

---

并发进阶  
Advanced Concurrency

---

# 现代C++基础 Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

---

- **Memory Order Basics**
- **Atomic Variable Details**
- **Advanced Memory Order**
- **Coroutine**

Initially, I think that most of the committee members underestimated the problem. We knew that Java had a good memory model [Pugh 2004] and hoped to adopt that. I was highly amused to find that representatives from Intel and IBM effectively vetoed that idea by pointing out that by adopting the Java memory model for C++ we would slow down all JVMs by a factor of at least two. Consequently, to preserve the performance of Java, we had to adopt a far more complex model for C++. Ironically and predictably, C++ was then criticized for having a more complicated memory model than Java.

最开始，我想大多数委员都小瞧了这个问题。我们知道 Java 有一个很好的内存模型 [Pugh 2004]，并曾希望采用它。令我感到好笑的是，来自英特尔和 IBM 的代表坚定地否决了这一想法，他们指出，如果在 C++ 中采用 Java 的内存模型，那么我们将使所有 Java 虚拟机的速度减慢至少一半。因此，为了保持 Java 的性能，我们不得不为 C++ 采用一个复杂得多的模型。可以想见而且讽刺的是，C++ 此后因为有一个比 Java 更复杂的内存模型而受到批评。

# Advanced Concurrency

## Memory Order Basics

*“Even with C++11 support, I consider  
lock-free programming expert-level work.”*

-- Bjarne Stroustrup, HoPL4, P33

# Advanced Concurrency

- Memory Order Basics

- Overview
- Sequentially consistent model
- Acquire-release model
- Relaxed model

- There also exists consume-release model, but since it's very difficult for users to annotate and for compilers to analyze better optimizations, all compilers strengthen consume-release model to acquire-release model.

- C++20: *[Note 1: Prefer `memory_order::acquire`, which provides stronger guarantees than `memory_order::consume`. Implementations have found it infeasible to provide performance better than that of `memory_order::acquire`. Specification revisions are under consideration. — end note]*

- C++26: consume operations are deprecated.

Defang and deprecate  
`memory_order::consume`

# Memory order

- Current programming world stands on the foundation of sequential execution...
  - Compiler / JIT may do aggressive optimization...
    - Here we will “cache” global variables to registers, and eliminate redundant expressions (i.e.  $b = \text{addend} + 1$ ).

```
int a, b, addend;  
  
void test()  
{  
    b = addend + 1;  
    a = b - 5;  
    b = addend + 3;  
}
```

```
void test(int addend)  
{  
    int tempb = addend;    // mov esi, DWORD PTR addend[rip]  
    int tempa = tempb - 4; // lea edi, [rsi - 4]  
    tempb += 4;            // add esi, 4  
  
    a = tempa;              // mov DWORD PTR a[rip], edi  
    b = tempb;              // mov DWORD PTR b[rip], esi  
}
```

- Processors may do out-of-order execution and speculative computation...
- Each processor may have its own L1/L2 cache...

# Memory order

- These optimizations are smart and correct in sequential world, but when it comes to parallelism, some assumptions are not that intuitive...

```
int a, b, addend;  
  
void test()  
{  
    b = addend + 1;  
    a = b - 5;  
    b = addend + 3;  
}
```

```
void test(int addend)  
{  
    int tempb = addend;    // mov esi, DWORD PTR addend[rip]  
    int tempa = tempb - 4; // lea edi, [rsi - 4]  
    tempb += 4;            // add esi, 4  
  
    a = tempa;              // mov DWORD PTR a[rip], edi  
    b = tempb;              // mov DWORD PTR b[rip], esi  
}
```

- What if there is another thread that modifies **addend** here?
  - **b** can be something other than **tempb + 4**, but compiler optimizations make it impossible.

# Memory order

- Among so many compiler optimizations, processor ISA regulations, cache coherence protocols...
  - We need to find a way to unify “as-if” behaviors by abstraction!
- That is what *memory order* for in C++.
  - Three types of memory order:
    - Sequentially consistent model (`seq_cst`)
    - Acquire-release model (`acq_rel`)
    - Relaxed model (`relaxed`)
  - BTW, Rust has completely same regulations as C++.

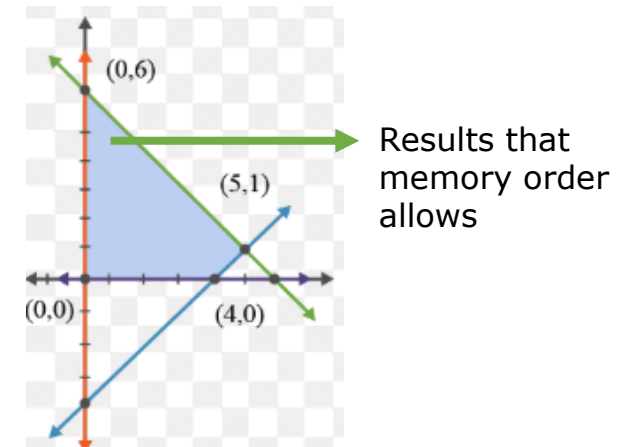
Rust pretty blatantly just inherits the memory model for atomics from C++20. This is not due to this model being particularly excellent or easy to understand. Indeed, this model is quite complex and known to have [several flaws](#). Rather, it is a pragmatic concession to the fact that *everyone* is pretty bad at modeling atomics. At very least, we can benefit from existing tooling and research around the

[Atomics - The Rustonomicon](#)

# Memory order

- But, how to describe memory order is still **an unsolved problem** even in academia (even `seq_cst` model has bug fix in C++20).
  - And C++ is pioneer in this field, so the standard has been revised nearly in every version.
  - But normally this is defect in theoretical model; real-world behaviors are not severely affected.
- The key problem is that memory order is *axiomatic*<sup>[1]</sup>, which is rather weak and cannot exactly describe what we want.
  - Memory order gives constraints, and every outcome that can fulfill the constraint is a valid solution.
    - While some solutions are not really valid...we'll see them later.

What memory order regulates:  
*constraints:*  $\begin{cases} x + y \geq 6 \\ x - y \geq 4 \\ x \geq 0 \\ y \geq 0 \end{cases}$





# Memory order

Formally, this is regulated by RR/RW/WR/WW coherence in standard; we rephrase it here.

- There are some intuitive basic regulations in memory model.
  1. Modification order: for a **single *atomic*** variable, all threads see the same operation sequences.
    - So can `r1 == 1 && r2 == 2 && r3 == 2 && r4 == 1`?
    - No!
    - Reason: r4 cannot read value newer than r3, and r2 cannot read value newer than r1.
      - `r1 == 1 && r2 == 2`: 2 is newer than 1;
      - `r3 == 2 && r4 == 1`: 1 is newer than 2; Conflict!
      - Compilers are not allowed to reorder.
    - But, operations for different atomic variables may have different orders in different threads.

```
-- Initially --
std::atomic<int> x{0};

-- Thread 1 --
x.store(1);

-- Thread 2 --
x.store(2);

-- Thread 3 --
int r1 = x.load();
int r2 = x.load();

-- Thread 4 --
int r3 = x.load();
int r4 = x.load();
```

# Memory order

2. Sequenced before: we've covered evaluation order previously...

## Expression

- Then, it's order of expression evaluation that computes the whole tree.
  - It is only determined that before the evaluation of root, the left child and the right child will be evaluated first; the order is **unspecified**.
  - e.g.  $f1() +_1 f2() +_2 f3()$ , it's  $root(+_2) \rightarrow lChild(f1() + f2()) \rightarrow rChild(f3())$ , while  $lChild$  is  $root(+_1) \rightarrow lChild(f1()) \rightarrow rChild(f2())$ ;
    - We can know before  $+_1$  is evaluated,  $lChild$  and  $rChild$  is first evaluated.
    - However, you can evaluate in the sequence of:
      - $lChild$  evaluates  $f1()$
      - $rChild$  evaluates  $f3()$ , gets the value.
      - $lChild$  evaluates  $f2()$ , gets the value.
      - This still obeys our rules, e.g.  $f1()$  and  $f2()$  evaluated before  $lChild$ .
    - So if we output  $a$  in  $f1()$ ,  $b$  in  $f2()$ ,  $c$  in  $f3()$ , any permutation of  $abc$  is possible!
  - To sum up, evaluation order is hugely determined by how compiler computes the tree.

# Memory order

- So if an evaluation **A** definitely computes before another one **B**, then we say **A** is ***sequenced before*** **B**.
  - For example, for different statements.

```
a += 1; // #1 happens before #2
b += 2; // #2
```
  - In the same statement:

```
a += 1, b += 2; // a += 1 is evaluated first
                // then it's sequenced before 'b += 2'
// Yet another smelly example.
a += b++; // b++ is evaluated first since C++17,
          // so here 'b++' sequenced before 'a +='.
```
- And function parameters are *indeterminately sequenced* since C++17, so there is some order but it's unspecified;
- And some evaluations are not regulated at all, which means they're *unsequenced* (e.g. **a = b++ + b** is UB, since **b++** and **b** are unsequenced while **b++** has side effect).
- Again, such order is in the sequential view...

# Memory order

Data races occur when non-atomic operations on the same memory location do NOT have some certain happens-before relationship.

3. Happens before: in parallel world, which evaluation is executed first is regulated by ***happens-before***.

- If A is sequenced before B, then A happens before B (single-thread case);
  - If A ***synchronizes with*** B, then A happens before B (inter-thread case);
  - Or A happens before B & B happens before C, then A happens before C.
- 
- For non-atomic variables, only when A happens before B will effects of A be visible to B.
    - So compilers can do aggressive optimizations, as long as they aren't visible.
  - For atomic variables, HB order is part of MO; if two operations have no HB relationship, then their order in MO is also random.
    - Namely, if B doesn't happen before A, then effects of A may be visible to B.
  - Memory order mainly regulates such "synchronize-with" relationship.

