TP 1

1 Circular buffers

The goal of this exercise is three folds :

- Implement a queue data structure through a circular buffer.
- Build a simple application program using a queue, and learn how to deal with multiple source files in C.
- Modify your C program into a C++ one to make use of templates for genericity.

These notions will be clarified in the course of the exercise.

In class we have already discussed the notion of stack, where pushes and pops are always performed at the top. A queue is similar in nature, the only difference being that the notion of “top” is replaced by both a front and a tail, and pushes are always performed at the tail while pops always occur at the front.

1) If necessary, get familiar (e.g. from Wikipedia `https://en.wikipedia.org/wiki/Queue_(abstract_data_type)`) with the notion of queue, and understand how it can be implemented using so-called circular buffers (`https://en.wikipedia.org/wiki/Circular_buffer`).

2) In the file `include/circular_buffer_queue.h`, located in the directory `TP1`, an API for a queue is proposed to you. The C struct itself has the form :

```c
struct Queue {
    size_t front;     // index of the first element in the queue
    size_t length;    // number of items presently in the queue
    size_t capacity;  // capacity of the queue (in nbr of items)
    size_t elem_size; // length in bytes of each item in the queue
    void   *data;     // address of the array
};
```

As you can observe, the type of the items in the queue is not prescribed (and hence not known by the compiler). Since C is a strongly typed language, we rely instead on the *size* (in bytes) of each item, which must be prescribed at the level of the queue initialization.

```c
struct Queue * queue_init(size_t elem_size, size_t capacity);
```

The enqueue and dequeue operations, since the type is unknown, have the form:

```c
void queue_enqueue(struct Queue *q, const void *src);
void queue_dequeue(struct Queue *q, void *dest);
```

The data will be copied from `src` into `data` or from `data` into `dest` using the function `memcpy` (found in the `<string.h>` header).

Your are asked to create a directory called `src` at the same level as the directory `include` and edit a file `src/circular_buffer_queue.c` implementing all the functions declared in the header `include/circular_buffer_queue.h`. You should take care of increasing the capacity of the buffer if the length of the queue reaches its capacity. For that purpose, implement a function

```
static void enlarge_queue_capacity(struct Queue *q);
```

That last function need not be part of the API (the user doesn't care), the reason why it is only declared and defined in the `.c` file. The keyword `static` implies that the function will be unknown to external compile units, like our `test_queue` here below.

3) Write a file `src/test_queue.c` that will contain a single `main` function which is expected to do the following. First initiate an empty queue `q` of integers, then iteratively pick random integers `p` (through the `rand` function in the `stdlib` library) and then enqueue `p` in `q` if `p` is even and instead dequeue one item (and do nothing with it) from `q` when `p` is odd. The number `n` of successive random integers to be picked should be given as a command line argument, and the main function should return the largest length `l_max` of the queue `q` during the whole process.

   In order to compile your program, you may use the command:
   `gcc src/circular_buffer_queue.c src/test_queue.c -I include -o test_queue`
   We have already encountered the `-o` (output file) gcc option. The `-I` option allows to add some directories to the include path for header files, here the `include` directory.

4) (*Bonus*) Could you compute from a theoretical point of view the expectation of `l_max` in terms of `n`, or at least an estimate of it ? If you can't do the maths you could try to guess the result by running multiple tests with different orders of magnitude for `n` and observing the graph of $\mathtt{n} \mapsto \mathtt{l\_max}$. Empirical methods are sometimes of great help.

5) The genericity obtained by *type erasure* (i.e. use of cast to `void *`) in C has a few drawbacks. One of them is that fewer optimizations are possible for the compiler, in particular when the data is a fundamental scalar type, because we must call the external `memcpy` function to perform simple operations such as read and write. The language C++ has a feature called *templates* that allows to write generic code without type erasure. Our queue structure could be transformed in C++ into[1]

```
template <typename T>
struct Queue {
    size_t front;      // index of the first element in the queue
    size_t length;     // number of items presently in the queue
    size_t capacity;   // capacity of the queue (in nbr of items)
    T      *data;      // address of the array
};
```

   Discover about the basics of templates online and adapt your code accordingly, in particular removing the use of `memcpy`. It is common to use the `.cpp` extension for C++ source files (but keep `.h` for headers), and to switch to a C++ compiler replace `gcc` by `g++`.

[1]Note the disappearance of the `elem_size` field, which is no longer needed when the type `T` is instanciated hence known to the compiler.

2 Bubble, insertion and merge sort

We are interested here in the problem of sorting a sequence of N arbitrary integers in increasing order. For the sequence of 10 integers e.g.

11	54	23	6	7	89	4	2	8	45

,

we wish to obtain as a result the sequence :

2	4	6	7	8	11	23	45	54	89

.

For that purpose, there exists a large number of sorting algorithms. In this exercise you will implement three among the simplest of them (we shall study an additional two that have theoretical importance in class).

