Exercises C++
SHEET 2

Pointers, References and Arrays

Problem 1. Write a function **sum** that which takes a pointer to a function **int funct(int)** and returns

$$\sum_{0 \le i < n} funct(i).$$

Hint:

a. Declaration of the function sum.

```
int sum(int (*funct)(int), int n);
```

b. Calling the function sum.

```
...
int g(int x)
{return x;};

int h(int x)
{ return x*x;};

int main()
{
   cout<<sum(g,n);
   cout<<endl;
   cout<<sum(h,n);
}
```

Recall that the function name is a pointer to the function, so you don't need to dereference using *. For the same reason, you don't have to use the symbol & in order to get the address of the function, as in the case of variables.

Problem 2. Implement the following program. Explain the results.

```
int& f(int & a)
{   a=a+5;
    return a;}
int main(){
 int a=5;
 for (int i=0;i<2;i++)
```

```
    { f(a)++;}
    cout<<f(a);
    return 0;}
```

What does it happen if

- we replace **int & f(int & a)** by **int f(int & a)**.
- we replace **int & f(int & a)** by **int & f(int a)**.
- we replace **int & f(int & a)** by **int f(int a)**.

Problem 3. Write a C++ program to accept five integer values from the keyboard. The five values will be stored in an array using a pointer. Then print the elements of the array on the screen.

Hint:

```
int a[5]; // Declaration of the array
int *p=a; // Declaration of a pointer pointing to the first element of the
   array
for (int i=0;i<5;i++)
   cin>>*(p+i); // this is equivalent to a[i];
```

Problem 4. Write a function **swap** which interchanges two real numbers. Propose two different implementations of this function, such that the interchange of the two real numbers **reflects back** in the caller of the function swap.

Problem 5. Write a function **sumDoubles** which takes a pointer to a list of doubles and the length of the list and returns the total.

Problem 6. Write a function **reverseDoubles** which takes a pointer to a list of doubles and the length of the list and reverses it. So $(1, 2, 3, 4)$ should be changed to $(4, 3, 2, 1)$ for example.

Problem 7. Write a function **mystrlen** to count the number of characters of a string, without the null-terminated character (of course, you should use the library function strlen() rather than write your own function. This exercise has only been given for educational purpose).

Problem 8. Write two functions **reverseString** which takes a **char*** string and reverse it. One should use the function **strlen** and the second one not.

Problem 9. Write a function **concatenate** to append one null-terminated string to another. I've deliberately not told you the parameters or return type, this is a design question for you to solve (of course, you should use the library functions rather than write your own function. This exercise has been given only for an educational purpose, in order to learn you manipulate pointers).

Problem 10. Write a function **count** which takes as input a char* string and a character. The function should return the number of occurences of the given character in the string (For example, if the string is "I'm a student" and the character is 't', the result should be 2. **Hint**: use the library function **strchr**).

Problem 11.

1. Write a function **mystrstr** which takes as input two char* strings. The first should be **phrase**, a phrase to search for, the second should be **text**, some text to scan through. The function should return a pointer to the first character of the found substring (**phrase**) in **text** if **phrase** is contained in **text** , or a null pointer if it cannot (of course, you should use the library function **strstr** rather than write your own function. This exercise has only been given for educational purpose).

2. Write a function **search** which counts the number of times **phrase** appears in **text**. The function **search** should call the function **mystrstr**.

Problem 12.. Write a sort algorithm of an array **a**[], which is implemented in a function **void sort(int a[], int n, int (*compare)(int, int))**. The sort routine calls the function **compare** when deciding how to order the values.

Problem 13. Quick Sort - An Efficient Recursive Procedure for Sorting The aim of this exercise is to implement a standard recursive procedure for sorting. Quick Sort is a an *efficient* recursively defined procedure for rearranging the values stored in an array in ascendind or descending order.

Suppose we start with the following array of 11 integers:

14	3	2	11	5	8	0	2	9	4	20
a[0]										a[10].

The idea is to use a process which separates the list into two parts, using a distinguished value in the list called *a pivot*. At the end of the process, one part will contain only values less than or equal to the pivot, and the other will contain only values greater than or equal to the pivot. So, if we pick 8 **as the pivot**, at the end of the process we will end up with something like:

4 3 2 2 5 0 **8** 11 9 14 20

a[0] a[10].

We can reapply exactly the same process to the left-hand and right-hand parts separately. This re-application of the same procedure leads to a recursive definition.

The detail of the rearranging procedure is as follows. The index of the pivot value is chosen simply by evaluating

(first+last)/2

where "first" and "last" are the indices of the initial and final elements in the array representing the list. We then identify "left_arrow" and a "right_arrow" on the far left and on the far right respectively. This can be envisioned as:

14 3 2 11 5 8 0 2 9 4 20

left_arrow=0 **pivot** **right_arrow=10**.

- **left_arrow** and **right_arrow** initially represent the lowest and highest indices of the array components. Starting on the right, the **right_arrow** is moved left until a value less than or equal to the pivot is encountered. This produces:

14 3 2 11 5 8 0 2 9 4 20

left_arrow=0 **pivot** **right_arrow=9** .

- In a similar manner, **left_arrow** is moved right until a value greater than or equal to the pivot is encountered. This is already the situation in our example. Now the contents of the two array components are swapped to produce:

4 3 2 11 5 8 0 2 9 14 20

left_arrow **pivot** **right_arrow** .

- We continue by moving **right_arrow** left to produce:

4 3 2 11 5 8 0 2 9 14 20

left_arrow **pivot** **right_arrow** .

- and then **left_arrow** right to produce:

4 3 2 11 5 8 0 2 9 14 20

left_arrow **pivot** **right_arrow** .

- These values are exchanged to produce:

4 3 2 2 5 8 0 11 9 14 20

left_arrow **pivot** **right_arrow** .

This part of the process only stops when the condition **left_arrow>right_arrow** becomes **true**.

Problem 14. Card games Create a structure representing a card game (color and value). Write a program allowing to create a game of 32 or 52 cards (depending on the user's choice) then display the deck of cards in a pleasant way (for example [8 of spades], [Jack of diamonds]). Add a function allowing to mix the game then to display it. The excahnge of two cards can be done using a two-card excahnge function. To mix, we can use the Fischer-Yates (or Knuth) method whose pseudo-code is as follows:

For i from n-1 to 1:
j = random number from 0 to i
exchange t[i] and t[j]