Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 8 – Multi-Threading and Mutexes

Thomas Corbat / Felix Morgner
Rapperswil / St. Gallen, 18.04.2024
FS2024

Agenda

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

Recap Week 7

Ostschweizer
Fachhochschule

- **Let the compiler compute constant values for us:**

```
constexpr auto 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;
};
```

Make Our Types Convenient to Use (UDL)

```
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++

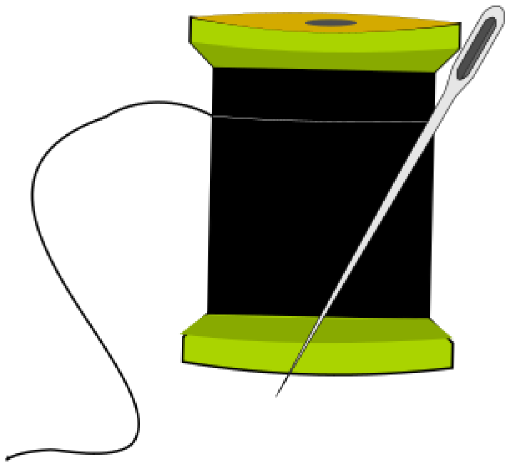
- **Participants should…**
  - know how to work with the C++ standard threading API
  - be able to use `std::async` to run computations asynchronously
  - be familiar with the C++ standard locking mechanisms
  - be able to use `std::promise` and `std::future` to obtain asynchronous results

- **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 explicitely 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`



Ostschweizer Fachhochschule

```
class std::thread
```

```
auto main() -> int {
  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 {
  auto operator()() const -> void {
    std::cout << "Functor" << std::endl;
  }
};

auto function() -> void {
  std::cout << "Function" << std::endl;
}

auto main() -> int {
  std::thread functionThread{function};
  std::thread functorThread{Functor{}};

  functorThread.join();
  functionThread.join();
}
```

```cpp
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!**

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

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

}

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

What If We Don't Join?

- **Guess what happens**

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

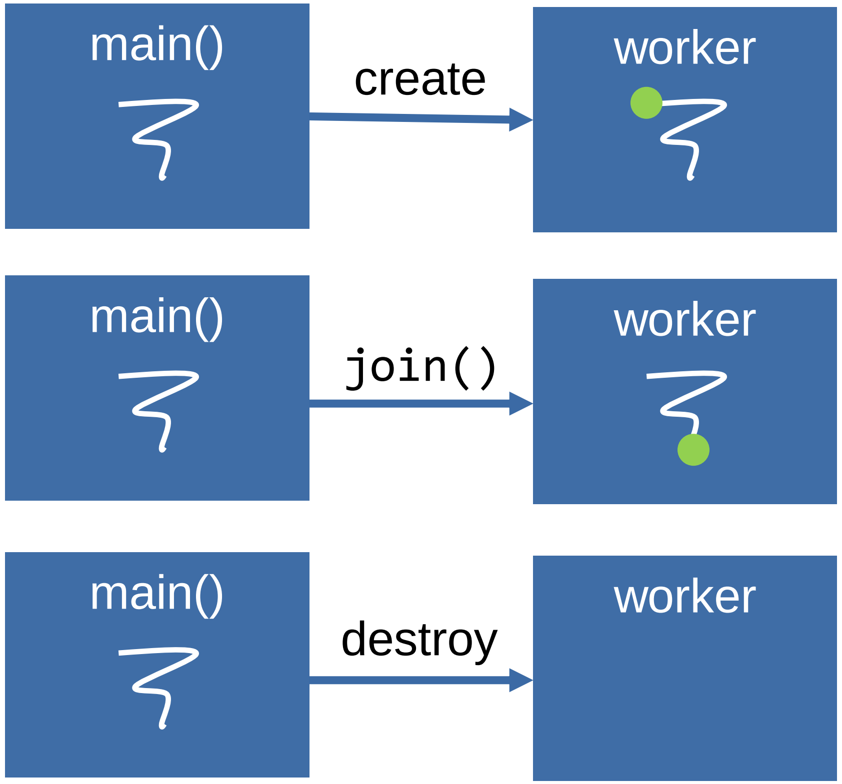
```
Main
Lambda
terminate called without an active exception
```

