Lecture 3 – Atomics and Memory Model in Modern C++

a.k.a. Atomic<> Weapons*

CS3211 Parallel and Concurrent Programming

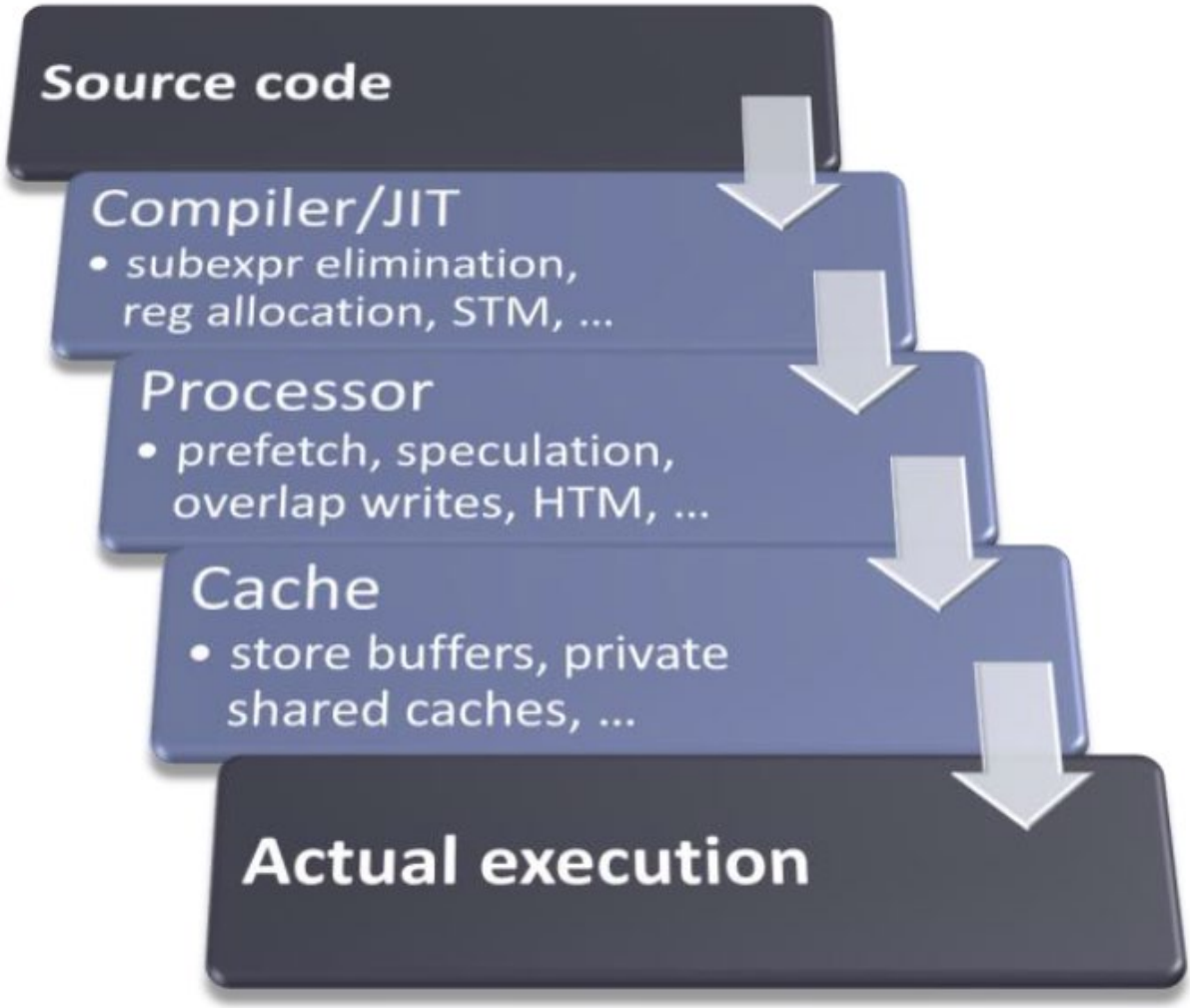
* Herb Sutter

Outline

- Atomics in C++
  - Types and atomic operations
- Memory model at language level
  - Sequential consistent order
  - Relaxed order
  - Acquire-release order
  - Fences

Changes to your program

- Correctly synchronized program should behave as if:
  - memory ops are actually executed in an order that appears equivalent to some sequentially consistent inter-leaved execution of the memory ops of each thread in your source code
  - including that each write appears to be atomic and globally visible simultaneously to all processors.
    - ❖ https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/



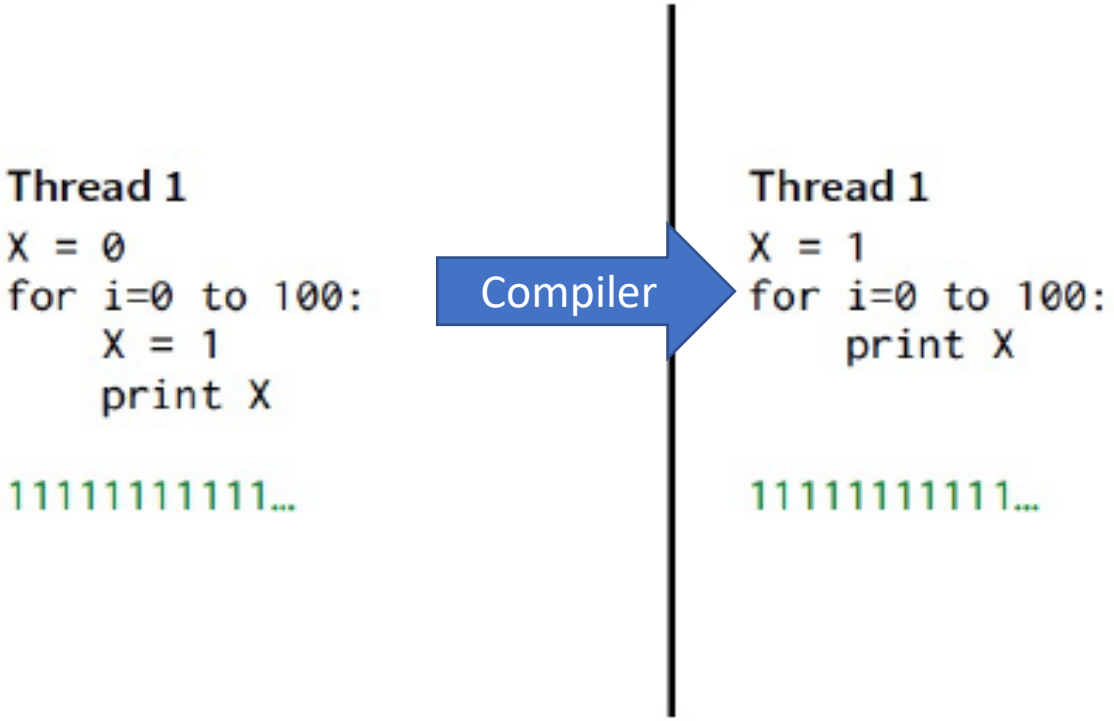
Aggressive operation reordering is useful!

- W→W: processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)
- R→W, R→R: processor might reorder independent instructions in an instruction stream (out-of-order execution)

- Valid optimizations when a program consists of a single instruction stream, but what about multiple threads?

Languages need memory models

Optimization **not** visible to programmer

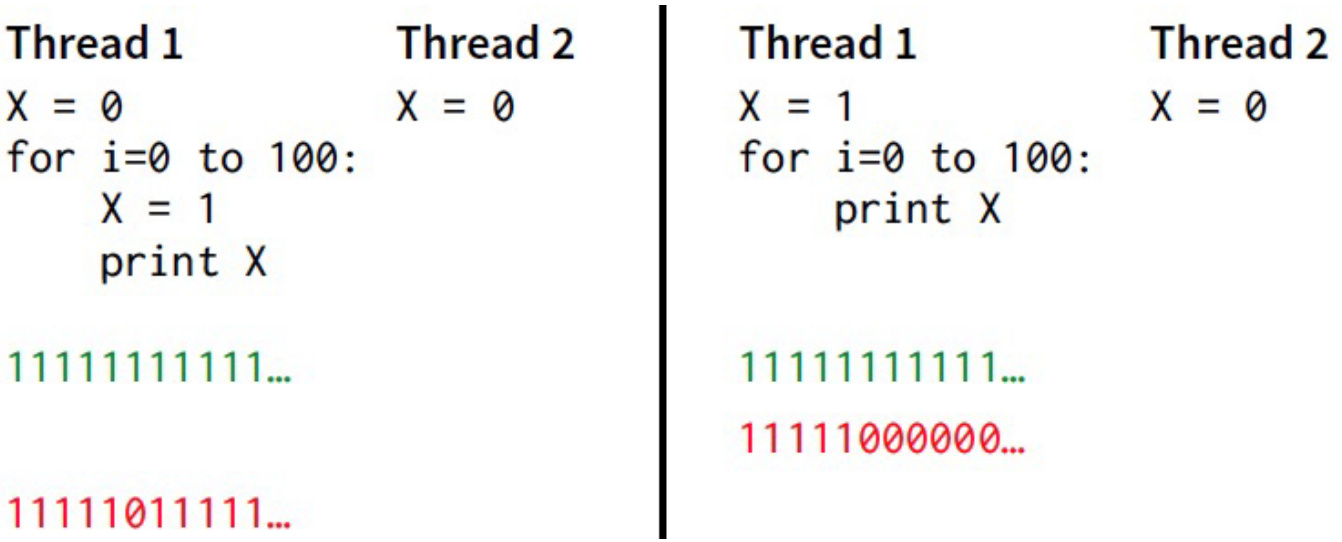


As-if rule

- The C++ compiler is permitted to perform any changes to the program as long as the following remains true:
  - Accesses (reads and writes) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered with respect to other volatile accesses on the same thread.
  - At program termination, data written to files is exactly as if the program was executed as written.
  - Prompting text which is sent to interactive devices will be shown before the program waits for input.
- Programs with undefined behavior are free from the *as-if rule*
  - Often these programs change observable behavior when recompiled with different optimization settings.

