Safety: off
How not to shoot yourself in the foot with C++
atomics

Anthony Williams

Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

25th April 2015
Safety: off
How not to shoot yourself in the foot with C++ atomics
Safety: off
How not to shoot yourself in the foot with C++ atomics

- C++ Atomic types and operations
- Worked examples
- Guidelines
We use atomic operations rather than locks to improve performance. We therefore need to specify the aspect we care about:

- Throughput
- Latency
- Something else

It is vital to profile before and after changing to atomic operations.
Atomic types
Atomic types

- `std::atomic<T>` provides an atomic type that can store objects of type `T`.
  - `T` can be a built-in type, or a class type of any size
  - `T` must be *trivially copyable*
  - `compare_exchange_xxx` operations require that you can compare `T` objects with `memcmp`
  - `std::atomic<T>` may not be lock free — especially for large types

- `std::atomic_flag` provides a guaranteed-lock-free flag type.

- The Concurrency TS provides `atomic_shared_ptr` and `atomic_weak_ptr`.
atomic
Adjective
Meaning
Of or forming a single irreducible unit or component in a larger system.
Origin
Late 15th century: from Old French atome, via Latin from Greek atomos ‘indivisible’, based on a- ‘not’ + temnein ‘to cut’
Atomic Operations

General ops

load(), store(), exchange(),
compare_exchange_weak(),
compare_exchange_strong()
=

Arithmetic ops for atomic<Integral> and atomic<T*>:
fetch_add(), fetch_sub()
++, --, +=, -=

Bitwise ops for atomic<Integral>:
fetch_and(), fetch_or(), fetch_xor()
&, |, ^,

Flag ops for atomic_flag:
test_and_set(), clear()
### Atomic Operations

#### General ops
- `load()`, `store()`, `exchange()`,
- `compare_exchange_weak()`,
- `compare_exchange_strong()`

#### Arithmetic ops for `atomic<Integral>` and `atomic<T*>`
- `fetch_add()`, `fetch_sub()`
- `++`, `--`, `+=`, `-=`

#### Bitwise ops for `atomic<Integral>`
- `fetch_and()`, `fetch_or()`, `fetch_xor()`
- `&=`, `|=`, `^=`

#### Flag ops for `atomic_flag`
- `test_and_set()`, `clear()`

---

MAY NOT BE LOCK FREE

GUARANTEED LOCK FREE
6 values for the ordering on an operation:

- `memory_order_seq_cst` (the default)
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel` (RMW ops only)
- `memory_order_relaxed` (Experts only)
- `memory_order_consume` (Optimized form of `memory_order_acquire`, for special circumstances, for experts only)
All `memory_order_seq_cst` operations to all variables form a single total order.

Initially $x=0$, $y=0$

Store $x=1$  
Load $x==1$  
Load $y==1$

Load $y==1$  
Load $x==0$

Store $y=1$
A `memory_order_release` operation synchronizes with a `memory_order_acquire` operation that reads the value written.
Release/acquire non-synchronization

Unrelated reads do not synchronize.

Initially x=0, y=0

Store x=1 → Release

Load x==1 → Acquire

Load y==1 → Acquire

Store y=1 → Release

Load y==0

Load x==0
Relaxed atomics: anything can happen

Relaxed atomics can read out of order.

Initially $x=0, y=0$

- Store $x=1$
- Store $y=1$
- Load $y==1$
- Load $x==0$

Release

Relaxed
Fences
C++ has two kinds of fences:

- `std::atomic_thread_fence` ⇒ Used for synchronizing between threads
- `std::atomic_signal_fence` ⇒ Used for synchronizing between a thread and a signal handler in that thread
Fences

Fences in C++ effectively modify the ordering constraints on neighbouring atomic operations rather than providing any direct ordering constraints themselves.

```cpp
x.load(memory_order_relaxed);
atomic_thread_fence(memory_order_acquire);
⇒ x.load(memory_order_acquire);

atomic_thread_fence(memory_order_release);
x.store(memory_order_relaxed);
⇒ x.store(memory_order_release);
```

Anthony Williams
Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

Safety: off How not to shoot yourself in the foot with C++ atomics
Fences

memory_order_acq_rel fences behave as both
memory_order_acquire and memory_order_release
fences.

memory_order_seq_cst fences are special: they form part
of the total order of memory_order_seq_cst operations, and
can therefore enforce orderings beyond the direct pairwise
acquire-release orderings. If you’re relying on this, you’ve
probably done something wrong.
Lock-free examples
Obstruction free (**Weakest guarantee**)
If all other threads are paused then any given thread will complete its operation in a bounded number of steps.

