Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 11 – Memory Model

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 02.05.2019
FS2019

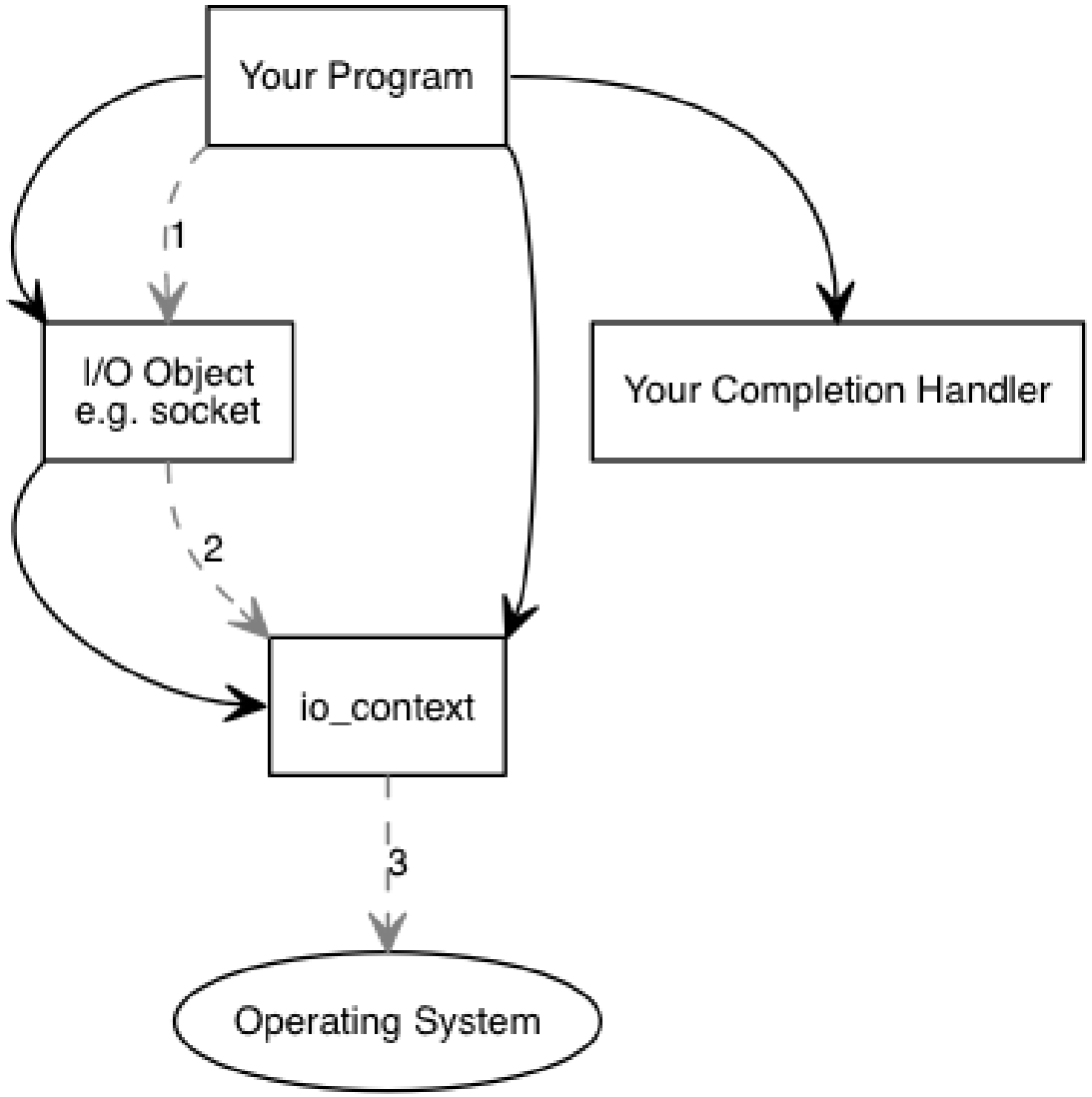
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

