

Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 8 – Multi-Threading and Mutexes

Thomas Corbat / Felix Morgner
Rapperswil, 26.04.2022
FS2022



- **Week 7 Recap**
- **Introduction to asynchronous computation in C++**
- **Threading in C++**
- **Communicating results from asynchronous operations**



Recap Week 7



- Let the compiler compute constant values for us:

```
constexpr double pi = 3.14159;  
  
constexpr auto area = [](double r) {  
    return pi * r * r;  
};  
  
constexpr auto circleArea = area(2.0);
```

- Values are determined at compile time -> no runtime overhead!
- Compiler prevents Undefined Behavior!



- Create a tag type for the unit

```
struct Kph;  
struct Mph;  
struct Mps;
```

- Create a quantity type template for speed

```
template <typename Unit>  
struct Speed {  
    constexpr explicit Speed(double value)  
        : value{value}{};  
    constexpr explicit operator double() const {  
        return value;  
    }  
private:  
    double value;  
};
```



```
namespace velocity::literals {  
  
constexpr inline Speed<Kph> operator"" _kph(unsigned long long value) {  
    return Speed<Kph>{safeToDouble(value)};  
}  
  
constexpr inline Speed<Kph> operator"" _kph(long double value) {  
    return Speed<Kph>{safeToDouble(value)};  
}  
  
//...  
}
```

```
auto speed1 = 5.0_kph;  
auto speed2 = 5.0_mph;  
auto speed3 = 5.0_mps;
```



Asynchronous Computation in C++



- **Participants should...**

- know how to work with the C++ standard threading API
- be able to use `std::async` to run computations asynchronously
- be familiar with the C++ standard locking mechanisms
- be able to use `std::promise` and `std::future` to obtain asynchronous results



- **Every C++ program has at least one thread of execution**
 - Computations run synchronously
 - Every computation has to wait for prior computations
- **C++11 introduced standard library support for asynchronous execution**
 - `std::thread` to explicitly run a new thread
 - `std::async` to easily wrap a computation (possibly with a result)
 - `std::mutex` and co. to facilitate synchronization



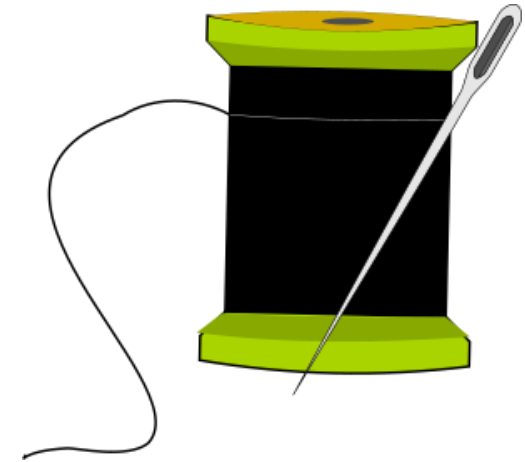
Multi-Threading



- **You might find a lot of legacy C++ code using “pthreads” (aka POSIX-Threads) API**
 - There is no portability guarantee
 - Your compiler must know about your use of “pthreads” to generate safe code
- **C++11 removed the need to rely on POSIX-Threads**
 - Not 100% functionally equivalent for all tasks, so some implementations still use POSIX (or even Microsoft) Threads
 - BUT, it is guaranteed to be portable across platforms and compilers
 - BECAUSE
C++11 and later define a Memory Model for the execution with concurrent execution agents (aka threads)
- **The slides distinguish `std::thread` (C++ class) and `thread` (OS execution agent)**



API of `std::thread`



```
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



- Besides lambdas also functions or functor objects can be executed in a thread
- Calls the given "function" in that thread
- Return value of the function is ignored
- Threads are default-constructible and moveable (construction and assignment)

```
struct Functor {  
    void operator()() const {  
        std::cout << "Functor" << std::endl;  
    }  
};  
  
void function() {  
    std::cout << "Function" << std::endl;  
}  
  
int main() {  
    std::thread functionThread{function};  
    std::thread functorThread{Functor{}};  
  
    functorThread.join();  
    functionThread.join();  
}
```



```
template<class Function, class... Args>
explicit thread(Function&& f, Args&&...args);
```

- `std::thread` constructor takes a function/functor/lambda and arguments to forward
- You should pass all arguments by value to avoid data races and dangling references (if possible)
- Capturing by reference in lambdas creates shared data as well!