Lock free (**Most common guarantee**)
If multiple threads are operating on a data structure then after a bounded number of steps **one** of them will complete its operation.

Wait free (**Strongest guarantee**)
**Every** thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure.
Why Queues?

- Core facility for communication between threads
- Many types of queue:
  - SPSC / MPSC / MPMC / SPMC
  - bounded / unbounded
  - FIFO / priority / unordered
  - intrusive / non-intrusive
- Good for demonstrating issues
Lock-based queue

Lock-based, unbounded, MPMC, FIFO queue

template<typename T>
class queue1{

private:
    std::mutex m;
    std::condition_variable c;
    std::queue<T> q;
};
void push_back(T t) {
    {
        std::lock_guard<std::mutex> guard(m);
        q.push(t);
    }
    c.notify_one();
}
T pop_front(){
    std::unique_lock<std::mutex> guard(m);
    c.wait(guard, [=]{return !q.empty();});
    auto ret=q.front();
    q.pop();
    return ret;
}
Lock-free SPSC FIFO queue

Let’s start simple with our lock-free queue:

- One producer thread
- One consumer thread
- Bounded, so no memory allocation
- Assume $T$ has a noexcept copy constructor
template<typename T, unsigned buffer_size=42>
class queue2{

  unsigned pop_pos{0};
  typedef typename std::aligned_storage<
      sizeof(T), alignof(T)>::type storage_type;
  struct entry{
    std::atomic<bool> initialized{false};
    storage_type storage;
  };
  entry buffer[buffer_size];
Lock-free SPSC FIFO queue: pushing

template<typename T,unsigned buffer_size=42>
class queue2{
    unsigned push_pos{0};

public:
    void push_back(T t) {
        unsigned my_pos=push_pos;
        push_pos=(push_pos+1)%buffer_size;
        auto& my_entry=buffer[my_pos];
        while (my_entry.initialized.load()) {}
        new(&my_entry.storage) T(t);
        my_entry.initialized.store(true);
    }
};
Aside: avoid busy waits

Busy waits are to be avoided: they consume processor power for no purpose.

It is acceptable for a compare_exchange_weak loop to have no body: we’re hoping to avoid spinning more than a couple of times.

If you need to wait, use a proper wait mechanism such as std::condition_variable.
Lock-free SPSC FIFO queue: popping

unsigned pop_pos{0};

public:

T pop_front() {
    while (!buffer[pop_pos].initialized.load()) {}{ }
    auto ptr = static_cast<T*>(
        static_cast<void*>(&buffer[pop_pos].storage));
    auto ret = *ptr;
    ptr->~T();
    buffer[pop_pos].initialized.store(false);
    pop_pos = (pop_pos + 1) % buffer_size;
    return ret;
}
Now let’s try and make an MPSC FIFO based on `queue2`. A naive attempt would be to make `push_pos` atomic:

```cpp
std::atomic<unsigned> push_pos{0};
void push_back(T t){
    unsigned my_pos=push_pos.load();
    while(!push_pos.compare_exchange_weak(
        my_pos, (my_pos+1)%buffer_size)){}
```
Now let’s try and make an MPSC FIFO based on `queue2`. A naive attempt would be to make `push_pos` atomic:

```cpp
std::atomic<unsigned> push_pos{0};

void push_back(T t) {
    unsigned my_pos = push_pos.load();
    while (!push_pos.compare_exchange_weak(
        my_pos, (my_pos + 1) % buffer_size)) {}
}
```