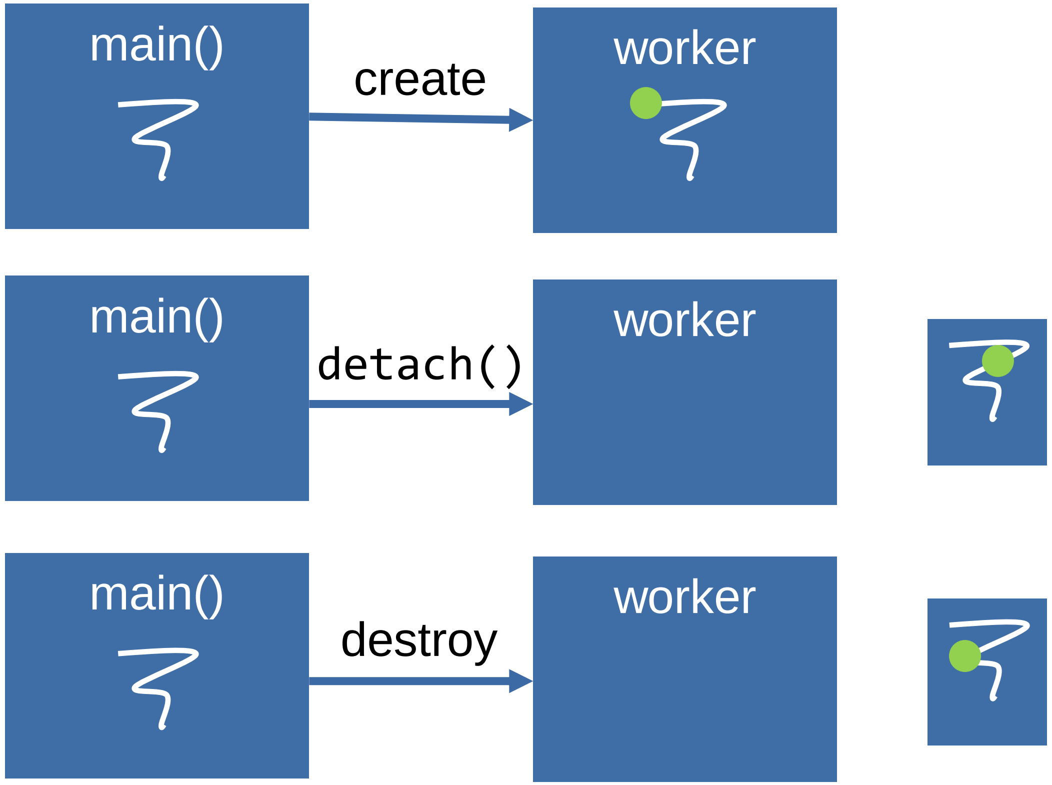
Joining/Detaching `std::thread`

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

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

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





Joining/Detaching `std::thread`

- **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

```
auto startThread() -> void {
  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();
}

auto main() -> int {
  using namespace std::chrono_literals;
  startThread();
  std::this_thread::sleep_for(2s);
}
```

- **RAII wrapper that automatically calls `join()`**
  - `std::jthread`
  - Also supports external stop requests
    - `t.request_stop()`
    - Multiple threads can request the stop of an `std::jthread`

```
auto main() -> int {
  std::jthread t {[]{
    std::cout << "Hello Thread"<< std::endl;
  }};
  std::cout << "Hello Main" << std::endl;
}
```

```
auto main() -> int {
  std::jthread t {[](std::stop_token token){
    while (!token.stop_requested()) {
      std::this_thread::sleep_for(100ms);
    }
    std::cout << "Stop requested"<< std::endl;
  }};
  std::this_thread::sleep_for(2s);
  t.request_stop();
}
```

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

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

auto main() -> int {
  using std::cout;
  using std::endl;

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

?

```
Hello Main
Hello Thread
```

```
HHeelllloo MTahirnead
```

```
<Arbitrary Combination>
```

Knowing Your Thread

- **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 threat for a duration
- **`sleep_until(time_point)`**
- **`yield()`**
  - Allows OS to schedule another thread
- **NB: Timing doesn't guarantee sequence!**

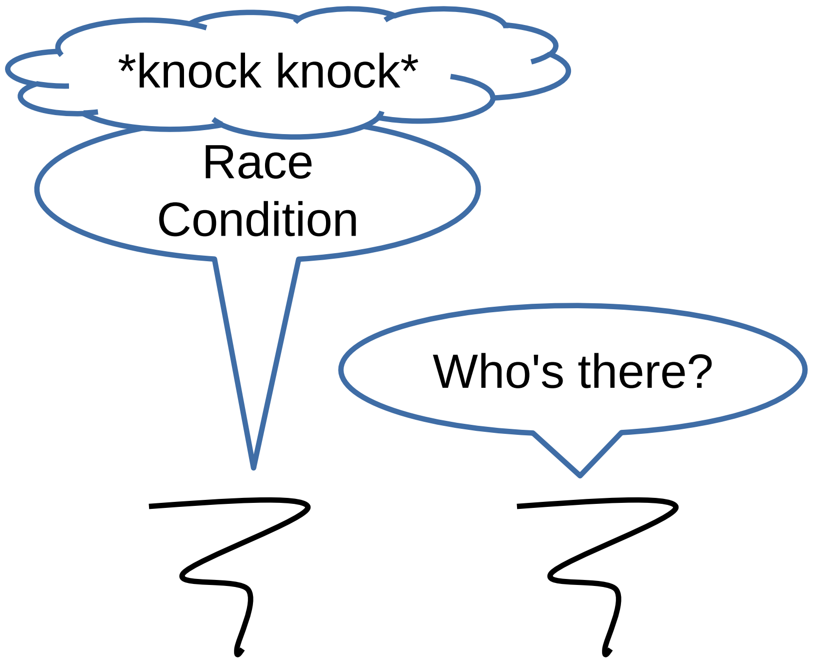
```
auto main() -> int {
  using std::cout;
  using std::endl;
  using namespace std::chrono_literals;

  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();
}
```