Languages need memory models too

Optimization **is** visible to programmer



Memory models provide a contract to programmers about how their memory operations will be reordered by the **compiler**

Multithreading-aware memory model (MM) in C++

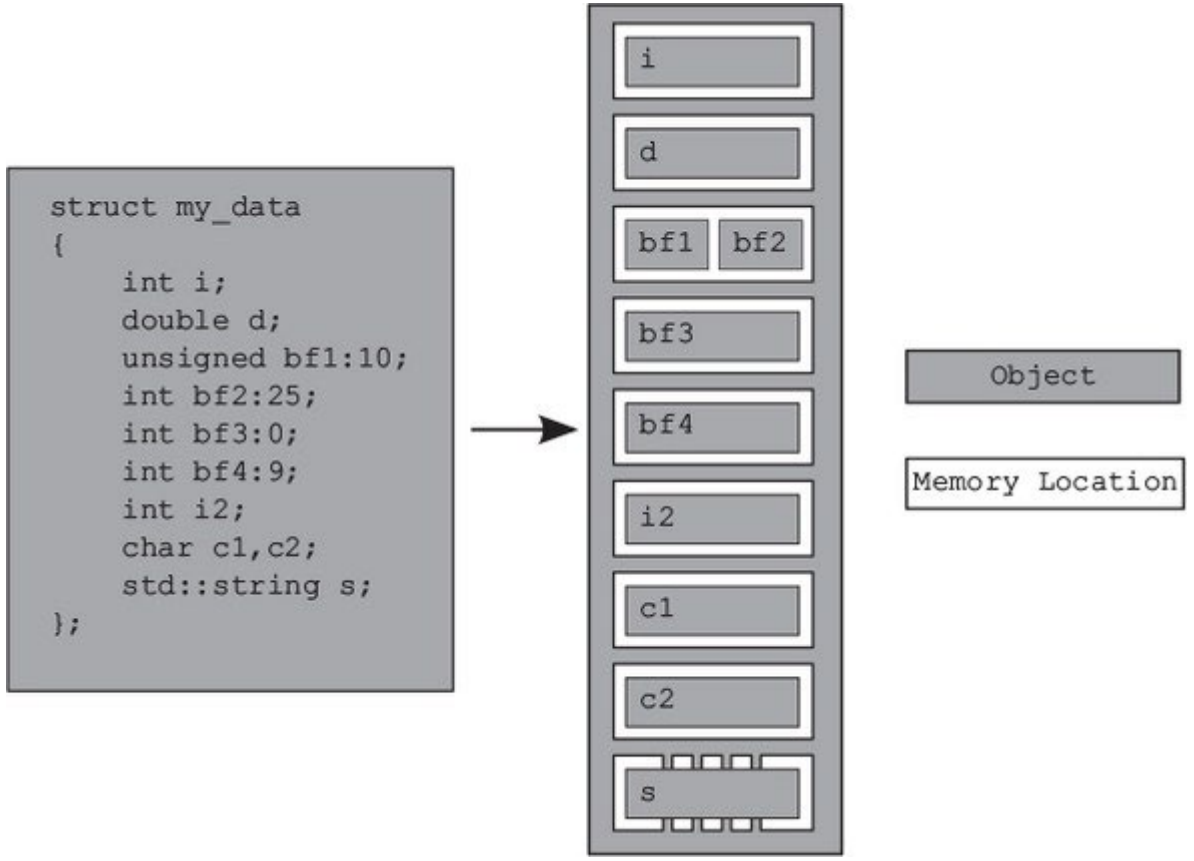
- Programmers might not notice the MM
  - Using mutexes, futures, barriers do not require an understanding of the MM
- MM is essential to make all multithreading facilities work
- C++ is a systems programming language
  - Should not be needed to work with another lower-level language than C++
  - Allow programmers to get “close to the machine”

Memory model

- Structure
- Concurrency

Structure

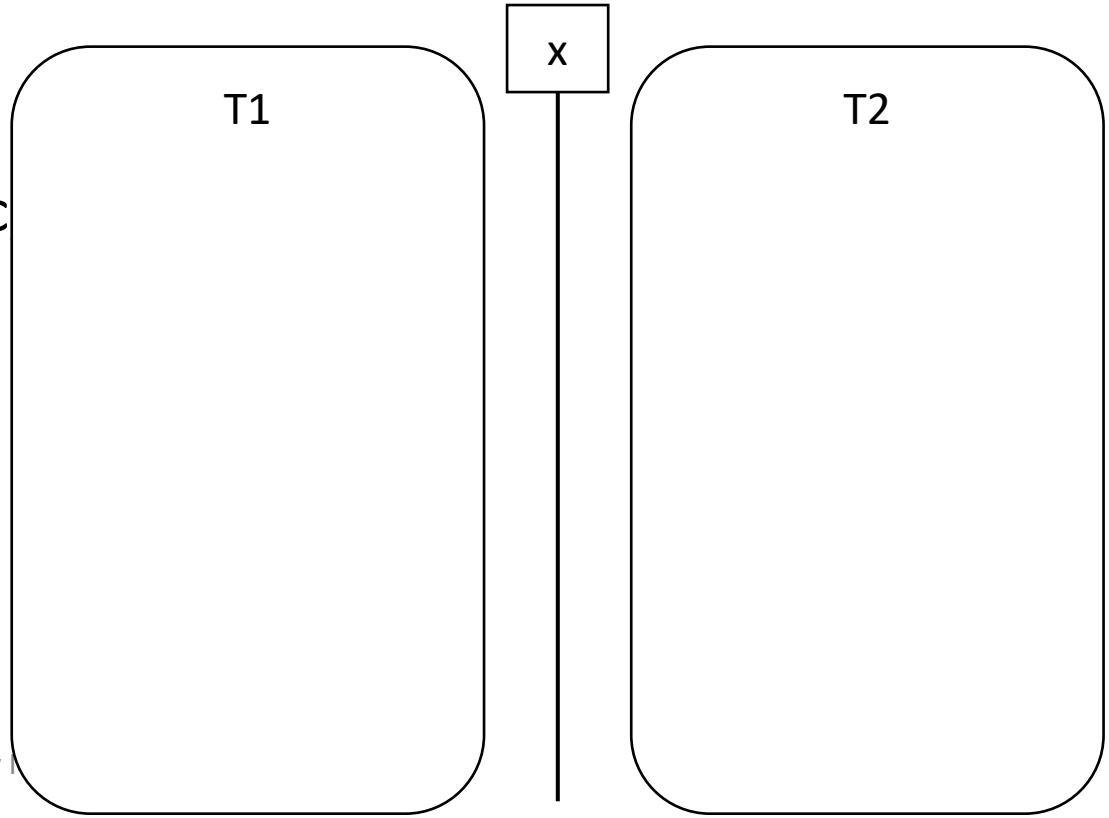
- Every variable is an object, including those that are members of other objects.
- Every object occupies *at least one* memory location.
- Variables of fundamental types such as `int` or `char` occupy *exactly one* memory location, whatever their size, even if they're adjacent or part of an array.
- Adjacent bit fields are part of the same memory location.



Concurrency

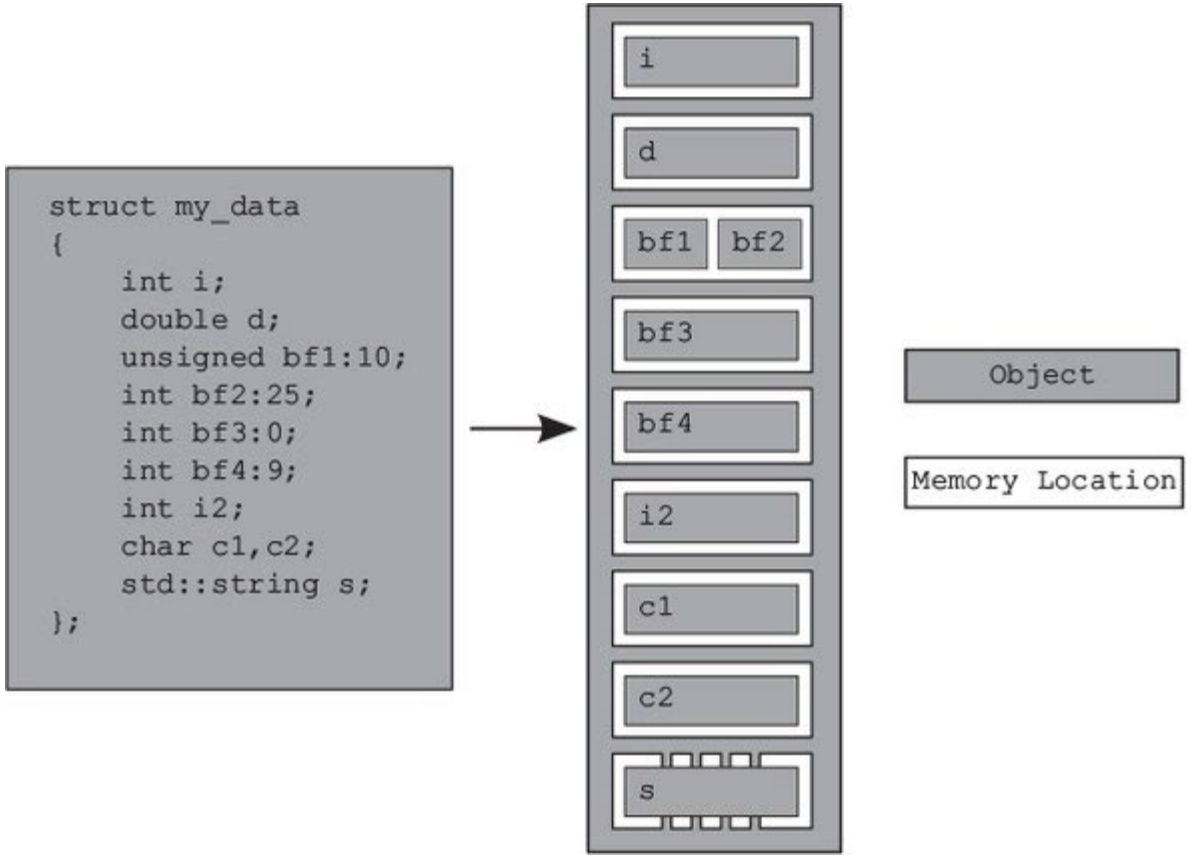
- Two threads access the *same* memory location
  - Read-only data doesn't need protection or synchronization
  - Either thread is modifying the data: potential for a race condition
- **Data Race** - for two accesses to a *single* memory location from separate threads
  - no enforced *ordering* between access
  - one or both of those accesses is not atomic
  - AND if one or both accesses is a write

