```cpp
template<typename T>
class mutex_queue
{
public:
    mutex_queue (size_t capacity_)
        : capacity (capacity_)
    {
    }

    bool try_push (const T&);
    bool try_pop (T&);

private:
    std::mutex mutex;
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    size_t head { 0 }, tail { 0 };

    size_t next_index (size_t current) const {
        return (current + 1) % capacity;
    }
};
```

```cpp
bool try_push (const T& v)
{
    const std::lock_guard _ (mutex);

    size_t current_tail = tail;
    size_t next_tail = next_index (current_tail);

    if (next_tail == head)
        return false;

    data[current_tail] = v;
    tail = next_tail;
    return true;
}
```

```
bool try_pop (T& v)
{
    const std::lock_guard _ (mutex);

    size_t current_head = head;

    if (current_head == tail)
        return false;

    v = data[current_head];
    head = next_index (current_head);
    return true;
}
```

```cpp
template<typename T>
class mutex_queue
{
public:
    mutex_queue (size_t capacity_)
        : capacity (capacity_)
    {
    }

    bool try_push (const T&);
    bool try_pop (T&);

private:
    std::mutex mutex;
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    size_t head { 0 }, tail { 0 };

    size_t next_index (size_t current) const {
        return (current + 1) % capacity;
    }
};
```

```cpp
    bool try_push (const T& v)
    {
        const std::lock_guard _ (mutex);

        size_t current_tail = tail;
        size_t next_tail = next_index (current_tail);

        if (next_tail == head)
            return false;

        data[current_tail] = v;
        tail = next_tail;
        return true;
    }
```

```cpp
    bool try_pop (T& v)
    {
        const std::lock_guard _ (mutex);

        size_t current_head = head;

        if (current_head == tail)
            return false;

        v = data[current_head];
        head = next_index (current_head);
        return true;
    }
```

```cpp
template<queue_end queue_end, typename queue_type>
struct queue_thread
{
    queue_thread(queue_type &queue_, std::latch &start_latch_, size_t num_iterations)
        : queue(queue_), start_latch(start_latch_), iters(num_iterations)
    {}

    void run_async()
    {
        thread = std::thread([this]
        {
            start_latch.arrive_and_wait();


            stopwatch sw;



            if constexpr (queue_end == queue_end::producer)
            {
                //...
            }
            else
            {
                //...
            }

            res.duration = sw.get();


            run_result = res;
        });
    }

    queue_result join()
    {
        assert (thread.joinable());
        thread.join();
        return run_result;
    }

private:
    queue_type& queue;
    std::latch& start_latch;
    const size_t iters;
    std::thread thread;
    queue_result run_result;
};
```