Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 9 – Memory Model and Atomics

Thomas Corbat / Felix Morgner
Rapperswil / St. Gallen, 25.04.2024
FS2024

Agenda

- **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

Ostschweizer
Fachhochschule

```
class std::thread
```

```
auto main() -> int {
  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**

Right or Wrong?

<pre>auto calcAsync() -> void {
  std::thread t{longRunningAction};
  doSomethingElse();
  t.join();
}</pre>
	Depends
	
	If doSomethingElse() does not throw an exception the code is correct.
	If an exception is thrown you are doomed!

<pre>auto countAsync(std::string_view input) -> void {
  std::thread t{[&] {
    countAs(input);
  }};
  t.detach();
}</pre>
	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	`std::mutex`	`std::recursive_mutex`
	Yes	`std::timed_mutex`	`std::recursive_timed_mutex`

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

`std::lock_guard`	RAII wrapper for a single mutex: • Locks immediately when constructed • Unlocks when destructed
`std::scoped_lock`	RAII wrapper for mutliple mutexes • Locks immediately when constructed • Unlocks when destructed
`std::unique_lock`	Mutex wrapper that allows defered and timed locking: • Similar interface to timed mutex • Allows explicit locking/unlocking • Unlocks when destructed (if still locked)
`std::shared_lock`	Wrapper for shared mutexes • 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>;
  auto push(T const & t) -> void {
    guard lk{mx};
    q.push(t);
    notEmpty.notify_one();
  }
  auto pop() -> T {
    lock lk{mx};
    notEmpty.wait(lk, [this] {
      return !q.empty();
    });
    T t = q.front();
    q.pop();
    return t;
  }
private:
  mutable MUTEX 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`

```
auto main() -> int {
    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

A Computer for C++

- **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**

Definitions

- **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!

```
#include <atomic>
```



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

```
class atomic_flag;
```

- **Template class to create atomic types**
- **Atomics are guaranteed to be data-race free!**
- **Several spezializations 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()`