  Causes **undefined behavior**



Avoid data races: use critical section

- Code that must execute in isolation w.r.t. other program code
- To implement critical section:
  - Locks
  - Ordered atomics
  - Transactional memory



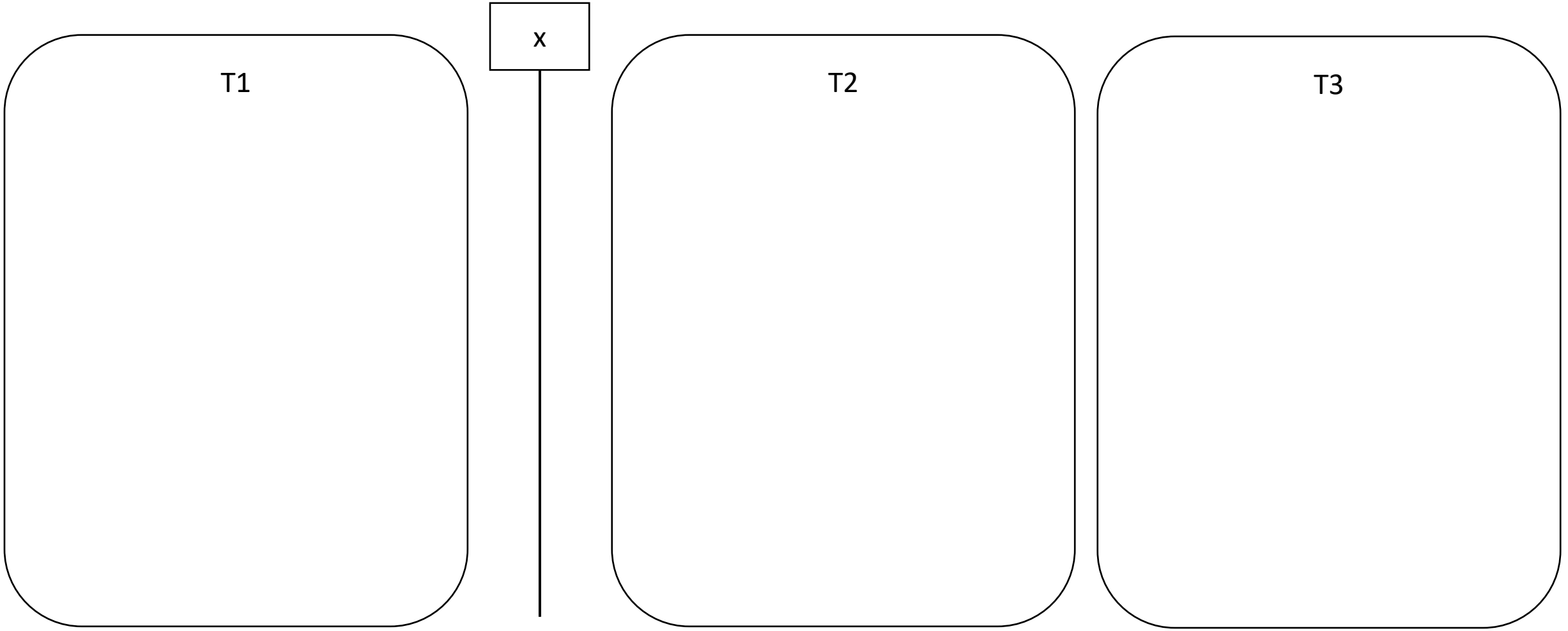
Undefined behavior

- The behavior of the complete application is now undefined, and it may do anything at all
- Serious bug and should be avoided at all costs
- To avoid undefined behavior
  - Understand modification orders

Modification order (MO)

- Composed of all writes to an object from all threads in the program
  - MO varies between runs
  - For every object in the program
- If the object is not one of the atomic types
  - Programmer is responsible for ensuring the threads agree on the modification order of each variable (through sufficient synchronization)
  - Data race and undefined behavior: different threads see distinct sequences of values for a single variable
- If atomic operations are used
  - Compiler is responsible for ensuring that the necessary synchronization is in place (this does not mean “race-free”)

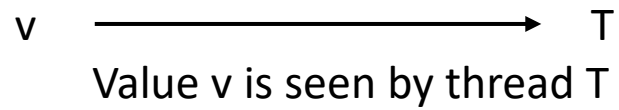
Modification order

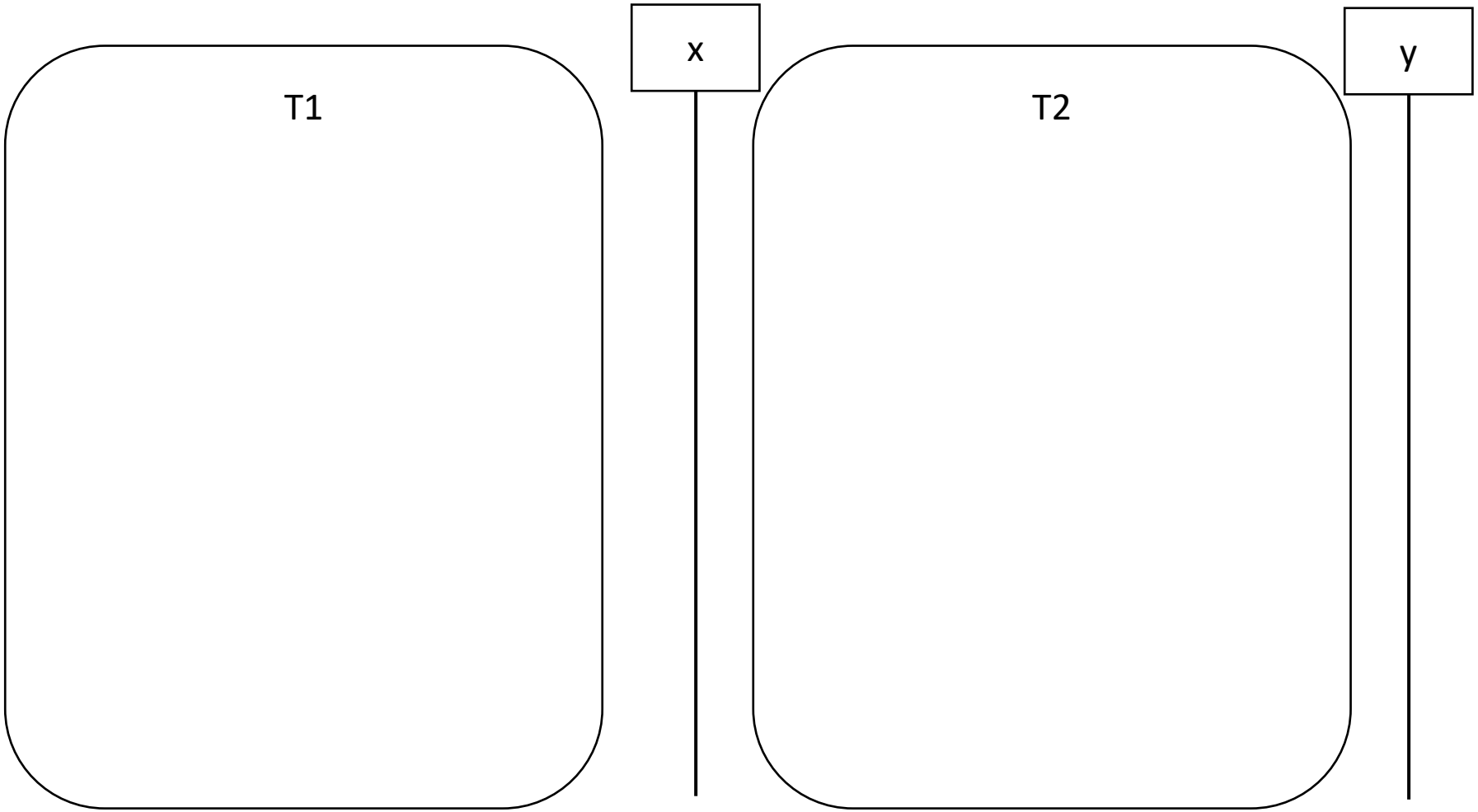


Modification order - requirements

- Once a thread has seen a particular entry in the modification order
  - Subsequent reads from that thread must return later values, and
  - Subsequent writes from that thread to that object must occur later in the modification order.
- A read of an object that follows a write to that object in the same thread must either return the value written or another value that occurs later in the modification order of that object
- All threads must agree on the modification orders of each individual object in a program
  - BUT they don't necessarily have to agree on the relative order of operations on separate objects

Modification order





Atomic operations and types

- Used to enforce modification order
- Atomic operation is an indivisible operation
  - Always observed fully done from any thread in the system
- Atomic load and stores for an object
  - Load will retrieve either the initial value of the object or the value stored by one of the modifications
- (Non-atomic operation might be seen as half-done by another thread)
  - Non-atomic operation composed of atomic operations (for example, assignment to a struct with atomic members): other threads may observe mixed-up combination of completion of the operations

“atomic<> weapons” in C++

- Enable low-level synchronization
- Programmers need to understand and deal with the memory model
- Atomic types and operations on atomics
  - Low-level synchronization operations that reduce to 1-2 CPU instructions

Atomic types

- Found in the `<atomic>` header
- All operations on such types are atomic
- `is_lock_free()` member function returns `true` if operations on a given type are done directly with **atomic instructions**
  - X::is_always_lock_free is true if and only if the atomic type X is lock-free for all supported hardware that the current compilated code might run on
- Might emulate atomicity by using an internal mutex
  - The hoped-for performance gains will probably not materialize

Outline

- Recap
- Atomics in C++
  - Types and atomic operations
- Memory model at the language level
  - Sequentially consistent order
  - Relaxed order
  - Acquire-release order
  - Fences

Memory order in C++ (atomics)

- Each of the operations on the atomic types has an optional memory-ordering argument which is one of the values of the `std::memory_order` enumeration
- `std::memory_order` specifies how memory accesses, including regular, non-atomic memory accesses, are to be ordered around an atomic operation
- Reminder: with no constraints on a multi-core system:
  - Lack of modification order: when multiple threads simultaneously read and write to several variables, one thread can observe the values change in an order different from the order another thread wrote them
  - The apparent order of changes can even differ among multiple reader threads
  - Some similar effects can occur even on uniprocessor systems due to compiler transformations allowed by the memory model