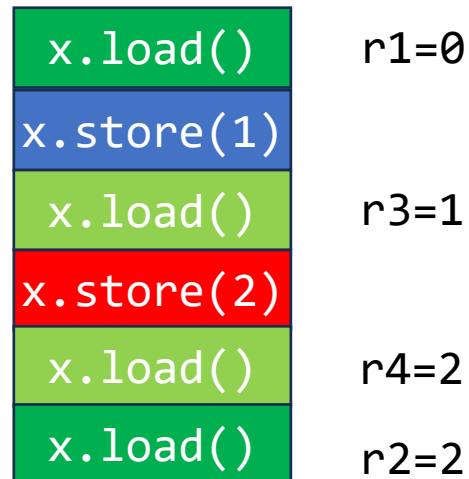
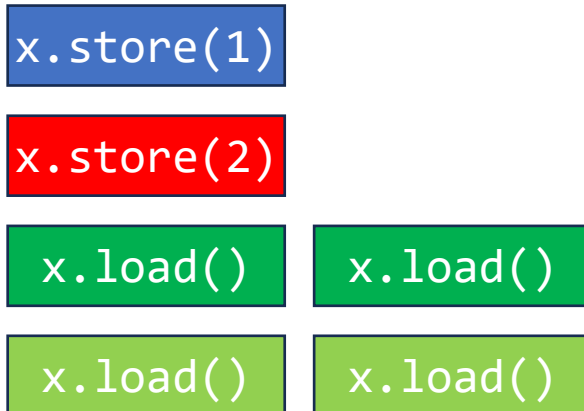
Note: actually, what we teach here is happens-before since C++26; before that (since C++20) this is called simply happens-before, but it's equivalent to happens-before (since C++11) when no consume operation is involved (and again, we've said that consume operations are never implemented).

# Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model

# Sequential Consistency

- In real world, all events are sequenced in some way, and all observers will see the same sequence.
- Similarly, we may think operations to have some total order, and all threads observe the same order.
  - This is the core of sequentially consistent model!
  - Back to our example:



Interleaving them randomly,  
we get a total order.

```
-- Initially --  
std::atomic<int> x{0};  
  
-- Thread 1 --  
x.store(1);  
  
-- Thread 2 --  
x.store(2);  
  
-- Thread 3 --  
int r1 = x.load();  
int r2 = x.load();  
  
-- Thread 4 --  
int r3 = x.load();  
int r4 = x.load();
```

# Sequential Consistency

- Formally, when an atomic load operation **B** loads a value that's stored by an atomic store operation **A**, then **A** synchronizes with **B**.
  - Then all previous outcomes are visible since **B**.

- For example:

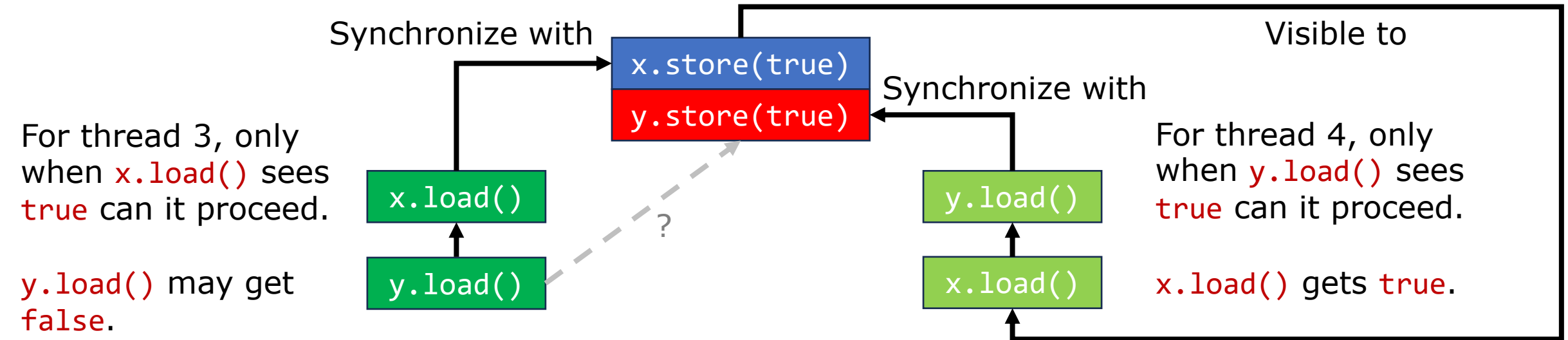
```
std::atomic<bool> x{false}, y{false};
std::atomic<int> z{0};

void write_x() { x.store(true); }
void write_y() { y.store(true); }
void read_x_then_y()
{
    while (!x.load());
    if (y.load())
        ++z;
}
void read_y_then_x()
{
    while (!y.load());
    if (x.load())
        ++z;
}
```

```
int main()
{
    { // 等四个线程全部结束。
        std::jthread a{ write_x }, b{ write_y }, c{ read_x_then_y },
                      d{ read_y_then_x };
    }
    assert(z.load() != 0);
}
```

Can this **assert** fire?

- No, `z.load()` is always non-zero.
- Reason: there is a total order, so either `x.store(true)` or `y.store(true)` occurs first.
  - Let's assume `x.store(true)` happens first since it's completely symmetric.



- Synchronization is implicitly established through reading value.
- Note: `x.store(true)` and `y.store(true)` **do NOT have happens-before relationship**; the order is imposed by total order.

seq\_cst model actually uses strongly-happens-before relationship but here they're equivalent; we'll cover it in the next section.



# Sequential Consistency

- Note 2: start of threads & joining threads will also establish synchronize-with relationship with function start & return.
  - So here thread joining happens before `z.load()`, and function return happens before thread joining, and `++z` happens before function return. Thus `z.load()` can get 1 or 2 correctly.
- Note 3: operations on atomic variables are indivisible (and thus prevent data races), which is not affected by memory order.
  - Called *atomicity*.
  - Our example in the last lecture:
    - If `a` is atomic variable, then lock protection is not needed.

```
void Inc(int& a, std::mutex& mut) {  
    for (int i = 0; i < 100000; i++)  
    {  
        std::lock_guard _{ mut };  
        a++;  
    }  
}
```

# Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model

# Acquire-release Model

- In many architectures like RISC-V, ARM and Power, such total-order assumption is quite expensive, while they support weaker model better.
  - Acquire-release is a commonly supported order!
- So what does acquire-release model guarantee?
  - Only read operations can be “acquire”, and only write operations can be “release”.
  - For an acquire operation **B**, if it reads the value from a release operation **A**, **then A synchronizes with B** (and thus **A** happens before B).
  - **There is no total order.**

# Acquire-release Model

- For example:

Sequenced before store, thus happens before store.

Only when `ptr` loads some value will the program proceed, then store synchronizes with load (and thus happens before load).

Sequenced after load, thus load happens before asserts.

```
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // never fires
    assert(data == 42); // never fires
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Through three happens-before, we know that `data` is always 42.

# Acquire-release Model

- On the other hand, since it doesn't have total order:

```
void write_x() { x.store(true, std::memory_order_release); }
void write_y() { y.store(true, std::memory_order_release); }
void read_x_then_y()
{
    while (!x.load(std::memory_order_acquire));
    if (y.load(std::memory_order_acquire))
        ++z;
}
void read_y_then_x()
{
    while (!y.load(std::memory_order_acquire));
    if (x.load(std::memory_order_acquire))
        ++z;
}
```

Store of x and y have no happens-before relationship.

Here we only know that x is true (synchronize-with), while **y.load** and **y.store** don't necessarily have happens-before relationship.

Similarly, **x.load** and **x.store** don't necessarily have happens-before relationship.

Thus, **z.load()** can be 0 here.

# Acquire-release Model

- Another example for transitivity:
  - SB(#0, #1)
  - SW(#1, #2)
    - As only when #2 reads **true** can **thread\_2** proceed.
  - SB(#2, #3)
  - SW(#3, #4)
  - SB(#4, #5)
- Thus we know HB(#0, #5).

Obviously, acquire-release model can be used to implement spinlock.

```
int data = 0;
std::atomic<bool> sync1{ false }, sync2{ false };

void thread_1()
{
    data = 442; // #0
    sync1.store(true, std::memory_order_release); // #1
}

void thread_2()
{
    while(!sync1.load(std::memory_order_acquire)); // #2
    sync2.store(true, std::memory_order_release); // #3
}

void thread_3()
{
    while(!sync2.load(std::memory_order_acquire)); // #4
    assert(data == 442); // #5
}
```

# Acquire-release Model

- By happens-before relationship, acquire-release model implicitly disables compiler reorder optimization.
  - An acquire operation **B** may happen after another release operation **A**...
    - If a compiler reorders statements **S1** after **B** to before **B**;
    - Or if a compiler reorders statements **S2** before **A** to after **A**;
    - Then **S1** may fail to observe results in **S2**.
  - Thus, acquire & release offers a **one-way instruction barrier** implicitly.
    - All operations that will cause side effects (that may be used by another threads) cannot go below beyond a release operation;
    - All operations that may rely on side effects cannot go above beyond an acquire operation.
  - Intuitively, acquire-release forms some critical section; you cannot move out code in between.

# Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model



# Relaxed Model

- Sometimes we may want even weaker order...
  - That is, we only need to maintain atomicity; no synchronize-with relationship is needed.
  - This is relaxed model.
- For example:

```
std::atomic<int> x{0}, y{0};

void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_acquire); // #1
    x.store(r1, std::memory_order_release); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_acquire); // #3
    y.store(42, std::memory_order_release); // #4
}
```

```
int main()
{
    int r1 = 0, r2 = 0;
    std::jthread{ read_y_then_write_x, std::ref(r1) },
                { read_x_then_write_y, std::ref(r2) };
    assert(!(r1 == 42 && r2 == 42));
}
```

Exercise: Can this assert fire?

# Relaxed Model

- No!
- Assuming that  $r1 == 42$ ,
  - Then #1 reads value from #4, and acquire-release model makes SW(#4, #1).
  - And SB(#3, #4), SB(#1, #2), thus we know HB(#3, #2).
  - Thus, effects of #2 are not visible to #3, and r2 is definitely 0.
- Then what about relaxed model?
  - This assertion may fire...
  - That is,  $r1 == 42 \ \&\& \ r2 == 42$  may be **true**.