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

auto main() -> int {
  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)**

`void` **`store`**`(T)` set the new value	`T` **`load`**`()` get the current value	`T` **`exchange`**`(T)` set the new value and return the old one
`bool` **`compare_exchange_weak`**`(T & expected, T desired)` 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`). **`compare_exchange_strong`** 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`

```
auto outputWhenReady(std::atomic_flag & flag,
                     std::ostream & out) -> void {
  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);
}

auto main() -> int {
  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

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

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

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

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

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

Example taken from Anthony Williams: C++ Concurrency in Action

Sequential Consistency (z == 2)

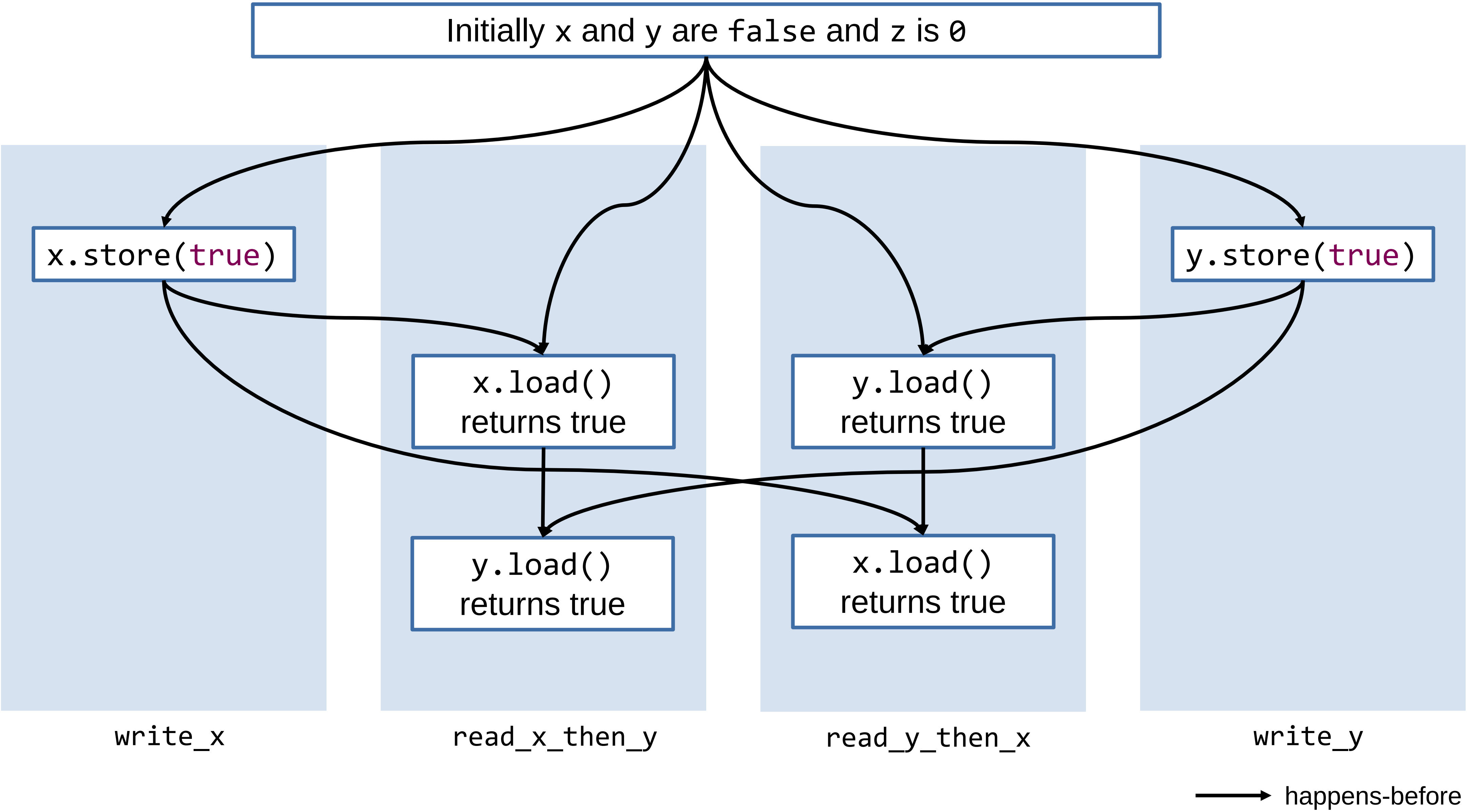


Illustration adapted from Anthony Williams: C++ Concurrency in Action

Sequential Consistency (z == 1)

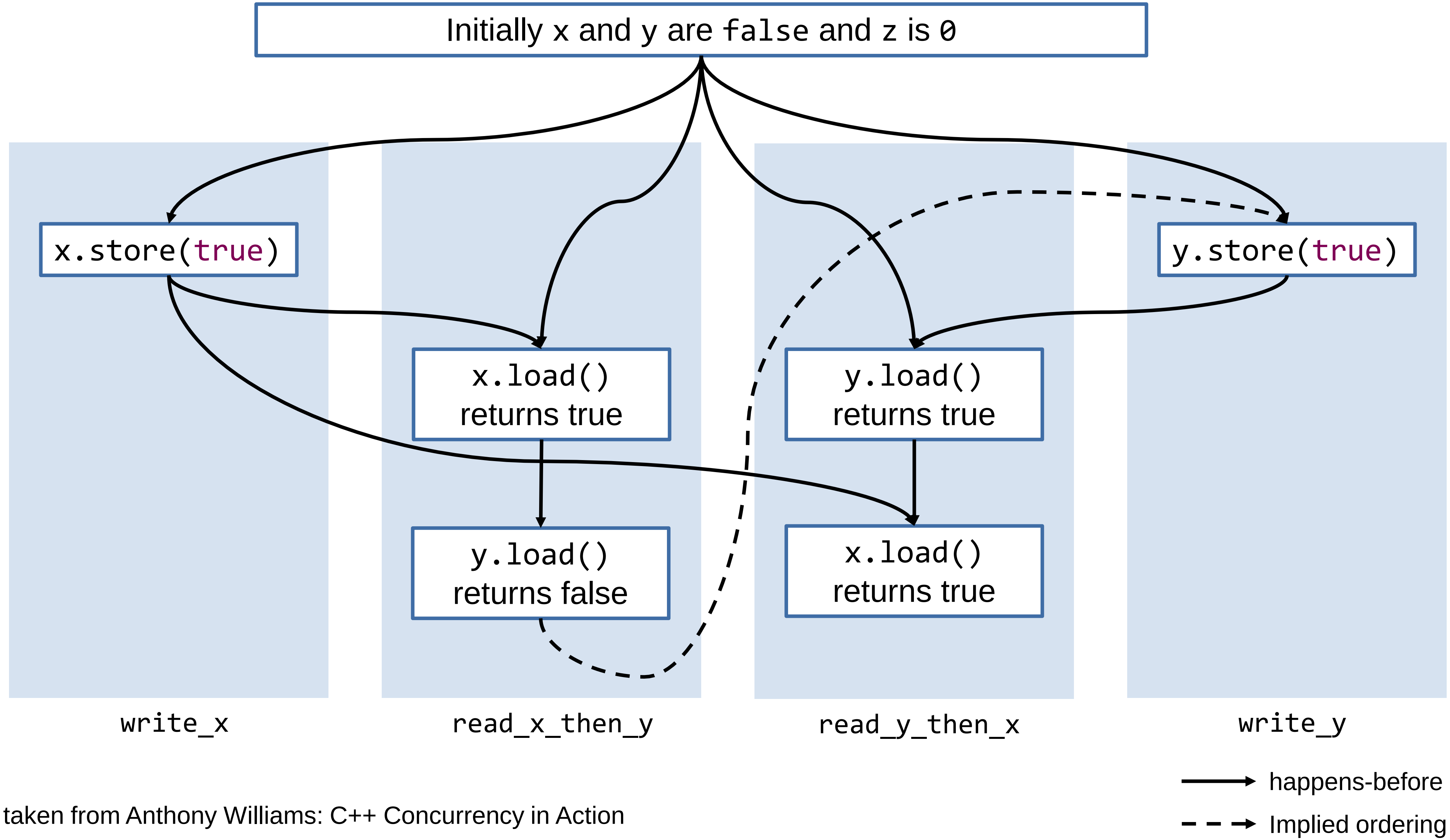


Illustration taken from Anthony Williams: C++ Concurrency in Action

- **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);
```