This is still broken.
Broken Lock-free MPSC FIFO queue

1. Queue is empty, `push_pos` is 0.
2. Thread 1 calls `push_back`, gets `my_pos` is 0, and increments `push_pos` to 1.
3. Thread 1 checks the cell is empty.
4. Thread 1 gets suspended by scheduler.
5. Thread 2 calls `push_back buffer_size-1` times, so `push_pos` loops round to 0.
6. Thread 2 calls `push_back` again. Thread 2 gets `my_pos` of 0, and sets `push_pos` to 1.
7. Thread 2 checks that the cell is empty.
8. Thread 2 populates the cell.
9. Thread 1 is woken by the scheduler.
10. Thread 1 populates the cell. **DATA RACE.**
The problem on the previous slide only occurs if the buffer is full. Can we prevent this by checking for a full buffer?

```cpp
std::atomic<unsigned> size{0};

void push_back(T t){
    unsigned old_size=size.load();
    for(;;){
        if(old_size==buffer_size)
            old_size=size.load();
        else if(size.compare_exchange_weak(old_size,old_size+1))
            break;
    }
}
```
Not-lock-free MPSC queue

Our queue is now not even obstruction free.

1. Queue is empty. `push_pos` is 0. `pop_pos` is 0.
2. Thread 1 calls `push_back` and increases size.
3. Thread 1 gets `my_pos` as 0, increments `push_pos`
4. Thread 1 is suspended by scheduler.
5. Thread 2 pushes `buffer_size-1` entries.
6. Thread 2 tries to push another entry, but `size==buffer_size`
7. Thread 3 calls `pop_front`, but `pop_pos` is 0 and the entry at 0 hasn’t been filled in.
8. All threads now stalled waiting for thread 1.
Can we fix this? First we need to identify the problem.

Pushing a value consists of 3 steps:

1. Find a free slot in the buffer
2. Construct the pushed value in the slot
3. Mark the value as available to the consuming thread
Fixing queue4

Can we fix this? First we need to identify the problem.

Pushing a value consists of 3 steps:

1. Find a free slot in the buffer
2. Construct the pushed value in the slot
3. Mark the value as available to the consuming thread

We need to publish in step 3, rather than step 1.
Fixing queue4

We need to separate the buffer ordering from the queue ordering, so we need to redo steps 1 and 3.

1 Hunt the buffer for a free slot
2 Construct the pushed value in the slot
3 Link that entry into the queue
Fixing queue4: Linking entries into the queue

Let’s use a linked list — that’s easy, isn’t it? Just push entries on the tail, and pop them off the head.

We still have two locations to update: the next pointer in the previous node, and the tail pointer.

Having the push thread do them in either order can lead to a race.
Fixing queue4: Linking entries into the queue

Answer: update the next pointers from the (one and only) pop thread.

In push_back we record the previous tail entry:

```cpp
class MyQueue {
public:
    void push_back(T t) {
        auto my_entry = allocate_entry();
        new(&my_entry->storage) T(t);
        my_entry->next = nullptr;
        my_entry->prev = tail.load();
        while (!tail.compare_exchange_weak(
            my_entry->prev, my_entry)) {} 
    }
};
```

Anthony Williams
Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

Safety: off How not to shoot yourself in the foot with C++ atomics
Fixing queue4: Linking entries into the queue

In `pop_front`, if there is no `next` value for the current entry we can start at the `tail` and fill them all in:

```cpp
T pop_front()
{
    entry* old_head=head;
    while(!old_head)
        old_head=chase_tail();
    head=old_head->next;
    auto ptr=static_cast<T*>(
        static_cast<void*>(
            &old_head->storage));
    auto ret=*ptr;
    ptr->~T();
    recycle_node(old_head);
    return ret;
}
```
Fixing queue4: Linking entries into the queue

```c++
entry* chase_tail(){
    entry* next=tail.exchange(nullptr);
    if(!next)
        return nullptr;
    while(next->prev)
        next->prev->next=next;
    next=next->prev;
}
return next;
}
```
A lock-free? MPSC FIFO queue

Our queue is now obstruction free, but is it lock-free or wait-free?

- If the queue is full then we have to wait.
  ⇒ Use a lock-free allocator instead of a fixed buffer.
- If the queue is empty then we have to wait.
- Otherwise, only waiting is in compare-exchange loops
  ⇒ No upper limit on loops, so cannot be wait-free.
- `compare_exchange_weak` can fail spuriously
  ⇒ If it does then there is no bound to the number of steps.

