

Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 9 – Memory Model and Atomics

Thomas Corbat / Felix Morgner
Rapperswil, 27.04.2019
FS2020



- **Recap Week 8**
- **The C++ Memory Model**
- **Atomic Types**



- **Participants should ...**

- know the properties of the C++ memory model
- be able to explain the differences of different memory orderings
- be able to successfully employ different memory orderings
- be able to use atomic types



Recap Week 8



```
class std::thread
```

```
int main() {  
    std::thread printer {[] (int answer) {  
        std::cout << "The answer is " << answer << std::endl;  
    }, 42};  
    printer.join();  
}
```

- Any callable can be run on a thread
- Parameters are passed at construction time
- `join()` waits for the thread to finish



```
void calcAsync() {  
    std::thread t{longRunningAction};  
    doSomethingElse();  
    t.join();  
}
```

```
void countAsync(std::string_view input) {  
    std::thread t{[&] {  
        countAs(input);  
    }};  
    t.detach();  
}
```

```
void calcAsync() {  
    std::thread t{longRunningAction};  
    doSomethingElse();  
    t.join();  
}
```

Depends

If doSomethingElse() does not throw an exception the code is correct.
If an exception is thrown you are doomed!

```
void countAsync(std::string_view input) {  
    std::thread t{[&] {  
        countAs(input);  
    }};  
    t.detach();  
}
```

Incorrect

The value parameter is captured by reference. It runs out of scope after the function execution. Furthermore, string_view is a reference wrapper itself. The string it is referring to might be destroyed as well.



- All mutexes provide the following operations

- Acquire:
 - lock() – blocking
 - try_lock() – non-blocking
- Release:
 - unlock() – non-blocking

- Two properties specify the capabilities

- Recursive – Allow multiple nested acquire operations of the same thread
 - Prevents self-deadlock
- Timed - Also provide timed acquire operations:
 - try_lock_for(<duration>)
 - try_lock_until(<time>)

		Recursive	
		No	Yes
Timed	No	<code>std::mutex</code>	<code>std::recursive_mutex</code>
	Yes	<code>std::timed_mutex</code>	<code>std::recursive_timed_mutex</code>



- Usually you will not acquire and release mutexes directly through the supplied member functions
- Instead you use a lock that manages the mutex

<code>std::lock_guard</code>	RAII wrapper for a single mutex: <ul style="list-style-type: none">• Locks immediately when constructed• Unlocks when destructed
<code>std::scoped_lock</code>	RAII wrapper for multiple mutexes <ul style="list-style-type: none">• Locks immediately when constructed• Unlocks when destructed
<code>std::unique_lock</code>	Mutex wrapper that allows deferred and timed locking: <ul style="list-style-type: none">• Similar interface to timed mutex• Allows explicit locking/unlocking• Unlocks when destructed (if still locked)
<code>std::shared_lock</code>	Wrapper for shared mutexes <ul style="list-style-type: none">• Allows explicit locking/unlocking• Unlocks when destructed (if still locked)



```
std::condition_variable
```

- **Similar to Java Condition**

- But is not bound to a lock at construction

- **Waiting for the condition**

- wait(<mutex>) – requires surrounding loop
- wait(<mutex>, <predicate>) – loops internally
- Timed waits wait_for and wait_until

- **Notifying a (potential) change**

- notify_one()
- notify_all()

- **std::unique_lock as condition releases lock (wait)**

```
template <typename T,
          typename MUTEX = std::mutex>
struct threadsafe_queue {
    using guard = std::lock_guard<MUTEX>;
    using lock = std::unique_lock<MUTEX>;
    void push(T const & t) {
        guard lk{mx};
        q.push(t);
        notEmpty.notify_one();
    }
    T pop() {
        lock lk{mx};
        notEmpty.wait(lk, [this] {
            return !q.empty();
        });
        T t = q.front();
        q.pop();
        return t;
    }
private:
    mutable MUX mx{};
    std::condition_variable notEmpty{};
    std::queue<T> q{};
};
```