Acquire/Release Ordering (z == 0)

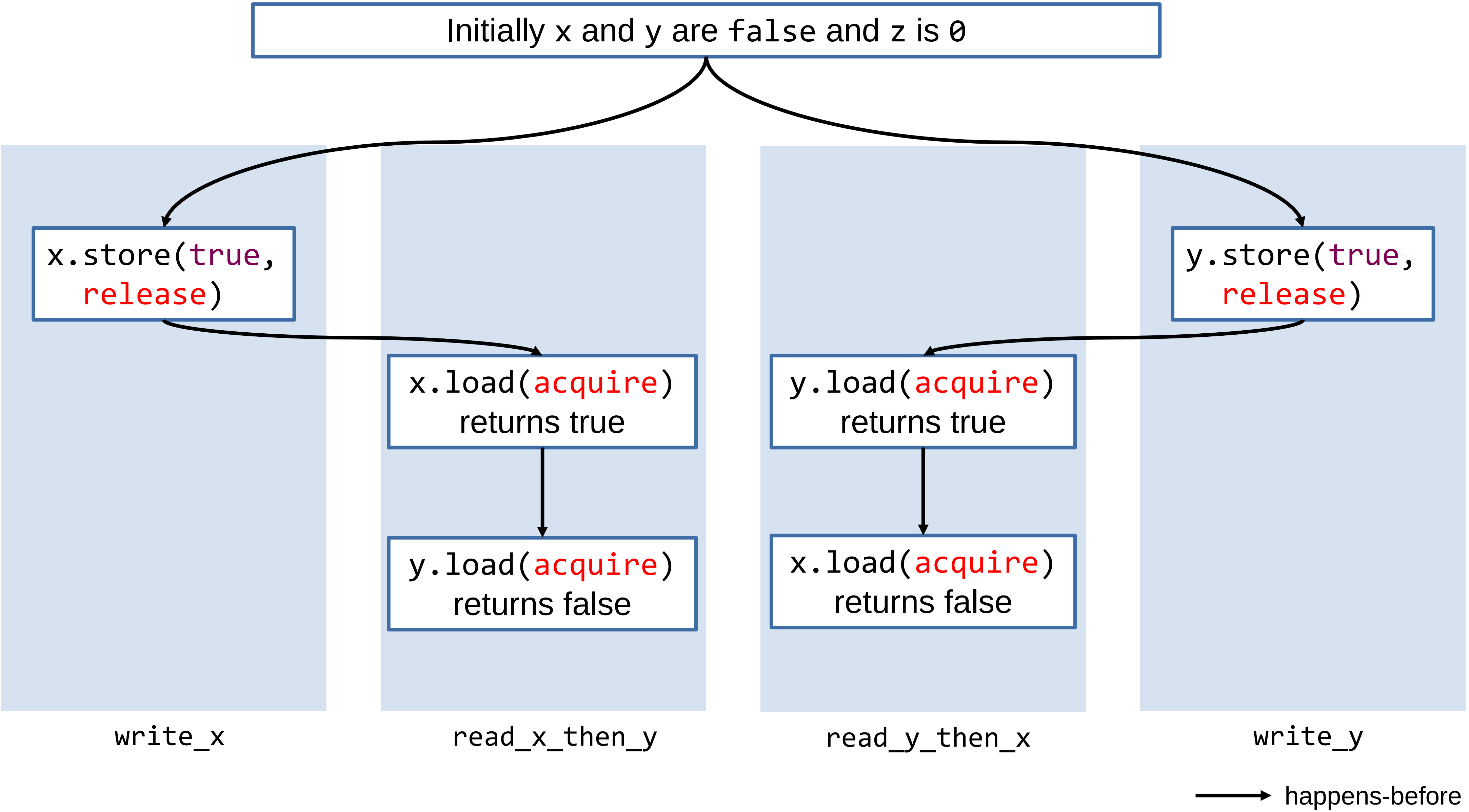


Illustration taken from Anthony Williams: C++ Concurrency in Action

Relaxed Ordering

- **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 may 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.**

```cpp
#include <atomic>
#include <thread>
#include <cassert>

std::atomic<bool> x{};
std::atomic<bool> y{};
std::atomic<int> z{};

auto write_x_then_y() -> void {
  x.store(true, std::memory_order::relaxed);
  y.store(true, std::memory_order::relaxed);
}

auto read_y_then_x() -> void {
  while(!y.load(std::memory_order::relaxed)); // Spin
  if(x.load(std::memory_order::relaxed))
    ++z;
}

auto main() -> int {
  auto a = std::thread{write_x_then_y};
  auto b = std::thread{read_y_then_x};
  a.join();
  b.join();
  assert(z.load() != 0);
}
```

Example adapted from Anthony Williams: C++ Concurrency in Action

Relaxed Ordering (z == 0)