`std::memory_order` enumeration

- **Default** ordering, the strongest ordering: `std::memory_order_seq_cst`
- **Store** operations can have `memory_order_relaxed`, `memory_order_release`, or `memory_order_seq_cst` ordering
- **Load** operations can have `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, or `memory_order_seq_cst` ordering
- **Read-modify-write** operations can have `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, or `memory_order_seq_cst` ordering

Sequentially consistent ordering

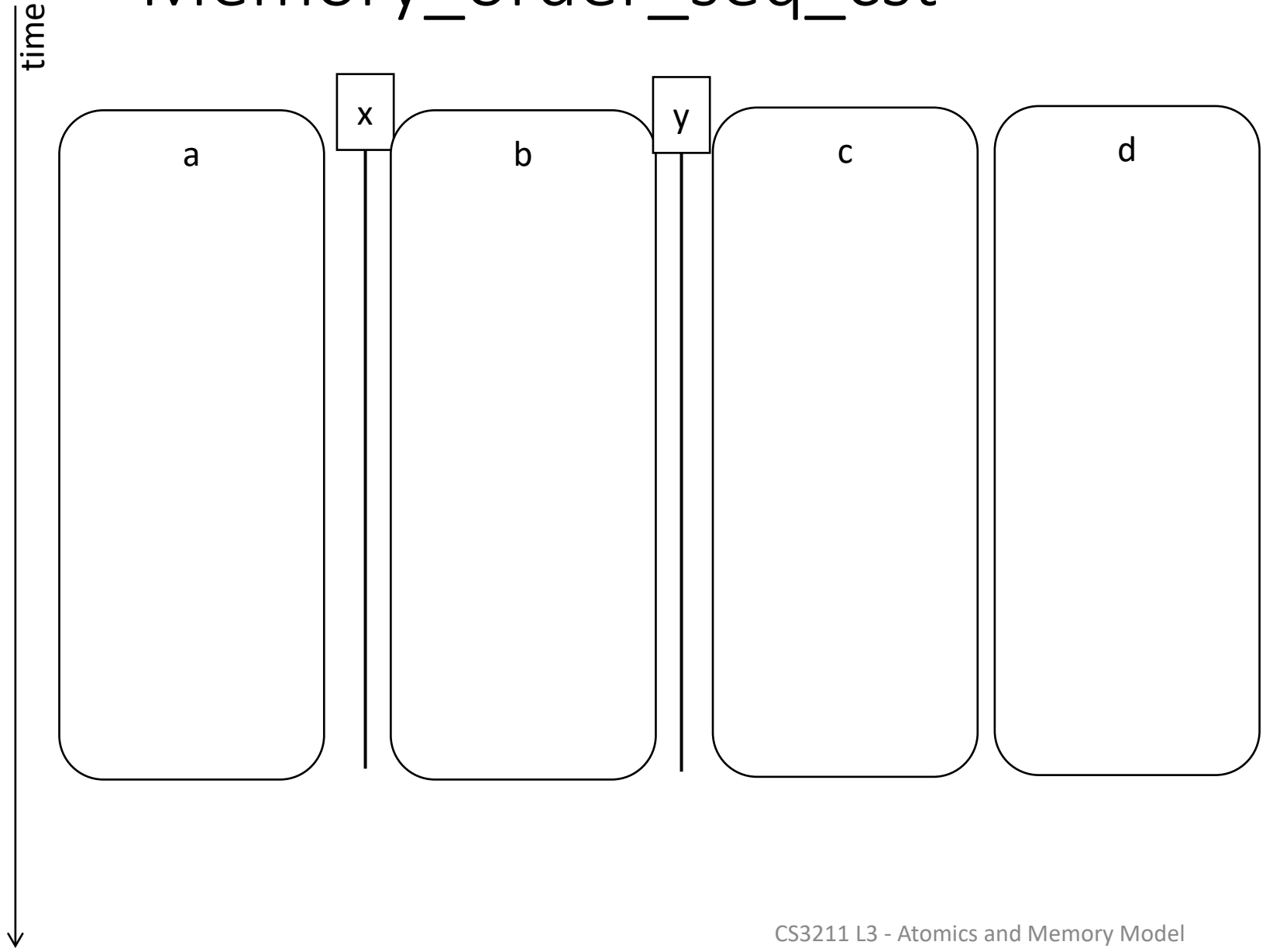
- Implies that the behavior of the program is consistent with a simple sequential view of the world
- All threads must see the same order of operations
- Operations can't be re-ordered
- A sequentially consistent store *synchronizes-with* a sequentially consistent load of the same variable that reads the value stored
  - Does not apply to atomic operations with relaxed memory orderings
  - Advice: use sequentially consistent operations on all threads
- Noticeable performance penalty on a *weakly-ordered machine* with many cores
  - The overall sequence of operations must be kept consistent between the cores, possibly requiring extensive (and expensive!) synchronization operations between the processors

Memory_order_seq_cst

- Line 39: assert never fires
- Either line 8 or 12 happens first
  - If line 17 returns false -> Line 8 happened first
  - If line 23 returns false -> Line 12 happened first

```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x()
7  {
8      x.store(true,std::memory_order_seq_cst);
9  }
10 void write_y()
11 {
12     y.store(true,std::memory_order_seq_cst);
13 }
14 void read_x_then_y()
15 {
16     while(!x.load(std::memory_order_seq_cst));
17     if(y.load(std::memory_order_seq_cst))
18         ++z;
19 }
20 void read_y_then_x()
21 {
22     while(!y.load(std::memory_order_seq_cst));
23     if(x.load(std::memory_order_seq_cst))
24         ++z;
25 }
26 int main()
27 {
28     x=false;
29     y=false;
30     z=0;
31     std::thread a{write_x};
32     std::thread b{write_y};
33     std::thread c{read_x_then_y};
34     std::thread d{read_y_then_x};
35     a.join();
36     b.join();
37     c.join();
38     d.join();
39     assert(z.load()!=0);
40 }
```