```
std::atomic<int> x{0}, y{0};
```

```
void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_acquire); // #1
    x.store(r1, std::memory_order_release); // #2
}
```

```
void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_acquire); // #3
    y.store(42, std::memory_order_release); // #4
}
```

```
void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_relaxed); // #1
    x.store(r1, std::memory_order_relaxed); // #2
}
```

```
void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_relaxed); // #3
    y.store(42, std::memory_order_relaxed); // #4
}
```

# Relaxed Model

- Since relaxed model doesn't establish any synchronize-with relationship...
- Remember our effect rules?

- For atomic variables, HB order is part of MO; if two operations have no HB relationship, then their order in MO is also random.
  - Namely, if B doesn't happen before A, then effects of A may be visible to B.

- So here #1 doesn't happen before #4, then effects of #4 can be read by #1 so `r1 == 42` can be true.
  - And #1 happens before #2, so x can store 42.
  - And #3 doesn't happen before #2, then effects of #2 can be read by #3 so `r2 == 42` can be true.
- Thus, `r1 == 42 && r2 == 42` can be true.

```
void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_relaxed); // #1
    x.store(r1, std::memory_order_relaxed); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_relaxed); // #3
    y.store(42, std::memory_order_relaxed); // #4
}
```

# Relaxed Model

- Note 1: again, we emphasize that there is no total order.
  - If there is, then in thread 2 SB(#3, #4) prevents any possible order to make `r2 == 42`.
  - In practice, compilers are allowed to reorder #3 and #4, since destroying such HB doesn't affect any visible effects.
- Note 2: this outcome doesn't violate modification order constraint of a single atomic variable.

All threads see this same modification order.

x	<table><tr><td>0</td><td>r1</td></tr></table>	0	r1
0	r1		
y	<table><tr><td>0</td><td>42</td></tr></table>	0	42
0	42		

#3 can read r1,  
so it can be 42.

#1 can read 42,  
so r1 can be 42.

Notice that here it's  
**can** instead of **must**;  
#1 and #3 can read  
older values.

# Relaxed Model

- Another complex example:

```
void increment(std::atomic<int>* var, ValueContainer* values)
{
    start.wait(false); Like a spinlock, covered later.
    for (unsigned int i = 0; i < loop_num; i++)
    {
        values[i].x = x.load(std::memory_order_relaxed);
        values[i].y = y.load(std::memory_order_relaxed);
        values[i].z = z.load(std::memory_order_relaxed);

        var->store(i + 1, std::memory_order_relaxed);
    }
}
```

```
void read_status(ValueContainer* values)
{
    start.wait(false);
    for (unsigned int i = 0; i < loop_num; i++)
    {
        values[i].x = x.load(std::memory_order_relaxed);
        values[i].y = y.load(std::memory_order_relaxed);
        values[i].z = z.load(std::memory_order_relaxed);
    }
}
```

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

constexpr unsigned int loop_num = 10;
struct ValueStatus { int x, y, z; };

using ValueContainer = std::array<ValueStatus, loop_num>;
```

```
int main()
{
    std::array<ValueContainer, 5> values;
    {
        std::jthread a{ increment, &x, &values[0] }, b{ increment, &y,
&values[1] }, c{ increment, &z, &values[2] };
        std::jthread d{ read_status, &values[3] }, e{ read_status, &values[4] };

        start.store(true);
        start.notify_all(); All threads start now.
    }

    for (const auto& cont: values)
    {
        std::print("[");
        for (auto val : cont)
            std::print("({}, {}, {}) ", val.x, val.y, val.z);
        std::println("]");
    }
}
```

# Relaxed Model

- So what it does is:
  - Three threads, with each one only modifying one of the atomic variables, and reading all of them;
  - Two threads that only read all atomic variables.
- It can only guarantee that:
  - The thread that modifies the variable will see it increases one by one, constrained by happens-before relationship.
    - For example, `values[0]` will have `(0, ..., ...)`, `(1, ..., ...)`, ..., `(9, ..., ...)`.
  - And constrained by single-atomic modification order, other variables that are not modified by itself will have non-decreasing values.
    - That is, once a value is read (not necessarily the newest), values older than it cannot be read.

<sup>15</sup> [Note 16: The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

# Relaxed Model

- Courtesy of *C++ Concurrency in Action, 2<sup>nd</sup> ed.* by Anthony Williams.

One possible output from this program is as follows:

```
(0,0,0) , (1,0,0) , (2,0,0) , (3,0,0) , (4,0,0) , (5,7,0) , (6,7,8) , (7,9,8) , (8,9,8) ,  
(9,9,10)  
(0,0,0) , (0,1,0) , (0,2,0) , (1,3,5) , (8,4,5) , (8,5,5) , (8,6,6) , (8,7,9) , (10,8,9) ,  
(10,9,10)  
(0,0,0) , (0,0,1) , (0,0,2) , (0,0,3) , (0,0,4) , (0,0,5) , (0,0,6) , (0,0,7) , (0,0,8) ,  
(0,0,9)  
(1,3,0) , (2,3,0) , (2,4,1) , (3,6,4) , (3,9,5) , (5,10,6) , (5,10,8) , (5,10,10) ,  
(9,10,10) , (10,10,10)  
(0,0,0) , (0,0,0) , (0,0,0) , (6,3,7) , (6,5,7) , (7,7,7) , (7,8,7) , (8,8,7) , (8,8,9) ,  
(8,8,9)
```

# Relaxed Model

- Relaxed model may cause very astonishing results, so it needs to be used with extreme caution...
  - Usually it either cooperates with other sync operations (like acquire-release model)...
  - Or it's used to do very simple job that only needs atomicity.
    - For example, `std::shared_ptr` has a counter to count its copies; when all copies are destructed, the memory is finally freed.
    - We can check the shared count by `.use_count()`, which is normally a relaxed load since it doesn't need to participate in synchronization.



# Advanced Concurrency

Atomic Variables

# Advanced Concurrency

- Atomic variables
  - Basic operations
    - `atomic_flag`
  - Specializations
    - `atomic_ref`

# Basic Operations

- We can divide atomic operations into three categories:
  - Read operations;
  - Write operations;
  - Read-Modify-Write (RMW) operations.
- And for the most general atomic types `std::atomic<T>`:
  - Read operations are `.load(memory_order=std::memory_order_seq_cst);`
    - And an `operator T`, which can only use `seq_cst` as order.
  - Write operations are `.store(T newObj, memory_order=std::memory_order_seq_cst);` Not `T&` or `std::atomic<T>&!`
    - And `operator=(T)`, which can only use `seq_cst` as order and returns `T newObj`.
    - Notice that atomic types are neither copyable nor moveable.

For methods of atomic operations, if it accepts memory order, then the default parameter is `std::memory_order_seq_cst`; if it doesn't accept memory order, then it just uses `std::memory_order_seq_cst`. We'll not repeat them in the following slides.

# Read-Modify-Write

- And we also need atomic RMW operations...

- This is not same as atomic read + atomic write, since two atomic operations are divisible.
- RMW operations are indivisible as a whole.
- For example: `a++`;
  - Read: `temp = a`; Modify: `temp++`; Write: `a = temp`.
  - If it's divisible, two threads running `Inc` may still get value other than `200000`.

```
void Inc(std::atomic<int>& a)
{
    for (int i = 0; i < 100000; i++)
    {
        a++;
    }
}
```

- And RMW operations are:

- `.exchange(T desired, memory_order) -> T`: read the original value and write `desired`; then return the original value.
  - Actually no modification, but read & write are atomic as a whole.

# Read-Modify-Write

- And a composite operation:
  - `.compare_exchange_strong(T& expected, T desired, memory_order success, memory_order failure) -> bool`:
    - Read op: read the value `v`, compare it with `expected`;
    - Modify op: no modification;
    - Write op: if equal (or called *success*), write back `desired`; otherwise (*failure*) write back nothing (but assign `expected = v`).
    - Return value means success or not.
  - So bit of strangely, its operation type depends on its read op:
    - If failure, then it's just a load operation;
    - If success, then it's RMW operation (and the whole operation is atomic).
  - And thus, you can assign two memory order.

BTW such operations are generally called CAS operations (compare-and-swap / compare-and-set).  
CAS is the basic operation for *lock-free data structures*.

# Read-Modify-Write

- Since RMW operations involve both read and write, the memory order will constrain both of them.
  - `seq_cst`: both read and write use sequential consistent model.
  - `relaxed`: both read and write use relaxed model.
  - But for acquire-release model, acquire and release are separate...
    - So you can use `std::memory_order::acq_rel`!
  - `acq_rel`: use acquire-release model, where read is acquire operation and write is release operation.
  - `acquire`: read is acquire operation while write is relaxed.
  - `release`: write is release operation while read is relaxed.
- And RMW ensures that it reads the newest value in MO, and writes the consequent result as the newest value in MO.

# Read-Modify-Write

- There also exists a one-memory-order overload:
  - `.compare_exchange_strong(T& expected, T desired, memory_order success) -> bool;`
    - Failure takes order from success, since RMW order includes read order.

Overloads	Memory model for	
	read-modify-write operation	load operation
(1,2,5,6)	success	failure
(3,4,7,8)	order	<ul style="list-style-type: none"><li>• <code>std::memory_order_acquire</code> if <code>order</code> is <code>std::memory_order_acq_rel</code></li><li>• <code>std::memory_order_relaxed</code> if <code>order</code> is <code>std::memory_order_release</code></li><li>• otherwise <code>order</code></li></ul>

- Notice that failure is not `seq_cst` by default.