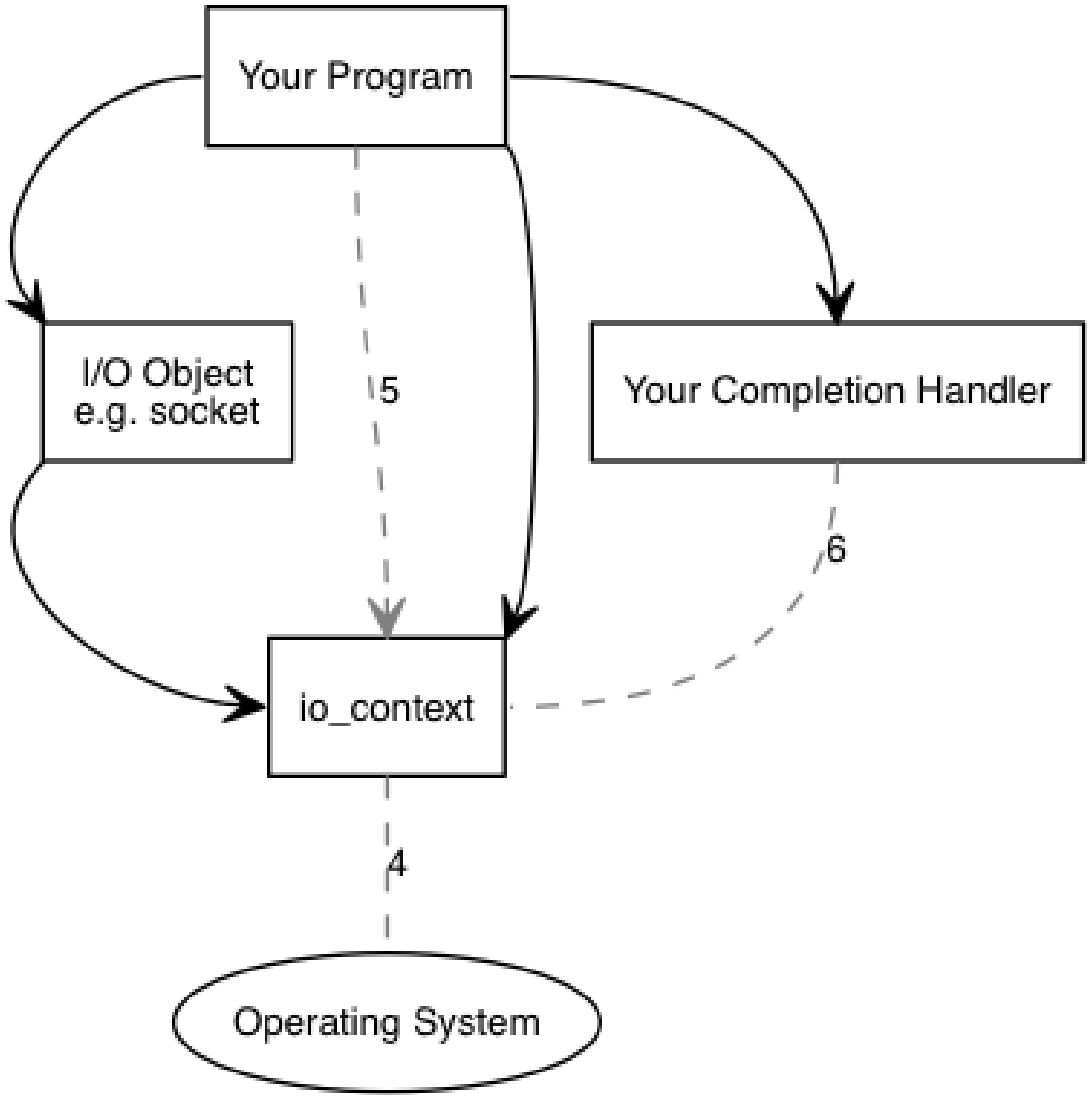
IFS
INSTITUTE FOR
SOFTWARE

Recap Week 10

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE





https://think-async.com/Asio/asio-1.12.2/doc/asio/overview/core/basics.html

```
struct Server {
  using tcp = asio::ip::tcp;
  Server(asio::io_context & context, unsigned short port)
      : acceptor{context, tcp::endpoint{tcp::v4(), port}}{
    accept();
  }

private:
  void accept() {
    auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
      if (!ec) {
        auto session = std::make_shared<Session>(std::move(peer));
        session->start();
      }
      accept();
    };
    acceptor.async_accept(acceptHandler);
  }
  tcp::acceptor acceptor;
};
```

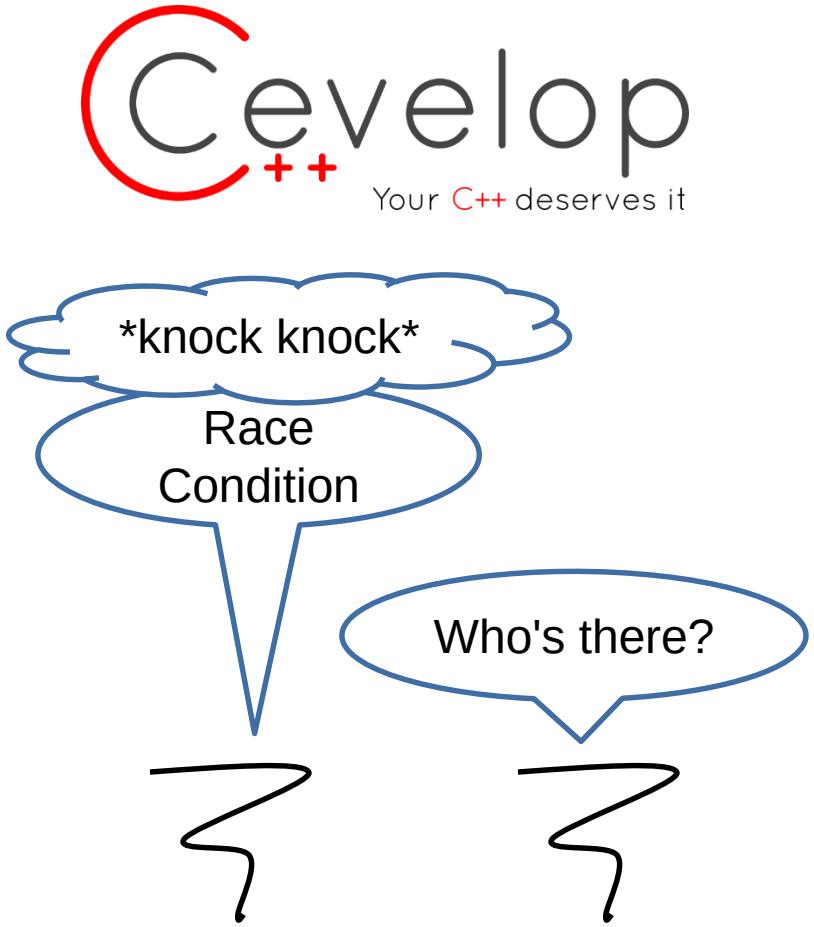
- **You know how threads safely communicate with each other**
- **You are aware of the pitfalls of synchronization**
- **You can use atomics**
- **You can implement thread-safe data structures**

```
class std::thread
```

