



# C++ User Group Aachen

## Multithreading in C++11

Fabian Kantereit  
Geschäftsbereich Produktion  
[fabian.kantereit@inform-software.com](mailto:fabian.kantereit@inform-software.com)



- Threading Basics
- Thread Synchronization
- Atomics
- Futures / Promises
- Exception Handling

## Threading Basics (1)

---

<thread> library

std::thread → Thread object

std::thread::hardware\_concurrency() [static] → Number of supported hardware threads

Member Functions:

joinable() → can the thread be joined ( active thread )

get\_id() → thread id

join() → joins the thread

detach() → executes the thread independently from the local handle, resources will be freed on thread termination

Note: Threads can be moved and swapped, but not copied / assigned!

# Passing arguments to threads:

- arguments can be passed as additional parameters
- references have to be wrapped with `std::ref`
- avoid passing local references to threads, that might go out of scope

## Threading Basics (3)

---

namespace `this_thread`:

`std::this_thread::get_id()` → returns the id of the current thread

`std::this_thread::sleep_for( std::chrono::duration )` → thread sleeps for the duration

Einmaliger Methodenaufruf:

`<mutex>` library

`once_flag`

`call_once` → will call method exactly once

Storage specifier:

`thread_local` → variable is allocated for each thread

# Thread Synchronization (Mutex)

---

<mutex> library

Mutex types:

`std::mutex` → standard mutex

`std::timed_mutex` → allows to specify a locking timeout

`std::recursive_mutex` → allows recursive locking

`std::recursive_timed_mutex` → combines the above

`std::shared_timed_mutex` → will be introduced with C++14

Mutex locking:

`std::mutex::lock()` → locks the mutex, waits until it is available

`std::mutex::try_lock()` → tries to lock the mutex without blocking

`std::mutex::unlock()` → unlocks the mutex

`std::timed_mutex::try_lock_for( duration )` → try to lock the mutex for the specified amount of time

Note: Mutexes are NOT exception-safe and should only be used in combination with a lock

# Thread Synchronization (Lock)

---

<mutex> library

Available Locks:

`std::lock_guard<mutex_type>` → scope-based locking

`std::unique_lock<mutex_type>` → movable lock, can be unlocked manually

`std::shared_lock<mutex_type>` → moveable and shareable lock (C++14)

Why use locks?

Exception-safety, preventing deadlock: Locks unlock on destruction

Lock Operations:

`lock_guard` → locks on creation, unlocks on destruction (RAII Pattern)

`unique_lock` → locks on creation, unlocks on destruction

`unique_lock::lock()` → manually lock the underlying mutex

`unique_lock::unlock()` → manually unlock the underlying mutex

Also available: `try_lock`, `try_lock_for`, `try_lock_until`

## Thread Synchronization (Condition Variables):

---

<condition\_variable> library

std::condition\_variable → provides inter-thread signaling, associated with a unique\_lock

std::condition\_variable\_any → works with all locks

Mechanism: Locks are unlocked on wait() call and reacquired, when notify\_one is called

condition\_variable::notify\_one() → notifies one thread to continue

condition\_variable::notify\_all() → resume all waiting threads

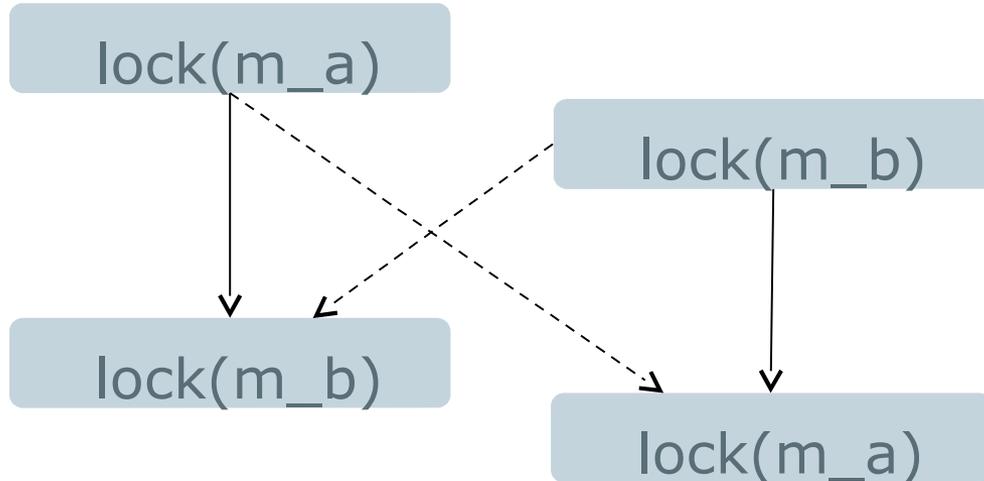
condition\_variable::wait( Lockable&, Predicate ) → waits for lockable, loops until Predicate is fulfilled

Also available: wait\_for(), wait\_until()

Note: wait can be used without a predicate, but using one prevents spurious wakeups

## Avoiding deadlocks:

---



**!DEADLOCK!**

### Solutions:

Keep it simple, avoid synchronization points!

`std::lock(Lockable&,...)` → locks all mutexes simultaneously

hierarchical locking → assign level to every lock and enforce locking order

## Threading Concepts (Singleton):

---

### Task:

Implement a thread-safe and exception- safe singleton

### Solution:

```
class A {  
    public:  
        static A &get_instance() {  
            static A instance;  
            return instance;  
        }  
};
```

## Atomic Types:

---

- `std::atomic<type>`
- thread-safe types, no race conditions
- read-modify-write is not interrupted
- Lock-free, if the platform supports atomic types
- Generally faster than locking

## Atomic Operations:

---

### Available Operations:

- load/store
- fetch\_add/sub/and/or/xor
- exchange
- compare\_exchange\_weak/strong

### atomic\_flag:

- test\_and\_set/clear

## Atomic fetch operations:

---

old\_val = read atomic\_val



Modify atomic\_val



Write atomic\_val



Return old\_val

## Threading Concepts( atomic spinlock ):

---

- atomic\_flag is used as a mutex

```
void lock() {  
    while(flag.test_and_set());  
}
```

```
void unlock() {  
    flag.clear();  
}
```

- further improvement: use RAI pattern lock

### What are Futures?

- result of asynchronous operation
- Contains value or exception
- can only be retrieved once
- associated with `std::async`,  
`std::packaged_task` or `std::promise`

What is the purpose of `std::async`?

- Asynchronous start of function, `packaged_task`, lambda expression or `std::bind`
- stores return value in future
- stores exception in future
- manages the threads

## What is a promise?

- Stores a value for asynchronous retrieval
- useful if a thread computes multiple values
- value/exception must be set manually
- value is available via `get_future()`

## Exceptions thrown by Futures and Promises

---

future\_error:

- broken\_promise
- future\_already\_retrieved
- promise\_already\_satisfied
- no\_state

## Exception Handling:

---

### Strategies:

- global try/catch(...) in every thread and catch/rethrow with exception\_ptr
- store exceptions in promise
- let std::async/packaged\_task store exceptions

## Sources

---

- C++ Concurrency in Action: Practical Multithreading (Anthony Williams)
- <http://www.cppreference.com>