# Read-Modify-Write

- Take our previous example:

```
int data = 0;
std::atomic<bool> sync1{ false }, sync2{ false };

void thread_1()
{
    data = 442; // #0
    sync1.store(true, std::memory_order_release); // #1
}

void thread_2()
{
    while(!sync1.load(std::memory_order_acquire)); // #2
    sync2.store(true, std::memory_order_release); // #3
}

void thread_3()
{
    while(!sync2.load(std::memory_order_acquire)); // #4
    assert(data == 442); // #5
}
```

Can be  
rewritten as:

But we notice that it's normally a bad idea to use RMW operation to do spinlock, since write is special in cache coherence protocol (like MESI) and harms efficiency. Here it's just an example.

```
int data = 0;
std::atomic<int> sync{0};

void thread_1()
{
    data = 442; // #0
    sync.store(1, std::memory_order_release); // #1
}

void thread_2()
{
    int expected = 1;
    while(!sync.compare_exchange_strong(expected, 2, // #2
                                        std::memory_order_acq_rel))
        expected = 1; // Restore expected to compare again.
}

void thread_3()
{
    while(sync.load(std::memory_order_acquire) != 2); // #4
    assert(data == 442); // #5
}
```



# Atomic Operations

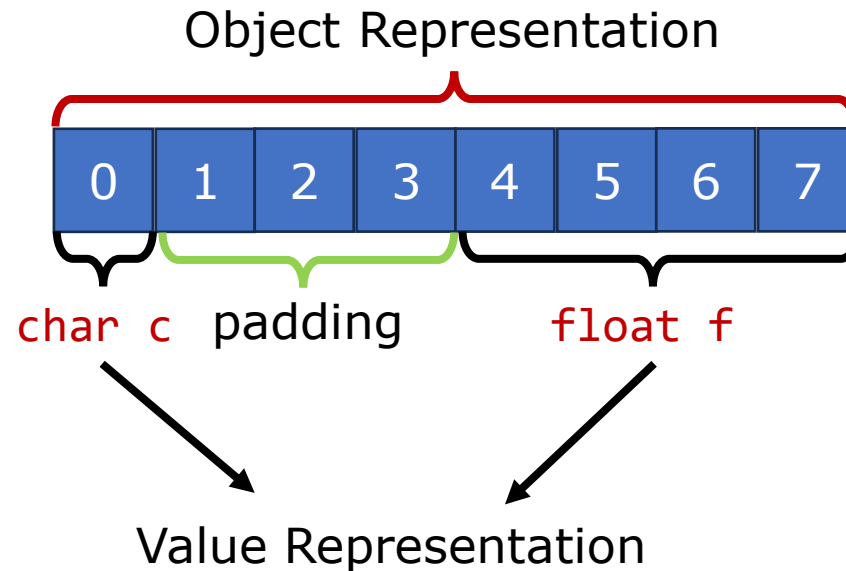
- Note 1: atomic variables are **bitwise-compared** and **bitwise-written**.
  1. A customized `operator==` doesn't affect CAS operation;
  2. Particularly, for floating points, bitwise-comparison is very misleading.
    - For example, -0.0 and 0.0 are not bitwise-equal.

```
std::atomic<float> f{ 0.0 };  
float expected = -0.0;  
f.compare_exchange_strong(expected, 1.0);
```
  3. To make bitwise-written reasonable, `std::atomic<T>` has constraint on **T**
    - **T** should be trivially copyable.

# Atomic Operations

```
struct S
{
    char c; // 1 byte value
           // 3 bytes of padding bits (assuming alignof(float) == 4)
    float f; // 4 bytes value (assuming sizeof(float) == 4)
};
```

- Note 2: padding bits are **NOT** compared **since C++20**.
  - Before C++20, they're compared.
  - Formally, object representation v.s. value representation.



We'll talk about memory layout in detail in *Memory Management*.

# Atomic Operations

```
struct S
{
    char c;
    char padding[3];
    float f;
};
```

- Reason: padding comparison may lead to astonishing false. If you really want to compare padding, you can manually pad as members.
- But, if it's atomic union, when different types have different value representations (i.e. padding positions are not same), only shared padding parts will be omitted.
- NOTICE:
  1. libc++ hasn't implemented it; libstdc++ currently implements it as DR (i.e. since gcc13, no matter what standard you specify, only value representation is compared).
  2. For union, no matter whether types have shared padding positions, MS-STL will only compare object representation (as of 2025/7). [A simple test.](#)

# Atomic Operations

- Note 3: there also exists `.compare_exchange_weak(...)`, with completely same parameters as `.compare_exchange_strong`.
  - The effects are also same, except that `weak` may fail spuriously.
    - That is, it may report failure when it's in fact equal; but when it reports success, then it's definitely equal.
    - So that in some platforms, it may be cheaper to use `weak` than `strong`.
  - Normally we don't want that spurious failure, so `weak` is usually used in a loop.

```
expected = current.load();
do {
    desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));
```

- So when the loop body is relatively cheap, `weak` can be beneficial to performance.
- But if you don't use in loop or loop is very expensive, `strong` is expected.

# spinlock

- Though we can implement spinlock by `acq_rel`, it's very inefficient.

The rules of spinlocks:

1. don't use spinlocks
2. if you do, make sure you spin on a `.load` operation
3. insert fallback strategies for when you couldn't acquire:  
always `PAUSE`, after a while switch to `UMWAIT` and if it still  
doesn't work, `futex`



- Normally we should rely on platform-dependent features.
  - Like in x86, there are lots of idle instructions (`PAUSE`, `UMWAIT`, etc.) to reduce busy-wait overhead.
  - And in OS layer, we can use lots of native utilities like `futex` on Linux, `WaitOnAddress` on Windows, etc.
- To maximize efficiency, C++20 introduces `.wait()` for atomics.

A brief but good article about idle instructions: [漫话Linux之“躺平”：IDLE 子系统](#)

- `.wait(T old, memory_order)`: block when `.load(memory_order)` is equal to `old`.
  - Similar to condition variables:
    1. You need to call `.notify_one()` and `.notify_all()` after modification to waken up waiting side;
      - And pay attention to possible ABA problem.
    2. It may spuriously wake up and do comparison, even if not notified.
- For example, again:

```
int data = 0;
std::atomic<bool> sync1{ false },sync2{ false };

void thread_1()
{
    data = 442; // #0
    sync1.store(true, std::memory_order_release); // #1
}

void thread_2()
{
    while(!sync1.load(std::memory_order_acquire)); // #2
    sync2.store(true, std::memory_order_release); // #3
}

void thread_3()
{
    while(!sync2.load(std::memory_order_acquire)); // #4
    assert(data == 442); // #5
}
```



```
int data = 0;
std::atomic<bool> sync1{ false },sync2{ false };

void thread_1()
{
    data = 442; // #0
    sync1.store(true, std::memory_order_release); // #1
    sync1.notify_one();
}

void thread_2()
{
    sync1.wait(false, std::memory_order_acquire); // #2
    sync2.store(true, std::memory_order_release); // #3
    sync2.notify_one();
}

void thread_3()
{
    sync2.wait(false, std::memory_order_acquire); // #4
    assert(data == 442); // #5
}
```

# spinlock\*

```
// Support for atomic waits.
// The "direct" functions are used when the underlying infrastructure can use WaitOnAddress directly; that is, _Size is
// 1, 2, 4, or 8. The contract is the same as the WaitOnAddress function from the Windows SDK. If WaitOnAddress is not
// available on the current platform, falls back to a similar solution based on SRWLOCK and CONDITION_VARIABLE.
int __stdcall __std_atomic_wait_direct(
    const void* _Storage, void* _Comparand, size_t _Size, unsigned long _Remaining_timeout) noexcept;
```

