1. Hacer un programa que dispare múltiples threads accediendo a datos compartidos sin sincronización. Ejecutarlo hasta mostrar el data race.
2. Implementar un semáforo utilizando **`mutex`** y **`condition_variable`**.
3. Implementar una cola sincronizada de un tamaño fijo de elementos con operaciones de get y put con variantes con límite temporal.

```
template<class T, size_t N>
  class Queue {
     public:
       void put(const T&);
       void get(T*);
       // variantes fancy
       bool try_put(const T&);
       template <class Rep, class Period>
         bool try_put_for(const T&, const duration<Rep,Period> &rel_time);
       template <class Clock, class Duration>
         bool try_put_until(const T&, const time_point<Clock,Duration> &abs_time);
       bool try_get(T*);
       template <class Rep, class Period>
         bool try_get_for(T*, const duration<Rep,Period> &rel_time);
       template <class Clock, class Duration>
         bool try_get_until(T*, const time_point<Clock,Duration> &abs_time);

  };
```

Hints: utilizar **`array<T,N>`** para almacenar los valores, un **`mutex`** y un par de **`condition_variable`**s para sincronización.

4. Hacer un programa que instancie una cola sincronizada del ejercicio anterior y varios threads productores y consumidores que efectúen operaciones put y get respectivamente sobre la cola.
5. Implementar una cola como la del ejercicio 3, pero que incluya un thread de dispatch de requests recibidos. Haga que la clase a implementar sea una clase template del tipo de request a recibir y procesar (dispatch).

```
template<class T, size_t QLEN_LOG2>
class ProcQueue {
    public:
        static const size_t QLEN = 1 << QLEN_LOG2;
        static const size_t QLEN_MASK = QLEN - 1;

        typedef void (*ProcFun)(const T &arg);

        void shutdown();  // para hacer un shutdown ordenado
```

```
        void put(ProcFun fun, const T &arg);

    private:
        // thread
        // contenedor para guardar QLEN Ts
        // sync objects

    // ...
};
```