Memory_order_seq_cst



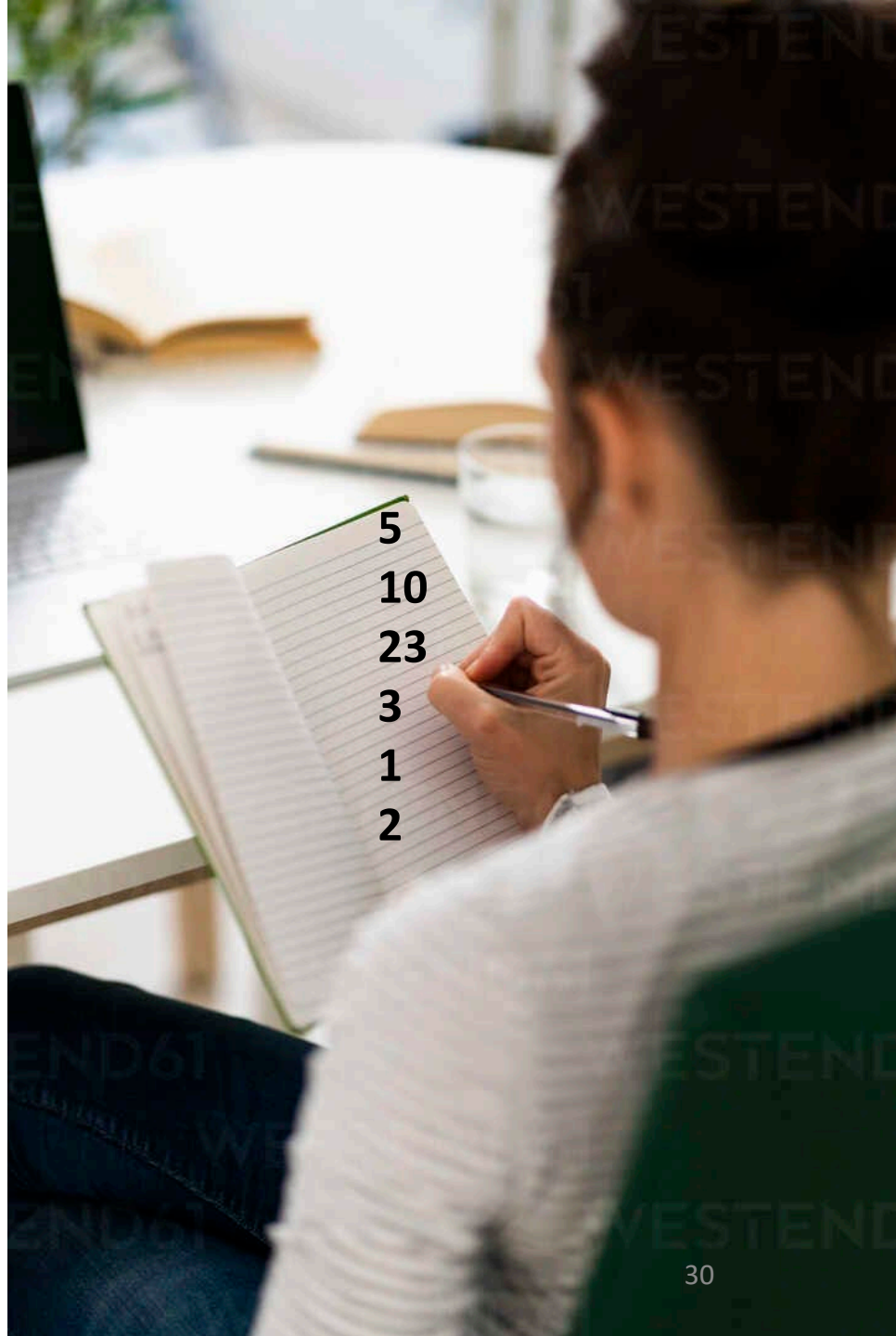

```cpp
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
    x.store(true,std::memory_order_seq_cst);
}
void write_y()
{
    y.store(true,std::memory_order_seq_cst);
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a{write_x};
    std::thread b{write_y};
    std::thread c{read_x_then_y};
    std::thread d{read_y_then_x};
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

Non-sequentially consistent orderings

- No single global order of events
  - Different threads can see different views of the same operations
  - **No** mental model of operations from different threads neatly interleaved one after the other
  - Throw out mental models based on the idea of the compiler or processor reordering the instructions
- The only requirement is that all threads agree on the modification order of each individual variable

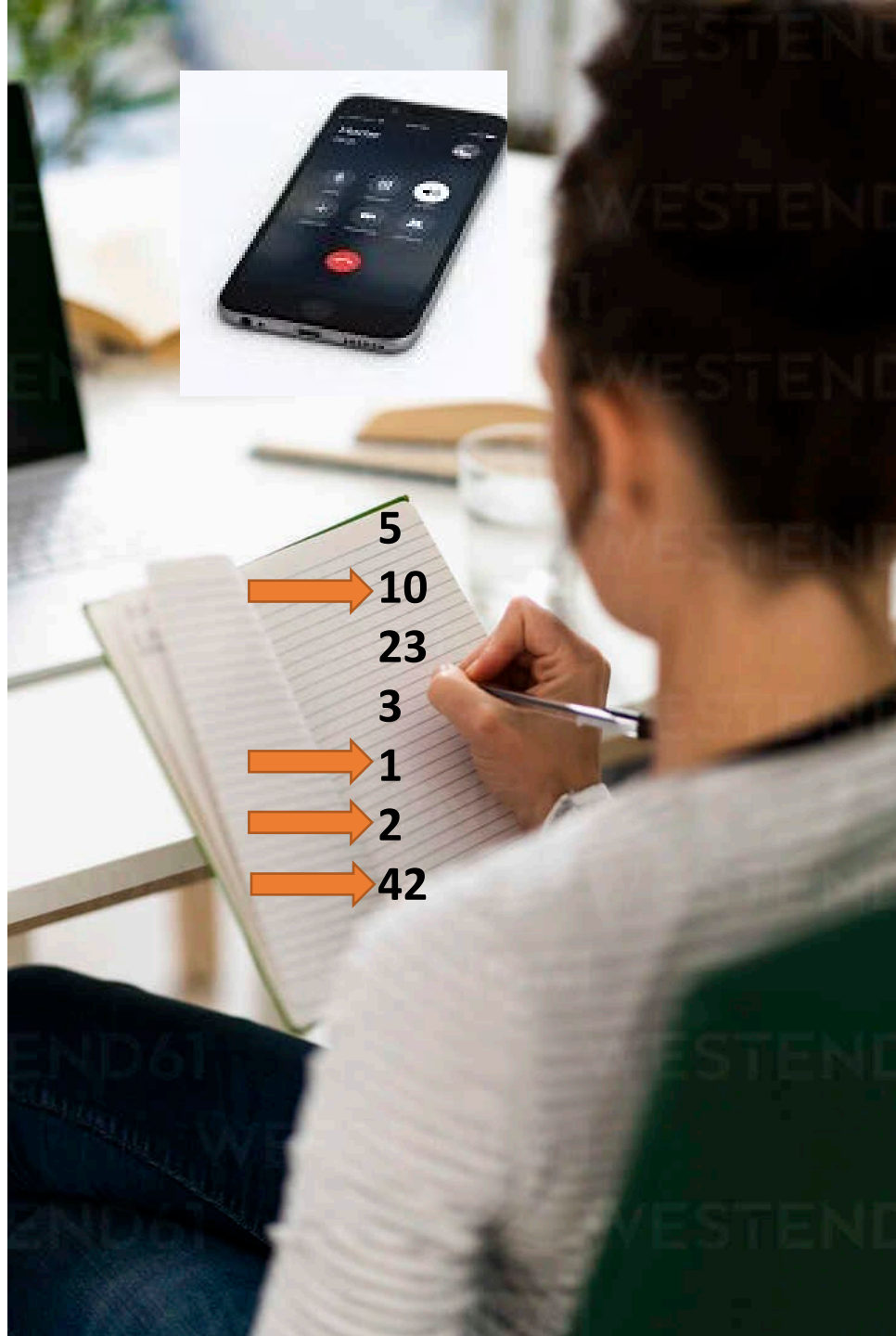
Understanding relaxed ordering

- Variables (memory locations) = woman in a cubicle with notepad
- On the notepad -> there is a list of values



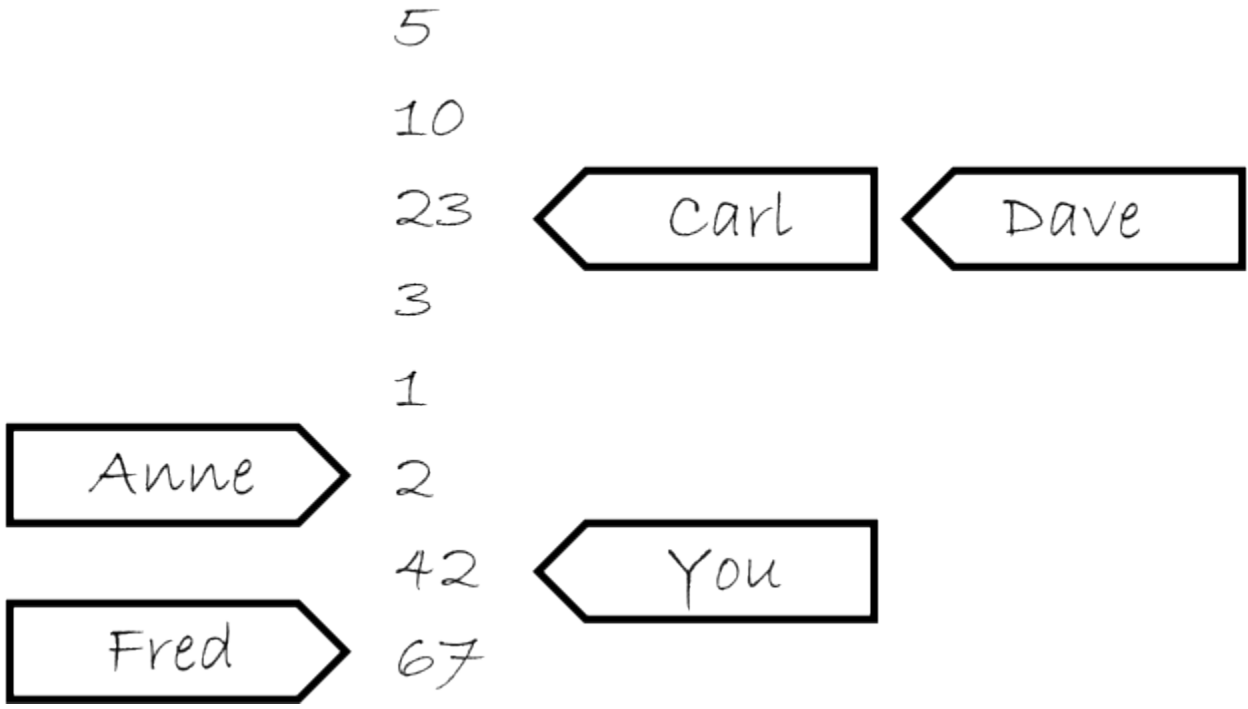
Understanding relaxed ordering

- Call her and ask her:
  - To give you a value (load) -> she will give
    - First time: any value on the list
    - Subsequent times: same value or a value from farther down the list
  - To write down a value (store) -> she will always write at the bottom of the list
    - If you ask for a value later, she must give you the value you added to the list or a value from farther down the list



Understanding relaxed ordering

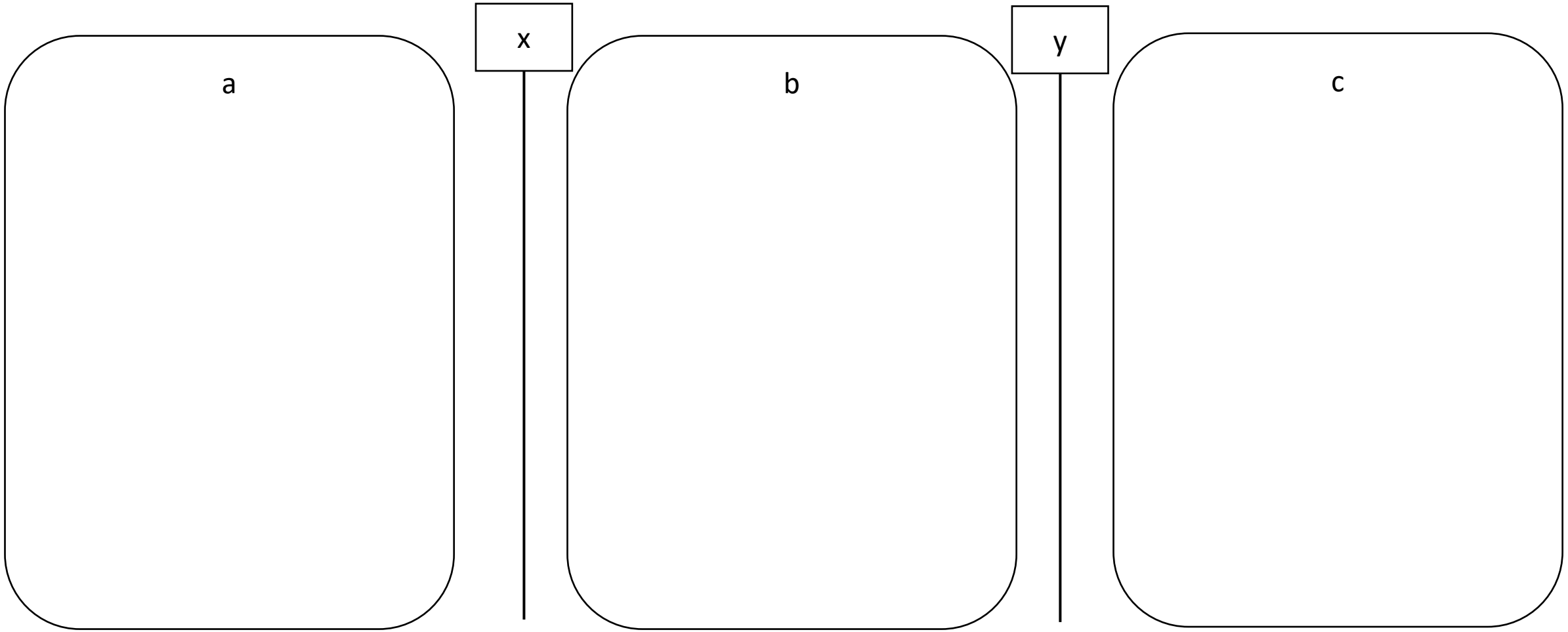
- Variable values = values in a notepad
- Thread performing stores/loads = a person calling to write/get values
  - Once or multiple times
- There are
  - Multiple notepads (variables)
  - Multiple persons (threads)
  - Multiple calls from each person (loads/stores)



Explaining memory order

- Basic building blocks of operation ordering in a program:
  - Sequenced-before
  - Synchronizes with
  - Inter-thread happens-before
  - Happens-before
  - Visible side-effect
  - Modification order

Building blocks



Sequenced before

- Within the same thread, evaluation A may be sequenced-before evaluation B
  - aka program order, but not exactly
- If A is sequenced before B (or, equivalently, B is *sequenced after* A), then evaluation of A will be complete before evaluation of B begins

Synchronizes-with relationship

- Appears between atomic load and store operations
- A suitably-tagged atomic write operation (release), W, in thread A on a variable, x, synchronizes with a suitably-tagged atomic read operation (acquire), R, in thread B on x
  - R reads the value stored by
    - either that write, W, or
    - a subsequent atomic write operation on x by the same thread that performed the initial write, W, or
    - a sequence of atomic read-modify-write operations on x (such as fetch_add() or compare_exchange_weak()) by any thread (where the value read by the first thread in the sequence is the value written by W)

Inter-thread happens-before