- About how `.wait()` is implemented, FYI.
  - Windows & MS-STL: by `WaitOnAddress` if supported in Windows SDK; otherwise by e.g. condition variable.
  - Linux & libstdc++:

```
#if __glibcxx_atomic_wait // C++ >= 20 && (linux_futex || gthread)
_GLIBCXX_ALWAYS_INLINE void
wait(bool __old,
     memory_order __m = memory_order_seq_cst) const noexcept
{
    const __atomic_flag_data_type __v
        = __old ? __GCC_ATOMIC_TEST_AND_SET_TRUEVAL : 0;

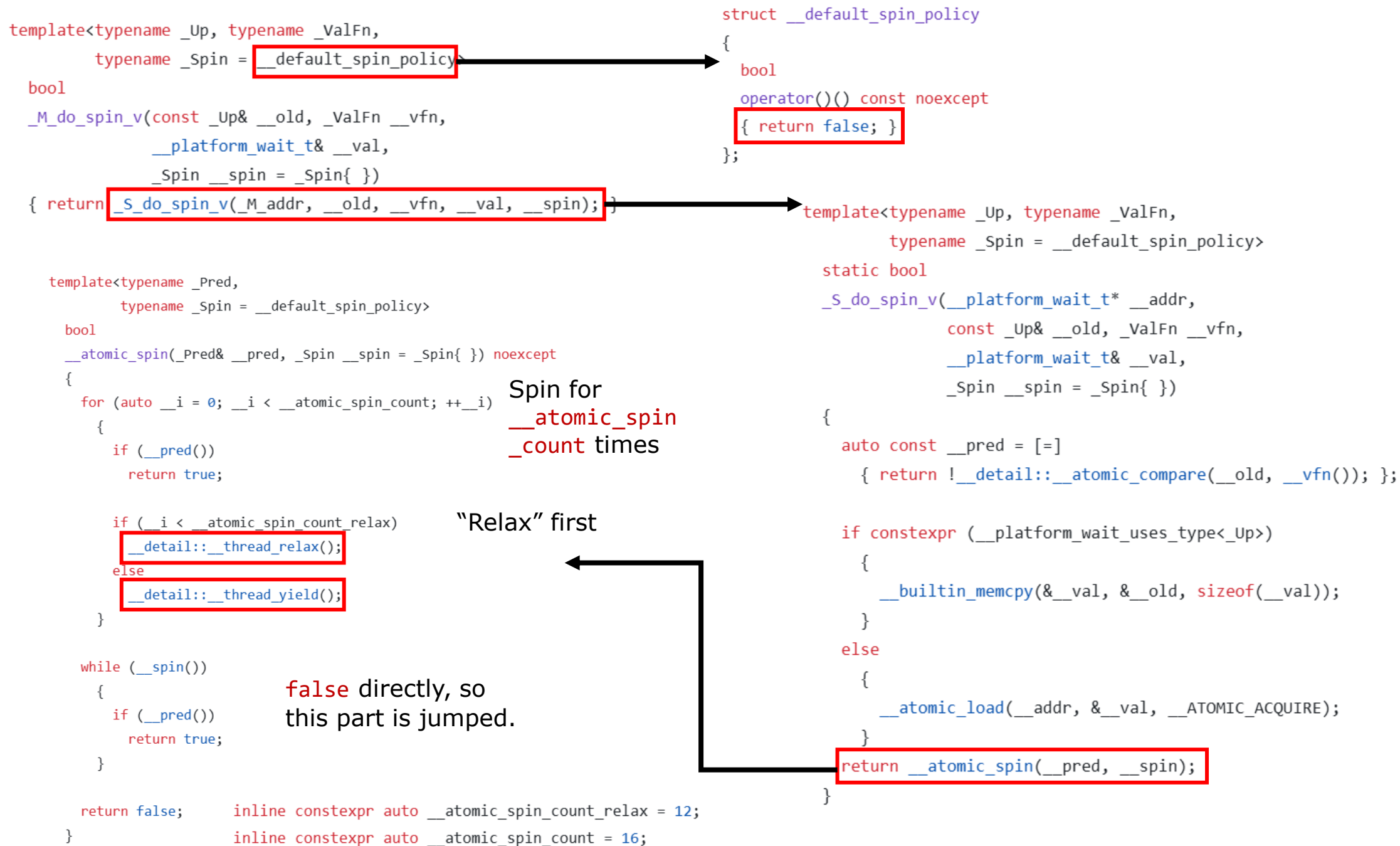
    std::__atomic_wait_address_v(&_M_i, __v,
        [__m, this] { return __atomic_load_n(&_M_i, int(__m)); });
}
```

```
template<typename _Tp, typename _ValFn>
void
__atomic_wait_address_v(const _Tp* __addr, _Tp __old,
    _ValFn __vfn) noexcept
{
    __detail::__enters_wait __w(__addr);
    __w._M_do_wait_v(__old, __vfn);
}
```

Memory order is  
contained in `__vfn`.

```
template<typename _Tp, typename _ValFn>
void
_M_do_wait_v(_Tp __old, _ValFn __vfn)
{
    do
    {
        __platform_wait_t __val;
        if (__base_type::_M_do_spin_v(__old, __vfn, __val))
            return;
        __base_type::_M_w._M_do_wait(__base_type::_M_addr, __val);
    }
    while (__detail::__atomic_compare(__old, __vfn()));
}
```

For article to introduce it, see [Implementing C++20 atomic waiting in libstdc++ | Red Hat Developer](#).





```

inline void
__thread_yield() noexcept
{
#ifdef _GLIBCXX_HAS_GTHREADS && defined _GLIBCXX_USE_SCHED_YIELD
    __gthread_yield();
#endif

    inline void
    __thread_relax() noexcept
    {
#ifdef defined __i386__ || defined __x86_64__
        __builtin_ia32_pause();
    #else
        __thread_yield();
    #endif
    }
}

```

By e.g. PAUSE instruction.

By  
futex

```

struct __waiter_pool : __waiter_pool_base
{
    void
    _M_do_wait(const __platform_wait_t* __addr, __platform_wait_t __old) noexcept
    {
#ifdef _GLIBCXX_HAVE_PLATFORM_WAIT
        __platform_wait(__addr, __old);
    #else
        __platform_wait_t __val;
        __atomic_load(__addr, &__val, __ATOMIC_SEQ_CST);
        if (__val == __old)
        {
            lock_guard<mutex> __l(_M_mtx);
            __atomic_load(__addr, &__val, __ATOMIC_RELAXED);
            if (__val == __old)
                _M_cv.wait(_M_mtx);
        }
    #endif // _GLIBCXX_HAVE_PLATFORM_WAIT
    };

    template<typename _Tp>
    void
    __platform_wait(const _Tp* __addr, __platform_wait_t __val) noexcept
    {
        auto __e = syscall (SYS_futex, static_cast<const void*>(__addr),
                           static_cast<int>(__futex_wait_flags::__wait_private),
                           __val, nullptr);

        if (!__e || errno == EAGAIN)
            return;
        if (errno != EINTR)
            __throw_system_error(errno);
    }
}

```

# Lock-free?

- Finally, usually the reason we use atomic variables instead of lock is that they're more efficient.
  - But are atomic variables really lock-free?
- No, not necessary...
  - C++ does **NOT** regulate that atomic variables should be lock-free.
  - Common platforms will support small types like integers to be lock-free by atomic instructions in ISA;
    - But if you use a very large struct, then no atomic instruction can do that!
    - Or if your platform only supports very weak ISA, then even not all atomic integers are lock-free...
- Instead, C++ provides interface to check whether it's lock-free.

For non-lock-free atomic types, you may need to link additional libraries; like in gcc and clang you need `-latomic`.

# Lock-free?

- A `constexpr static` boolean: `std::atomic<T>::is_always_lock_free`; only when on the current platform `std::atomic<T>` is definitely lock-free will it be `true`.
- A normal function: `.is_lock_free()`; some lock-free types may be only determined in runtime (e.g. when its address are over-aligned).
- C++20 adds some aliases that are guaranteed to be lock-free:

## Aliases for special-purpose types

<code>atomic_signed_lock_free</code> (C++20)	a signed integral atomic type that is lock-free and for which waiting/notifying is most efficient (typedef)
--	--

<code>atomic_unsigned_lock_free</code> (C++20)	an unsigned integral atomic type that is lock-free and for which waiting/notifying is most efficient (typedef)
--	---

Note: `std::atomic_intN_t`, `std::atomic_uintN_t`, `std::atomic_intptr_t`, and `std::atomic_uintptr_t` are defined if and only if `std::intN_t`, `std::uintN_t`, `std::intptr_t`, and `std::uintptr_t` are defined, respectively.

`std::atomic_signed_lock_free` and `std::atomic_unsigned_lock_free` are optional in freestanding implementations. (since C++20)

1. ↑ Support for always lock-free integral atomic types and presence of type aliases `std::atomic_signed_lock_free` and `std::atomic_unsigned_lock_free` are implementation-defined in a freestanding implementation. (since C++20)

# atomic\_flag

- Besides, C++ standard regulates a special type to be definitely lock-free: `std::atomic_flag`.
  - It's similar to `std::atomic<bool>`, but the latter is not regulated to be lock-free.
  - And since its value is either `true` or `false`, methods are renamed directly.
  - Read op:
    - `.test(memory_order)`, since C++20.
  - Write op:
    - `.clear(memory_order)`: set to `false`.
  - RMW op:
    - `.test_and_set(memory_order)`: set to `true` and return the previous test result.
  - Spinlock:
    - `.wait(bool old, memory_order)`, `.notify_one()`, `.notify_all()`, since C++20.

# atomic\_flag

## std::atomic\_flag::atomic\_flag

- And for ctor:

Defined in header `<atomic>`

<code>atomic_flag() noexcept = default;</code>	(1)	(since C++11) (until C++20)
<code>constexpr atomic_flag() noexcept;</code>		(since C++20)
<code>atomic_flag( const atomic_flag&amp; ) = delete;</code>	(2)	(since C++11)

Constructs a new `std::atomic_flag`.

- 1) Trivial default constructor, initializes `std::atomic_flag` to unspecified state. (until C++20)
- 1) Initializes `std::atomic_flag` to clear state. (since C++20)
- 2) The copy constructor is deleted; `std::atomic_flag` is not copyable.

In addition, `std::atomic_flag` can be value-initialized to clear state with the expression `ATOMIC_FLAG_INIT`. For an `atomic_flag` with static `storage duration`, this guarantees `static initialization`: the flag can be used in constructors of static objects.

## ATOMIC\_FLAG\_INIT

Defined in header `<atomic>`

```
#define ATOMIC_FLAG_INIT /* implementation-defined */ (since C++11)
```

Defines the initializer which can be used to initialize `std::atomic_flag` to clear (false) state in the form `std::atomic_flag v = ATOMIC_FLAG_INIT;`. It is unspecified if it can be used with other initialization contexts.

If the flag has is a `complete object` with `static storage duration`, this `initialization is static`.

This is the only way to initialize `std::atomic_flag` to a definite value: the value held after any other initialization is unspecified. (until C++20)

This macro is no longer needed since default constructor of `std::atomic_flag` initializes it to clear state. It is kept for the compatibility with C. (since C++20)

Before C++20



# Advanced Concurrency

- Atomic variables
  - Basic operations
    - `atomic_flag`
  - Specializations
    - `atomic_ref`

# Specializations

- Some atomic types are specialized to provide more convenient methods; we list them here.
  - Integers:
    - The character types `char`, `char8_t` (since C++20), `char16_t`, `char32_t`, and `wchar_t`;
    - The standard signed integer types: `signed char`, `short`, `int`, `long`, and `long long`;
    - The standard unsigned integer types: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`;
    - Any additional integral types needed by the typedefs in the header `<cstdint>`.
  - Floating points (since C++20):

When instantiated with one of the cv-unqualified floating-point types (`float`, `double`, `long double` and cv-unqualified `extended floating-point types` (since C++23)), `std::atomic` provides additional atomic operations
  - And raw pointers.