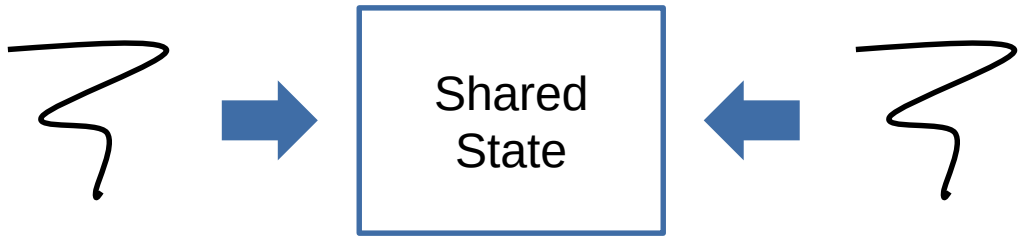
```
int main() {
  std::thread greeter {
    [] { std::cout << "Hello, I'm thread!" << std::endl; }
  };
  greeter.join();
}
```

- **A new thread is created and started automatically**
- **Creates a new execution context (thread)**
- **`join()` waits for the thread to finish**

Communication Between Threads



- **Mutable Shared State**
- **Problem: Data Race**
  - Two operations on the same memory location
  - At least one is not atomic (at the same time)
  - At least one is a modifying operation



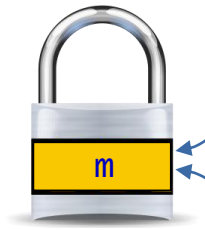
```cpp
int main () {
  UnsynchronizedCounter counter{};
  auto run = [&counter] {
    for (int i = 0; i < 100'000'000; ++i) {
      counter.increment();
    }
  };
  std::thread t1{run}, t2{run};
  t1.join();
  t2.join();
  std::cout << counter.current();
}
```

```cpp
struct UnsynchronizedCounter {
  void increment() {
    ++value;
  }

  int current() const {
    return value;
  }

private:
  int value{};
};
```

- **What is the expected output?**
- **Solutions**
  - Locking the shared access
  - Make all accesses atomic

`#include <mutex>`

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




```
struct ConcurrentCounter {
  void increment() {
    m.lock();
    ++value;
    m.unlock();
  }

  int current() const {
    m.lock();
    int const current = value;
    m.unlock();
    return current;
  }

private:
  mutable std::mutex m{};
  int value{};
};
```

```
#include <mutex>
```


- **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>)`

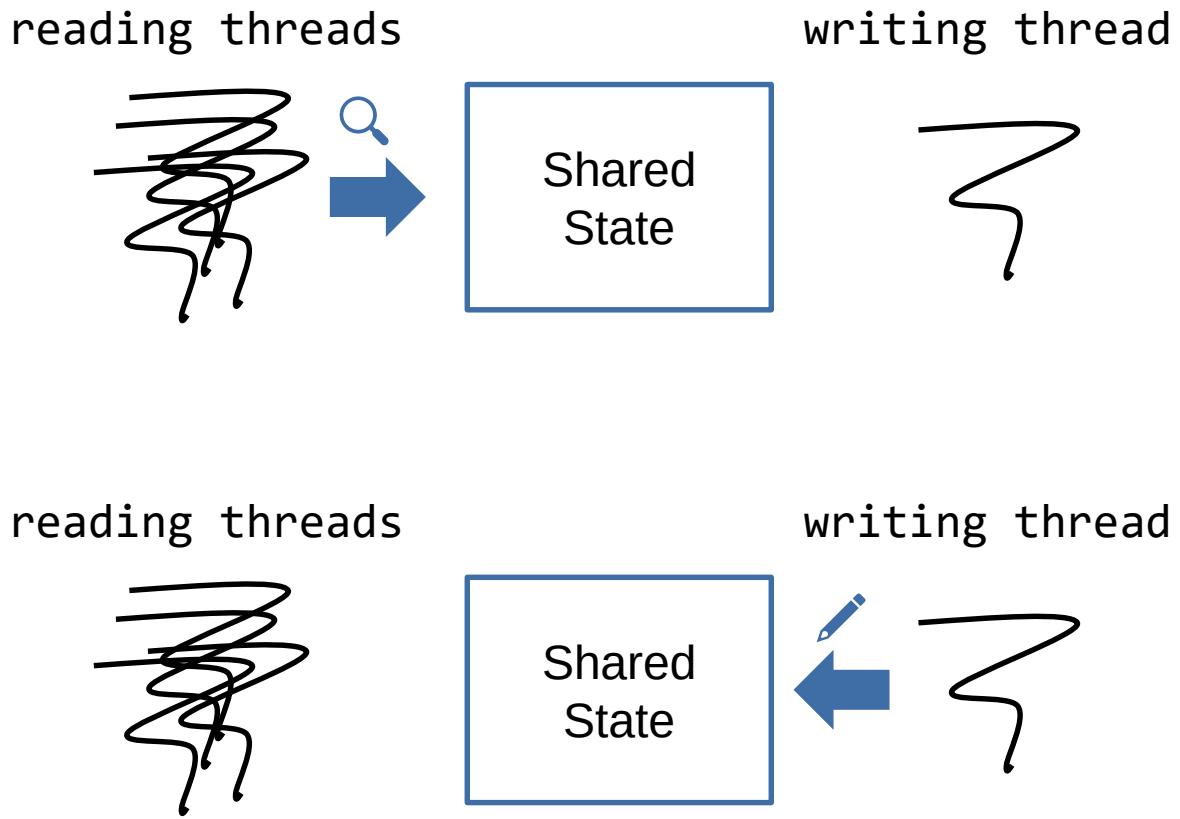
		Recursive	
		No	Yes
Timed	No	`std::mutex`	`std::recursive_mutex`
	Yes	`std::timed_mutex`	`std::recursive_timed_mutex`

```
#include <shared_mutex>
```


