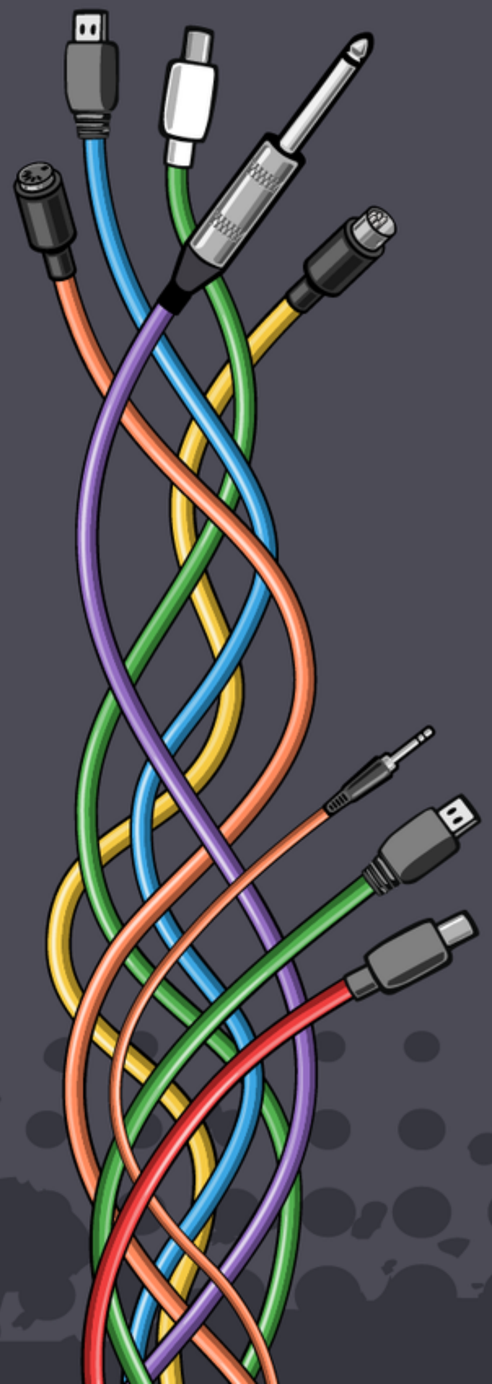


DEMYSTIFYING STD::MEMORY_ORDER

TIMUR DOUMLER

ADC25 Bristol

Lock-free programming

- One "real-time" thread that must not block
  (audio processing)
- One or more non-"real-time" threads
  (GUI rendering, MIDI I/O, file I/O, ...)
- Data that must be shared between those threads

Lock-free programming

- One "real-time" thread that must not block
  (audio processing)
- One or more non-"real-time" threads
  (GUI rendering, MIDI I/O, file I/O, ...)
- Data that must be shared between those threads
- Lock-free algorithms (CAS loop, double buffering, RCU, SeqLock...)
- Lock-free data structures (queue, stack, linked list...)

`std::atomic<T>`

std::atomic<T>

```
// atomic load operations
load

// atomic store operations
store
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add
fetch_sub

fetch_and
fetch_or
fetch_xor
```

std::atomic<T>

```
// atomic load operations
load, operator T

// atomic store operations
store, operator =
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add, operator +=, operator ++
fetch_sub, operator -=, operator --

fetch_and, operator &=
fetch_or,  operator |=
fetch_xor, operator ^=
```

std::atomic<T>

```
// atomic load operations
load, operator T

// atomic store operations
store, operator =
```

```
struct X {
  std::atomic<int> a;

  void f(int b) {
    int a_now = a.load();
    a.store(a_now + b);
  }
}
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add, operator +=, operator ++
fetch_sub, operator -=, operator --

fetch_and, operator &=
fetch_or,  operator |=
fetch_xor, operator ^=
```

std::atomic<T>

```
// atomic load operations
load, operator T

// atomic store operations
store, operator =
```

```
struct X {
  std::atomic<int> a;

  void f(int b) {
    int a_now = a;
    a = a_now + b;
  }
}
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add, operator +=, operator ++
fetch_sub, operator -=, operator --

fetch_and, operator &=
fetch_or,  operator |=
fetch_xor, operator ^=
```

std::atomic<T>

```
// atomic load operations
load, operator T

// atomic store operations
store, operator =
```

```
struct X {
  std::atomic<int> a;

  void g() {
    a.fetch_add(1);
  }
}
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add, operator +=, operator ++
fetch_sub, operator -=, operator --

fetch_and, operator &=
fetch_or,  operator |=
fetch_xor, operator ^=
```

std::atomic<T>

```
// atomic load operations
load, operator T

// atomic store operations
store, operator =
```

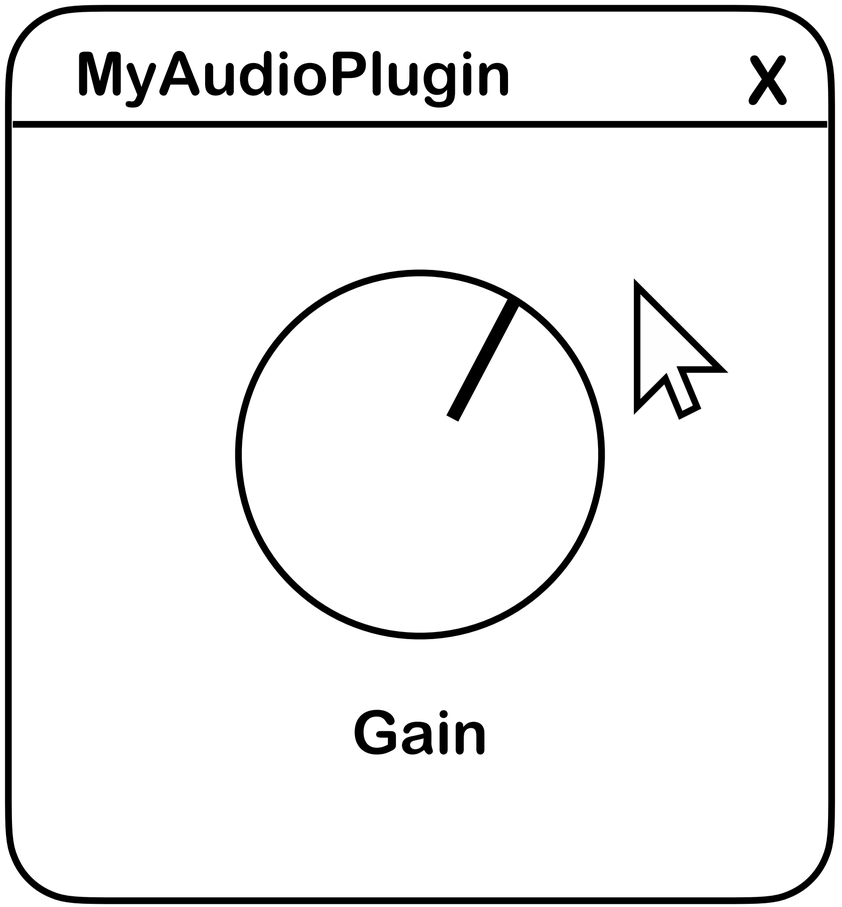
```
struct X {
  std::atomic<int> a;

  void g() {
    a++;
  }
}
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add, operator +=, operator ++
fetch_sub, operator -=, operator --

fetch_and, operator &=
fetch_or,  operator |=
fetch_xor, operator ^=
```

std::atomic<T>

```
// atomic load operations
load, operator T

// atomic store operations
store, operator =
```

```
// atomic read-modify-write operations
exchange
compare_exchange_strong
compare_exchange_weak

fetch_add, operator +=, operator ++
fetch_sub, operator -=, operator --

fetch_and, operator &=
fetch_or,  operator |=
fetch_xor, operator ^=
```