For each sorting algorithm below, you are requested to write a function (and possibly additional subroutines if needed) implementing it. The function shall take as input :

- a `int*` representing the adress of the array of integers to be sorted,
- an `int` representing the number of elements in the array,

and will have no return value (it will thus modify directly its input - such routines are called *in place*). Once this is done, you will write a `main` function which

1) loops over N for $N = 10, 20, 50, 100, 200, 500, 1000$ and for each value of N:
    * generates an array of N random integers,
    * calls your sorting algorithm over it,
    * measures the execution time and records it.
2) writes the data (N versus exectution time) into a file suitable for gnuplot,
3) launches gnuplot through a `system()` call, using a logarithmic scale for both the axis.

2.1 Bubble sort

This simple algorithm and easy to understand and to implement, but as you will observe it is often the less efficient in terms of speed. Starting from the first element, if it is larger than the next one we swap them.

11	54	23	6	7	89	4	2	8	45

We next proceed accordingly with the second element, comparing it with the third.

11	23	54	6	7	89	4	2	8	45

And we go on until we reach the end of the sequence.

11	23	6	7	54	4	2	8	45	89

At this stage it is clear that the last element of the array is the greatest one, and it is therefore in final position. We repeat then the whole process on the restrictied sequence obtained by omitting the last element (here the first nine elements). At the end of this step the last two elements of the array are in final position.

11	6	7	23	4	2	8	45	54	89

This is repeated until the size of the restricted subsequence is one, which yields the sorted array

2	4	6	7	8	11	23	45	54	89

.

2.2 Insertion sort

This algorithm is often used to sort short arrays, as part of more involved sorting algorithms. It is based on the following strategy. Assume that for some k the first k elements of the sequence are in increasing order (here $k = 4$), but not necessarily in final position.

6	11	23	54	7	89	4	2	8	45

The goal is then to **insert** the element $k + 1$ th position in the correct position **within** that restricted sequence composed of the first k elements. For that prupose, we swap the elements in positions k and $k+1$ (if needed), then $k-1$ et k (if needed) etc, until the sequence composed of the first $k + 1$ elements is sorted (this happens as soon as one of the swaps is not needed).

6	7	11	23	54	89	4	2	8	45

One then proceeds to the next step, that is inserting the element in position $k + 2$ into the sequence composed of the first $k+1$ elements, etc. The algorithm starts by inserting the second element in the sequence composed of the first element only (and which is obviously ordered).

2.3 Merge sort

This is an optimal algorithm from the point of view of algorithmic complexity, if counting the number of comparisons and/or swaps. It is based on the very powerful idea of "divide and conquer". The algorithm is based one an intermediate routine called `merge` which we present first.

Subroutine MERGE That routine takes as input an `int*` representing an array of integers which we denote here by T, and three `int` which we denote here by p, q, r and such that $p \leq q < r$. It assumes that the subsequences $\mathrm{T}[p], \mathrm{T}[p+1], \ldots \mathrm{T}[q]$ one one hand, and $\mathrm{T}[q+1], \mathrm{T}[q+2], \ldots \mathrm{T}[r]$ on the other hand, are already sorted in increasing order each, and will output the whole sequence in increasing order.

The process goes as follows. First, we compare $\mathrm{T}[p]$ et $\mathrm{T}[q]$. We retain the smallest of the two, and we save it. Let's say it was $\mathrm{T}[p]$ here, we then compare $\mathrm{T}[p+1]$ et $\mathrm{T}[q]$, retain the smallest of the two, and save it for positioning in second place in the final sequence. Et caetera.

Here is an example at work, where the left most arrays are the ones that need to be merged, and the right most array is the (intermediate) merged array, one iteration after the other. The reyed cells are the ones considered for comparison at each step.

Iterate 1:	**6**	7	11	23		**2**	3	9	20		2							
Iterate 2:	**6**	7	11	23		2	**3**	9	20		2	3						
Iterate 3:	**6**	7	11	23		2	3	**9**	20		2	3	6					
Iterate 4:	6	**7**	11	23		2	3	**9**	20		2	3	6	7				
Iterate 5:	6	7	**11**	23		2	3	**9**	20		2	3	6	7	9			
Iterate 6:	6	7	**11**	23		2	3	9	**20**		2	3	6	7	9	11		
Iterate 7:	6	7	11	**23**		2	3	9	**20**		2	3	6	7	9	11	20	23

MERGE SORT algorithm Once that subroutine implemented, the merge sort can then be described in the following short pseudo-code form:

```
MergeSort(T, $p$, $r$)
if (p<r) {
    $q = \lfloor (p+r)/2 \rfloor$
    MergeSort(T, $p$, $q$)
    MergeSort(T, $q+1$, $r$)
    Merge(T, $p$, $q$, $r$)
}
```

In the above, $\lfloor n \rfloor$ denotes the integer part of n. Note the recursive nature of the algorithm. To sort the whole array, it suffices to call `MergeSort` with $p = 0$ and $r =$ length(T) -1.

Note: The `merge` cannot fully work in place (understanding this is part of the exercise, perhaps the most interesting part!). One possibility is to allocate space as needed inside that routine (but pay attention to avoid memory leaks by releasing in due time). A better solution (*more challenging*) is to pre-allocate an array S with the same size of T, initially copy T into it, and then ping-pong between T and S in the `MergeSort` recursive call (besides, it is actually possible and probably recommended to avoid recursivity complitely and replace it with a while loop).