- **Reading operations don't need exclusive access**
  - Only concurrent writes need exclusive locking
- **`std::shared_mutex` and `std::shared_timed_mutex` provide exclusive and shared locking**
  - `lock_shared()`
  - `try_lock_shared()`
  - `try_lock_shared_for(<duration>)`
  - `try_lock_shared_until(<time>)`
  - `unlock_shared()`

`#include <shared_mutex>`


```
struct ConcurrentCounter {
  void increment() {
    m.lock();
    ++value;
    m.unlock();
  }

  int current() const {
    m.lock_shared();
    int const current = value;
    m.unlock_shared();
    return current;
  }

private:
  mutable std::shared_mutex m{};
  int value{};
};
```



- **Beware: a `std::shared_mutex` might be slower due to more overhead**
  - Use only if writes are scarce and reads very common

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

`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)

- **Scoped Locking Pattern**
  - Create a lock guard that locks and unlocks the mutex automatically
- **Code is shorter**
- **Code is safer**
  - In case of an exception the mutex is released

```
struct ConcurrentCounter {
  void increment() {
    std::scoped_lock lock{m};
    ++value;
  }

  int current() const {
    std::scoped_lock lock{m};
    return value;
  }

private:
  mutable std::mutex m{};
  int value{};
};
```

Why `mutable`?

- **Strategized Locking Pattern**
  - Template parameter for mutex type
  - Could also be `null_mutex` (boost)

```
template <typename Mutex = std::mutex>
struct ConcurrentCounter {
  using Lock = std::scoped_lock<Mutex>;
  void increment() {
    Lock lock{m};
    ++value;
  }

  int current() const {
    Lock lock{m};
    return value;
  }

private:
  mutable Mutex m{};
  int value{};
};
```

Multiple Locks with `std::scoped_lock`

- **`std::scoped_lock`**
  - Acquires multiple locks in the constructor
  - Avoids deadlocks, by relying on internal sequence
  - Blocks until all locks could be acquired
  - Class template argument deduction avoids the need for specifying the template arguments

```
struct ConcurrentCounter {
  // can't be noexcept, because locks might throw
  void swap(ConcurrentCounter & other) {
    if (this == &other) return;

    std::scoped_lock both{m, other.m};

    std::swap(value, other.value);
    // no need to swap mutex
  }
};
```

Multiple Locks (Before `std::scoped_lock`)

- **`std::lock`**
  - Acquires multiple locks in a single call
  - Avoids deadlocks
  - Blocks until all locks could be acquired
- **`std::try_lock`**
  - Tries to acquire multiple locks in a single call
  - Does not block
  - When it returns...
    - `true`, all locks have been acquired
    - `false`, no lock has been acquired

```
struct ConcurrentCounter {
  // can't be noexcept, because locks might throw
  void swap(ConcurrentCounter & other) {
    if (this == &other) return;

    // std::defer_lock prevents immediate locking
    lock my_lock{m, std::defer_lock};
    lock other_lock{other.m, std::defer_lock};

    // blocks until all locks are acquired
    std::lock(my_lock, other_lock);

    std::swap(value, other.value);
    // no need to swap
  }
};
```

`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` is required as condition might release lock (`wait`)**

#include <condition_variable>


```
template <typename T, typename MUTEX = std::mutex>
struct ThreadsafeQueue {
  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 MUTEX mx{};
  std::condition_variable notEmpty{};
  std::queue<T> q{};
};
```

```
struct ConcurrentCounter {
  void incrementAndGet() {
    std::scoped_lock{m};
    return ++value;
  }
  mutable std::mutex m{};
  int value{};
};
```

```
template<typename T>
struct ThreadsafeQueue {
  std::optional<T> tryPop() {
    guard lk{ mx };
    if (empty()) { return std::nullopt; }
    return pop();
  }
  bool empty() const {
    std:: scoped_lock lk { mx };
    return q.empty();
  }
private:
  mutable std::mutex mx { };
  std::queue<T> q{};
};
```

```
struct ConcurrentCounter {
  void incrementAndGet() {
    std::scoped_lock{m};
    return ++value;
  }
  mutable std::mutex m{};
  int value{};
};
```

Incorrect

`std::scoped_lock` creates a temporary variable. It is immediately destroyed at the end of the statement and the rest of the function is not locked.

```
template<typename T>
struct ThreadsafeQueue {
  std::optional<T> tryPop() {
    guard lk{ mx };
    if (empty()) { return std::nullopt; }
    return pop();
  }
  bool empty() const {
    std:: scoped_lock lk { mx };
    return q.empty();
  }
private:
  mutable std::mutex mx { };
  std::queue<T> q{};
};
```

Incorrect

An std::mutex is not recursive. If `empty()` or `pop()` are called in `tryPop()` it self deadlocks. A private non-locking helper function is required that can be called.