MyAudioPlugin
X
Gain

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  float gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      // Thread 1
      gain = knob.getValue();
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      // Thread 2
      sample *= gain;
  }
};
```

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  float gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      // Thread 1
      gain = knob.getValue();
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      // Thread 2
      sample *= gain;
  }
};
```

Data race
==
undefined behaviour

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  std::atomic<float> gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      // Thread 1
      gain = knob.getValue();  // OK, but not recommended
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      // Thread 2
      sample *= gain;  // OK, but not recommended
  }
};
```

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  std::atomic<float> gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      // Thread 1
      gain.store(knob.getValue());  // Better
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      // Thread 2
      sample *= gain.load();  // Better
  }
};
```

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  std::atomic<float> gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      // Thread 1
      gain.store(knob.getValue(), std::memory_order::relaxed);  // Optimal
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      // Thread 2
      sample *= gain.load(std::memory_order::relaxed);  // Optimal
  }
};
```

`std::memory_order`

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  float gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      gain = knob.getValue();
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      sample *= gain;
  }
};
```

Why is a data race undefined behaviour?

Why is a data race undefined behaviour?

Why is a data race undefined behaviour?

- **Compiler optimisations** can reorder or elide any operations
  as long as the observable behaviour of the program is the same,
  assuming that no undefined behaviour happens

```
extern int flag;

void process(float* data, int n) {
  for (int i = 0; i < n; ++i) {
    if (flag != 0) {
      data[i] = 1.0;
    }
  }
}
```

```
extern int flag;

void process(float* data, int n) {
  for (int i = 0; i < n; ++i) {
    if (flag != 0) {
      data[i] = 1.0;
    }
  }
}
```

```
extern int flag;

void process(float* data, int n) {
  if (flag == 0)
    return;

  for (int i = 0; i < n; ++i)
    data[i] = 1.0;
}
```

```
volatile extern int flag;

void process(float* data, int n) {
  for (int i = 0; i < n; ++i) {
    if (flag != 0) {
      data[i] = 1.0;
    }
  }
}
```

Why is a data race undefined behaviour?

- **Compiler optimisations** can reorder or elide any operations
  as long as the observable behaviour of the program is the same,
  assuming that no undefined behaviour happens

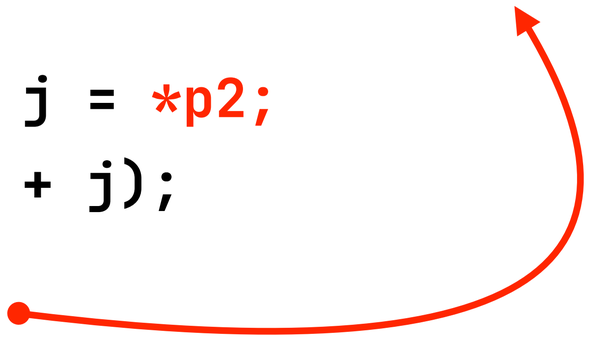
Why is a data race undefined behaviour?

- **Compiler optimisations** can reorder or elide any operations as long as the observable behaviour of the program is the same, assuming that no undefined behaviour happens
- **Hardware optimisations** (instruction reordering, pipelining, store buffering, etc.) can modify your program inside the CPU

```
int i = *p1;
int j = *p2;
f(i + j);

// code that does not use i or j...
```

```
int i = *p1;
int j = *p2;  // not in cache
f(i + j);

// code that does not use i or j...
```

```
int i = *p1;

// code that does not use i or j...

int j = *p2;
f(i + j);
```

