Memory Ordering

- Relaxed:
  - Only guarantees atomicity
  - Ordering between the same *atomic* variable

- Acquire/release
  - Synchronise with each other
  - Work in pairs (store = release, load = acquire)
  - Allows some reordering
  - Writes before a release store become visible after an acquire load in another thread

- Sequential consistency
  - Acquire/release semantics
  - All threads agree on ordering (consistent with program order)

- Implemented like read/write locks per-cache-line

Memory Ordering

- Relaxed:
  - Only guarantees atomicity
  - Ordering between the same *atomic* variable
- Acquire/release
  - Synchronise with each other
  - Work in pairs (store = release, load = acquire)
  - Allows some reordering
  - Writes before a release store become visible after an acquire load in another thread
- Sequential consistency
  - Acquire/release semantics
  - All threads agree on ordering (consistent with program order)
- Implemented like read/write locks per-cache-line

```cpp
template<typename T>
class drow_queue_v3
{
public:
    drow_queue_v3 (size_t capacity_)
        : capacity (capacity_)
    {}

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

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

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