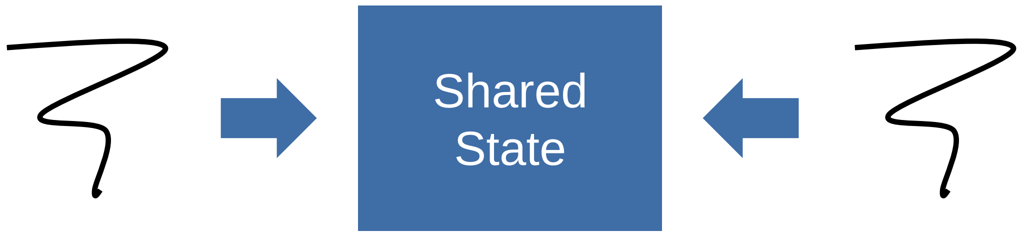
<pre>auto calcAsync() -> void {
  std::thread t{longRunningAction};
  doSomethingElse();
  t.join();
}</pre>
	Depends
	
	If doSomethingElse() does not throw an exception the code is correct.
	If an exception is thrown you are doomed!

<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.

Communication Between Threads



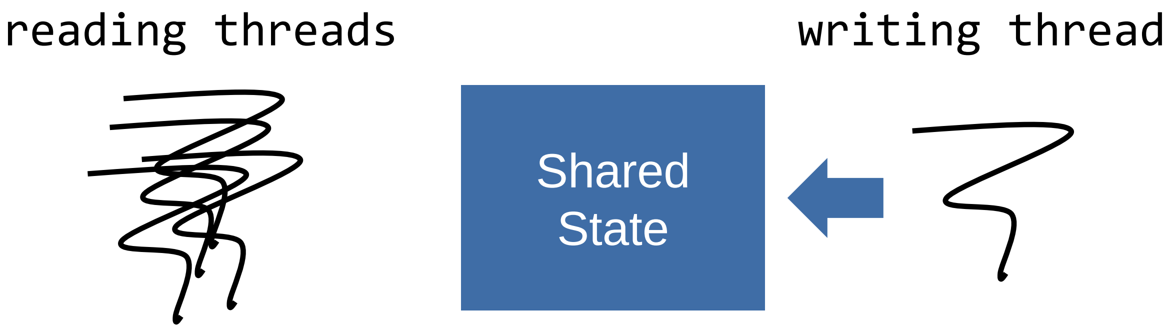
- **Mutable Shared State**
- **Problem: Data Race**
  - Two operations on the same memory location
  - At least one is not atomic (at the same time)
  - At least one is a modifying operation
- **Solution**
  - Locking the shared access
  - Make access atomic



- **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	`std::mutex`	`std::recursive_mutex`
	Yes	`std::timed_mutex`	`std::recursive_timed_mutex`

- **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()`



- **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)

- **Threadsafe queue**
  - Delegate functionality to `std::queue`
  - Make every member function **mut**ually **ex**clusive
- **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>;
  auto push(T const &t) -> void {
    guard lk{mx};
    q.push(t);
  }
  T pop() { /* later */ return T{};}
  auto try_pop(T & t) -> bool {
    guard lk{mx};
    if (q.empty()) return false;
    t = q.front();
    q.pop();
    return true;
  }
  auto empty() const -> bool{
    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
auto swap(threadsafe_queue<T> & other) -> void {
  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
auto swap(threadsafe_queue<T> & other) -> void {
  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)**

```cpp
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{};
};
```

Standard Containers and Concurrency

- **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

Results via Shared State and Locking

- **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

```
auto main() -> int {
    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 computes 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 may waits for the result to become available**
  - Unless they have no shared state (see `std::async` later)

Class `std::promise`

`#include <future>`

- **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

```
auto main() -> int {
    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 intesive 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>
auto async(Function&& f, Args&&... args) -> std::future<...>;
```

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

Behavior of `std::async`

- **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'
}
```

`std::launch::deferred`

- **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`

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

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

Summary

- **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*!