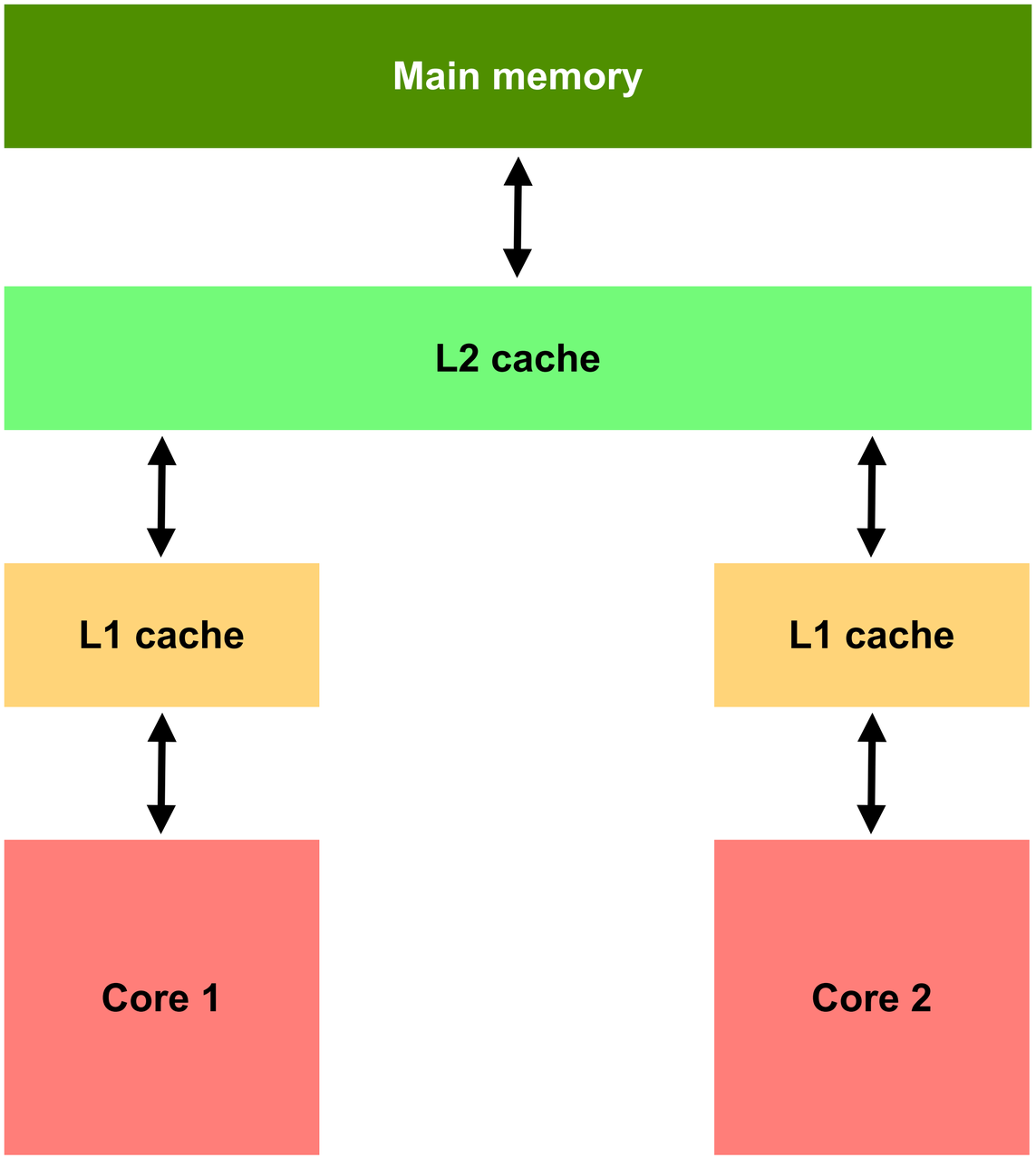


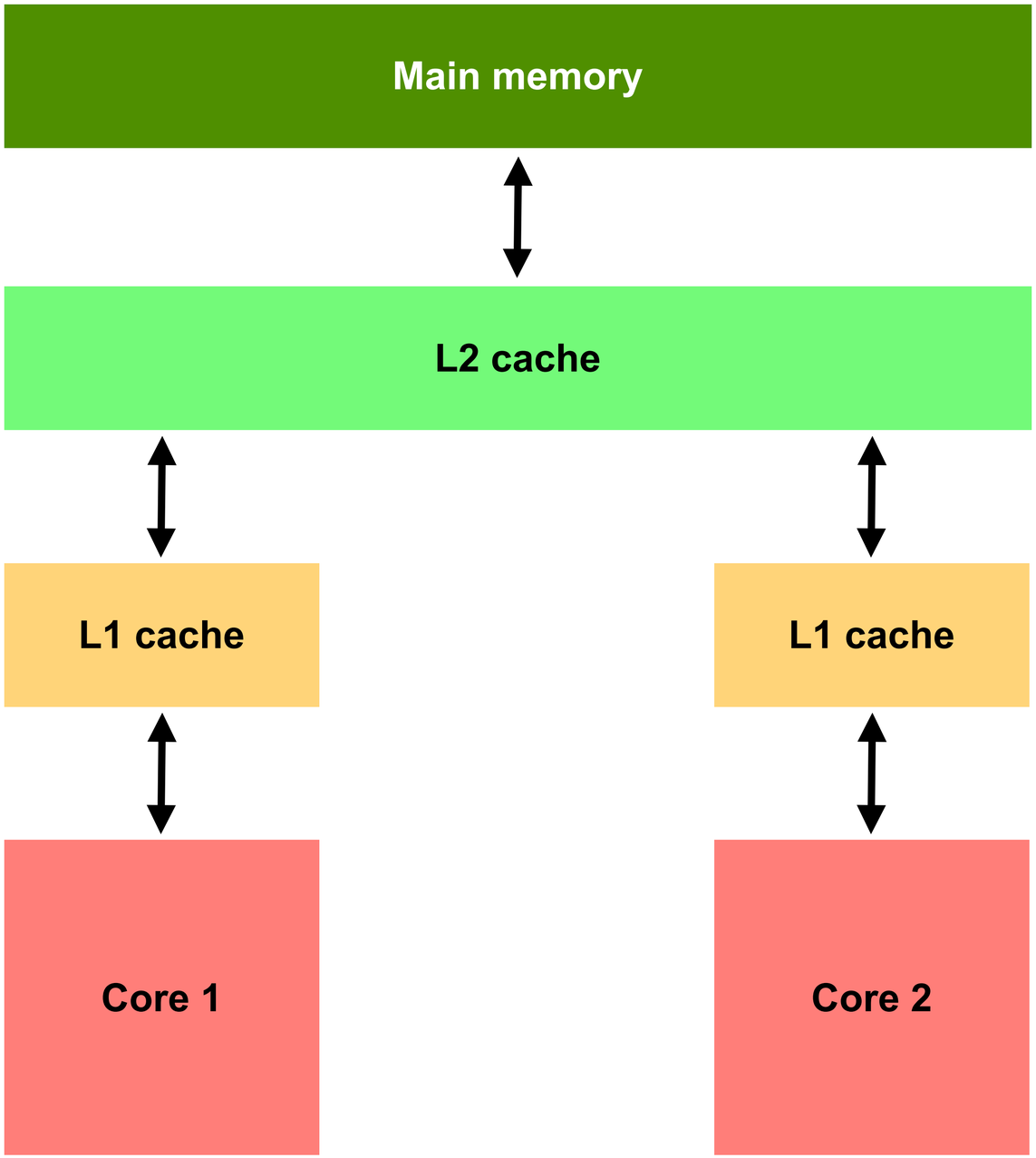
Why is a data race undefined behaviour?

- **Compiler optimisations** can reorder or elide any operations as long as the observable behaviour of the program is the same, assuming that no undefined behaviour happens
- **Hardware optimisations** (instruction reordering, pipelining, store buffering, etc.) can modify your program inside the CPU

Why is a data race undefined behaviour?

- **Compiler optimisations** can reorder or elide any operations as long as the observable behaviour of the program is the same, assuming that no undefined behaviour happens
- **Hardware optimisations** (instruction reordering, pipelining, store buffering, etc.) can modify your program inside the CPU
- **Multi-core cache coherence protocols** can cause different threads to observe operations happening in a different order or not at all


Main memory
L2 cache
L1 cache
L1 cache
Core 1
Core 2



Cache coherence protocols (MESI, MOESI, ...) ensure per-address consistency of each individual memory location, but not order of operations on different memory locations!

```
volatile int data = 0;
volatile bool ready = false;

int main() {
  std::jthread t1([] {
    data = 42;
    ready = true;
  });

  std::jthread t2([] {
    while (!ready) /* spin */;
    int x = data;
    assert(x == 42);
  });
}
```

```
volatile int data = 0;
volatile bool ready = false;

int main() {
  std::jthread t1([] {
    data = 42;
    ready = true;  // ready = true may become visible to t2 before data = 42!
  });

  std::jthread t2([] {
    while (!ready) /* spin */;
    int x = data;
    assert(x == 42);
  });
}
```

Why is a data race undefined behaviour?

