Types of Locks: std::unique_lock

This lesson gives an overview of std::unique_lock which is a type of lock used in C++.

WE'LL COVER THE FOLLOWING

- Features
- Methods
  - More on `lk.try_lock` and `lk.release` methods
  - How to solve deadlock with `std::unique_lock`?

Features

In addition to what's offered by an `std::lock_guard`, an `std::unique_lock` enables us to:

- create it without an associated mutex.
- create it without locking the associated mutex.
- explicitly and repeatedly set or release the lock of the associated mutex.
- move the mutex.
- try to lock the mutex.
- delay the lock on the associated mutex.

Methods

The following table shows the methods of an `std::unique_lock lk`.

Method	Description
`lk.lock()`	Locks the associated mutex.

`std::lock(lk1, lk2, ... )`	Atomically locks the arbitrary number of associated mutexes.
`lk.try_lock()`, and `lk.try_lock_for(relTime)`, and `lk.try_lock_until(absTime)`	Tries to lock the associated mutex.
`lk.release()`	Releases the mutex. The mutex remains locked.
`lk.swap(lk2)` and `std::swap(lk, lk2)`	Swaps the locks.
`lk.mutex()`	Returns a pointer to the associated mutex.
`lk.owns_lock()`	Checks if the lock has a mutex.

More on `lk.try_lock` and `lk.release` methods

`lk.try_lock_for(relTime)` needs a relative time duration; `lk.try_lock_until(absTime)` needs an absolute time point.

`lk.try_lock` tries to lock the mutex and returns immediately. On success, it returns true, but otherwise, it's false. In contrast, the methods `lk.try_lock_for` and `lk.try_lock_until` block the release until the specified timeout occurs or the lock is acquired, whichever comes first. we should use a steady clock for our time constraint. A steady clock cannot be adjusted.

The method `lk.release()` returns the mutex; therefore, we have to unlock it manually.

How to solve deadlock with `std::unique_lock`?

Thanks to `std::unique_lock`, it is quite easy to lock many mutexes in one

Thanks to `std::unique_lock`, it is quite easy to lock many mutexes in one atomic step; therefore, we can overcome deadlocks by locking mutexes in a different order. Remember the deadlock from the subsection Issues of Mutexes?

```
// deadlock.cpp

#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>

struct CriticalData{
  std::mutex mut;
};

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

  a.mut.lock();
  std::cout << "get the first mutex" << std::endl;
  std::this_thread::sleep_for(std::chrono::milliseconds(1));
  b.mut.lock();
  std::cout << "get the second mutex" << std::endl;
  // do something with a and b
  a.mut.unlock();
  b.mut.unlock();

}

int main(){

  CriticalData c1;
  CriticalData c2;

  std::thread t1([&]{deadLock(c1,c2);});
  std::thread t2([&]{deadLock(c2,c1);});

  t1.join();
  t2.join();

}
```

Let's solve this issue. The function `deadLock` has to lock its mutexes atomically and that's exactly what happens in the following example.

```
// deadlockResolved.cpp

#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>
```

```
using namespace std;

struct CriticalData{
  mutex mut;
};

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

  unique_lock<mutex> guard1(a.mut,defer_lock);
  cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;

  this_thread::sleep_for(chrono::milliseconds(1));

  unique_lock<mutex> guard2(b.mut,defer_lock);
  cout << "  Thread: " << this_thread::get_id() << " second mutex" <<  endl;

  cout << "    Thread: " << this_thread::get_id() << " get both mutex" << endl;
  lock(guard1,guard2);
  // do something with a and b
}

int main(){

  cout << endl;

  CriticalData c1;
  CriticalData c2;

  thread t1([&]{deadLock(c1,c2);});
  thread t2([&]{deadLock(c2,c1);});

  t1.join();
  t2.join();

  cout << endl;

}
```

If we call the constructor of `std::unique_lock` with `std::defer_lock`, the underlying mutex will not be locked automatically. At this point (lines 16 and 21), the `std::unique_lock` is just the owner of the mutex. Thanks to the variadic template `std::lock`, the lock operation is performed in an atomic step (line 25). A variadic template is a template which can accept an arbitrary number of arguments. `std::lock` tries to get all locks in one atomic step, so it either gets all of them or none of them and retries until it succeeds.

In this example, `std::unique_lock` manages the lifetime of the resources and `std::lock` locks the associated mutex; we can also do it the other way around. In the first step the mutexes are locked, in the second `std::unique_lock` manages the lifetime of resources. Here is an example 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);
```

Let us see this approach in action:

```
// deadlockResolved.cpp

#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>

using namespace std;

struct CriticalData{
  mutex mut;
};

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

  lock_guard<std::mutex> guard1(a.mut, std::adopt_lock);
  cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;

  this_thread::sleep_for(chrono::milliseconds(1));

  lock_guard<std::mutex> guard2(b.mut, std::adopt_lock);
  cout << "  Thread: " << this_thread::get_id() << " second mutex" <<  endl;

  cout << "    Thread: " << this_thread::get_id() << " get both mutex" << endl;
  lock(a.mut, b.mut);
  // do something with a and b
}

int main(){

  cout << endl;

  CriticalData c1;
  CriticalData c2;

  thread t1([&]{deadLock(c1,c2);});
  thread t2([&]{deadLock(c2,c1);});

  t1.join();
  t2.join();

  cout << endl;

}
```

Resolving the deadlock with an std::scoped_lock

With C++17, the resolution to the deadlock becomes quite easy. We get the `std::scoped_lock` that can lock an arbitrary number of mutexes atomically - so as long as we only have to use an `std::lock_guard` instead of the `std::lock` call. That's all. Here is the modified function `deadlock`.

```
// deadlockResolvedScopedLock.cpp
void deadLock(CriticalData& a, CriticalData& b){

  cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;
   this_thread::sleep_for(chrono::milliseconds(1));
   cout << "  Thread: " << this_thread::get_id() << " second mutex" <<  endl;
   cout << "    Thread: " << this_thread::get_id() << " get both mutex" << endl;

   std::scoped_lock(a.mut, b.mut);
   // do something with a and b
 }
```

With C++14, C++ adds support for `std::shared_lock`; let's see this in the next lesson.