**Lock-free vs obstruction-free strictly depends on the compare_exchange_weak implementation.**
Performance: Cache Ping-Pong
Cache Ping-Pong is where a cacheline is continuously shuttled back and forth between two processors. This occurs when two threads are accessing either:

- **the same** atomic variable
- **different** variables on the same cache line

This can have a big performance impact, because transferring cache lines is slow.
Cache Ping-Pong in `queue5`

`queue5` can be accessed by many threads in `push_back`, and one more thread in `pop_front` simultaneously.

```cpp
std::atomic<unsigned> push_hint{0};
entry* head{nullptr};
std::atomic<entry*> tail{nullptr};
entry buffer[buffer_size];
```

`head` and `tail` are adjacent, but accessed by different threads ⇒ unnecessary cache ping-pong.
There are many examples in this data structure.
The solution to cache ping-pong is to put data on different cache lines by adding padding. This trades memory space for performance.

```cpp
std::atomic<unsigned> push_hint{0};
char padding1[padding_size];
entry* head{nullptr};
char padding2[padding_size];
std::atomic<entry*> tail{nullptr};
char padding3[padding_size];
entry buffer[buffer_size];
```
Cache Ping-Pong avoidance in \texttt{queue5}

Times for 10,000,000 pushes of an integer on each of 3 threads, with another thread popping all 30,000,000 entries.

<table>
<thead>
<tr>
<th>Run</th>
<th>No padding</th>
<th>With padding</th>
<th>With Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.4s</td>
<td>11.4s</td>
<td>27.1s</td>
</tr>
<tr>
<td>2</td>
<td>22.4s</td>
<td>9.8s</td>
<td>17.8s</td>
</tr>
<tr>
<td>3</td>
<td>22.1s</td>
<td>15.4s</td>
<td>25.4s</td>
</tr>
<tr>
<td>4</td>
<td>14.3s</td>
<td>9.0s</td>
<td>24.3s</td>
</tr>
<tr>
<td>Mean</td>
<td>21.3s</td>
<td>11.4s</td>
<td>23.7s</td>
</tr>
</tbody>
</table>
Performance: Memory Ordering Constraints

All the examples so far have used the default ordering constraint: `memory_order_seq_cst`.

You should use `memory_order_seq_cst` unless you have a strong reason not to.
Performance: Memory Ordering Constraints

For x86, only `store` is affected by the memory order, but for architectures like POWER and ARM with weaker default synchronization, all operations can be affected.

You **must** test on a weakly-ordered system like POWER or ARM if you’re using anything other than `memory_order_seq_cst`. 
Stacks
A stack is a simpler data structure than a queue. It’s great for examples, but bad for real use, as all threads are contending to access the top-of-stack.

I’m going to use it to demonstrate a specific problem: the **A-B-A** problem.
A simple MPSC stack: pushing

template<typename T>
class stack1{
    struct node{
        T val;
        node* next;
    };
    std::atomic<node*> head{nullptr};
public:
    void push(T newval) {
        auto newnode = new node{newval, head.load()};
        while (!head.compare_exchange_weak(newnode->next, newnode)) {}
    }
}
A simple MPSC stack: popping

T pop() {
    auto old_head=head.load();
    for(;;){
        if(!old_head)
            old_head=head.load();
        else if(head.compare_exchange_strong(
            old_head,old_head->next)){
            auto res=old_head->val;
            delete old_head;
            return res;
        }
    }
}
A simple stack: A-B-A issues

Why is this a single-consumer stack?
Answer: the A-B-A problem.
A simple stack: A-B-A issues

1. Thread 1 calls `pop()`
2. Thread 1 reads `head` into `old_head` (A)
3. Thread 1 reads `old_head->next`
4. Thread 1 is suspended
5. Thread 2 pops two items, `head` has new value (B)
6. Thread 2 pushes two items
7. **Second new item is given address of old item**, `head` has original value (A)
8. Thread 1 resumes and calls `compare_exchange_strong`, which **succeeds because the address is the same**
9. Stack is now corrupt
The setup:

1. A value changes from A to B and back to A,
2. Other aspects of the data structure have changed, and
3. A thread makes a change based on the first time the value was A that is inconsistent with the new state of the data structure.