- By default, `std::async` might spawn a thread... or not
 - `std::async` can take an argument of type `std::launch` (called a *launch policy*)
 - `std::launch` is an enumeration with enumerators `async` and `deferred`
 - `std::launch::async` launches a new thread
 - `std::launch::deferred` defers execution until the result is obtained from the `std::future`
 - The default is `std::launch::async` | `std::launch::deferred`

```
int main() {  
    auto the_answer = std::async(std::launch::async, [] {  
        // Calculate for 7.5 million years  
        return 42;  
    });  
    std::cout << "The answer is: " << the_answer.get() << '\n'  
}
```



The C++ Memory Model



- **The C++ Standard defines an abstract machine**

- Describes how a program is executed
- Abstracts away platform specifics
- Represents the “minimal viable computer” required to execute a valid C++ program

- **Compare to other Languages**

- Java defines the JVM
- .NET defines the CLR

- **The C++ abstract machine defines...**

- in what order Initialization takes place
- in what order a program is executed
- **what a thread is**
- **what a memory location is**
- how threads interact
- **what constitutes a data race**



- **Memory Location**

- An object of scalar type
 - arithmetic
 - pointer
 - enum
 - `std::nullptr`

- **Conflict**

- Two expression evaluations run in parallel
- Both access the same **Memory Location**
 - one reads
 - one writes

- **Data Race**

- The program contains two conflicting actions
- **Undefined Behavior!**



- **The C++ Memory Model defines**

- When the effect of an operation is visible to other threads
- How and when operations might be reordered

- **Visibility of effects:**

- *sequenced-before*: within a single thread
- *happens-before*: either sequenced-before or inter-thread happens-before
- *synchronizes-with*: inter-thread sync.

- **Note: Reads/writes in a single statement are “unsequenced”!**

```
std::cout << ++i << ++i; //don't know output
```

- **Memory orderings define when effects become visible**

- Sequentially-consistent
 - “Intuitive” and the default behavior
- Acquire/Release
 - Weaker guarantees than sequentially-consistent
- Consume (discouraged)
 - Slightly weaker than acquire-release
- Relaxed
 - No guarantees besides atomicity!



```
template<typename T>
struct atomic;
```

```
class atomic_flag;
```

- **Template class to create atomic types**
- **Atomics are guaranteed to be data-race free!**
- **Several specializations in the standard library**
- **Most basic atomic: `std::atomic_flag`**
 - Guaranteed to be lock-free
 - `clear()` – sets the flag to false
 - `test_and_set()` – set flag to true and return old value
- **Other atomics might use locks internally**
 - Check with `is_lock_free()`

```
void outputWhenReady(std::atomic_flag & flag,
                    std::ostream & out) {
    while (flag.test_and_set())
        yield();
    out << "Here is thread: "
        << get_id()
        << std::endl;
    flag.clear();
}

int main() {
    std::atomic_flag flag { };
    std::thread t { [&flag] {
        outputWhenReady(flag, std::cout);}
    };
    outputWhenReady(flag, std::cout);
    t.join();
}
```



- You can use your own types with `std::atomic`!

- However, they must be *trivially-copyable*

- Member operations (all atomic)

<code>void store(T)</code> set the new value	<code>T load()</code> get the current value	<code>T exchange(T)</code> set the new value and return the old one
<code>bool compare_exchange_weak(T & expected, T desired)</code> compare expected with current value, if equal replace the current value with desired, otherwise replace expected with current value. May spuriously fail (even when current value == expected). <code>compare_exchange_strong</code> cannot fail spuriously, but might be slower		

- Specializations like `std::atomic<int>` additionally provide atomic operators like `++`, `--`, `+=`, etc.