- **Compiler optimisations** can reorder or elide any operations as long as the observable behaviour of the program is the same, assuming that no undefined behaviour happens
- **Hardware optimisations** (instruction reordering, pipelining, store buffering, etc.) can modify your program inside the CPU
- **Multi-core cache coherence protocols** can cause different threads to observe operations happening in a different order or not at all

`std::memory_order`

```
namespace std {
  enum class memory_order {
    relaxed,
    consume,
    acquire,
    release,
    acq_rel,
    seq_cst
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,
    consume, // does not work, deprecated in C++26 - never use
    acquire,
    release,
    acq_rel,
    seq_cst
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,
    acquire,
    release,
    acq_rel,
    seq_cst
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,
    release,
    acq_rel,
    seq_cst
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst
  };
}
```

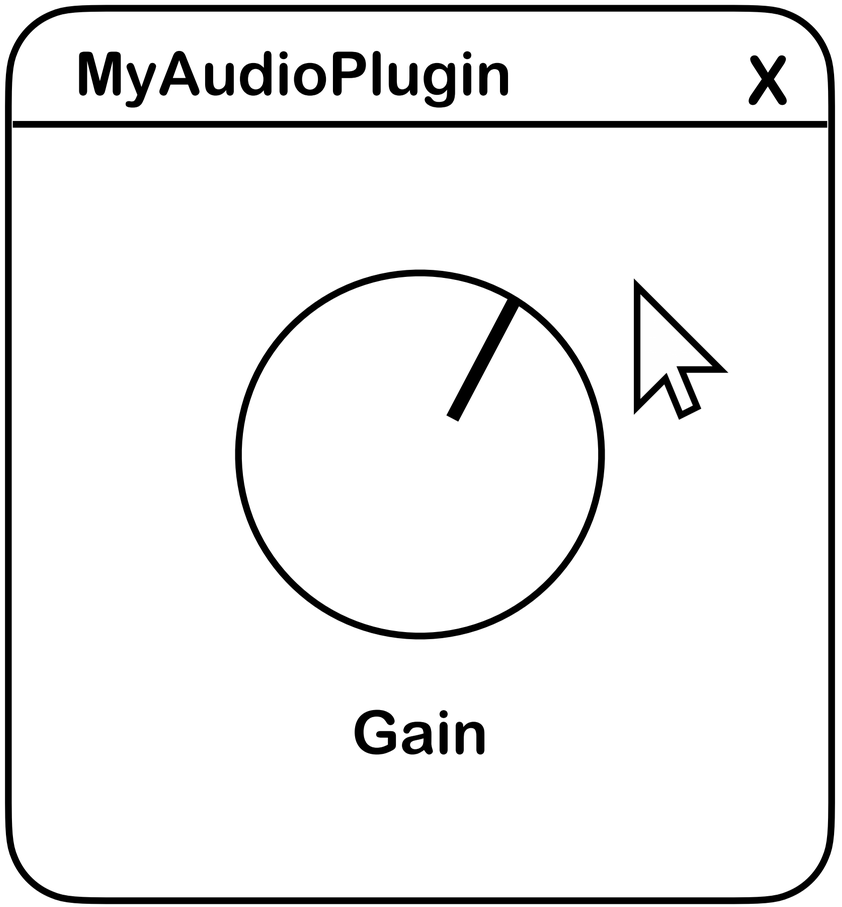
```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst    // Option 3. Sequentially consistent memory order
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst    // Option 3. Sequentially consistent memory order
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst
  };
}
```

Relaxed accesses to <u>one</u> particular atomic variable **`b`**:

- can happen concurrently from multiple threads
- are guaranteed to be observed in a consistent order
- do <u>not</u> impose any memory order on any other operations (that are not on **`b`**), atomic or otherwise


MyAudioPlugin
X
Gain

```
class MyAudioPlugin {
  Knob knob = {0, 1, Scale::log};
  std::atomic<float> gain = 1;

public:
  MyAudioPlugin() {
    knob.onValueChange = [this](const Knob& knob){
      // Thread 1
      gain.store(knob.getValue(), std::memory_order::relaxed);  // 👍
    };
  }

  void process(AudioBuffer& buffer) {
    for (auto& sample : buffer)
      // Thread 2
      sample *= gain.load(std::memory_order::relaxed);  // 👍
  }
};
```

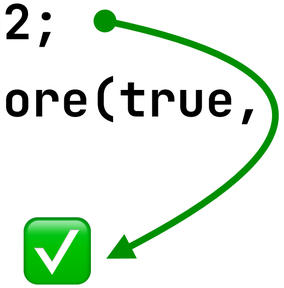
```cpp
int data = 0;
std::atomic<bool> ready = false;


// Thread 1

data = 42;
ready.store(true, /* ??? */);
```

```cpp
// Thread 2


while (!ready.load(/* ??? */))
  /* spin */ ;

assert(data == 42);  // OK or not?
```

```
int data = 0;
std::atomic<bool> ready = false;
using enum std::memory_order;

// Thread 1

data = 42;
ready.store(true, relaxed);
```

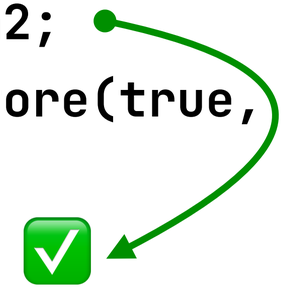
```
// Thread 2


while (!ready.load(relaxed))
  /* spin */ ;

assert(data == 42);  // OK or not?
```

```
int data = 0;
std::atomic<bool> ready = false;
using enum std::memory_order;

// Thread 1

data = 42;
ready.store(true, relaxed);
```

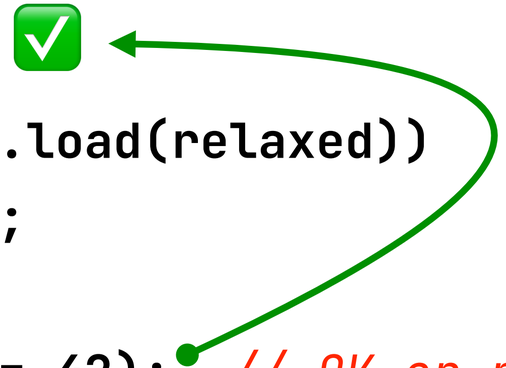


```
// Thread 2

while (!ready.load(relaxed))
  /* spin */ ;

assert(data == 42);  // OK or not?
```

```
int data = 0;
std::atomic<bool> ready = false;
using enum std::memory_order;

// Thread 1

data = 42;
ready.store(true, relaxed);
```



```
// Thread 2
```



```
while (!ready.load(relaxed))
  /* spin */ ;

assert(data == 42); // OK or not?
```

```
int data = 0;
std::atomic<bool> ready = false;
using enum std::memory_order;

// Thread 1

data = 42;
ready.store(true, memory_order_relaxed);
```

```
// Thread 2


while (!ready.load(memory_order_relaxed))
  /* spin */ ;

assert(data == 42);
```

Data race
==
undefined behaviour

```
std::atomic<int> data = 0;
std::atomic<bool> ready = false;
using enum std::memory_order;

// Thread 1

data.store(42, relaxed);
ready.store(true, relaxed);
```

```
// Thread 2


while (!ready.load(relaxed))
  /* spin */ ;

int current_data = data.load(relaxed);
assert(current_data == 42);
```

```cpp
std::atomic<int> data = 0;
std::atomic<bool> ready = false;
using enum std::memory_order;

// Thread 1

data.store(42, relaxed);
ready.store(true, relaxed);
```

```cpp
// Thread 2


while (!ready.load(relaxed))
  /* spin */ ;

int current_data = data.load(relaxed);
assert(current_data == 42); // may fail :(
```

```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst    // Option 3. Sequentially consistent memory order
  };
}
```

The C++ Memory Model

[intro.races]

- Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location.