- Between threads, evaluation A inter-thread happens before evaluation B if any of the following is true

1) A *synchronizes-with* B

2) A is dependency-ordered before B

3) A *synchronizes-with* some evaluation X, and X is *sequenced-before* B

4) A is *sequenced-before* some evaluation X, and X inter-thread happens-before B

5) A inter-thread happens-before some evaluation X, and X inter-thread happens-before B

Happens-before

- Regardless of threads, evaluation A *happens-before* evaluation B if any of the following is true:

1) A is *sequenced-before* B

2) A *inter-thread happens before* B

Happens-before relationship

- Specifies which operations see the effects of which other operations
- Intra-thread happens-before
  - Based on program order, and *sequenced-before*
  - One operation (A) occurs in a statement prior to another (B) in the source code, then A *happens before* B (sequenced-before)
  - There is no happens-before relationship between operations that occur in the same statement
- Inter-thread happens-before
  - Based on *synchronizes-with*, and *sequenced-before*
- Transitive relationship

Visible side effects

- The side-effect A on a scalar M (a write) is visible with respect to value computation B on M (a read) if both of the following are true:
  1) A happens-before B
  2) There is no other side effect X to M where A happens-before X and X happens-before B
- If side-effect A is visible with respect to the value computation B, then the longest contiguous subset of the side-effects to M, in modification order, where B does not happen-before it is known as the visible sequence of side-effects. (the value of M, determined by B, will be the value stored by one of these side effects)

Note: inter-thread synchronization boils down to preventing data races (by establishing happens-before relationships) and defining which side effects become visible under what conditions

Modification order

- All modifications to any particular atomic variable occur in a total order that is specific to this one atomic variable.
- The following four requirements are guaranteed for all atomic operations:

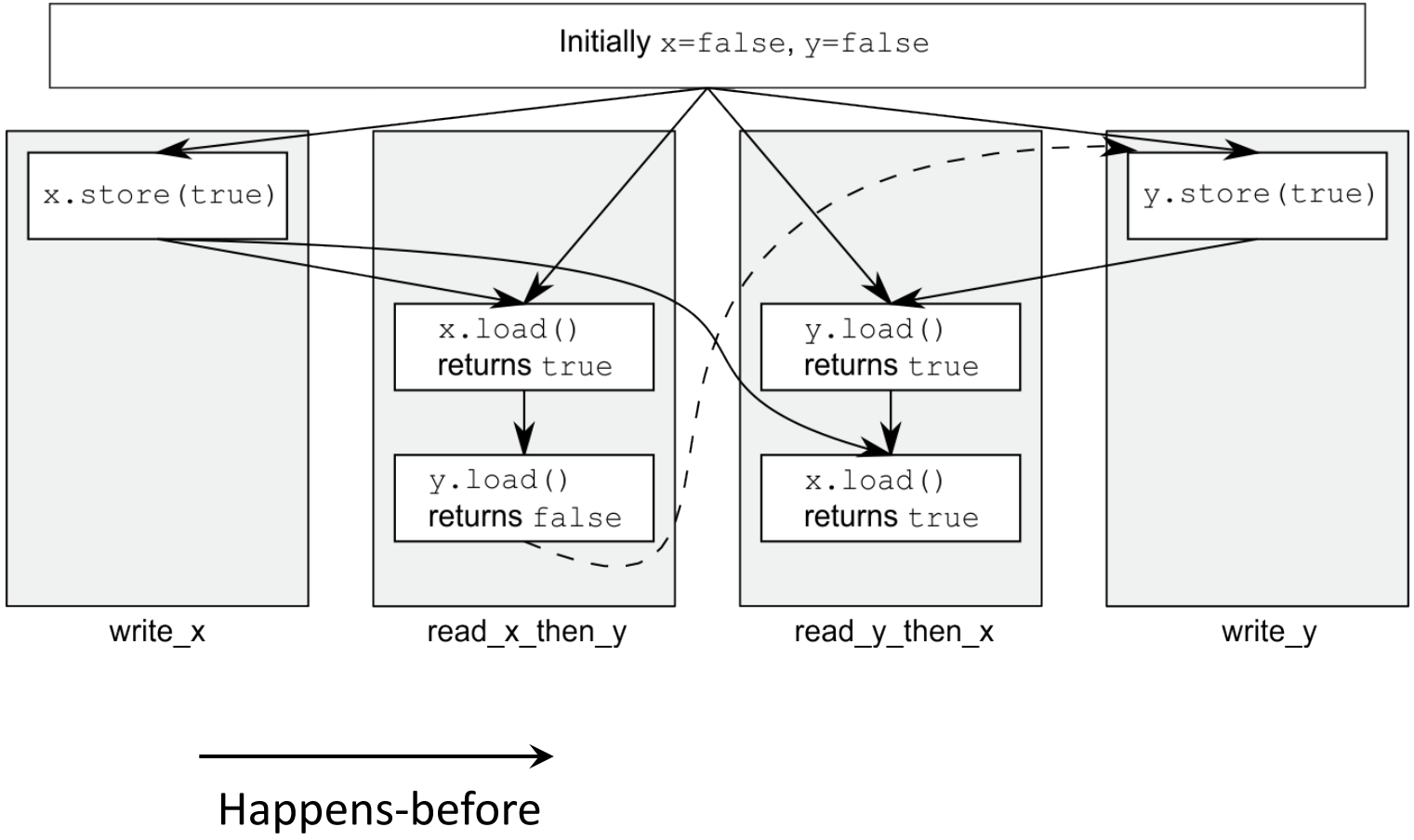
  1) **Write-write coherence**: If evaluation A that modifies some atomic M (a write) happens-before evaluation B that modifies M, then A appears earlier than B in the modification order of M

  2) **Read-read coherence**: if a value computation A of some atomic M (a read) happens-before a value computation B on M, and if the value of A comes from a write X on M, then the value of B is either the value stored by X, or the value stored by a side effect Y on M that appears later than X in the modification order of M.

  3) **Read-write coherence**: if a value computation A of some atomic M (a read) happens-before an operation B on M (a write), then the value of A comes from a side-effect (a write) X that appears earlier than B in the modification order of M

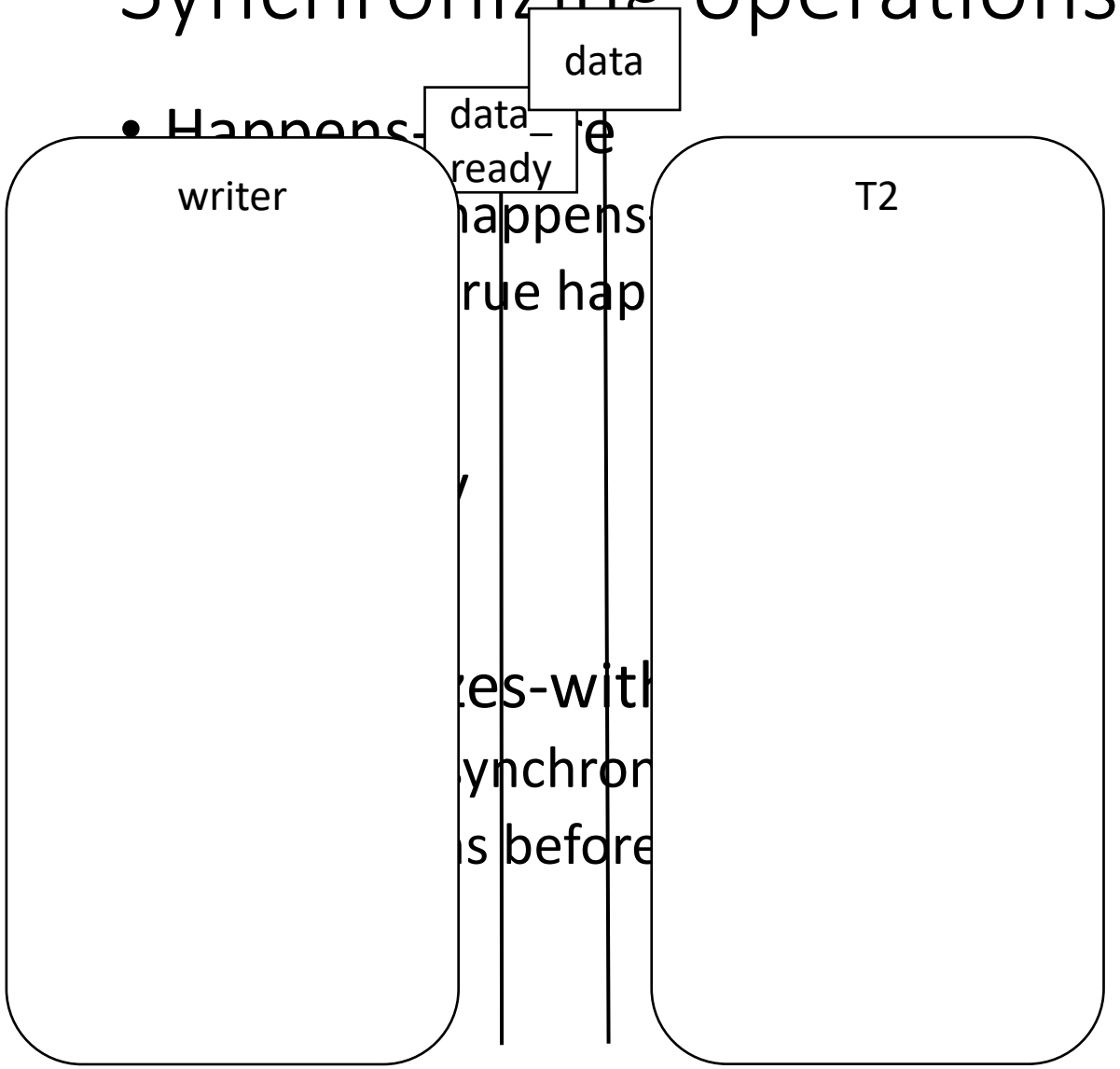
  4) **Write-read coherence**: if a side effect (a write) X on an atomic object M happens-before a value computation (a read) B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M

Memory_order_seq_cst




