# Homework 1

Due by: 18/5/23 23:59

- Make sure your submission includes all files (e.g., MSVS project files) and it can successfully be built and run on another computer.
- Your code should run in release/x64 (in MSVS) or "g++ -std=c++11 -o3" in Linux/WSL.
- Avoid command-line parameters and special compilation flags/configuration for your programs.
- Don't use any external libraries. Only Native C++11.

## Grading

- **Think HARD if and where synchronization is needed !!**
- Each exercise will be graded for
  - Correct functionality (behavior) - 60%
  - Efficient synchronization (only where/if needed) between producers and consumers – 40%

## Exercise 1: parallel bounded queue – 30 points

Implement a bounded (FIFO) queue with finite capacity. Its memory is allocated during queue creation. The queue capacity should not grow at any point in time.

1. Implement a C++ class for the bounded queue which is fully complied with the following interface.

```
// Pop the next element from the queue.
// If the buffer is empty, the calling thread waits until being notified of
// new elements in the queue
int pop();

// Push a new integer to the queue.
// If the buffer is full, the calling thread should wait until being notified //
that the queue is not full anymore
// v - the new integer to push into the queue.
void push(int v);
```

2. Implement a program to use the bounded queue implementation:
   a. Create 4 threads: 2 consumers (doing get) and 2 producers (doing put).
   b. Each thread calls push()/pop() in a finite loop (of same size).
   c. Once all treads complete their loop, the program terminates.
3. Validate that you see interleaved "print" messages by different threads on your screen.

### Exercise 2: parallel bounded queue with one producer /one consumer – 20 points

Implement another version of a bounded queue but with different requirements.

1. This bounded queue is always used by **only** 1 producer and 1 customer threads.
2. This bounded queue should be complied with the following interface:

```
// Pop the next element (integer value) from the queue.
// if the buffer is empty, ruturn_false. Otherwise, true.
bool pop(int &val);

// Push a new integer to the queue.
// if queue is full, return false. Otherwise, true.
// v - the new integer to push into the queue.
bool push(int v);
```

3. Pop includes a *val* parameter to return the value from the queue. Also, in case the queue is empty, it returns false. Otherwise, it returns true.
4. Push returns false if queue is full. Otherwise, return true.

Write a program to validate this bounded queue implementation. Each thread should complete a loop of push/get ops.

### Exercise 3: parallel unbounded queue with one producer /one consumer – 40 points

Implement an unbounded queue (meaning it can grow as needed).

1. Implementation should be based on pointer-based data structures (e.g., list), as opposed to arrays or vectors which are resized.
2. Assume it is always used by at most **only** 1 producer and 1 customer threads.
3. It should have the exact following interface:

```
bool pop(int &value); //  returns false   if  queue   is  empty

void push(int val); // v - the new integer to push into the queue.
```

4. The implementation should also include memory reclamation (free unused elements of the pointer-based data structure). The reclamation should be efficient (e.g., minimizing memory copies etc.)

Good luck!