Exercise 5a. (Automatically expanding simple queue, 1p)

The simplest way to implement a queue is to use an array as a storage for queued elements. In the enqueue() operation a new item is always put as the last element and the number of elements is incremented. In the dequeue() operation an item is always taken from the beginning of the array (index 0) and all other items are moved forward in the array. This is not the most efficient way, because data needs to be moved in each dequeue() operation. But anyway, it is very easy to understand, because it behaves like the real queue of persons in the bank for example (all people take one step forward, when one customer moves to get the service). We saw the data definition and basic principles of how operation functions of this kind of queue can be implemented. You can find this kind of working queue (queue.h) and a test program (main.cpp) from the Oma-portal. This queue uses fixed sized array, so that the program gives an error message, if you try to enqueue data when the queue is full.

It is possible to modify the queue to automatically expand its size when needed. Dynamic memory (e.g. new and delete –operations) is needed here. Start for example with the queue size of 5 (when the queue is initialized allocate space for five elements) and every time when more space is needed, automatically increase the size of the array by 5 for example. In order to increase the size you need to do the following tasks:

1. Create a new array with larger size
2. Copy all elements from the old (smaller) array to the new (larger) array
3. Release (delete) the old array from the memory
4. Switch to use the new array (copy address of the new array to the place of array pointer)
5. Update the new size information of the new array

To make testing easy, put an output like "size is increased and is now xx items" to the enqueue() operation, when size is increased to make it easier to follow the working of your program. Do not also use global variables in your solution, and don't ever decrease the size of the array in enqueue() operation.

The most important principle here is that expanding really is automatic. It is automatic for the user of the queue. The use of the queue is as easy as before. The only difference for the user is that enqueue() function never returns false, because enqueue() operation always succeeds (assuming that the operating system has infinite amount of memory available).

Download the program queue.h and main.cpp from the Oma portal and compile and run them so that you understand how they work. After that do all modifications needed to make the queue automatically expandable.

Remark 1. Do not modify the main program (main), because you still can use it to test your more advanced queue. This also means that you must not modify the prototypes of the operation functions of the queue. The implementation of operation functions of course needs modifications because we need to move to use dynamic array instead of fixed size array.

Extra Exercise 5b. (Automatically expanding circular buffer queue, 0.25p)

In the simple queue implementation data needs to be moved in each dequeue() operation. Therefore, the implementation is not very efficient. To improve the operation of the queue, implement the queue using the circular buffer technique presented in the lectures. Notice that this implementation must also be automatically expanding, i.e. when the space for the queue fills, more space will be allocated (this is not so easy task in this case).

Use the given (below) test application to measure the time it takes for the dequeue() operation. For a very high-performance system, this test application does not give accurate measurements. For the Metropolia Virtual Machine system, accuracy of the performance measurement is enough. Measure the time consumed in the dequeue() operation function in the original simple array and this circular queue implementations. Then calculate how many times better the circular implementation is compared to the array implementation. Ensure also that your program works by checking that the last item value is ok.

Remark 1. Efficient memory allocation would need also automatically shrinking mode. When there is about 20% free space in the queue, some memory should be released (not required in this exercise).

Remark 2. Parameter N is chosen for the Metropolia's virtual machine. It may need to be increased in higher performance environments.

```
/*
 * Performance measurement application for the queue.
 *
 * Not a very accurate measurement system if the dequeue() function is relative short
(compared to the
 * time consumed in for-loop)
 *
 * In order to run this program in Visual Studio 2012 select
 * Debug-Start-without-Debugging (or pressing Ctrl-F5)
 */

#include <iostream>
#include <stdlib.h>
#include <time.h>
#include "arrque.h"

using namespace std;

#define N 100000     // number of elements inserted/retrieved from the queue, must be even


int main() {
    Queue<int> queue;
    int        item;
    clock_t    tic, tac;
    double     duration;
    int        i;
```

```
	// first we play a little with the queue to test that it really works
    cout << "Fill the queue with " << N << " items\n";
	item = 0;
    for (i = 0; i < N; i++) {
		if (!queue.enqueue(item++)) {
			cerr << "Enqueue failed\n";
			exit(1);
		}
	}
    cout << "Remove half of them\n";
    for (i = 0; i < N/2; i++) {
		if (!queue.dequeue(item)) {
			cerr << "Dequeue failed\n";
			exit(1);
		}
	}
    cout << "Add then new " << N/2 << " items to the queue\n";
	item++;
    for (i = 0; i < N/2; i++) {
		if (!queue.enqueue(item++)) {
			cerr << "Enqueue failed\n";
			exit(1);
		}
	}

	// then we deque elements and measure the execution time
	cout << "Then dequeue them\n";
	tic = clock();
	for (i = 0; i < N; i++)
		queue.dequeue(item);
	tac = clock();
	cout << "Last item value " << item << " (should be " << N-1 << ")\n";
	cout << "(tic " << tic << ", tac " << tac << ")\n";

	duration = (double)(tac - tic) / CLOCKS_PER_SEC;
	cout << "\ndequeue took " << duration / (double)N*1e6  << " us to run\n";
}
```