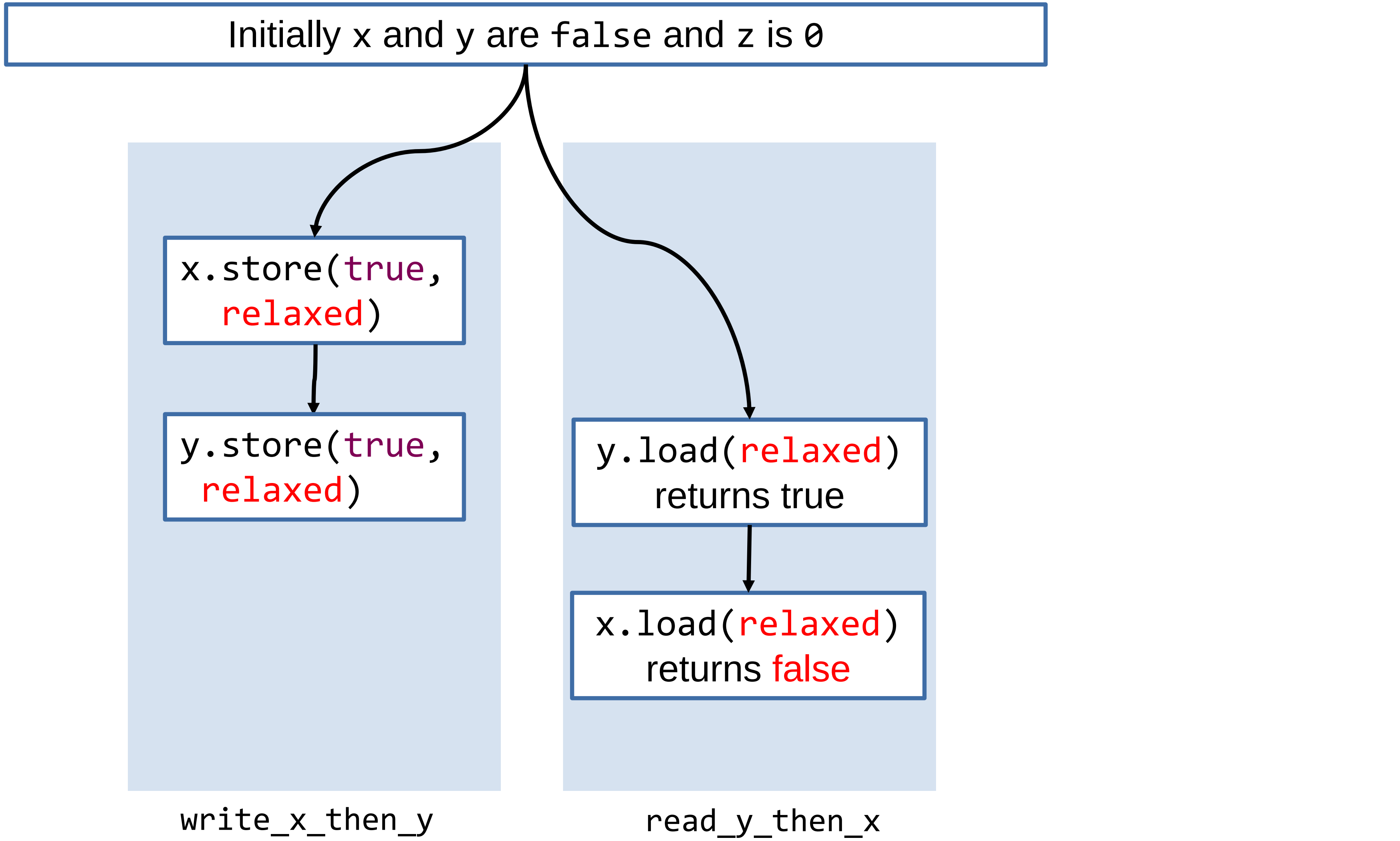


Illustration taken from Anthony Williams: C++ Concurrency in Action

- **Somewhat similar to Acquire/Release, but**
  - Introduces data-dependency concept
    - dependency-ordered-before
    - carries-a-dependency-to
  - Only dependent data is synchronized
  - Subtle difference == hard to use
- **DO NOT USE CONSUME!**
  - Even the standard recommends against it!

> ***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.*
>
> \- ISO 14882:2017

Custom Types with `std::atomic`

- **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`

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

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

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

Bibs and Bobs

`volatile` and Interrupts

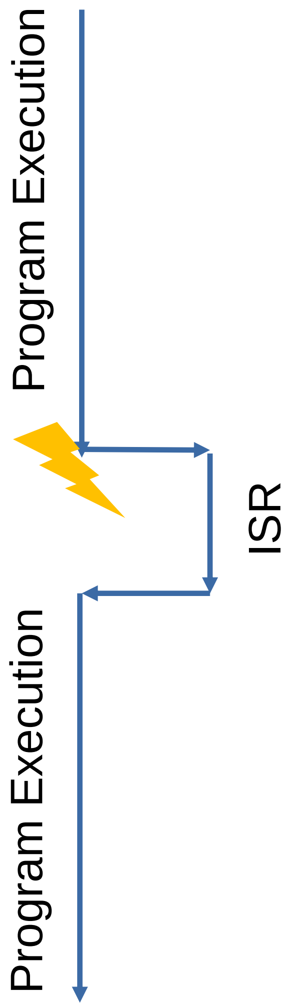
Ostschweizer Fachhochschule

Volatile

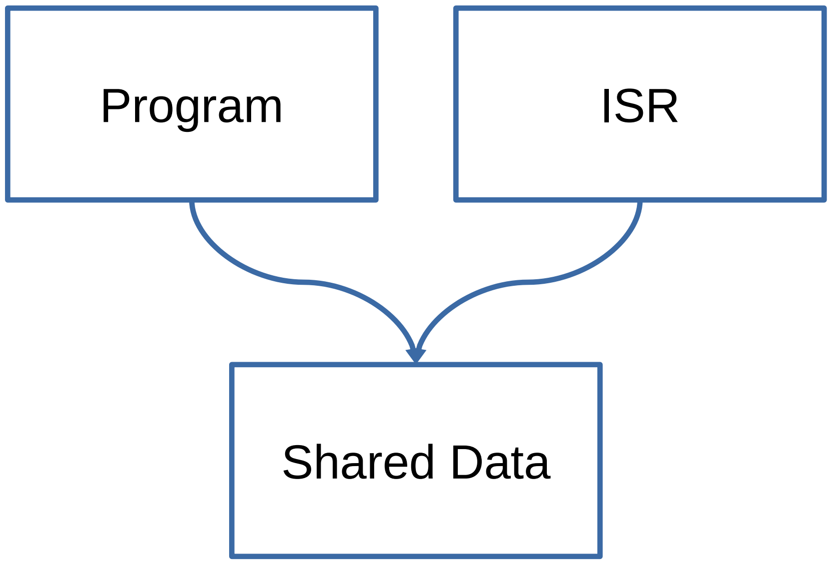

```
volatile int mem{0};
```

- **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**



Interrupts Example: Arduino

- **On AVR-based Arduinos, interrupts cannot be interrupted**
  - Other platforms support so-called Interrupt-Nesting (e.g. ARM, Risc V, …)
- **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