- **All atomic operations take an additional argument to specify the memory order (type `std::memory_order`)**

- `std::memory_order_seq_cst`
- `std::memory_order_acquire`
- `std::memory_order_release`
- `std::memory_order_acq_rel`
- `std::memory_order_relaxed`
- `std::memory_order_consume`

```
void outputWhenReady(std::atomic_flag & flag,
                    std::ostream & out) {
    while (flag.test_and_set(std::memory_order_seq_cst))
        yield();
    out << "Here is thread: "
        << get_id()
        << std::endl;
    flag.clear(std::memory_order_seq_cst);
}

int main() {
    std::atomic_flag flag { };
    std::thread t { [&flag] {
        outputWhenReady(flag, std::cout);
    } };
    outputWhenReady(flag, std::cout);
    t.join();
}
```



- **Sequential Consistency**

- Global execution order of operations
- Every thread observes the same order

- **Memory order flag**

- `std::memory_order_seq_cst`

- **Default behavior**

- **The latest modification (in the global execution order) will be available to a read**

```
std::atomic<bool> x, y;  
std::atomic<int> z;
```

Every function runs on its own thread

```
void write_x() {  
    x.store(true);  
}
```

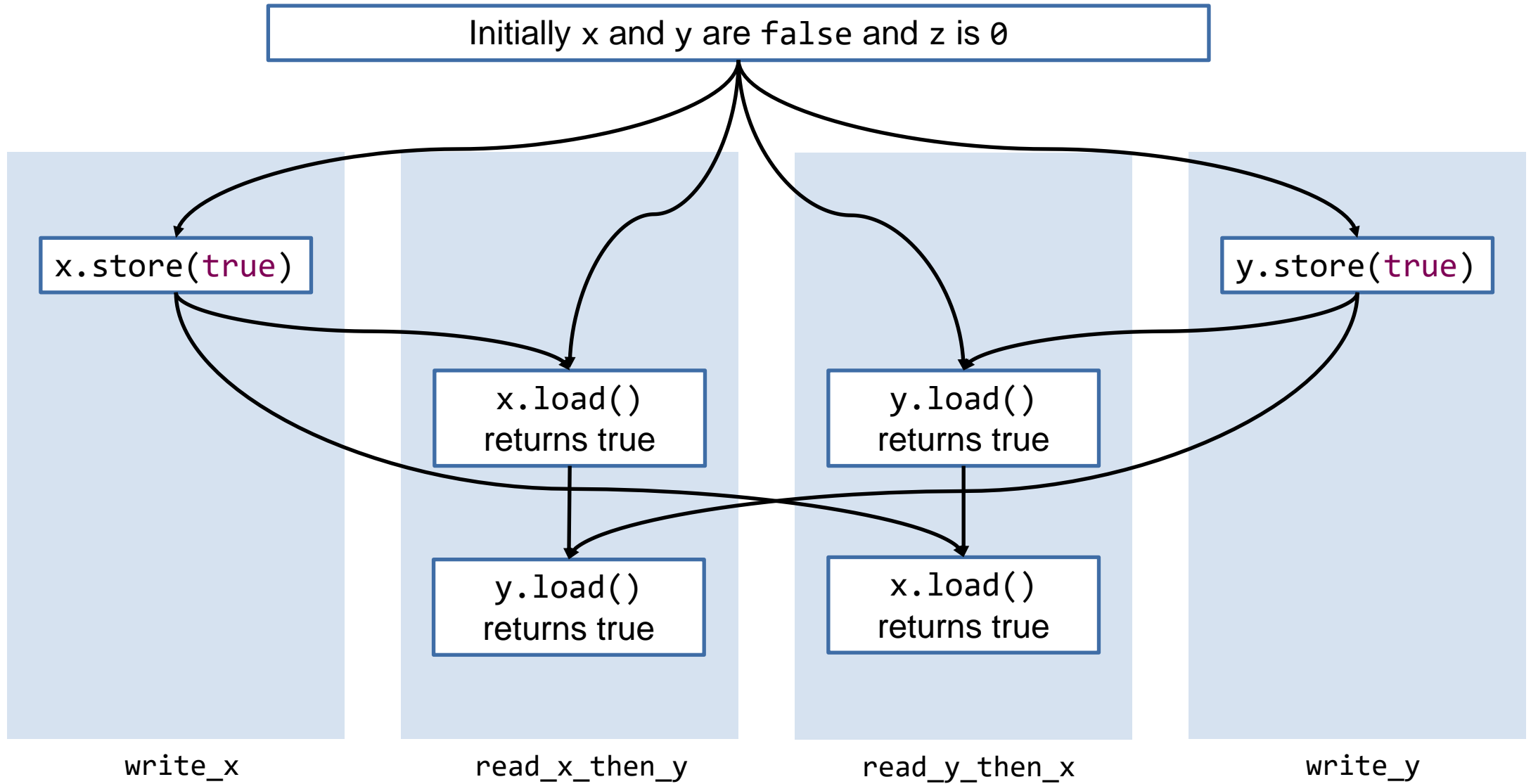
```
void read_x_then_y() {  
    while (!x.load());  
    if (y.load()) ++z;  
}
```