[intro.races]

- Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location.
- The execution of a program contains a **data race** if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and **neither happens before the other** [...]

[intro.races]

- Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location.
- The execution of a program contains a **data race** if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and **neither happens before the other** [...]
- Any such data race results in undefined behavior.

[intro.races]

- An evaluation *A* **happens before** an evaluation *B* if either
  - *A* is **sequenced before** *B*, or
  - *A* **synchronizes with** *B*, or
  - *A* happens before *X* and *X* happens before *B*.

[intro.races]

- An evaluation *A* **happens before** an evaluation *B* if either
  - *A* is **sequenced before** *B*, or
  - *A* **synchronizes with** *B*, or
  - *A* happens before *X* and *X* happens before *B*.

[intro.execution]

- Sequenced before is an asymmetric, transitive, pair-wise **relation between evaluations executed by a single thread**, which induces a partial order among those evaluations.
- Given any two evaluations *A* and *B*, if *A* is sequenced before *B* (or, equivalently, B is sequenced after *A*), then the execution of *A* shall precede the execution of *B*.
- If *A* is not sequenced before *B* and *B* is not sequenced before *A*, then *A* and *B* are unsequenced.

```
void f(int i) {
  std::print("{}", i);
}

int main() {
  f(1);
  f(2);  // call to f(1) is sequenced before call to f(2)
}
```

```
void f(int i) {
  std::print("{}", i);
}

int main() {
  std::thread t1([]{ f(1); });
  std::thread t2([]{ f(2); });
  // calls to f(1) and f(2) are unsequenced with respect to each other
}
```

[intro.races]

- An evaluation *A* **happens before** an evaluation *B* if either
  - *A* is **sequenced before** *B*, or
  - *A* **synchronizes with** *B*, or
  - *A* happens before *X* and *X* happens before *B*.

[atomics.order]

- An atomic operation *A* that performs a **release operation on an atomic object *M*** synchronizes with an atomic operation *B* that performs **an acquire operation on *M*** and takes its value from any side effect in the release sequence headed by *A.*

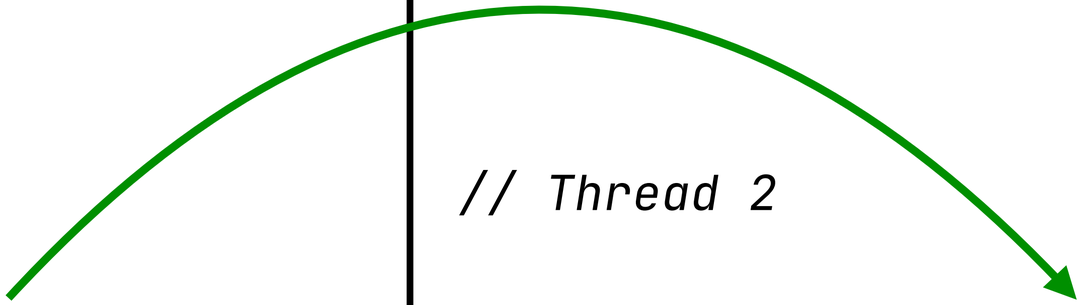
[atomics.order]

- An atomic operation *A* that performs a **release operation on an atomic object *M*** synchronizes with an atomic operation *B* that performs **an acquire operation on *M*** and takes its value from any side effect in the release sequence headed by *A.*
- An atomic store operation with **`memory_order::release`** performs a release operation on the affected memory location.
- An atomic load operation with **`memory_order::acquire`** performs an acquire operation on the affected memory location.

```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```

```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```

```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```




```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```

```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```

"synchronises with"

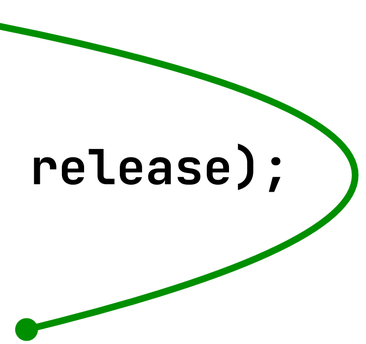
```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```

```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```

"synchronises with"

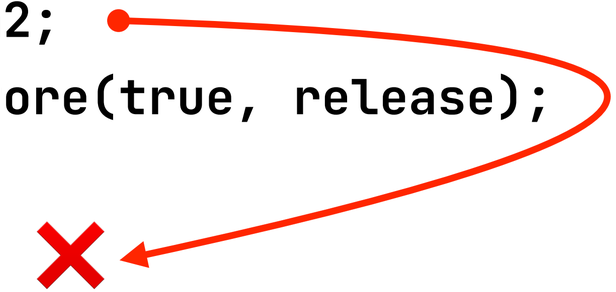
```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```

```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1
                ✅
data = 42;
ready.store(true, release);

other_data = 43;
```



```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```

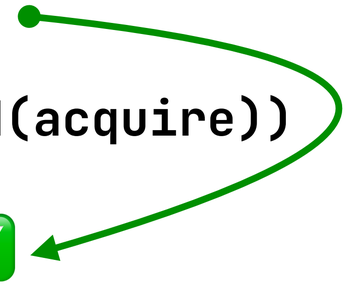
```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```



```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```

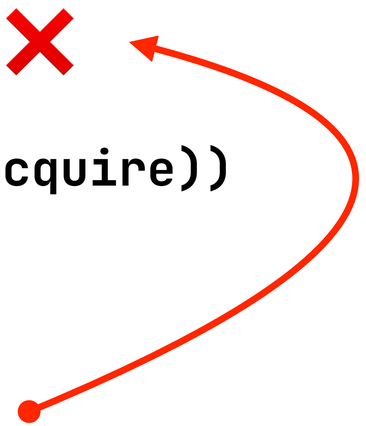
```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```

```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```



```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, release);
```

```
// Thread 2

while (!ready.load(acquire))
  /* spin */ ;

assert(data == 42);
```



```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, memory_order_release);
```

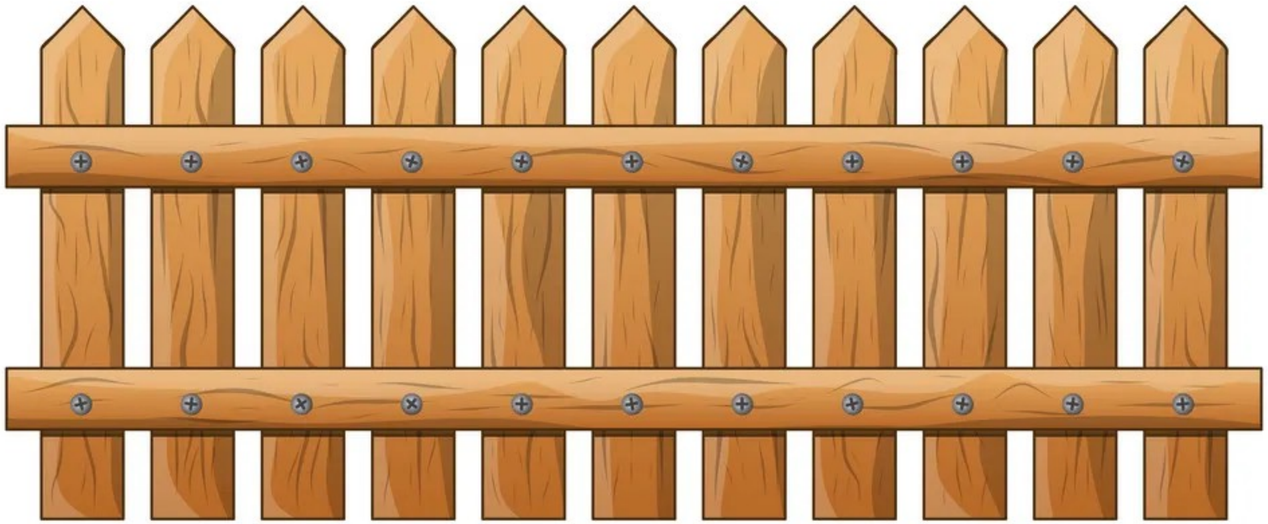
No data race :)

