pj?

   c) After the first statement, what values are represented by pj and *pj?

   d) After the second statement, what value is represented by i?

   e) After the third statement, what values are represented by pi and *pi?

   f) After the fourth statement, what values are represented by pi and *pi?

2. Assume that the following expressions are true:

   sizeof(short) == 2
   sizeof(int) == 4
   sizeof(float) == 4
   sizeof(double) == 8

   Also, assume that we have declared the array

   int arr[5] = { 0, 0, 0, 0, 0 };

   Explain with a picture (hand-drawn is fine) where the following statements store values (both the location and the amount of bytes written). Assume that all the assignments succeed.
a) arr[3] = 42;
b) arr[9] = 7;
c) arr[-4] = 1;
d) ((short*)arr)[7] = 128;
e) ((double*)arr)[2] = 3.14;
f) ((char*)&arr[1])[6] = 'A';
g) ((float*)((&((short*)&arr[3])[-3]))[0] = 7.5; (This is challenging!)

(Hint: Remember that array indexing [] binds more tightly than the address-of operator and type conversions. Start analysing the more complex expressions from the inside.)

If all the previous assignments succeed, which of the assignments have affected the value of:

i) arr[0]
ii) arr[2]
iii) arr[4]

3. Write a function

```c
void print_doubles(double* array, int len);
```

that prints the double values stored in the given array. The parameter len contains the number of double values in the array.

Example call:
```c
print_doubles(nums, 5);
```

Result of the example call:
3.14159, 2.71828, 1.41421, 1.61803, 4.66920

4. Write a function

```c
void shuffle_ints(int* array, int len);
```

that shuffles the contents of the given array of integers. The parameter len contains the number of int values in the array. Write the shuffling code so that, given a random function with uniform distribution, every possible ordering of the contents is as likely. (Use the rand function provided by the standard C library.)

Example call:
```c
int nums[] = { 0, 1, 2, 3, 4, 5, 6 };  
shuffle_ints(nums, 7);
```
Example result:

1, 0, 6, 4, 3, 2, 5

5. In the exercise 1 we implemented operations for fractions. Represent now a fraction with the help of a structure:

```c
typedef struct {
    int numerator;
    int denominator;
} fraction;
```

Implement the addition operation so that the function returns the sum in a structure:

```c
fraction addFractions(fraction r, fraction s);
```

6. Consider a doubly linked list (see the figures in the appendix) which contains integer values. Write the type definitions for this kind of a list (the name of the type could be dList).

7. Write a function

```c
dList* insertElementD(dList *L, dList *p, int value);
```

which adds an integer value into the list L to the right side of position p. Take into account in this and other exercises that initially the list may be empty. The function returns a pointer to the start of the list.

8. Write a function

```c
int printElementsD(dList *L);
```

which prints the integers in the list.

9. Write a function

```c
dList* deleteElementD(dList *L, dList *p);
```

which deletes an element pointed by p from a doubly linked list. The function returns a pointer to the start of the list.

10. Write a function

```c
int orderListD(dList *L);
```

which orders the values in the doubly linked list L from the smallest to the largest. The function returns 0, if everything succeeds.
11. Write a function

   dList* mergeListsD(dList *L1, dList *L2);

which merges ordered doubly linked lists $L_1$ and $L_2$ into a third list which is ordered, too.

12. Consider a singly linked list which contains integer values. The lists have a header node. The header node contains at least the number of nodes and a pointer to the first data node. Write the type definitions and the function

   sList* createSList(void);

which creates an empty list.

13. Write a function

   int insertElementS(sList *L, sList *p, int value);

which adds an integer value into the list $L$ to the right side of position $p$. Check the next task and notice how the position pointer should be defined. The function returns 0, if everything succeeds. Test your function by printing the elements in the list just created with the help of insertElementS.

14. Write a function

   int deleteElementS(sList *L, sList *p);

which deletes an element pointed by $p$ from a singly linked list. The operation should work in a constant time. Thus you should think where the pointer $p$ should point (see the appendix). The function returns 0, if the operation succeeds.

15. Write a function

   sList* mergeLists(sList *L1, sList *L2);

which merges ordered singly linked lists $L_1$ and $L_2$ into a third singly linked list which is ordered, too.
APPENDIX: List Diagrams

Figure 1: Doubly linked list

Figure 2: After deleting the item 4 pointed by p
Figure 3: Singly linked list and position s pointed by p

Figure 4: Singly linked list after deleting the node s