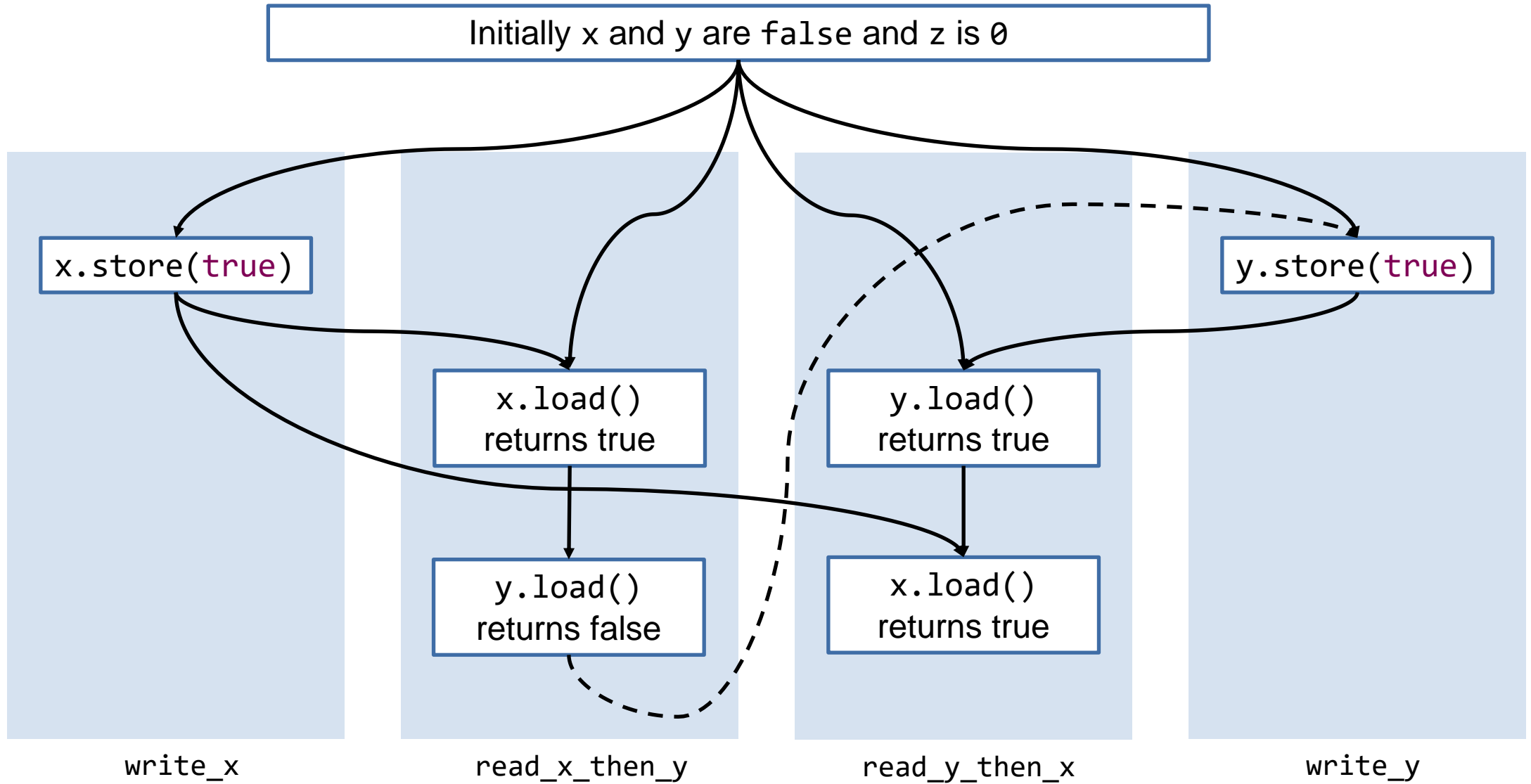
```
void write_y() {  
    y.store(true);  
}
```

```
void read_y_then_x() {  
    while (!y.load());  
    if (x.load()) ++z;  
}
```