```
// Thread 2

while (!ready.load(memory_order_acquire))
  /* spin */ ;

assert(data == 42);
```

```
int data = 0;
std::atomic<bool> ready = false;

// Thread 1

data = 42;
ready.store(true, memory_order_release);
```

No data race :)

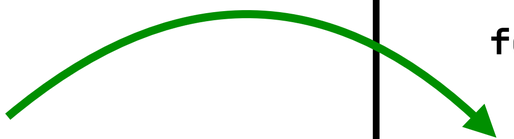
```
// Thread 2

while (!ready.load(memory_order_acquire))
  /* spin */ ;

assert(data == 42);  // always holds :)
```



```
struct LockFreeFIFO {
  std::atomic<float> buffer[kSize];
  std::size_t rpos, wpos;
};

// Thread 1
void captureCallback(LockFreeFIFO& fifo,
float const* buffer, size_t n) {
  auto const wpos = fifo.wpos;

  for (size_t i = 0; i < n; ++i) {
    fifo.buffer[(i + wpos) % kSize]
    .store(buffer[i], release);
  }

  fifo.wpos = (wpos + n) % kSize;
}
```

```
// Thread 2
void playbackCallback(LockFreeFIFO& fifo,
float* buffer, size_t n) {
  auto const rpos = fifo.rpos;

  for (size_t i = 0; i < n; ++i)
    buffer[i] = fifo.buffer[(i + rpos) % kSize]
    .exchange(0.f, acq_rel);

  fifo.rpos = (rpos + n) % kSize;
}
```

```
struct LockFreeFIFO {
  std::atomic<float> buffer[kSize];
  std::size_t rpos, wpos;
};

// Thread 1
void captureCallback(LockFreeFIFO& fifo,
float const* buffer, size_t n) {
  auto const wpos = fifo.wpos;

  for (size_t i = 0; i < n; ++i) {
    fifo.buffer[(i + wpos) % kSize]
    .store(buffer[i], release);
  }

  fifo.wpos = (wpos + n) % kSize;
}
```



```
// Thread 2
void playbackCallback(LockFreeFIFO& fifo,
float* buffer, size_t n) {
  auto const rpos = fifo.rpos;

  for (size_t i = 0; i < n; ++i)
    buffer[i] = fifo.buffer[(i + rpos) % kSize]
    .exchange(0.f, acq_rel);

  fifo.rpos = (rpos + n) % kSize;
}
```

```
struct LockFreeFIFO {
  std::atomic<float> buffer[kSize];
  std::size_t rpos, wpos;
};

// Thread 1
void captureCallback(LockFreeFIFO& fifo,
float const* buffer, size_t n) {
  auto const wpos = fifo.wpos;

  for (size_t i = 0; i < n; ++i) {
    fifo.buffer[(i + wpos) % kSize]
    .store(buffer[i], relaxed);
  }

  std::atomic_thread_fence(release);

  fifo.wpos = (wpos + n) % kSize;
}
```

```
// Thread 2
void playbackCallback(LockFreeFIFO& fifo,
float* buffer, size_t n) {
  auto const rpos = fifo.rpos;

  std::atomic_thread_fence(acquire);

  for (size_t i = 0; i < n; ++i)
    buffer[i] = fifo.buffer[(i + rpos) % kSize]
    .exchange(0.f, relaxed);

  fifo.rpos = (rpos + n) % kSize;
}
```

```
struct LockFreeFIFO {
  std::atomic<float> buffer[kSize];
  std::size_t rpos, wpos;
};

// Thread 1
void captureCallback(LockFreeFIFO& fifo,
float const* buffer, size_t n) {
  auto const wpos = fifo.wpos;

  for (size_t i = 0; i < n; ++i) {
    fifo.buffer[(i + wpos) % kSize]
    .store(buffer[i], relaxed);
  }

  std::atomic_thread_fence(release);

  fifo.wpos = (wpos + n) % kSize;
}
```

```
// Thread 2
void playbackCallback(LockFreeFIFO& fifo,
float* buffer, size_t n) {
  auto const rpos = fifo.rpos;

  std::atomic_thread_fence(acquire);

  for (size_t i = 0; i < n; ++i)
    buffer[i] = fifo.buffer[(i + rpos) % kSize]
    .exchange(0.f, relaxed);

  fifo.rpos = (rpos + n) % kSize;
}
```

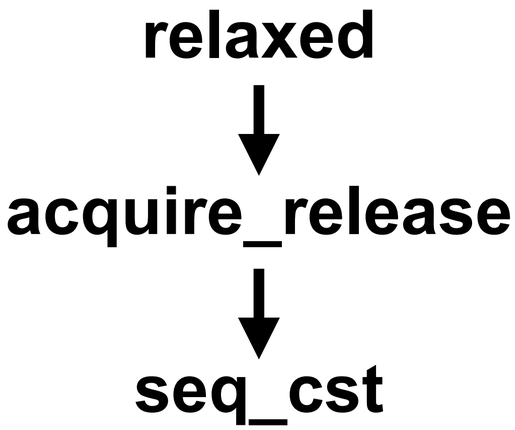
```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst    // Option 3. Sequentially consistent memory order
  };
}
```

```
namespace std {
  enum class memory_order {
    relaxed,   // Option 1. Relaxed memory order
    acquire,   // Option 2. Acquire-release memory order
    release,
    acq_rel,
    seq_cst    // Option 3. Sequentially consistent memory order
  };
}
```

Sequentially consistent atomic operations are all observed in a consistent order from all threads.
The default if you do not specify any memory order!