This most commonly happens where the value is a pointer.
A-B-A: Solutions

Do not allow a variable to return to its previous value while a thread can do something based on the old value.

- Use a change count as part of the variable:
  ```cpp
  struct Value { T* ptr; unsigned count; };
  std::atomic<Value> v;
  ```

- Ensure that objects are not recycled when still accessible, so A-B-A never happens.
  ⇒ Reference count the objects, e.g. with std::shared_ptr and atomic_shared_ptr or use hazard pointers, or something similar.
Reference Counting
Reference Counting

How hard can it be?

(Actually, quite hard)
Basic reference counting

Basic reference counting like `std::shared_ptr` is relatively simple.
⇒ Use an atomic counter, with `fetch_add()` and `fetch_sub()` to adjust the count.
Basic reference counting like `std::shared_ptr` is relatively simple.

⇒ Use an atomic counter, with `fetch_add()` and `fetch_sub()` to adjust the count.

This assumes that you have a reference to the count (a `std::shared_ptr` instance) which your thread has exclusive access to (for now).
Basic reference counting

```cpp
struct refcount{
    std::atomic<unsigned> count{1};
    void add_ref(){
        count.fetch_add(1);
    }
    bool release_ref(){
        if(count.fetch_sub(1)==1){
            return true;
        }
        return false;
    }
};
```
We can even put ordering constraints in place:

```cpp
void add_ref(){
    count.fetch_add(1, memory_order_relaxed);
}

bool release_ref(){
    if(count.fetch_sub(1, memory_order_release) == 1){
        std::atomic_thread_fence(
            memory_order_acquire);
        return true;
    }
    return false;
}
```
Advanced reference counting

What if we can’t guarantee that we have exclusive access to our instance pointer (e.g. a global `std::shared_ptr`)?

We need to synchronize somehow and avoid data races and A-B-A issues.
Advanced reference counting

What if we can’t guarantee that we have exclusive access to our instance pointer (e.g. a global `std::shared_ptr`)?

We need to synchronize somehow and avoid data races and A-B-A issues.

One way is to count the threads currently accessing each instance.
Advanced reference counting

Wrap our global reference in a type which holds the pointer to the shared object and the local access count:

class atomic_ref_count_ptr{
    struct counted_ptr{
        refcount* p;
        unsigned count;
    };
    std::atomic<counted_ptr> ptr;
public:
    ref_count_ptr load();
    void store(ref_count_ptr newval);
};
Advanced reference counting: \texttt{load()}

1. Read the value
2. Increase the local count
3. Increase the “proper” reference count
4. Decrease the local count if we’re still pointing to the same object
5. If not, release a reference
Advanced reference counting: `load()`

```cpp
ref_count_ptr atomic_ref_count_ptr::load() {
    counted_ptr local = ptr.load();
    while (local.p && !ptr.compare_exchange_weak(
        local, counted_ptr{local.p, local.count + 1}))) {}
    if (!local.p) return ref_count_ptr(nullptr);

    ref_count_ptr res(local.p);

    counted_ptr newlocal = ptr.load();
    while ((newlocal.p == local.p) &&
        !ptr.compare_exchange_weak(
            newlocal,
            counted_ptr{newlocal.p, newlocal.count - 1}))) {}
    if (newlocal.p != local.p)
        local.p->release_ref();
    return res;
}
```
Advanced reference counting: `store()`

1. Add an external reference to the new value
2. Store the new value and fetch the old one
3. Release the external references on the old one
4. If that was the last reference, delete it
Advanced reference counting: \texttt{store}()

```cpp
void atomic_ref_count_ptr::store(
    ref_count_ptr newval){
    refcount* newptr=newval.release();
    if(newptr) newptr->add_external();
    counted_ptr old_ptr=ptr.exchange(
        counted_ptr{newptr,0});
    if(old_ptr.p &&
        old_ptr.p->release_external(old_ptr.count))
        delete old_ptr.p;
}
```
Advanced reference counting: split counters