- **There is no thread-safety wrapper for standard containers! (yet)**
- **Access to different individual elements from different threads is not a data race**
  - Container must not change during the concurrent access to elements!
  - Using different elements of a `std::vector` from different threads is OK!
- **Almost all other concurrent uses of containers is dangerous**
- **`std::shared_ptr` copies to the same object can be used from different threads, but accessing the object itself can race if non-const**
  - Reference counter is an atomic

Async, Future and Promise

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

```
template< class Function, class...Args >
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
async(std::launch policy, Function && f, Args &&... args );
```

- **`std::async` provides a simple interface for**
  - Background tasks (in separate threads)
  - Postponed computation
- **Launch policies (arguments)**
  - `std::launch::async` – actual asynchronous computation
  - `std::launch::deferred` – lazy evaluation
- **Executes a function `f` with the given arguments `args` and returns an `std::future` of the return type of the called function `f`**

```
int main() {
  httplib::Client cli("www.w3.org", 80);
  auto fetchUrl = [&] (std::string_view url) {
    return cli.Get(url.data());
  };
  auto future =
      std::async(std::launch::async, fetchUrl, "/");
  std::cout << "Do something else...\n";
  auto result = future.get();
  if (result) {
    printHeaders(std::cout, result->headers);
  }
}
```

```
template<typename T>
class future;
```

- **An `std::future` instance is a surrogate object for retrieving a value**
- **`get()` retrieves the value**
  - Or throws the exception that occured while computing its value
  - Blocks until the value is available (possibly forever)
  - Releases the shared state -> must only be called once!
- **`wait()`, `wait_for()`, `wait_until()`**
  - Waits until the value is available or the time expires
- **An `std::future` is an exclusive handle for the value and cannot be copied (only moved)**
  - `share()` creates an `std::shared_future` that can be copied – required when multiple threads wait for the same value (multiple `get()` calls allowed – even to the same `std::shared_future` object)

Destruction of `std::future` From `std::async`

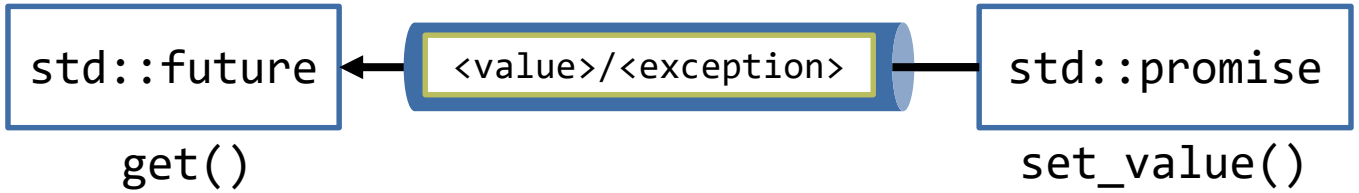
- **Beware: If you get an `std::future` through a call to `std::async` the destructor of the `std::future` may block until the `std::async` execution is completed**

- **This might sound innocent, but has significant consequences**

```
void someAsyncStuff();

void startAsyncStuff() {
  std::async(std::launch::async, someAsyncStuff);
  std::async(std::launch::async, someAsyncStuff);
  std::async(std::launch::async, someAsyncStuff);
}
```

- **Each call of `std::async` will block because the `std::future` return value will be destroyed immediately**

```
template<typename T>
class promise;
```

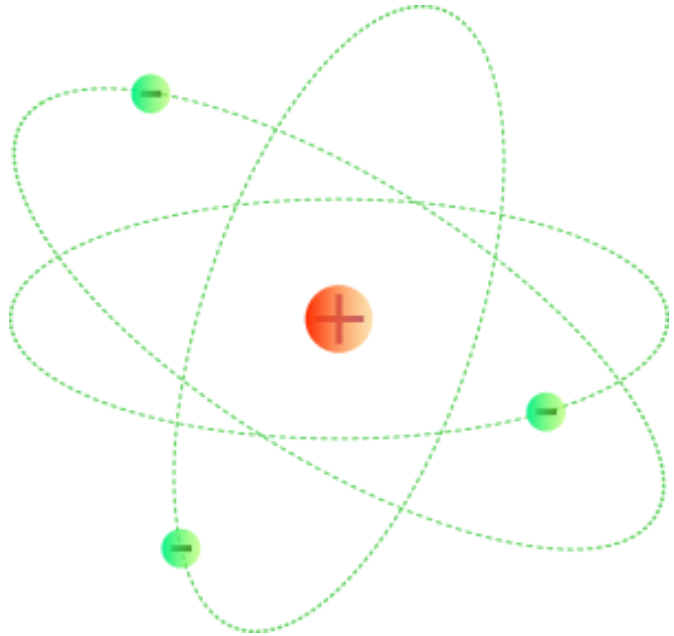


- **`std::promise` is the other end of the `std::future`, which allows setting the value (or exception)**
  - `set_value(T const &)` – makes the value available
  - `set_exception(std::exception_ptr)` – the exception will be thrown in the get() call
- **`get_future()` – gets the corresponding `std::future` object**
  - Must not be called more than once

```
void compute(int num, int den, std::promise<int> p) {
  if (den == 0) {
    std::logic_error e{"must not divide by 0"};
    auto e_ptr = std::make_exception_ptr(e);
    p.set_exception(e_ptr);
  }
  auto result = num / den;
  p.set_value(result);
}

std::future<int> compute_in_thread(int n, int d) {
  std::promise<int> res_promise{};
  std::future<int> res_future{res_promise.get_future()};
  std::thread t{compute, n, d, std::move(res_promise)};
  t.detach();
  return res_future;
}
```

Atomics and the Memory Model

Cevelop
Your C++ deserves it



HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

```
#include <atomic>
```



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

```
class atomic_flag;
```