```
std::size_t fibonacci(std::size_t n) {
    if (n < 2) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

void printFib(std::size_t n) {
    auto fib = fibonacci(n);
    std::cout << "fib(" << n << ") is "
              << fib << '\n';
}

int main() {
    std::thread function { printFib, 46 };
    std::cout << "waiting..." << std::endl;
    function.join();
}
```



- **Guess what happens**

```
int main() {  
    std::thread lambda {  
        [] {std::cout << "Lambda" << std::endl; }  
    };  
    std::cout << "Main" << std::endl;  
}
```

Main

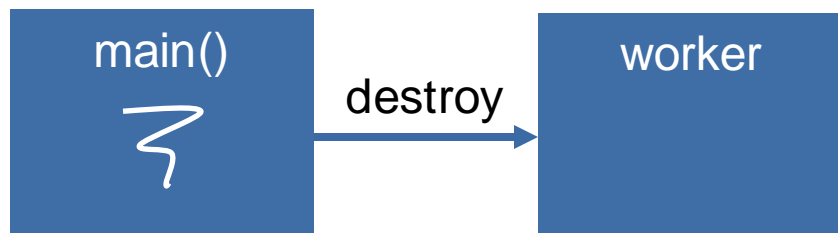
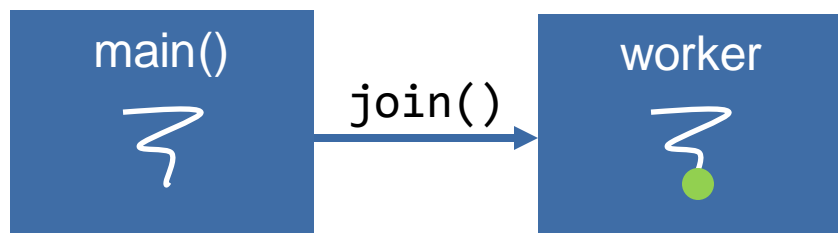
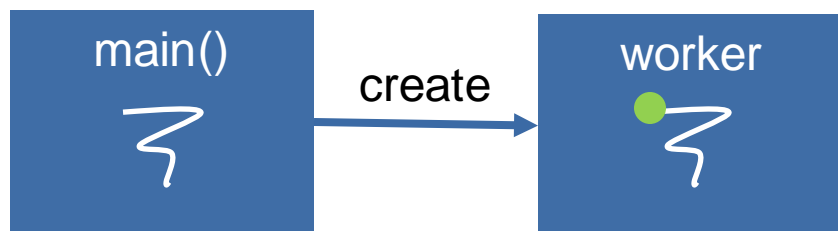
Lambda

terminate called without an active exception

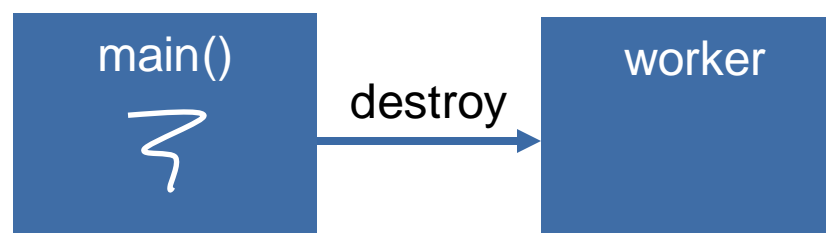
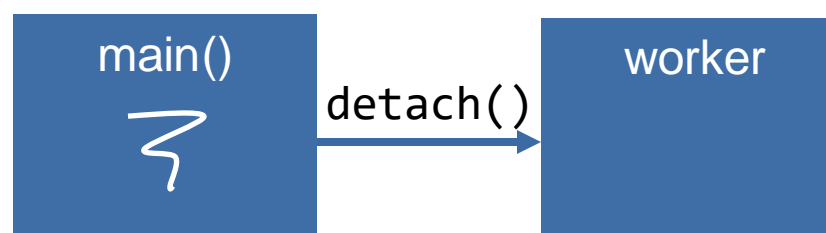
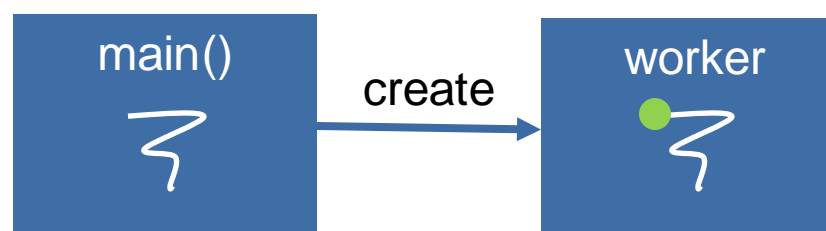


- Before the `std::thread` object is destroyed you must `join()` or `detach()` the thread