## Member types

Type	Definition
<code>value_type</code>	<code>T</code> (regardless of whether specialized or not)
<code>difference_type</code> <sup>[1]</sup>	<code>value_type</code> (only for <code>atomic&lt;Integral&gt;</code> and <code>atomic&lt;Floating&gt;</code> (since C++20) specializations) <code>std::ptrdiff_t</code> (only for <code>std::atomic&lt;U*&gt;</code> specializations)

# Specializations

- They just add common RMW operators and corresponding function overloads (to provide memory order).
  - Operators: `+=`, `-=`, `++`, `--`, `&=`, `|=`, `^=`;
    - But they do NOT return `*this`; except for postfix `++`, they return the **new** value (i.e. the stored value).
  - Functions: `fetch_xxx`, i.e. `fetch_add/sub/and/or/xor(T, memory_order)`.
    - And they return the **original** value.
  - And another two functions since C++26: `fetch_max/min(T, mo)`, which writes maximum/minimum value back.
- Floating points only provide `+=`, `-=`, `add`, `sub`;
- Pointers only provide `+=`, `-=`, `++`, `--`, `add`, `sub`, `max`, `min`;



# Specializations

- Note 1: since C++20, there also exist specializations for `std::shared_ptr` and `std::weak_ptr`, and we'll talk about them in *Memory Management*.
- Note 2: atomic pointers do NOT mean you access underlying objects atomically; they mean pointer themselves are atomic.
  - And that's why there are no `operator*` and `operator->` for atomic pointers.
  - Since C++20, `std::atomic_ref` is introduced for that atomic access.

# atomic\_ref

- An example adjusted from C++20 the Complete Guide by *Nicolai M. Josuttis*.
- Most of methods in `std::atomic_ref<T>` are same as `std::atomic<T>` as if operating on it directly.
  - So not listed again.

```
// create and initialize an array of integers with the value 100:
std::array<int, 1000> values;
std::fill_n(values.begin(), values.size(), 100);

// initialize a common stop token for all threads:
std::stop_source allStopSource;
std::stop_token allStopToken{ allStopSource.get_token() };

// start multiple threads concurrently decrementing the value:
std::vector<std::jthread> threads;
for (int i = 0; i < 9; ++i)
{
    threads.push_back(std::jthread{
        [&values](std::stop_token st) {
            while (!st.stop_requested())
            {
                std::size_t idx = GetRandomIndex(values.size());
                std::atomic_ref val{ values[idx] };
                auto newVal = --val;
                if (newVal <= 0)
                    std::println("index {} is zero", idx);
            }
        },
        allStopToken // pass the common stop token
    });
}

allStopSource.request_stop();
```

# atomic\_ref

1. To denote const reference, you can use `std::atomic_ref<const T>`; then write operations will be disabled.
  - `const std::atomic_ref<T>` is shallow const; as reference itself is already `const`, this shallow const does nothing.
2. When an object is accessed by `std::atomic_ref`, you shouldn't access it by normal reference and pointers to avoid data races.
  - And of course, you need to ensure the lifetime of referenced object doesn't end (i.e. not dangling reference).
  - And different `std::atomic_ref` shouldn't overlap. Formally:  
through those atomic\_ref instances. No subobject of the object referenced by atomic\_ref shall be concurrently referenced by any other atomic\_ref object.

# atomic\_ref

## 3. And some unique members:

- Data members:
  - `static constexpr std::size_t required_alignment`, the referenced object should align with `required_alignment`; otherwise UB.
- Methods:
  - `.address()`: since C++26, returning pointer to the referenced object.
  - copy ctor: reference the same object as another `std::atomic_ref`.
    - But it's not copy assignable.

## 4. Even if `std::atomic<T>` is lock free, `std::atomic_ref<T>` may not be lock free; their implementations are different.

# Advanced Concurrency

Advanced Memory Order

# Advanced Concurrency

- Advanced Memory Order
  - Release Sequence
  - Out-of-thin-air Problem
  - Memory Model Conflict
  - Fence

# Release Sequence

- Observe code below:

```
std::vector<int> items;
std::atomic<int> readySize{0};

void Producer()
{
    int size = 10;
    for (int i = 0; i < size; i++)
        items.push_back(i); // #0
    readySize.store(10, std::memory_order_release); // #1
}
```

```
void Consumer()
{
    while (true)
    {
        int idx = readySize.fetch_sub(1, std::memory_order_acquire); // #2
        if (idx <= 0) {
            wait_for_random_time();
            continue;
        }
        else {
            int item = items[idx - 1]; // #3
            Process(item);
        }
    }
}
```

# Release Sequence

- Assuming that there is only one producer and one consumer, then it's definitely correct.
  - If producer is not ready, then consumer will wait;
  - The first time `idx > 0`, it means that #2 read value from #1 and thus `SW(#1, #2)`.
    - And `SB(#0, #1)`, `SB(#2, #3)`, thus `HB(#0, #3)`.
    - That is, when the consumer extracts a value, it's guaranteed that the producer has already stored it, which ensures correctness.
  - And for following `fetch_sub`, since they're performed in the same thread, `SB` makes it still correct.
- But, what about one producer + two consumers?



# Release Sequence

- For the consumer that first sees  $idx > 0$ , it's still correct (as we analyzed before).
- But for the second consumer, #2<sub>2</sub> reads value from write in #2<sub>1</sub>.
  - But write in #2<sub>1</sub> is relaxed, so there is no SW(#2<sub>1</sub>, #2<sub>2</sub>);
  - Thus, we cannot conclude HB(#0, #3<sub>2</sub>)...
  - That is, when the second consumer extracts a value, it's **NOT** guaranteed that the producer has already stored it.
- To solve it, we can use `acq_rel` instead of `acquire`;
  - But we've said that `acquire` only introduces one-way barrier, while `acq_rel` will introduce two-way barrier, which harms optimization.

```
void Consumer()
{
    while (true)
    {
        int idx = readySize.fetch_sub(1, std::memory_order_acquire); // #2
        if (idx <= 0) {
            wait_for_random_time();
            continue;
        }
        else {
            int item = items[idx - 1]; // #3
            Process(item);
        }
    }
}
```

# Release Sequence

- To overcome this counter-intuitive result, C++ introduces ***release sequence***.

<sup>5</sup> A *release sequence* headed by a release operation  $A$  on an atomic object  $M$  is a maximal contiguous sub-sequence of side effects in the modification order of  $M$ , where the first operation is  $A$ , and every subsequent operation is an atomic read-modify-write operation. [\[intro.multithread\]](#)

<sup>2</sup> 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  $A$ . [\[atomics\]](#)

- In other words, the release operation can be maintained through a continuous bunch of RMW operations (no matter what memory order they have), as long as there are no other new modifications kick in.
- In our example, this means  $SW(\#1, \#2_2)$  is thus guaranteed.

# Release Sequence

- This is slightly weaker than `acq_rel`.
  - If we use `acq_rel`, we can conclude  $SW(\#2_1, \#2_2)$  and thus infer  $HB(\#1, \#2_2)$ .
  - But by release sequence, we can only conclude  $SW(\#1, \#2_2)$ ; there is no HB relationship between  $\#2_1$  and  $\#2_2$ .
- Take our previous example, again:

```
int data = 0;
std::atomic<int> sync{0};

void thread_1()
{
    data = 442; // #0
    sync.store(1, std::memory_order_release); // #1
}
```

Can we weaken this order to `relaxed`?

```
void thread_2()
{
    int expected = 1;
    while(!sync.compare_exchange_strong(expected, 2, // #2
                                        std::memory_order_acq_rel))
        expected = 1; // Restore expected to compare again.
}

void thread_3()
{
    while(sync.load(std::memory_order_acquire) != 2); // #4
    assert(data == 442); // #5
}
```

# Release Sequence

```
int data = 0;
std::atomic<int> sync{0};

void thread_1()
{
    data = 442; // #0
    sync.store(1, std::memory_order_release); // #1
}
```

- Yes, #1 + #2 is a release sequence, and thus SW(#1, #4).
  - With SB(#0, #1) and SB(#4, #5), we thus know HB(#0, #5), which means **assert** never fire.

- But code right is incorrect:

```
void thread_2()
{
    int expected = 1;
    while(!sync.compare_exchange_strong(expected, 2, // #2
                                         std::memory_order_relaxed))
        expected = 1; // Restore expected to compare again.
    assert(data == 442); // #6
}
```

- Since there is no SW(#1, #2), thus we don't know HB(#1, #6), and thus no HB(#0, #6).
  - And we say that two non-atomic operations that have no HB relationship will cause *data races*.
  - Thus, #6 is UB.
- If we use **acq\_rel** here, then from SW(#1, #2) we know HB(#0, #6), then it's correct.

# Release Sequence before C++20\*

- This part is **optional**.
- Actually before C++20, release sequence can have more components: **Release sequence**

After a *release operation* A is performed on an atomic object M, the longest continuous subsequence of the modification order of M that consists of:

- 1) Writes performed by the same thread that performed A. (until C++20)
- 2) Atomic read-modify-write operations made to M by any thread.

Is known as *release sequence headed by A*.

- And C++20 weakens release sequence; but why do C++11 introduce it while C++20 delete it?