To handle the external counts we need a new `refcount` class:

```cpp
struct refcount{
    struct counters{
        unsigned internal;
        unsigned external;
    };
    std::atomic<counters> counts;
    void add_ref();
    bool release_ref();
    bool release_external(unsigned count);
    void add_external();
};
```
Advanced reference counting: split counters

The use of `std::atomic<counters>` makes the functions a bit more complex: everything is now a CAS loop.

```cpp
void refcount::add_ref()
{
    counters old=counts.load();
    while(!counts.compare_exchange_weak(
        old,
        counters{old.internal+1,old.external})){}
}

bool refcount::release_ref()
{
    counters old=counts.load();
    while(!counts.compare_exchange_weak(
        old,
        counters{old.internal-1,old.external})){}
    return (!old.external && (old.internal==1));
}
```
void refcount::add_external() {
    counters old = counts.load();
    while (!counts.compare_exchange_weak(
        old,
        counters{old.internal, old.external + 1})){}
}

bool refcount::release_external(unsigned count) {
    counters old = counts.load();
    while (!counts.compare_exchange_weak(
        old,
        counters{
            old.internal + count - 1, old.external - 1})){}
    return (old.external == 1) &&
           (old.internal == (1 - count));
}
### Advanced reference counting: example

<table>
<thead>
<tr>
<th>Step</th>
<th>Internal count</th>
<th>External count</th>
<th>Local count</th>
</tr>
</thead>
<tbody>
<tr>
<td>One non-atomic ptr</td>
<td>1</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Store in atomic ptr</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Destroy original non-atomic ptr</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>One thread starts load()</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Second thread starts load()</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Another thread calls store() and changes atomic ptr value</strong></td>
<td>1</td>
<td>1</td>
<td>(store-local == 2)</td>
</tr>
<tr>
<td>One load thread makes non-atomic ptr</td>
<td>2</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Second load thread makes non-atomic ptr</td>
<td>3</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>One thread finishes load()</td>
<td>2</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Second thread finishes load()</td>
<td>1</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First non-atomic ptr destroyed</td>
<td>0</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Second non-atomic ptr destroyed</td>
<td>-1</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Store thread calls release_external() and destroys object</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First non-atomic ptr destroyed</td>
<td>0</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>Store thread calls release_external()</td>
<td>1</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Second non-atomic ptr destroyed and destroys object</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Guidelines
Guidelines

- Don’t use atomics unless you have to
- Profile before and after
- Test on a weakly-ordered architecture such as POWER or ARM
- Don’t use atomics unless you **really** have to
Guidelines

Think in transactions
Do work off to the side and commit with a single atomic operation.

Split big operations
If the operation is too big to do in one step, split it into smaller steps that retain the data structure invariants.

Limit use cases
Restrict the permitted concurrency levels where possible to reduce implementation complexity.

Watch out for ABA problems
These require the circumstances to align just so, but will destroy your data structure when they happen. They can be easily missed in testing.

Anthony Williams Just Software Solutions Ltd http://www.justsoftwaresolutions.co.uk
Safety: off How not to shoot yourself in the foot with C++ atomics
Guidelines

Avoid cache ping pong
Add padding between variables that are accessed from different threads. Try and avoid too many threads accessing the same variable.

Stick to `memory_order_seq_cst`
Unless you really know what you’re doing, and really need the performance gain, stick to the default `memory_order_seq_cst`. Anything else can be a nightmare to prove correct.

Package things up
Wrap atomic operations with types that only expose the desired functionality, to clarify the user code and hide the complexity.
Guidelines

Aim for lock-free

Aim for your code to be at least *obstruction-free*, and preferably *lock-free*. Leave *wait-free* for those rare circumstances where you need it.

Avoid busy waits

If you’re actually waiting (as opposed to spinning on a `compare_exchange_weak` operation), use a proper wait mechanism.
Questions?
just::thread provides a complete implementation of the C++14 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX.

Just::Thread Pro gives you actors, concurrent hash maps, concurrent queues and synchronized values.
C++ Concurrency in Action: Practical Multithreading
http://stdthread.com/book
Safety: off How not to shoot yourself in the foot with C++ atomics