```
int main() {  
    std::thread worker { doWork };  
    worker.join();  
}
```



```
int main() {  
    std::thread worker { doWork };  
    worker.detach();  
}
```



- **The destructor doesn't call `join()` nor `detach()`**
 - If it called `join()` the program might hang when leaving the scope (possibly unexpected due to an exception)
 - If it called `detach()` the thread continues with possibly deallocated resources (references to local variables)
- **If an unjoined and undetached thread object is destroyed `std::terminate()` will be called**
- **When using `.detach()` beware of the lifetime of objects referred from the detached thread's function, global or passed, or even local ones.**
 - When the `main()` thread ends globals like `std::cout` will be destroyed

```
void startThread() {  
    using namespace std::chrono_literals;  
    std::string local{"local"};  
    std::thread t{[&] {  
        std::this_thread::sleep_for(1s);  
        std::cout << local << std::endl;  
    }};  
    t.detach();  
}  
  
int main() {  
    using namespace std::chrono_literals;  
    startThread();  
    std::this_thread::sleep_for(2s);  
}
```



- **Suggested by Anthony Williams in “C++ Concurrency in Action”**
 - Adapted version from the book
- **RAII wrapper that automatically calls `join()`**
 - Part of C++20 (`std::jthread`)
 - With a more powerful API

```
struct ScopedThread {
    explicit ScopedThread(std::thread && t)
        : the_thread{std::move(t)} {
        if (!the_thread.joinable())
            throw std::logic_error { "no thread" };
    }
    ~ScopedThread() {
        the_thread.join();
    }
private:
    std::thread the_thread;
};

int main() {
    ScopedThread t { std::thread {
        [] {std::cout << "Hello Thread"<< std::endl;}
    } };
    std::cout << "Hello Main" << std::endl;
}
```



- Using global streams does not create data races, but sequencing of characters could be mixed

```
#include <thread>
#include <iostream>

int main() {
    using std::cout;
    using std::endl;

    std::thread t {[] {cout << "Hello Thread" << endl;}};
    cout << "Hello Main" << endl;
    t.join();
}
```

?

Hello Main
Hello Thread

HHeellllloo MTahirnead

<Arbitrary Combination>



- **namespace `std::this_thread` provides some helper functions**
- **`get_id()`, also available as member function**
 - An id of the underlying OS thread
 - Distinguishes one thread from all others and can be used as a key in a map of threads
- **`sleep_for(duration)`**
 - Suspends thread for a duration
- **`sleep_until(time_point)`**
- **`yield()`**
 - Allows OS to schedule another thread
- **NB: Timing doesn't guarantee sequence!**

```
int main() {  
    using std::cout;  
    using std::endl;  
    using namespace std::chrono_literals;  
  
    std::thread t { [] {  
        std::this_thread::yield();  
        cout << "Hello ID: "  
              << std::this_thread::get_id()  
              << endl;  
        std::this_thread::sleep_for(10ms);  
    }};  
    cout << "main() ID: "  
          << std::this_thread::get_id()  
          << endl;  
    cout << "t.get_id(): "  
          << t.get_id()  
          << endl;  
    t.join();  
}
```



```
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!