What are possible values of z after the execution of all threads?







- **Acquire** (`std::memory_order_acquire`)

- No reads or writes in the current thread can be reordered **before** this load
- All writes in other threads that release **the same atomic** are visible in the current thread

```
x.load(std::memory_order_acquire);
```



Note: It is not guaranteed that you always see the latest write in a read operation, but what you see is consistent according to the ordering above regarding the same atomic!

- **Release** (`std::memory_order_release`)

- No reads or writes in the current thread can be reordered **after** this store
- All writes in the current thread are visible in other threads that acquire the same atomic

```
x.store(std::memory_order_release);
```

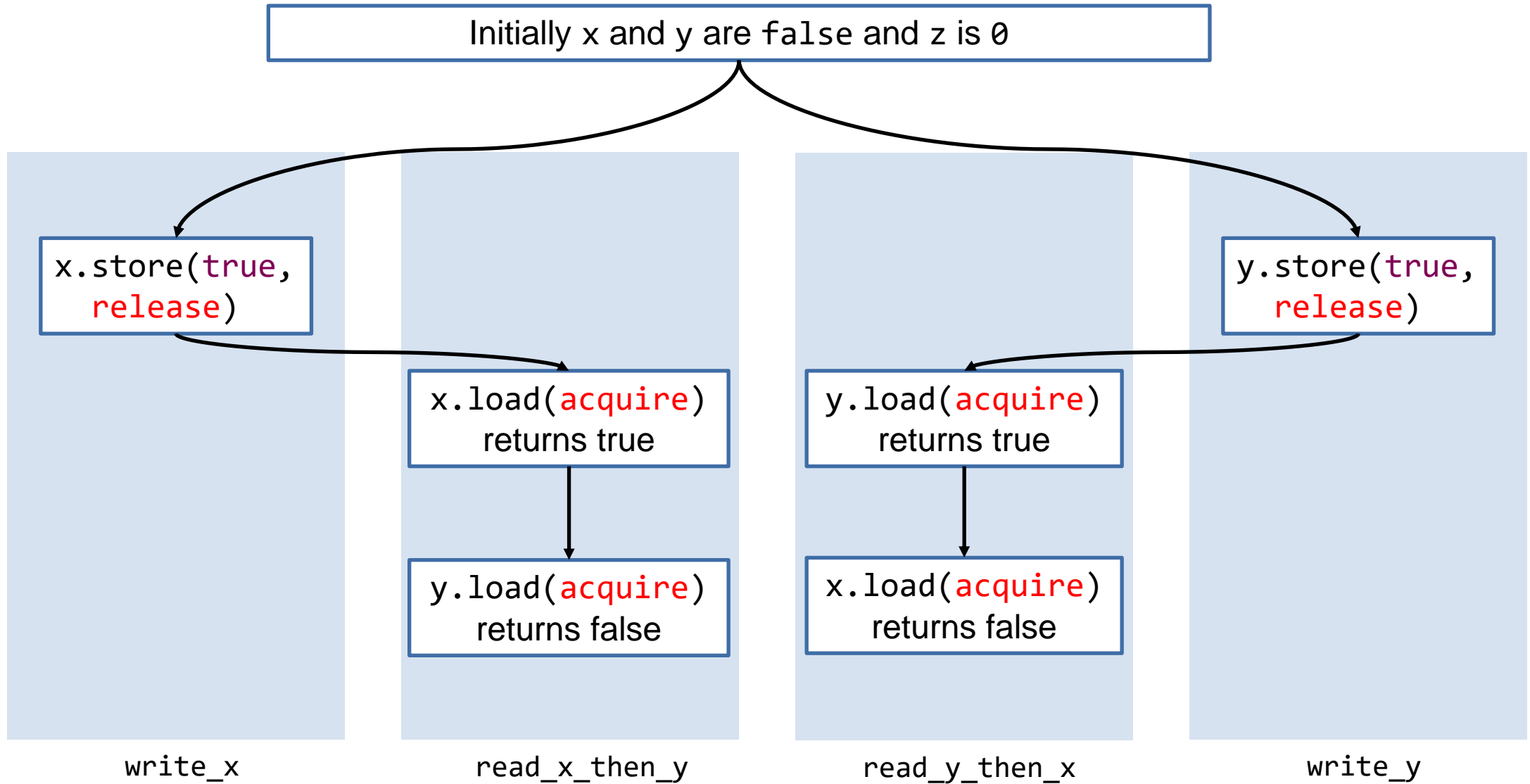


- **Acquire/Release** (`std::memory_order_acq_rel`)

- Works on the latest value

```
x.test_and_set(std::memory_order_acq_rel);
```





→ happens-before

- **Relaxed memory order**
 - Does not give promises about sequencing
 - No data-races for atomic variables
- **Order of observable effects can be inconsistent**
 - `load()` and `store()` operations happen in parallel
- **May be more “efficient” on certain platforms**
 - Less synchronization means less pipeline stalling or waiting for memory loads