- Consider code right:

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

void thread_1()
{
    y = 1; // #1
    x.store(1, std::memory_order_release); // #2
    x.store(3, std::memory_order_relaxed); // #3
}

void thread_2()
{
    if (x.load(std::memory_order_acquire) == 3) // #4
        assert(y == 1); // #5
}
```

# Release S

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

void thread_1()
{
    y = 1; // #1
    x.store(1, std::memory_order_release); // #2
    x.store(3, std::memory_order_relaxed); // #3
}
```

```
void thread_2()
{
    if (x.load(std::memory_order_acquire) == 3) // #4
        assert(y == 1); // #5
}
```

- We know that to execute #5, #4 needs to be true, which means that #4 needs to load value from #3.
  - But #3 is a relaxed store, thus there is no SW relationship, and thus we cannot infer HB(#1, #5).
  - So here #5 has data races with #1.
- But intuitively, since SB(#2, #3) and #2 is release operation, it's "natural" to think SW(#2, #4).
  - Thus, C++11 regulates that following writes in the same thread are also part of release sequence.
- However, this is not natural at all<sup>[1, 2]</sup>...

[1]: [Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it | POPL'15](#), Viktor et.al.

[2]: [P0982R1: Weaken release sequences](#)

# Release Sequence before C++20\*

- We may introduce a new thread:
  - If #6 kicks in between #2 and #3 in modification order of **x**...
  - Then this release sequence is destroyed and suddenly #5 has data races with #1 again.
- This is counter-intuitive again and will make program buggy.
  - That is, the relaxed order, which should not be engaged in any SW relationship, weirdly corrupts other HB relationship.

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

void thread_1()
{
    y = 1; // #1
    x.store(1, std::memory_order_release); // #2
    x.store(3, std::memory_order_relaxed); // #3
}

void thread_2()
{
    if (x.load(std::memory_order_acquire) == 3) // #4
        assert(y == 1); // #5
}

void thread_3()
{
    x.store(2, std::memory_order_relaxed); // #6
}
```

# Release Sequence before C++20\*

- Two ways to solve that:
  - [1], as an academic paper, proposes a very complex way to strengthen the memory model to make release sequence still valid;
  - [2], as a C++ proposal, proposes to minimize changes and thus weaken release sequence by cancelling the first rule.

## **Release sequence**

After a *release operation* A is performed on an atomic object M, the longest continuous subsequence of the modification order of M that consists of:

- 1) Writes performed by the same thread that performed A. (until C++20)
- 2) Atomic read-modify-write operations made to M by any thread.

Is known as *release sequence headed by A*.

- This means even if there is no thread 3, code has data races (as we reason before that HB(#1, #5) doesn't hold water).

[1]: [Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it | POPL'15](#), Viktor et.al.

[2]: [P0982R1: Weaken release sequences](#)



# Advanced Concurrency

- Advanced Memory Order
  - Release Sequence
  - Out-of-thin-air Problem
  - Memory Model Conflict
  - Fence

# Out-of-thin-air Problem

- We've said relaxed model can cause astonishing results:

- Here `r1 == 42 && r2 == 42` is possible.

- Then what about code below?

```
// Thread 1:
r1 = y.load(std::memory_order_relaxed);
if (r1 == 42)
    x.store(r1, std::memory_order_relaxed);
// Thread 2:
r2 = x.load(std::memory_order_relaxed);
if (r2 == 42)
    y.store(42, std::memory_order_relaxed);
```

```
void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_relaxed); // #1
    x.store(r1, std::memory_order_relaxed); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_relaxed); // #3
    y.store(42, std::memory_order_relaxed); // #4
}
```

- We only add two if without any new atomic operations, which should not affect any HB order, and theoretically `r1 == 42 && r2 == 42` is still a valid solution.

# Out-of-thin-air Problem

- Reasoning process is same as we point out before:

- Remember our effect rules?

- For atomic variables, if B doesn't happen before A, then effects of A may be visible to B (as long as there are no effects after A take place).

- So here #1 doesn't happen before #4, then effects of #4 can be read by #1 so `r1 == 42` can be true.

- And #1 happens before #2, so x can store 42.

- And #3 doesn't happen before #2, then effects of #2 can be read by #3 so `r2 == 42` can be true.

- However, this outcome is contradictory with logical causality (因果律) .

# Out-of-thin-air Problem

```
// Thread 1:
r1 = y.load(std::memory_order_relaxed);
if (r1 == 42)
    x.store(r1, std::memory_order_relaxed);
// Thread 2:
r2 = x.load(std::memory_order_relaxed);
if (r2 == 42)
    y.store(42, std::memory_order_relaxed);
```

- The logical preconditions are as follows ( $\rightarrow$  means “requires”):
  - #2 store happens  $\rightarrow r1 == 42 \rightarrow$  #1 loads 42  $\rightarrow$  #4 store happens  $\rightarrow r2 == 42 \rightarrow$  #3 loads 42  $\rightarrow$  #2 store happens.
  - So the precondition to make #2 happen, is that #2 has already happened.
  - This is logical fallacy, i.e. [begging the question](#) (循环论证, 即通过假设结果正确, 推出结果正确。)
- Compared with our normal example:
  - #2 store happen have **NO** precondition.
- Such logical fallacy is called “out-of-thin-air problem” in relaxed order;  **$r1 == 42 \ \&\& \ r2 == 42$**  comes from nowhere but it’s allowed by theoretical model.

```
// Thread 1:
r1 = y.load(std::memory_order_relaxed);
x.store(r1, std::memory_order_relaxed);
// Thread 2:
r2 = x.load(std::memory_order_relaxed);
y.store(42, std::memory_order_relaxed);
```

# Out-of-thin-air Problem

Thread 1	Thread 2
r1 = x; y = r1;	r2 = y; x = r2;

- If we allow it to happen, a scary example will be right too:
  - Initially `x == 0 && y == 0`, we can still get `r1 == 42 && r2 == 42`.
  - Because we can “assume” that r1 loads 42, and then we find that `x == 42 && y == 42 && r1 == 42 && r2 == 42` is a valid and consistent solution.
    - Again, we beg the question...
- However, these problems are still under investigation in academy:
  1. How can we describe out-of-thin-air problem in current model?
  2. How can compilers detect out-of-thin-air problem?
    - Currently we can only describe it by data dependency, which is almost not trackable in complex program.
  3. How can we avoid out-of-thin-air problem in the most efficient way?

[An Initial Study of Two Approaches to Eliminating Out-of-Thin-Air Results](#) is a good academic survey for this.

- Of course, lots of academic work tries to solve them with different approaches...
  - And before a widely-accepted model & description is proposed, C++ standard chooses the most conservative way to state it:

<sup>8</sup> Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

[*Note 5*: For example, with x and y initially zero,

```
// Thread 1:
r1 = y.load(memory_order::relaxed);
x.store(r1, memory_order::relaxed);

// Thread 2:
r2 = x.load(memory_order::relaxed);
y.store(r2, memory_order::relaxed);
```

this recommendation discourages producing `r1 == r2 == 42`, since the store of 42 to y is only possible if the store to x stores 42, which circularly depends on the store to y storing 42. Note that without this restriction, such an execution is possible. — *end note*]

And since out-of-thin-air problem is not well-defined now, we only assert that no processor will do operations that violate causality.

<sup>9</sup> [*Note 6*: The recommendation similarly disallows `r1 == r2 == 42` in the following example, with x and y again initially zero:

```
// Thread 1:
r1 = x.load(memory_order::relaxed);
if (r1 == 42) y.store(42, memory_order::relaxed);

// Thread 2:
r2 = y.load(memory_order::relaxed);
if (r2 == 42) x.store(42, memory_order::relaxed);
```

— *end note*]

# Advanced Concurrency

- Advanced Memory Order
  - Release Sequence
  - Out-of-thin-air Problem
  - Memory Model Conflict
  - Fence

# Memory Model Conflict

- We've said that sequential consistent model ensures total order, while acquire-release & relaxed model doesn't.
  - What if we mix their operations?
  - For example<sup>[1, 2]</sup>:

```
// Thread 1:
x.store(1, std::memory_order_seq_cst); // A
y.store(1, std::memory_order_release); // B
// Thread 2:
r1 = y.fetch_add(1, std::memory_order_seq_cst); // C
r2 = y.load(std::memory_order_relaxed); // D
// Thread 3:
y.store(3, std::memory_order_seq_cst); // E
r3 = x.load(std::memory_order_seq_cst); // F
```

- Initial value of x and y are both 0, can `r1 == 1 && r2 == 3 && r3 == 0`?

[1]: [Repairing sequential consistency in C/C++11 | PLDI'17](#), Lahav et.al.

[2]: [P0668R5: Revising the C++ memory model](#)



# Memory Model Co