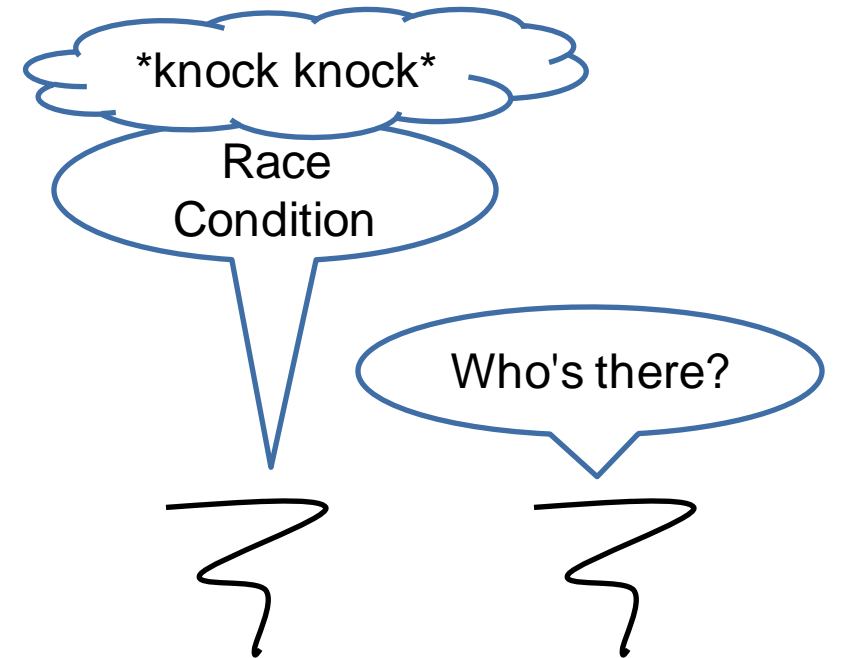
```
void countAsync(std::string_view input) {  
    std::thread t{[&] {  
        countAs(input);  
    }};  
    t.detach();  
}
```

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.



Communication Between Threads



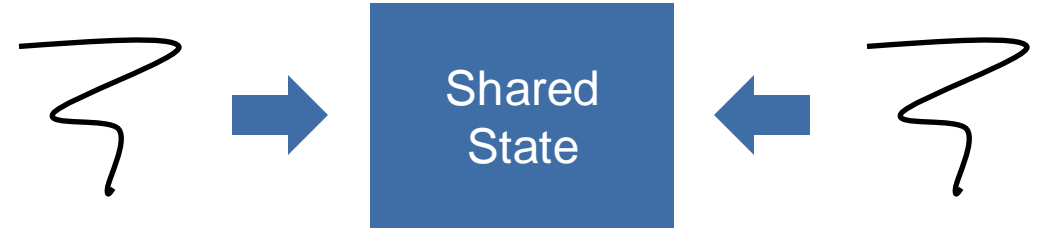
- **Mutable Shared State**

- **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

- **Solution**

- Locking the shared access
- Make access atomic



- 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>



- **Reading operations don't need exclusive access**

- Only concurrent writes need exclusive locking

- **`std::shared_mutex` (C++17) and `std::shared_timed_mutex` (C++14) provide exclusive and shared locking**

- **Additional functions for read-locking:**

- `lock_shared()`
- `try_lock_shared()`
- `try_lock_shared_for(<duration>)`
- `try_lock_shared_until(<time>)`
- `unlock_shared()`

reading threads



writing thread



reading threads



writing thread



- **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)



- **Threadsafe queue**

- Delegate functionality to `std::queue`
- Make every member function **mutually exclusive**

- **Scoped Locking Pattern**

- Create a lock guard that locks and unlocks the mutex automatically

- **Strategized Locking Pattern**

- Template parameter for mutex type
- Could also be `null_mutex` (boost)



```
template <typename T,  
         typename MUTEX = std::mutex>  
struct threadsafe_queue {  
    using guard = std::lock_guard<MUTEX>;  
    void push(T const &t) {  
        guard lk{mx};  
        q.push(t);  
    }  
    T pop() { /* later */ return T{}};  
    bool try_pop(T &t){  
        guard lk{mx};  
        if (q.empty()) return false;  
        t = q.front();  
        q.pop();  
        return true;  
    }  
    bool empty() const{  
        guard lk{mx};  
        return q.empty();  
    }  
private:  
    mutable MUTEX mx{};  
    std::queue<T> q{};  
};
```

Why not this->empty()?