```
std::atomic<bool> x = {false};
std::atomic<bool> y = {false};
std::atomic<int> z = {0};

std::thread a([&]{ x.store(true, seq_cst); });
std::thread b([&]{ y.store(true, seq_cst); });

std::thread c([&]{
  while (!x.load(seq_cst)) /* spin */ ;
  if (y.load(seq_cst))
    ++z;
});

std::thread d([&]{
  while (!y.load(seq_cst)) /* spin */ ;
  if (y.load(seq_cst))
    ++z;
});

assert(z.load() != 0); // is only guaranteed to pass when using seq_cst above!
```

Differences between hardware architectures


relaxed
acquire_release
seq_cst

Atomic reads:

relaxed

↓

acquire

↓

seq_cst

Atomic writes:

relaxed

↓

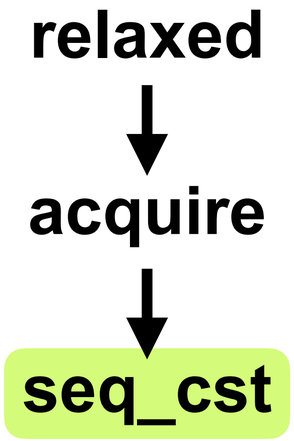
release

↓

seq_cst

x86_64: "strongly ordered" architecture

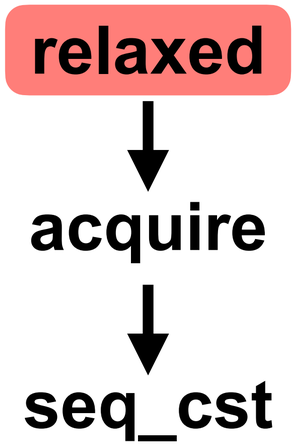
Atomic loads:



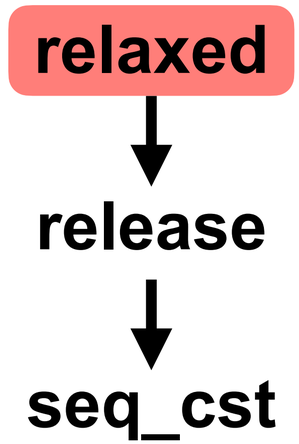
Atomic stores:

arm64: "weakly ordered" architecture

Atomic loads:



Atomic stores:



Takeaways

- When implementing lock-free algorithms in C++,
  - understand the C++ memory model and memory orderings
  - always use the "long form" for atomic reads/writes
    (makes it obvious that you are accessing a **`std::atomic`**!)
  - always specify the intended memory order
    (forces you to think about the algorithm!)
  - use **`relaxed`** whenever dealing with a single atomic in isolation
    - remember that atomic operations on the *same* atomic variable always have a consistent order)