```
// Thread 1:
x.store(1, std::memory_order_seq_cst); // A
y.store(1, std::memory_order_release); // B
// Thread 2:
r1 = y.fetch_add(1, std::memory_order_seq_cst); // C
r2 = y.load(std::memory_order_relaxed); // D
// Thread 3:
y.store(3, std::memory_order_seq_cst); // E
r3 = x.load(std::memory_order_seq_cst); // F
```

- In seq\_cst total order:
  - To make  $r1 == 1 \ \&\& \ r2 == 3$ , C needs to read  $y == 1$  but D needs to read  $y == 3$  and thus in total order  $C \rightarrow E$ .
    - If  $E \rightarrow C$ , then D can never get 3.
    - B is not seq\_cst operation so  $B \rightarrow C$  is not among total order.
  - To make  $r3 == 0$ , F needs to read  $x == 0$  and thus in total order  $F \rightarrow A$ .
  - And we know that SB restricts total order  $E \rightarrow F$ .
  - So in seq\_cst model, such result just needs total order  $C \rightarrow E \rightarrow F \rightarrow A$ .
- While in HB relationship...
  - We first note that in SW relationship, seq\_cst is equivalent to acq\_rel.

# Memory Model Co

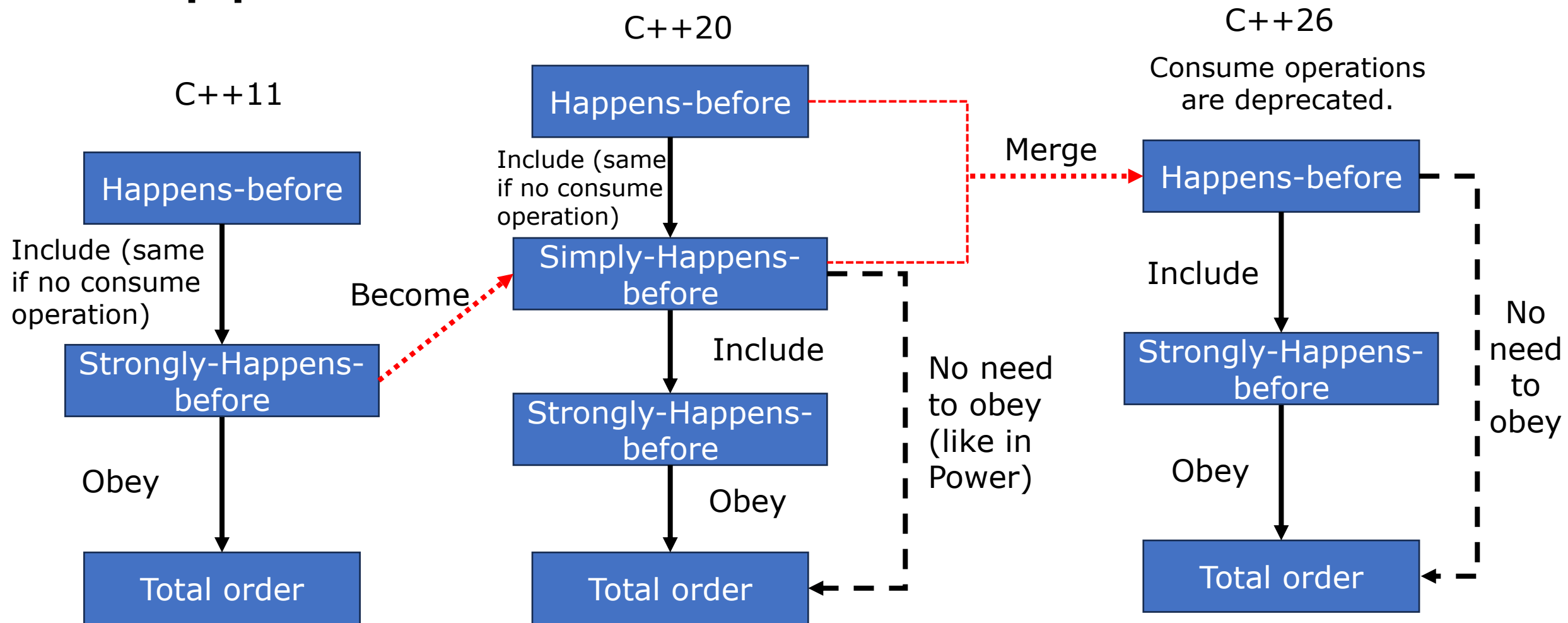
```
// Thread 1:
x.store(1, std::memory_order_seq_cst); // A
y.store(1, std::memory_order_release); // B
// Thread 2:
r1 = y.fetch_add(1, std::memory_order_seq_cst); // C
r2 = y.load(std::memory_order_relaxed); // D
// Thread 3:
y.store(3, std::memory_order_seq_cst); // E
r3 = x.load(std::memory_order_seq_cst); // F
```

- We only know:
  - SW(Init\_x, F), SB(A, B), SB(C, D), SB(E, F).
- To make  $r1 == 1$ , C reads value from B and thus SW(B, C).
  - Thus HB(A, B, C, D).
- To make  $r2 == 3$ , as long as HB(D, E) is not true.
  - And there is no way to deduce it's true, and thus it's Okay.
- So in different angles of views, we seem to get contradictory results:
  - In total order,  $C \rightarrow E \rightarrow F \rightarrow A$ ;
  - In HB order, HB(A, B, C, D).
- Before C++20, it's regulated HB order should be consistent with total order, i.e.  $r1 == 1 \ \&\& \ r2 == 3 \ \&\& \ r3 == 0$  is impossible.

# Memory Model Conflict

- However, Power and ARM allow it (especially Power)...
  - To maximize optimization, instead of fixing compilers, the C++20 standard is thus revised to allow such contradiction.
- Formally, only ***strongly-happens-before*** relationship should obey total order.
  - 1) A is *sequenced-before* B.
  - 2) A *synchronizes with* B, and both A and B are sequentially consistent atomic operations.
  - 3) A is *sequenced-before* X, X `simply(until C++26)` *happens-before* Y, and Y is *sequenced-before* B.
  - 4) A *strongly happens-before* X, and X *strongly happens-before* B.
- In our example, SW(B, C) are not all seq\_cst atomic operations, and thus SHB(A, B, C, D) is not true.
  - We can only assert SHB(A, D), since SB(A, B) && HB(B, C) && SB(C, D); but since D is not seq\_cst, SHB(A, D) is still not involved in total order.

# Happens-before Revision\*



What we teach is based on C++26; and since consume operations are never implemented, it can also be seen as based on C++20.

# Advanced Concurrency

- Advanced Memory Order
  - Release Sequence
  - Out-of-thin-air Problem
  - Memory Model Conflict
  - Fence

# Fence

- Sometimes we want to synchronize without an explicit atomic variable...
  - And fence, as a global barrier, is for that!
  - `std::atomic_thread_fence(memory_order)`.
- It somehow imposes memory order globally:
  - For a release fence, as if adding release order for following atomic writes.
    - BUT the relationship is just SW **from fence**.
  - For an acquire fence, as if adding acquire order for previous atomic reads.
    - BUT the relationship is just SW **from fence**.
- Specifically: Depending on the value of the `order` parameter, the effects of this call are:
  - When `order == std::memory_order_relaxed`, there are no effects.
  - When `order == std::memory_order_acquire` or `order == std::memory_order_consume`, is an acquire fence.
  - When `order == std::memory_order_release`, is a release fence.
  - When `order == std::memory_order_acq_rel`, is both a release fence and an acquire fence.
  - When `order == std::memory_order_seq_cst`, is a sequentially-consistent ordering acquire fence and release fence.

# Fence

- For example:
  - When #3 is true, it reads value from #2;
  - And we say #3 is “as if” an acquire operation since it’s atomic read before acquire fence...
    - So then SW relationship is established.
  - And SW starts from fence...
    - Thus it’s SW(#2, **#4**), not SW(#2, #3).
    - So HB(#1, #5), then it’s safe for this read.

```
constexpr int num_mailboxes = 32;
std::atomic<bool> mailbox_receiver[num_mailboxes]{};
std::string mailbox_data[num_mailboxes];

void writer(int i)
{
    mailbox_data[i] = compute(i); // #1
    mailbox_receiver[i].store(true, std::memory_order_release); // #2
}

void Reader()
{
    for (int i = 0; i < num_mailboxes; ++i)
    {
        if (std::mailbox_receiver[i].load(std::memory_order_relaxed)) // #3
        {
            std::atomic_thread_fence(std::memory_order_acquire); // #4
            do_work(mailbox_data[i]); // #5
        }
    }
}

int main()
{
    std::jthread writer_threads[num_mailboxes];
    for (int i = 0; i < num_mailboxes; i++)
    {
        writer_threads[i] = std::jthread{ writer, i };
    }
    std::jthread reader_thread{ Reader };
    return 0;
}
```

# Fence

```
void Reader()
{
    for (int i = 0; i < num_mailboxes; ++i)
    {
        if (std::mailbox_receiver[i].load(std::memory_order_relaxed)) // #3
        {
            do_work(mailbox_data[i]); // #5
            std::atomic_thread_fence(std::memory_order_acquire); // #4
        }
    }
}
```

- Is it still correct if we swap #4 and #5?
  - No, since SW(#2, **#4**) doesn't imply HB(#1, #5) then.
- So we can see that fence strengthens atomic operations globally, which also incurs higher overhead.
- Formally:
  - <sup>2</sup> A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X* and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X* modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written by any side effect in the hypothetical release sequence *X* would head if it were a release operation.
  - <sup>3</sup> A release fence *A* synchronizes with an atomic operation *B* that performs an acquire operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by any side effect in the hypothetical release sequence *X* would head if it were a release operation.
  - <sup>4</sup> An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced before *B* and reads the value written by *A* or a value written by any side effect in the release sequence headed by *A*.



# Fence

- Another example:

- If #6 is true, then #4 reads value from #3;
- And #4 is atomic read before acquire fence, #3 is atomic write after release fence.
  - So “as if” #3 is release operation, #4 is acquire operation.
  - Thus SW relationship from fence is established.
- Then SW(#2, #5), and thus HB(#1, #7), so #7 is safe to read.

```
std::atomic<int> arr[3] = {-1, -1, -1};
std::string data[1000]; //non-atomic data

// Thread A, compute 3 values.
void ThreadA(int v0, int v1, int v2)
{
    // assert(0 <= v0, v1, v2 < 1000);
    data[v0] = computation(v0); #1
    data[v1] = computation(v1);
    data[v2] = computation(v2);
    std::atomic_thread_fence(std::memory_order_release); #2
    std::atomic_store_explicit(&arr[0], v0, std::memory_order_relaxed); #3
    std::atomic_store_explicit(&arr[1], v1, std::memory_order_relaxed);
    std::atomic_store_explicit(&arr[2], v2, std::memory_order_relaxed);
}

// Thread B, prints between 0 and 3 values already computed.
void ThreadB()
{
    int v0 = std::atomic_load_explicit(&arr[0], std::memory_order_relaxed); #4
    int v1 = std::atomic_load_explicit(&arr[1], std::memory_order_relaxed);
    int v2 = std::atomic_load_explicit(&arr[2], std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire); #5
    // v0, v1, v2 might turn out to be -1, some or all of them.
    // Otherwise it is safe to read the non-atomic data because of the fences:
    if (v0 != -1) #6
        print(data[v0]);
    if (v1 != -1) #7
        print(data[v1]);
    if (v2 != -1)
        print(data[v2]);
}
```