Why mutable?



- `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

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

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

    std::swap(q, other.q);
    // no need to swap mutex or condition variable
}
```



- **`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

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

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

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

    std::swap(q, other.q);
    // no need to swap mutex or condition variable
}
```



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{};
};
```



- **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 are dangerous**
- **`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



Returning Results from Threads



- **We can use shared state to “return” results**

- Acquire lock in producer
- Write the shared result
- Wait for the result
- Read the result

- **We have some problems:**

- Getting the granularity right is hard
- We cannot communicate exceptions

```
int main() {  
    auto mutex = std::mutex{};  
    auto finished = std::condition_variable{};  
    auto shared = 0;  
  
    auto thread = std::thread{[&]{  
        std::this_thread::sleep_for(2s);  
        auto guard = std::lock_guard{mutex};  
        shared = 42;  
        finished.notify_all();  
    }};  
  
    std::this_thread::sleep_for(1s);  
    auto lock = std::unique_lock{mutex};  
    finished.wait(lock);  
  
    std::cout << "The answer is: "  
              << shared << '\n';  
    thread.join();  
}
```



- **Futures represent results that may be computed asynchronously**
- **They allow us to:**
 - Wait until the result is available:
 - `wait()` – blocks until available
 - `wait_for(<timeout>)` – blocks until available or timeout elapsed
 - `wait_until(<timepoint>)` – blocks until available or the timepoint has been reached
 - Get the result
 - `get()` – blocks until available and returns the result value or throws if the future contains an exception
- **Their dtor waits for the result to become available**
 - Unless they have no shared state (see `std::async` later)



- **Promises are one origin of futures**
- **They allow us to:**
 - Obtain a future using `get_future()`
 - Publish results or errors:
 - `set_result(<result_value>)` – set the associated future's result to `result_value`
 - `set_exception(<exception pointer>)` – set the associated future's exception



```
int main() {  
    using namespace std::chrono_literals;  
  
    std::promise<int> promise{};  
    auto result = promise.get_future();  
  
    auto thread = std::thread { [&]{  
        std::this_thread::sleep_for(2s);  
        promise.set_value(42);  
    }};  
  
    std::this_thread::sleep_for(1s);  
  
    std::cout << "The answer is: " << result.get() << '\n';  
    thread.join();  
}
```



`std::async`



- **Computing results asynchronously is a common task**
 - Offload intensive computations
 - Perform I/O operations
 - Building GUI applications
- **The C++ standard provides a ready made solution: `std::async`**
 - It allows us to just return our result from our computation function
 - Additionally it catches all exceptions propagates them



```
template<typename Function, typename ...Args>  
std::future<...> async(Function&& f, Args&&... args);
```

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

- Schedules the execution of the given lambda (NOTE: not necessarily in a different thread!)
- Returns a `std::future` that will store the result
- `get()` waits for the result to be available



- 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'  
}
```



- Operations with `std::launch::async` are executed, regardless if we need their results or not
- But what if we don't care about the result?
 - Maybe the original user of the result is gone
 - `std::launch::deferred` defers execution until the result is obtained from the `std::future`

```
int main() {  
    auto the_answer = std::async(std::launch::deferred, [] {  
        // Calculate for 7.5 million years  
        return 42;  
    });  
}
```

- This *launch policy* is also known as *lazy evaluation*
- However: The result will be computed on the thread calling `get()`!



- The C++ standard features API for using threads
- `std::mutex` can be used in conjunction with `std::lock_guard` etc. to safely access shared state
- `std::promise` and `std::future` can be used to asynchronously provide results
- Asynchronous operations can easily be started using `std::async`
 - Beware of the default *launch policy*!