- **Atomics are guaranteed to be data race free!**
- **Most primitive atomic: `std::atomic_flag`**
  - Lock-free
  - `clear()` – sets flag to false
  - `test_and_set()` – sets flag to true and returns previous value
- **Other atomic types might require using locks**
  - `std::atomic<bool>` and `std::atomic<int>` usually can be implemented without relying on mutexes/locks
  - Can be checked 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();
}
```

- **Can everything be used as template argument for `std::atomic<T>`?**
  - `T` 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 a new value and get the previous
`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>` also provide atomic operators like ++, --, +=, etc.**

- **The C++ memory model specifies**
  - When the effect of an operation is visible to other threads (storing and writing)
  - The possibility of reordering instructions
- **Different relations about visibility of effects**
  - sequenced-before: within a single thread
  - happens-before: sequenced-before or inter-thread happens-before
  - synchronizes-with: cross-thread sync.
- **Note: Read/writes in a single statement are "unsequenced"**

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

- **Atomic operations have an optional parameter for memory order**
  - `std::memory_order_seq_cst` (default)
  - `std::memory_order_acquire`
  - `std::memory_order_release`
  - `std::memory_order_acq_rel`
  - `std::memory_order_relaxed`
  - `std::memory_order_consume` (discouraged)

```
int main() {
  std::atomic<int> c{};
  c.fetch_add(1, std::memory_order_seq_cst);
  //...
}
```

- **Sequential consistency**
  - Global execution order of operations
  - Every thread observes the same order
- **Memory order flag**
  - `std::memory_order_seq_cst`
- **The latest modification (in the global execution order) will be available to a read**

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

Each function is executed in 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?

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

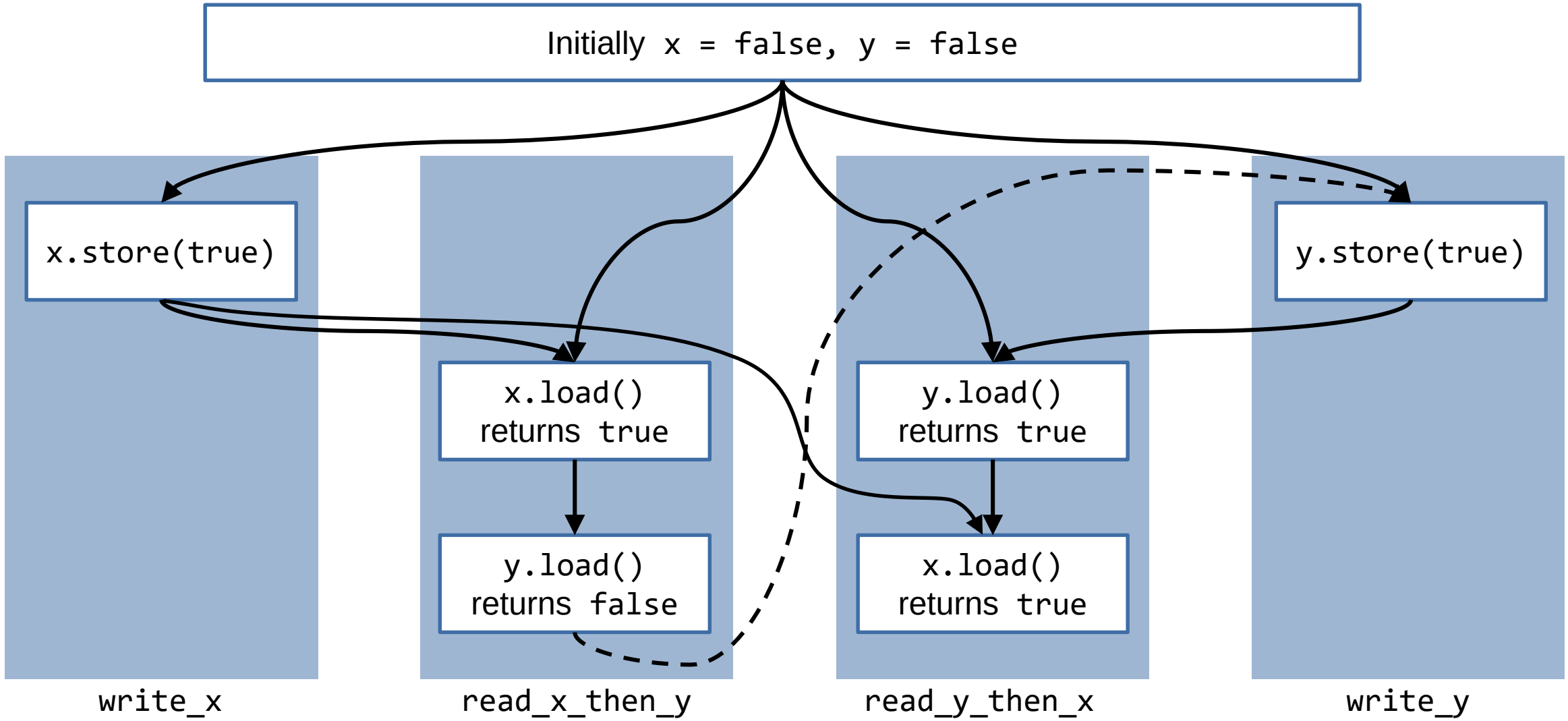


Illustration taken from Anthony Williams: C++ Concurrency in Action

- **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);
```