  - use **`seq_cst`** when you *actually* need full sequential consistency (very rarely)
  - you should end up using **`acquire`**/**`release`** most of the time
- Always measure!!!

Takeaways

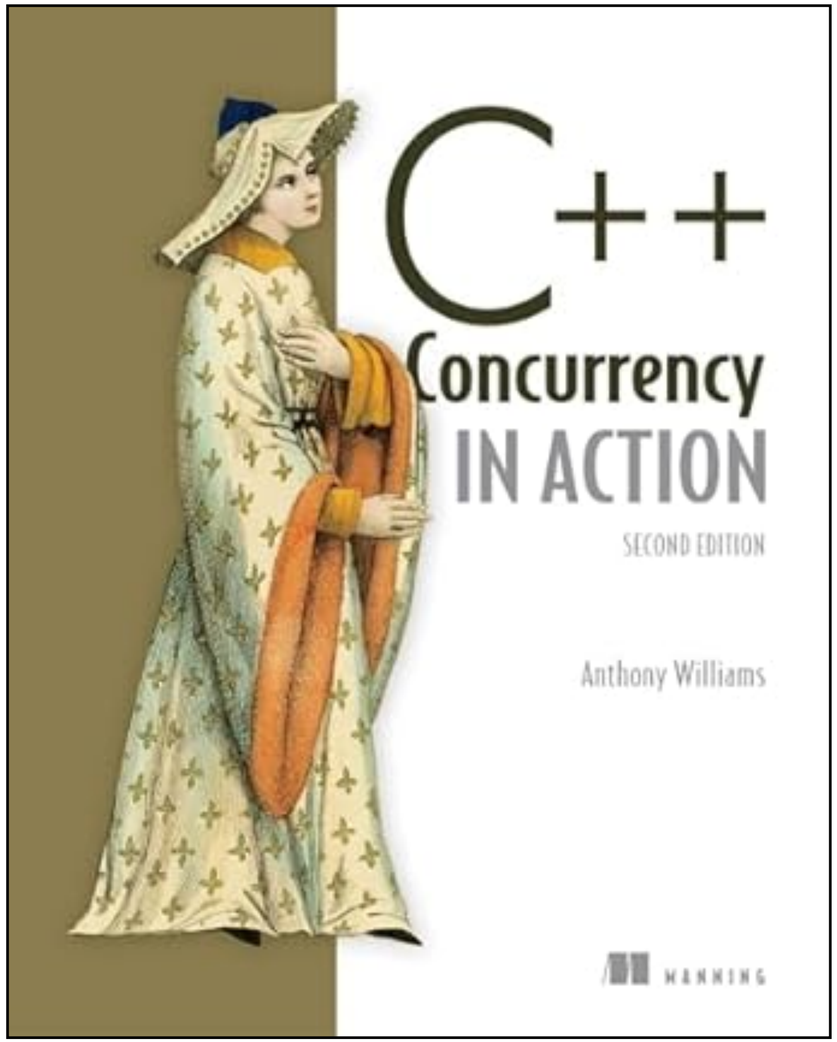
- When implementing lock-free algorithms in C++,
  - understand the C++ memory model and memory orderings
  - always use the "long form" for atomic reads/writes
    (makes it obvious that you are acc c!)
  - always specify the
    (forces you t
  - use **`relaxed`** a single atomic in isolation
    - remember atomic operations on the *same* atomic variable always have a consistent order)
  - use **`seq_cst`** when you *actually* need full sequential consistency (very rarely)
  - you should end up using **`acquire`**/**`release`** most of the time
- Always measure!!!

Actually, don't do any of this, just use a library!

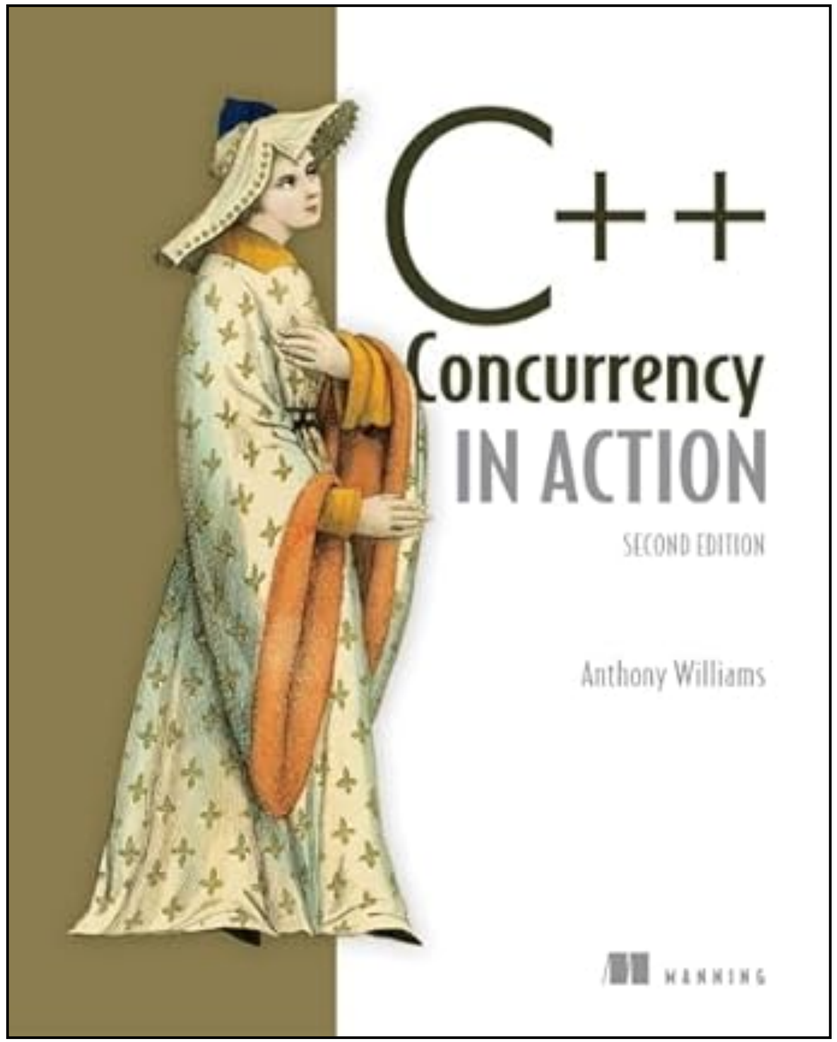
Talk recommendations

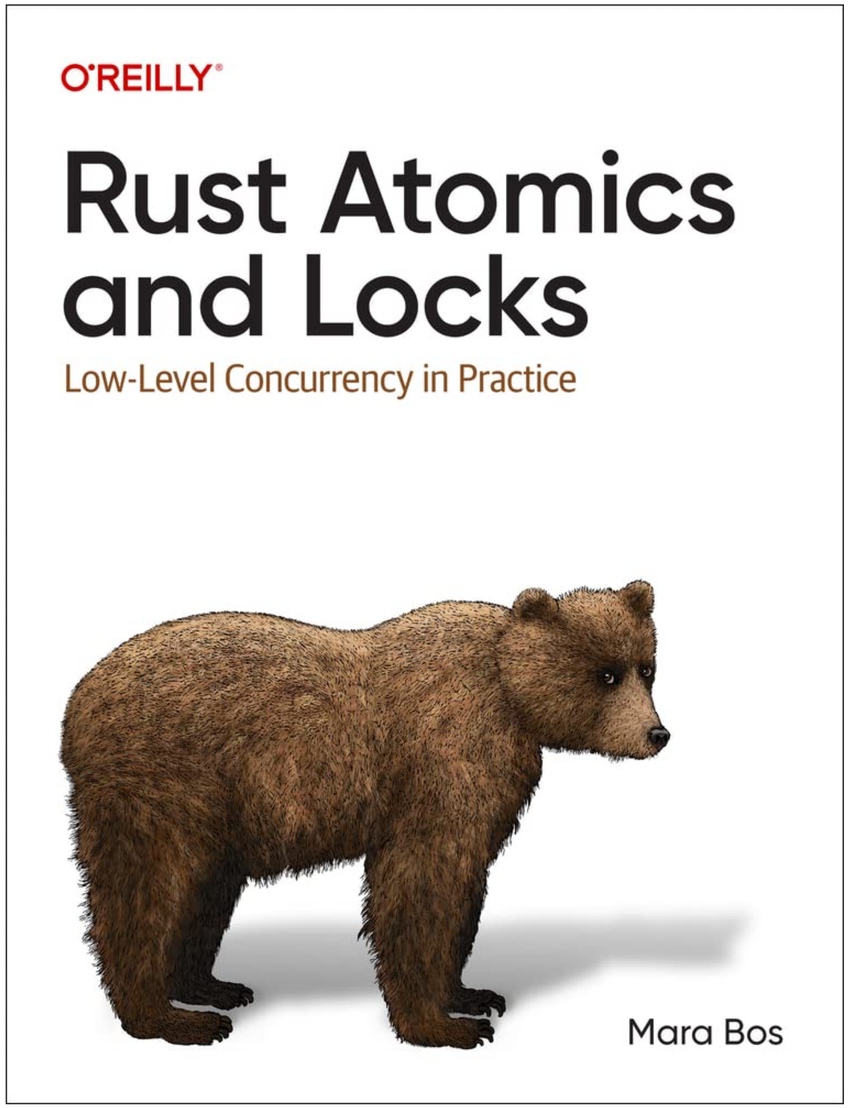
- Herb Sutter: "Atomic weapons"
  (C++ and beyond 2012)
- Dave Rowland: "Lock-free queues in the multiverse of madness"
  (ADC'25)
- Christopher Fretz: "Beyond Sequential Consistency"
  (CppOnSea 2025)

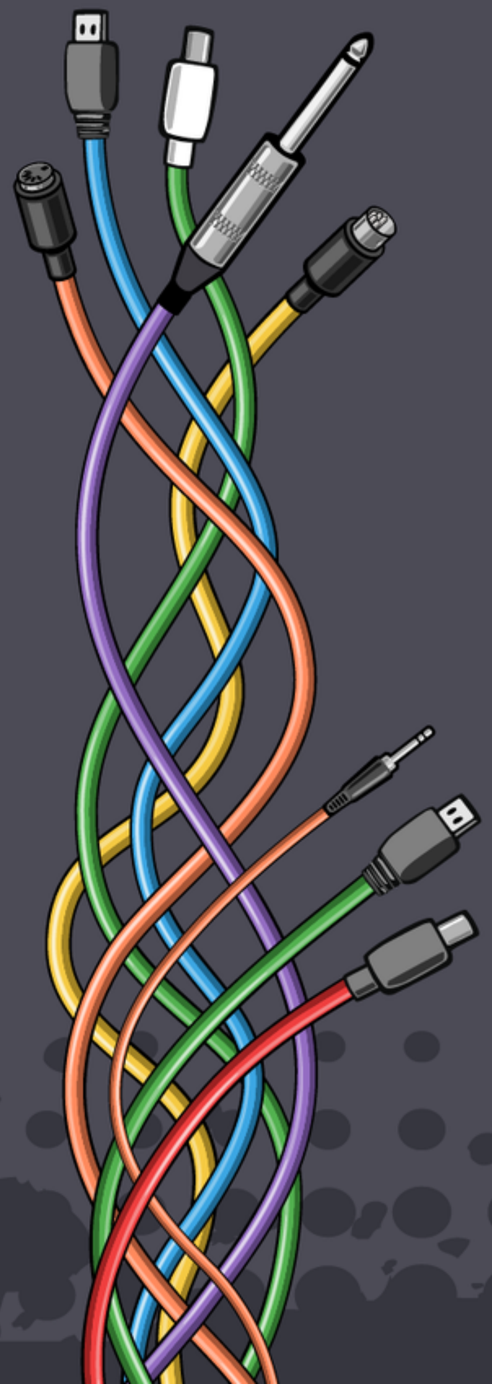
Book recommendations



Book recommendations







DEMYSTIFYING STD::MEMORY_ORDER

TIMUR DOUMLER

ADC25 Bristol