```cpp
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
    x.store(true,std::memory_order_seq_cst);
}
void write_y()
{
    y.store(true,std::memory_order_seq_cst);
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a{write_x};
    std::thread b{write_y};
    std::thread c{read_x_then_y};
    std::thread d{read_y_then_x};
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

Synchronizing operations and enforcing ordering




```
1  #include <vector>
2  #include <atomic>
3  #include <iostream>
4  #include <chrono>
5  #include <thread>
6  std::vector<int> data;
7  std::atomic<bool> data_ready{false};
8  void reader_thread()
9  {
10     while(!data_ready.load())
11     {
12         std::this_thread::sleep_for(
13             std::chrono::milliseconds(1));
14     }
15     std::cout<<"The answer="<<data[0]<<"\n";
16 }
17 void writer_thread()
18 {
19     data.push_back(42);
20     data_ready=true;
21 }
```

Outline

- Recap
- Atomics in C++
  - Types and atomic operations
- Memory model at the language level
  - Sequentially consistent order
  - Relaxed order
  - Acquire-release order
  - Fences

Memory ordering for atomic operations

- 3 memory models:
  - **Sequentially consistent ordering** (`memory_order_seq_cst`)
  - **Relaxed ordering** (`memory_order_relaxed`)
  - **Acquire-release ordering** (`memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, `memory_order_consume`)
- Varying costs on different CPU architectures
  - Additional synchro instructions are needed to achieve sequential consistent ordering over acquire-release ordering

Relaxed ordering

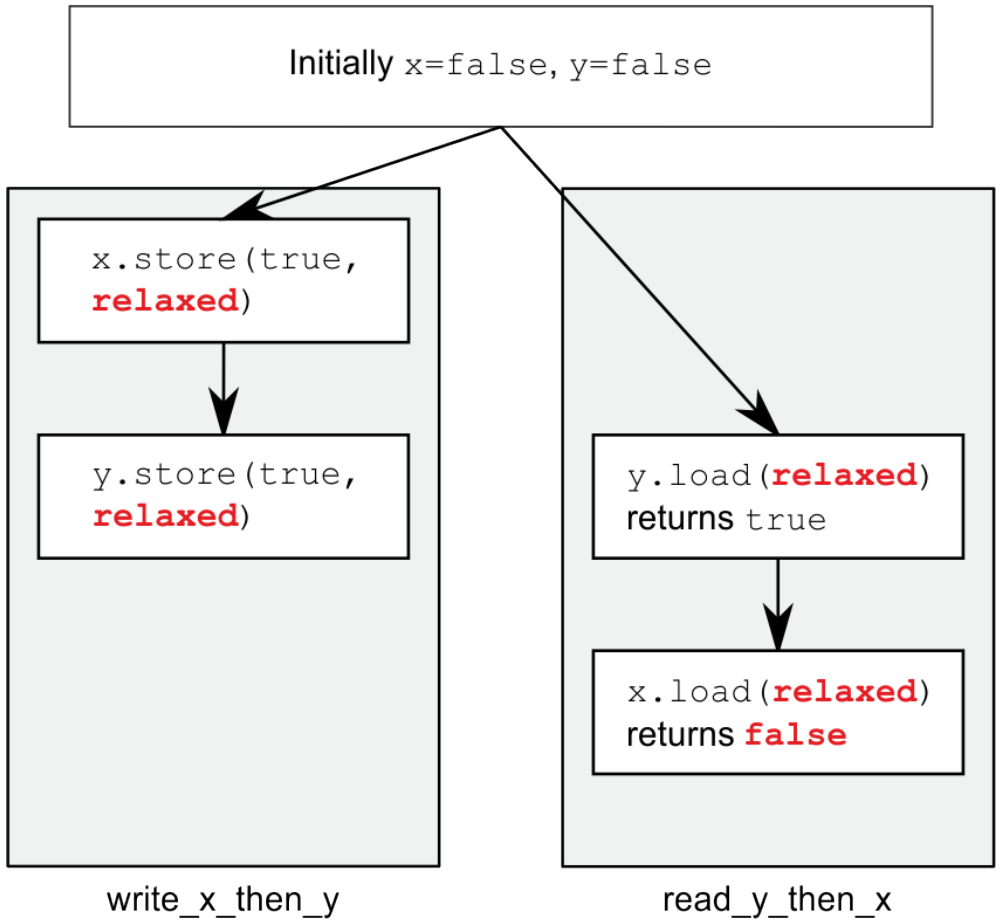
- Operations on atomic types performed with relaxed ordering don't participate in *synchronizes-with* relationships
- Operations on the same variable *within a single thread* still obey *happens-before* relationships
  - Almost no requirement on ordering relative to other threads
- Accesses to a single atomic variable from the same thread can't be reordered
  - Once a given thread has seen a particular value of an atomic variable, a subsequent read by that thread can't retrieve an earlier value of the variable

Memory_order_relaxed

- Line 26: assert can fire
  - Line 14 can read false even though 13 read true
  - Even though line 8 happens before (sequenced-before) 9
- Sequenced-before relationship
  - between the stores (lines 8-9)
  - between the loads (lines 13-14)
- No happens-before between either store and either load,
  - The loads can see the stores out of order

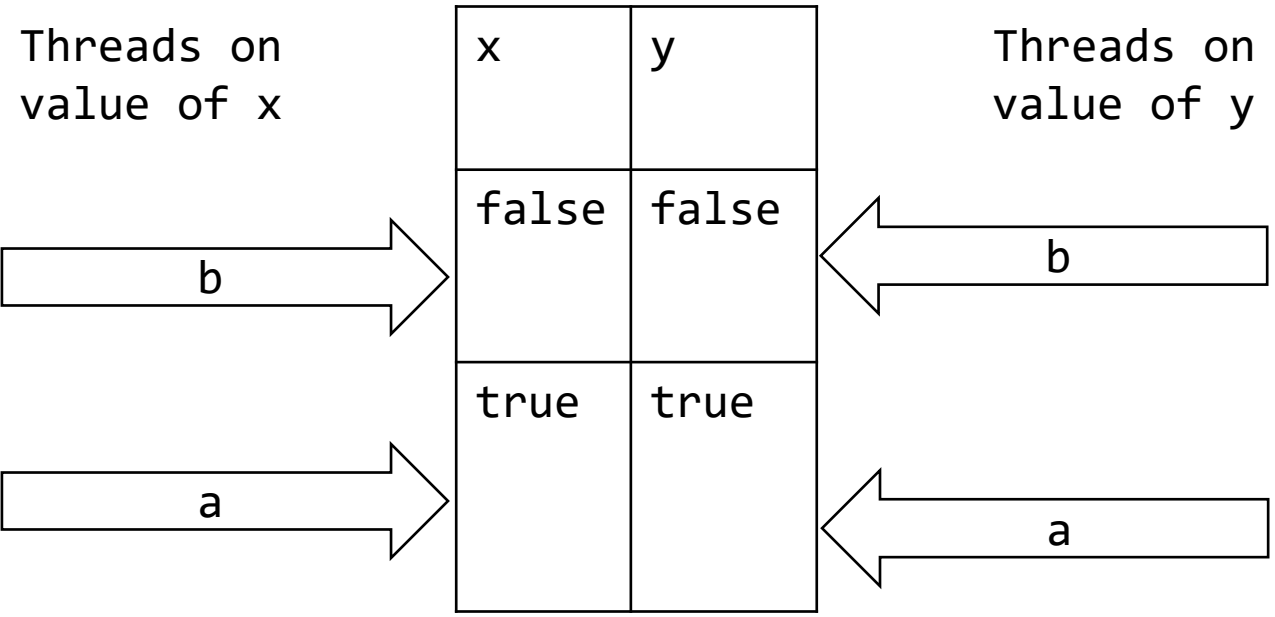
```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_relaxed);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_relaxed));
14     if(x.load(std::memory_order_relaxed))
15         ++z;
16 }
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a{write_x_then_y};
23     std::thread b{read_y_then_x};
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

Memory_order_relaxed




```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_relaxed);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_relaxed));
14     if(x.load(std::memory_order_relaxed))
15         ++z;
16 }
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a{write_x_then_y};
23     std::thread b{read_y_then_x};
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