http://en.cppreference.com/w/cpp/atomic/memory_order

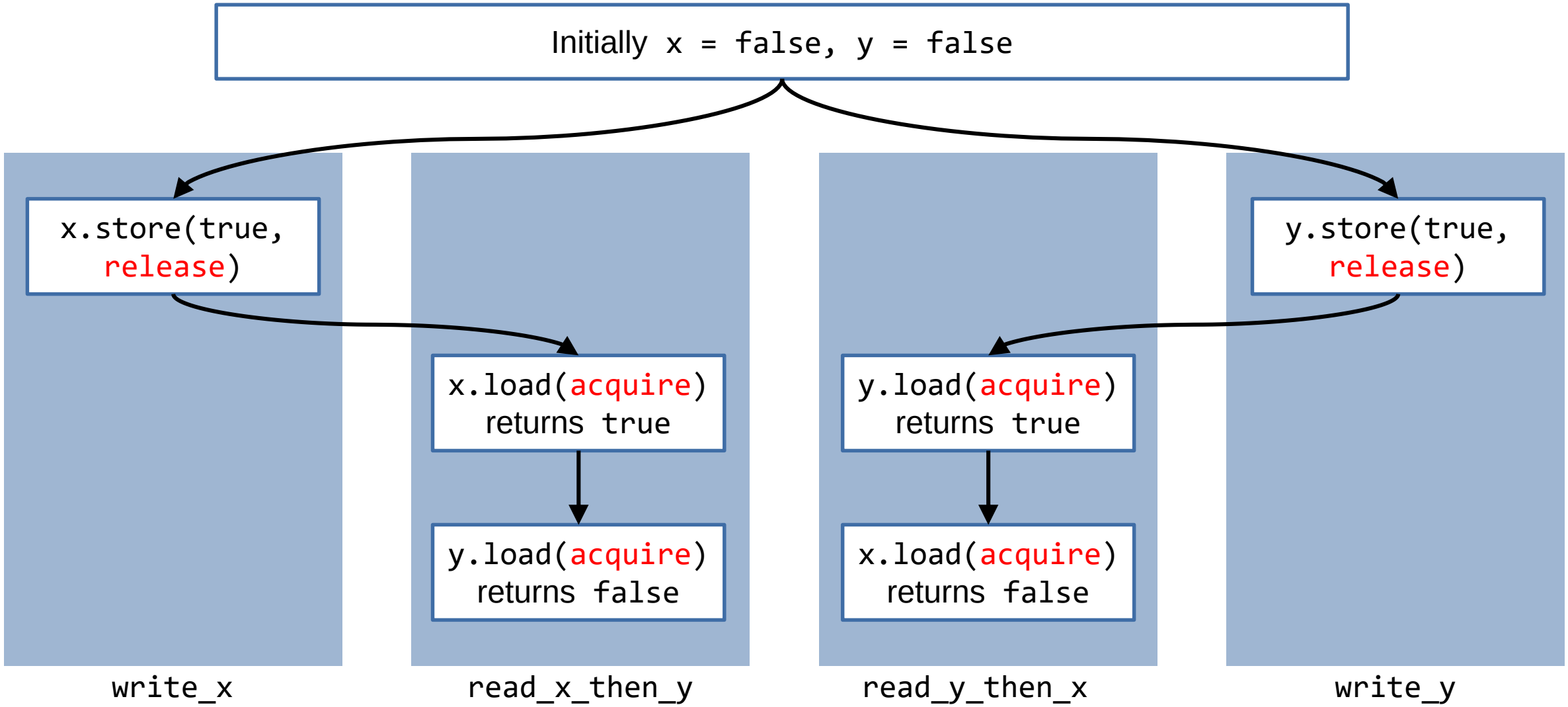


Illustration taken from Anthony Williams: C++ Concurrency in Action

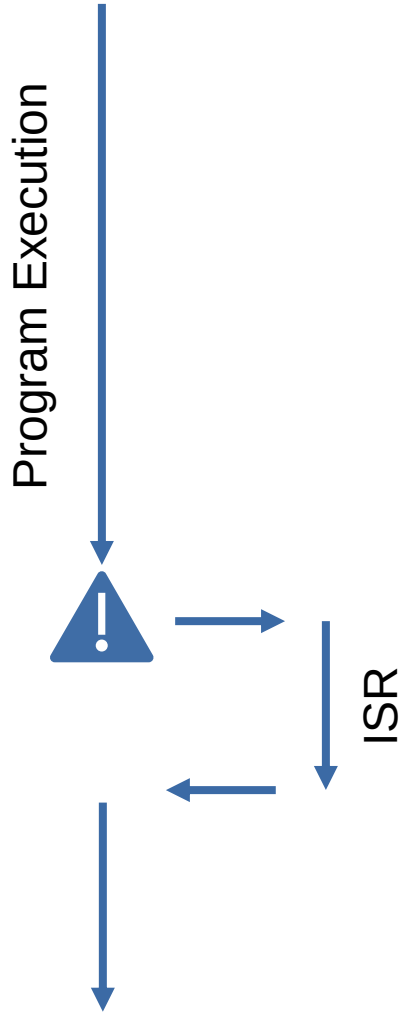
- **Relaxed memory order**
  - Does not give promises about sequencing
  - No data race for atomic variables
- **Order of observable effects can be inconsistent (in different threads)**
  - load() and store() operations happen in parallel
- **Depends on processor hardware if more "efficient" or not (and compiler support)**
  - Less synchronization effort means less processor pipeline stalling or needing to wait for memory loads
- **Using non sequentially consistent operations correctly is an art and requires proving correctness! (beyond this course)**

```
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();
  }
}
// see github repo, adapted from C++ Concurrency in Action
// DEMO!
```

