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

1. A program has the following definitions and statements:

```c
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:

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

Also, assume that we have declared the array

```c
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

    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:

print_doubles(nums, 5);

Result of the example call:

3.14159, 2.71828, 1.41421, 1.61803, 4.66920

4. Write a function

    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:

int nums[] = { 0, 1, 2, 3, 4, 5, 6 };  
shuffle_ints(nums, 7);

Example result:

1, 0, 6, 4, 3, 2, 5
5. Consider a singly linked list whose nodes are of the following type:

```c
struct list_node {
    int value;
    struct list_node *next;
};
```

Write a function that adds one item into the list. The position where the new item is to be added is given as a parameter. The operation should work in a constant time.

(Hint: Before you determine how to represent a position, you should check the hint of the next task in order to avoid two definitions for the position.)

6. Write a function that deletes one item from the list. The position of the item to be deleted is given as a parameter. The operation should work in a constant time.

(Hint: If you point directly to the node to be deleted, you must search for the previous node starting from the beginning of the list. This takes time $O(n)$ in the worst case, where $n$ is the length of the list. So you must define the position with the help of the previous node or you must use more complicated position definitions than a single pointer.)

7. Write the type definitions for doubly linked list. You can assume that the data values are integers.

8. Write a function that adds one item into the list. The position where the new item is to be added is given as a parameter.

9. Write a function that deletes one item from the list. The item to be deleted is given as a parameter.

10. Write a function

```c
    void shuffle(void* array, int len, size_t elem_size);
```

that shuffles the contents of arrays of any type. The parameter `elem_size` contains the size of a single element in bytes. Use the generic swap function presented in the lecture of the second week.

Example call:

```c
double nums[5] = { 1.5, 3.5, 5.5, 7.5, 9.5 };
shuffle(nums, 5, sizeof(double));
print_doubles(nums, 5); // implemented in a previous task
```

Example result:
11. Define a data type called DynArray that contains a dynamically expanding array of int values. (The constructor, destructor and accessors are implemented in the following tasks.)

12. Write a function

   DynArray* DynArray_create(void);

   that creates a dynamically allocated DynArray that does not contain any int values. It should reserve some space internally for upcoming values, though. Use the malloc function for memory allocation. The function returns a pointer to the dynamically allocated object on success, or NULL if memory allocation fails.

13. Write a function

   void DynArray_destroy(DynArray* arr);

   that frees resources allocated for the given DynArray. Decide how to handle the case where the caller passes NULL (pointer to nothing) as the argument.

14. Write a simple test program where you create and destroy a DynArray using the functions written in the previous two tasks. Run the program with valgrind to verify that you use memory correctly and free all the allocated resources before exit.

15. Write a function

   bool DynArray_append(DynArray* arr, int number);

   that inserts number to the end of the given DynArray. Use the realloc function to expand the internal storage by a factor of 2 if necessary. The function returns true if the operation succeeds, or false if memory allocation fails.
   Extend your DynArray test program with several test calls of DynArray_append so that the realloc function inside DynArray_append is called at least twice. Rerun with valgrind in order to make sure that there are no new memory errors.

16. Write a function

   int DynArray_get(DynArray* arr, int index);

   that retrieves a number from the given index in the given DynArray. Decide how to handle the case where the caller passes an invalid index as the index argument.