- **Extremely difficult to get right! You will need to prove correctness.**

```
size_t max_count{ 10 };
size_t max_threads{ 5 };
std::memory_order order = std::memory_order_relaxed;

void increment(std::atomic<int> & x_or_y,
               std::array<values_read, max_count> & val)
{
    waitForStartFlag();
    for (size_t i = 0; i < max_count; ++i) {
        val[i].x = x.load(order);
        val[i].y = y.load(order);
        val[i].z = z.load(order);
        x_or_y.store(i + 1, order);
        yield();
    }
}

void read(std::array<values_read, max_count> &val) {
    waitForStartFlag();
    for (size_t i = 0; i < max_count; ++i) {
        val[i].x = x.load(order);
        val[i].y = y.load(order);
        val[i].z = z.load(order);
        yield();
    }
}
```



- **Custom types need to be trivially copyable**

- You can not have a custom copy ctor
- You can not have a custom move ctor
- You can not have a custom copy assignment operator
- You can not have a custom move assignment operator

- **Object can only be accessed as a whole**

- No member access operator in `std::atomic`

```
struct SimpleType {  
    int first;  
    float second;  
};
```

```
struct NonTrivialCctor {  
    NonTrivialCctor(NonTrivialCctor const&) {  
        std::cout << "copied!\n";  
    }  
};
```

```
struct NonTrivialMember {  
    int first;  
    std::string second;  
};
```