What do threads see?




```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_relaxed);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_relaxed));
14     if(x.load(std::memory_order_relaxed))
15         ++z;
16 }
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a{write_x_then_y};
23     std::thread b{read_y_then_x};
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

Release-acquire ordering

- No total modification order, but it does introduce some synchronization
- Atomic loads use `memory_order_acquire`
- Atomic stores use `memory_order_release`
- Atomic read-modify-write operations (such as `fetch_add()` or `exchange()`) use = either *acquire*, *release*, or *acquire-release* (`memory_order_acq_rel`)

Release-acquire ordering

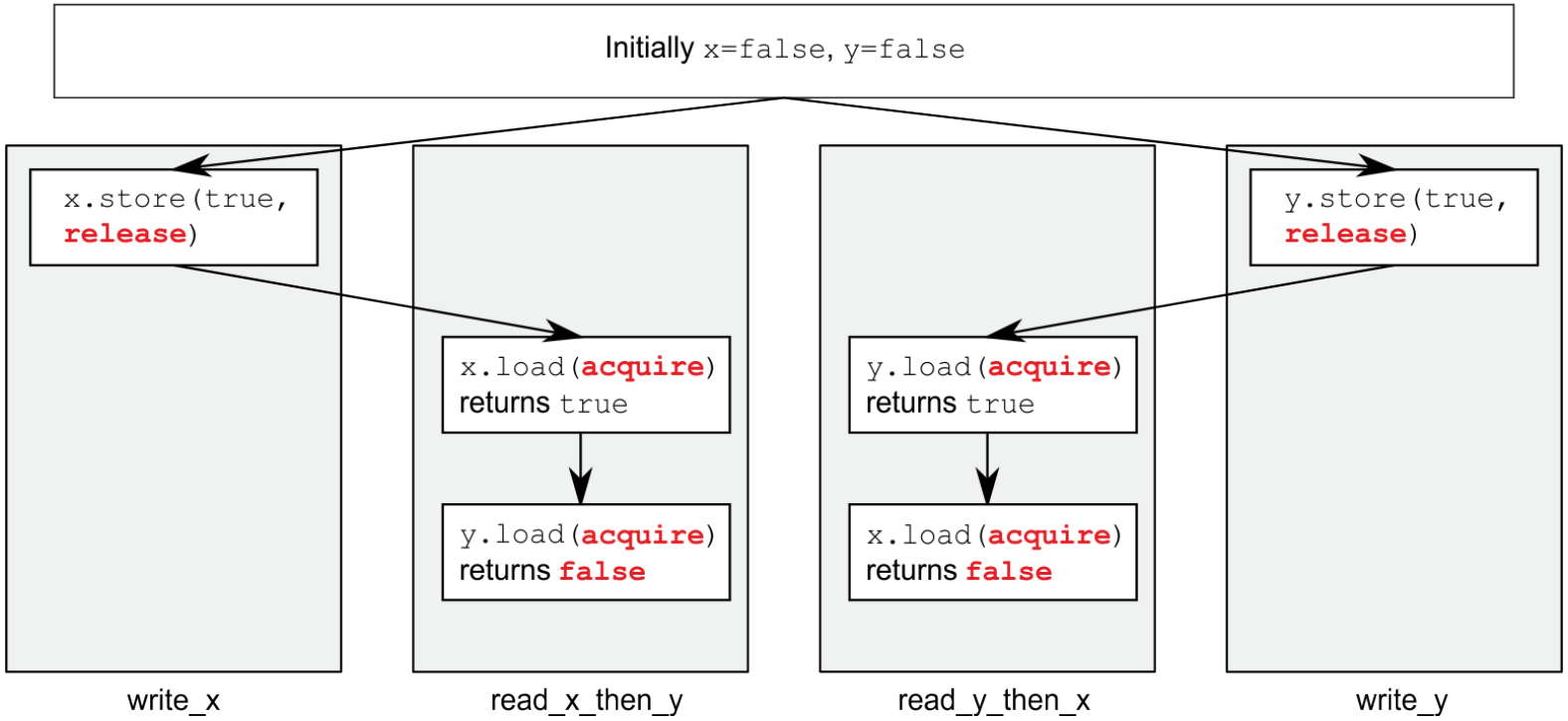
- If an atomic store in thread A is tagged `memory_order_release`, an atomic load in thread B from the same variable is tagged `memory_order_acquire`, and the load in thread B reads a value written by the store in thread A, then the store in thread A *synchronizes-with* the load in thread B.
  - All memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A, become *visible side-effects* in thread B
    - That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.
    - This promise only holds if B actually returns the value that A stored, or a value from later in the release sequence.
  - The synchronization is established only between the threads releasing and acquiring the same atomic variable.
    - Other threads can see different order of memory accesses than either or both of the synchronized threads.

Memory_order_acquire/release

- Line 39: assert can fire
  - Both lines 17 and 23 read false as x and y are written by different threads and there is no synchronization between them

```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x()
7  {
8      x.store(true,std::memory_order_release);
9  }
10 void write_y()
11 {
12     y.store(true,std::memory_order_release);
13 }
14 void read_x_then_y()
15 {
16     while(!x.load(std::memory_order_acquire));
17     if(y.load(std::memory_order_acquire))
18         ++z;
19 }
20 void read_y_then_x()
21 {
22     while(!y.load(std::memory_order_acquire));
23     if(x.load(std::memory_order_acquire))
24         ++z;
25 }
26 int main()
27 {
28     x=false;
29     y=false;
30     z=0;
31     std::thread a{write_x};
32     std::thread b{write_y};
33     std::thread c{read_x_then_y};
34     std::thread d{read_y_then_x};
35     a.join();
36     b.join();
37     c.join();
38     d.join();
39     assert(z.load()!=0);
40 }
```

Memory_order_acquire/release



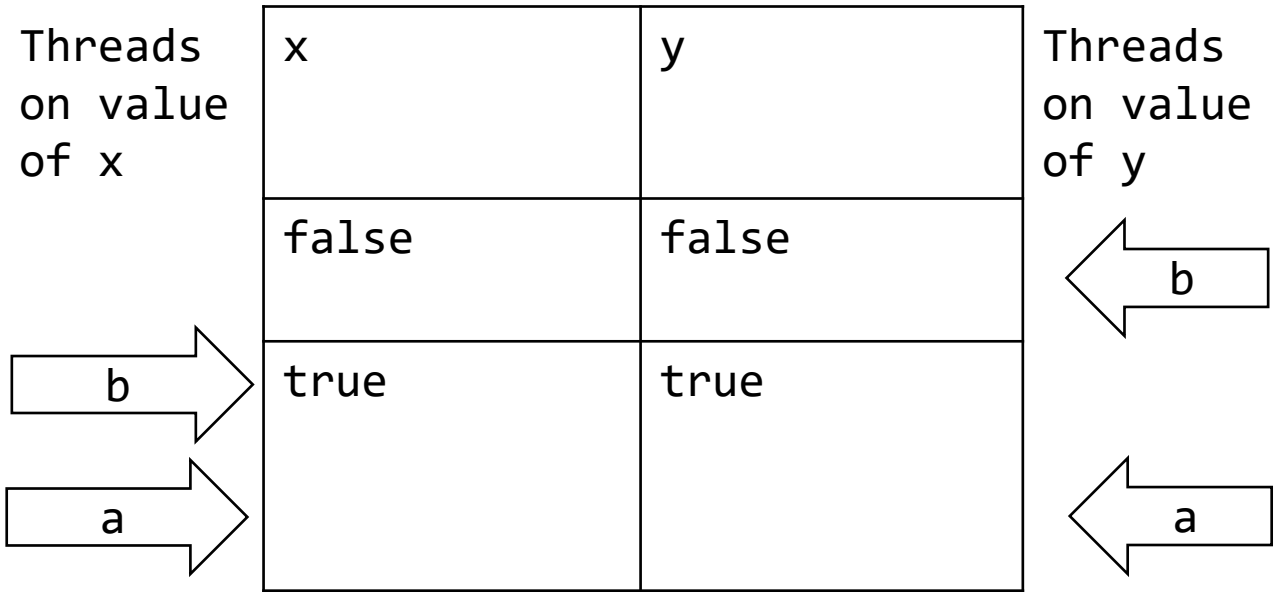

```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x()
7  {
8      x.store(true,std::memory_order_release);
9  }
10 void write_y()
11 {
12     y.store(true,std::memory_order_release);
13 }
14 void read_x_then_y()
15 {
16     while(!x.load(std::memory_order_acquire));
17     if(y.load(std::memory_order_acquire))
18         ++z;
19 }
20 void read_y_then_x()
21 {
22     while(!y.load(std::memory_order_acquire));
23     if(x.load(std::memory_order_acquire))
24         ++z;
25 }
26 int main()
27 {
28     x=false;
29     y=false;
30     z=0;
31     std::thread a{write_x};
32     std::thread b{write_y};
33     std::thread c{read_x_then_y};
34     std::thread d{read_y_then_x};
35     a.join();
36     b.join();
37     c.join();
38     d.join();
39     assert(z.load()!=0);
40 }
```

Synchronizing with acq/rel

- Line 26: assert never fires
- Line 9 synchronizes-with 13 (true)
  - 8 happens-before 9
  - 13 happens-before 14

```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_release);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_acquire));
14     if(x.load(std::memory_order_relaxed))
15         ++z;
16 }
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a(write_x_then_y);
23     std::thread b(read_y_then_x);
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

What do threads see?




```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_release);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_acquire));
14     if(x.load(std::memory_order_relaxed))
15         ++z;
16 }
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a(write_x_then_y);
23     std::thread b(read_y_then_x);
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

Language level memory models

- Modern (C++11) and not-so-modern (Java 5) languages guarantee **sequential consistency for data-race-free programs ("SC for DRF")**
  - Compilers will insert the necessary synchronization to cope with the hardware memory model
  - SC for DRF is guaranteed only if **every** atomic only uses `seq_cst`
    - No guarantees for other types of orderings, even if the program is DRF
  - No guarantees if your program contains data races!

Fences

- Operations that enforce memory-ordering constraints without modifying any data and are typically combined with atomic operations that use the `memory_order_relaxed` ordering constraints
- *Memory barriers*
  - Put a line in the code that certain operations can't cross
- An `atomic_thread_fence` with `memory_order_release` ordering prevents all preceding reads and writes from moving past all subsequent stores
  - Remember: an atomic store-release operation prevents all preceding reads and writes from moving past the store-release

Atomic_thread_fence

- Line 28: assert never fires
- Line 9 synchronizes-with 15
- Line 8 happens-before 16

- Swap 8 and 9?
  - Assert can fire

```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      std::atomic_thread_fence(std::memory_order_release);
10     y.store(true,std::memory_order_relaxed);
11 }
12 void read_y_then_x()
13 {
14     while(!y.load(std::memory_order_relaxed));
15     std::atomic_thread_fence(std::memory_order_acquire);
16     if(x.load(std::memory_order_relaxed))
17         ++z;
18 }
19 int main()
20 {
21     x=false;
22     y=false;
23     z=0;
24     std::thread a{write_x_then_y};
25     std::thread b{read_y_then_x};
26     a.join();
27     b.join();
28     assert(z.load()!=0);
29 }
```

Summary

- Covered the low-level details of the C++ memory model and the atomic operations

- References:
  - Chapter 5 from C++ Concurrency in Action
  - https://en.cppreference.com/w/cpp/atomic/memory_order#Relaxed_ordering
  - https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/