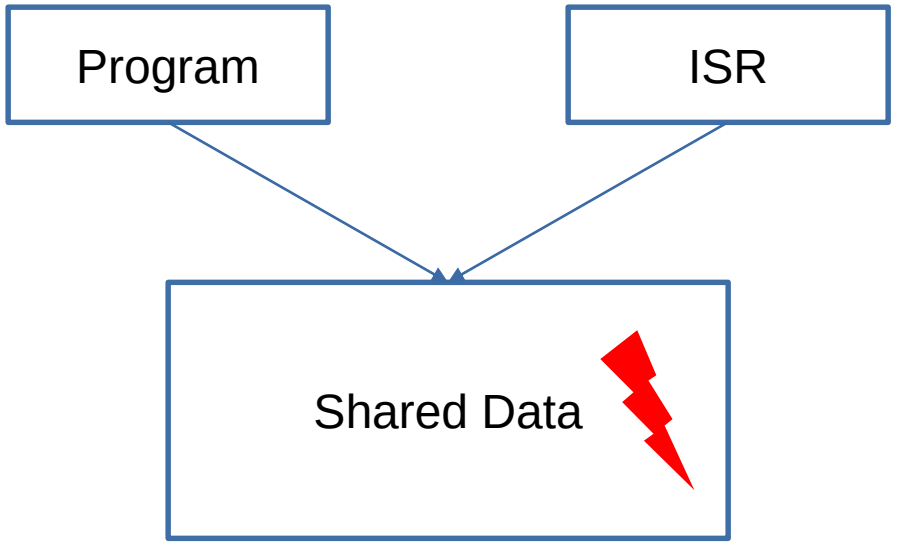
```
volatile int mem{0};
```

- **The semantics of the `volatile` specifier for variables in C++ is different from the same specifier in Java and C#**
- **Load and store operations of volatile variables must not be removed, even if the compiler cannot recognize a visible effect within the same thread**
- **Prevents the reordering by the compiler within the same thread**
  - The hardware may still reorder it
- **Only useful when writing memory to access hardware directly**
  - Not for inter-thread communication!

- **An interrupt is a high priority event in a system that interrupts the normal program execution**
- **It invokes by a function that is registered to handle this event specifically**
  - Interrupt Service Routine
  - ISRs need to be short and must not block (run to completion)
- **After the interrupt has been handled, normal program execution is continued**



- **Data shared between an interrupt and the normal program execution needs to be protected**
  - Access needs to be atomic
  - Modifications have to become visible
- **volatile helps regarding visibility**
- **For atomicity interrupts might need to be disabled temporarily when accessing the data from the "normal" program execution**
- **Exact details about how to deal with interrupts depend on the specific microcontroller**



- **On Ardiuno interrupts cannot be interrupted**
- **Before accessing shared data interrupts neet to be disabled**
- **Example**
  - Pin with ID 2 can be configured as interrupt pin.
- **`toggleLed()` is the ISR for the interrupt**
- **Every time before the ledState is read the interrupts are disabled. Afterwards they are renabled again.**

```
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();
}
```

- **Getting programs working on shared mutable data correct is terribly difficult**
- **Use the appropriate kind of mutex and lock for protecting common data**
- **C++ has a memory model featuring very fine grained configuration possibilities, but unless you really know what you are doing and you inevitably need something else, stick with the defaults.**
- **Interrupts require special treatment of shared data, depending the actual platform**

What Else?

Cevelop
Your C++ deserves it

- **Concurrency, Parallelism Technical Specification and Transactional Memory Support**
  - http://en.cppreference.com/w/cpp/experimental/concurrency (C++17)
  - http://en.cppreference.com/w/cpp/experimental/parallelism (C++17)
  - http://en.cppreference.com/w/cpp/language/transactional_memory
- **Continuations for futures (`std::future::then()`)**
- **Parallel versions of standard algorithms**
- **Latches and barriers**
- **Atomic smart pointers**
- **ACCU 2017 talk of Anthony Williams: https://www.youtube.com/watch?v=UhrIKqDADX8**
- **Still no thread pool and task abstraction**

`#include <future>`


```
template<typename R, typename...Args>
class packaged_task<R(Args...)>;
```

- **`std::packaged_task` wraps a function call with arguments**
  - The call is deferred until the task's call operator is called
- **Allows to obtain a future to the call's result**
  - Another thread can synchronize with the actual call through the future object
- **Call `operator()` of `std::packaged_task` makes the future ready when done**
  - Could be called from a thread
  - Need to pass by `std::move()` or `ref()` wrapper, can not be copied

```
std::string compute_the_answer(int i) {
  using namespace std::literals;
  std::this_thread::sleep_for(1s);
  return "the answer is "s + std::to_string(i);
}

using pt_fun = std::packaged_task<std::string(int)>;

int main() {
  std::cout << "computing" << std::endl;
  pt_fun task { compute_the_answer };
  auto future = task.get_future();
  std::thread compute { std::move(task), 42 };
  std::cout << future.get();
  compute.join();
}
```

- **For initialization code that might be needed by different threads, but called atomically only by one**
- **`std::once_flag` can be**
  - global (hard to test)
  - static local in a function (also hard to test)
  - or a class member
- **`std::call_once` takes the `std::once_flag`**
  - a callable
  - all arguments
  - need to wrap passed references with `std::ref` as with `std::thread`

```
void initfun(threadsafe_queue<int> & q){
  q.push(999);
}
//...
  std::once_flag init{};
  std::thread prod1{[&]{
    sleep_for(10ms);
    call_once(init, initfun, std::ref(queue));
    for(int i=0; i < 10; ++i) {
      queue.push(i);
      yield();
    }
  }};

  std::thread prod2{[&]{
    sleep_for(9ms);
    call_once(init, initfun, std::ref(queue));
    for(int i = 0; i < 10; ++i) {
      queue.push(i*10);
      yield();
    }
  }};
```

```
thread_local bool done{};
```

- **Thread-local variables are possible (global or static)**
  - Unfortunately not through a library type, but built-in prefix (storage class)
  - Compilers used to provide that through this mechanism
  - I (Peter Sommerlad) believe it is an error
- **No simple and easy examples**
  - One could implement recursive_mutex with a tread_local counter per mutex, but that requires a map of mutex to counter in a static thread_local member of the recursive_mutex class