Locks

Locks take care of thier resource following the [RAII](http://en.cppreference.com/w/cpp/language/raii) idiom.  A lock automatically binds its mutex in the constructor and releases it in the destructor. This considerably reduces risk of a deadlock, because the runtime takes care of the mutex.

### std::lock\_guard

First, the simple use case.

mutex m;

m.lock();

sharedVariable= getVar();

m.unlock();

With so little code mutex m ensures  access of the critical section sharedVariable= getVar() is sequential. Sequential means  - in this special case -  that each thread gains acces to critical section in order. The code is simple, but prone to deadlocks. Deadlock appears  if the critical section throws an exception or if the programmer simply forgets to unlock the mutex. With std::lock\_guard we can do this more elegant:

{

std::mutex m,

std::lock\_guard<std::mutex> lockGuard(m);

sharedVariable= getVar();

}

That was easy. But what's about the opening and closing brackets? The lifetime of std::lock\_guard is limited by the brackets (<http://en.cppreference.com/w/cpp/language/scope#Block_scope>). That means, its lifetime ends when it leaves the critical section. Exactly at that time point, the destructor of std::lock\_guard is called and - I guess, you know it - the mutex is released. It happens automatically, and, in addition, it  happens if  getVar() in sharedVariable = getVar() throws an exception. Of course, function body  scope or loop scope also limit the lifetime of an object.

### std::unique\_lock

std::unique\_lock is mightier but more expansive than its small brother std::lock\_guard.

A std::unique\_lock enables youin addition to std::lock\_guard

* create it without an associated mutex
* create it without a locked associated mutex
* explicitly and repeatedly set or release the lock of the associated mutex
* move the mutex
* try to lock the mutex
* delayed lock the associated mutex

the function deadlock has to lock their mutex in an atomic fashion.

// deadlockResolved.cpp

#include <iostream>

#include <chrono>

#include <mutex>

#include <thread>

struct CriticalData{

std::mutex mut;

};

void deadLock(CriticalData& a, CriticalData& b){

std::unique\_lock<std::mutex>guard1(a.mut,std::defer\_lock);

std::cout << "Thread: " << std::this\_thread::get\_id() << " first mutex" << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(1));

std::unique\_lock<std::mutex>guard2(b.mut,std::defer\_lock);

std::cout << " Thread: " << std::this\_thread::get\_id() << " second mutex" << std::endl;

std::cout << " Thread: " << std::this\_thread::get\_id() << " get both mutex" << std::endl;

std::lock(guard1,guard2);

// do something with a and b

}

int main(){

std::cout << std::endl;

CriticalData c1;

CriticalData c2;

std::thread t1([&]{deadLock(c1,c2);});

std::thread t2([&]{deadLock(c2,c1);});

t1.join();

t2.join();

std::cout << std::endl;

}

Here is a sketch of the second approach.

std::lock(a.mut, b.mut);

std::lock\_guard<std::mutex> guard1(a.mut, std::adopt\_lock);

std::lock\_guard<std::mutex> guard2(b.mut, std::adopt\_lock);

### A side note: Special deadlocks

It's an illusion that only a mutex can produce a deadlock.**Each time a thread has to wait for a resource, while it is holding a resource, a deadlock lurks near.**

Even a thread is a resource.

// blockJoin.cpp

#include <iostream>

#include <mutex>

#include <thread>

std::mutex coutMutex;

int main(){

std::thread t([]{

std::cout << "Still waiting ..." << std::endl;

std::lock\_guard<std::mutex> lockGuard(coutMutex);

std::cout << std::this\_thread::get\_id() << std::endl;

}

);

{

std::lock\_guard<std::mutex> lockGuard(coutMutex);

std::cout << std::this\_thread::get\_id() << std::endl;

t.join();

}

}

Two ways to solve this deadlock come to mind.

* The main thread locks the output stream std::cout after the call t.join().

{

t.join();

std::lock\_guard<std::mutex> lockGuard(coutMutex);

std::cout << std::this\_thread::get\_id() << std::endl;

}

* The main thread releases its lock by an additional scope. This is done before the t.join() call.

{

{

std::lock\_guard<std::mutex> lockGuard(coutMutex);

std::cout << std::this\_thread::get\_id() << std::endl;  
 }

t.join();

}