Bibs and Bobs

`volatile` and Interrupts

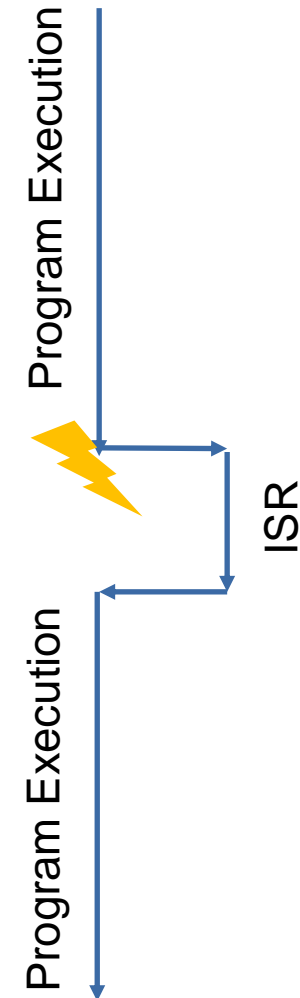


```
volatile int mem{0};
```

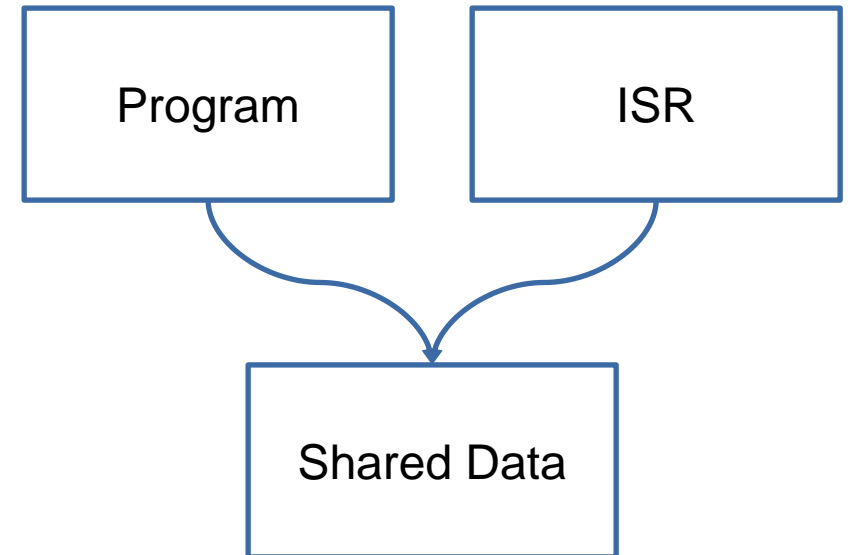
- The semantics of the volatile specifier in C++ is different from volatile in Java and C#
- Load and store operations of volatile variables must not be elided, even if the compiler cannot see any visible side-effects within the same thread
- Prevents the compiler from reordering within the same thread
 - However: The hardware might still reorder instructions!
- Useful when accessing memory-mapped hardware
 - Never use it for inter-thread communication!
- Currently there are proposals to reduce/remove volatile from the language
 - Goal is to replace it with library functionality and clean up the semantics



- **Interrupts are events originating from underlying system**
 - They interrupt the normal execution flow of the program
 - Depending on the platform, they can be suppressed
- **When an interrupt occurs, a previously registered function is called**
 - Such functions are called Interrupt Service Routines (ISRs)
 - ISRs should generally be short and must run to completion
- **After the interrupt was handled, execution of the program resumes**



- **Data shared between an ISR and the normal program execution needs to be protected**
 - All accesses must be atomic
 - Modifications need to become visible
- **Volatile helps regarding visibility**
 - Suppresses compiler optimizations
- **Interrupts may need to be disabled temporarily to guarantee atomicity**
- **Refer to your hardware manual for specific details on how to deal with interrupts**



- **On (AVR-based) Arduinos, interrupts cannot be interrupted**
 - Other platforms support so-called Interrupt-Nesting
- **Before accessing shared data, interrupts must be disabled and enabled afterwards**
 - `noInterrupts()` – disable interrupts
 - `interrupts()` – enable interrupts
- **Interrupts sources can be “external”, e.g. pins on the board**
 - Again, refer to the hardware manual

```
constexpr byte ledPin = 13;
constexpr byte switchPin = 2;

volatile bool ledState = LOW;

void toggleLed() {
    ledState = !ledState;
}

void setup() {
    pinMode(ledPin, OUTPUT);
    pinMode(switchPin, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(switchPin),
                    toggleLed,
                    CHANGE);
}

void loop() {
    noInterrupts();
    digitalWrite(ledPin, ledState);
    interrupts();
}
```



Summary



- **Writing correct multi-threaded programs is difficult**
 - Use the appropriate synchronization primitives
- **The C++ standard defines a fine-grained memory model**
 - Stick to the defaults unless:
 - You know what you are doing
 - You can demonstrate that you really need the alternatives
- **Interrupt handling requires special care**
 - Refer to the manual of the hardware you are targeting

