Types of Locks: std::lock_guard

This lesson gives an introduction to locks and explains how std::lock_guard is used in C++.

WE'LL COVER THE FOLLOWING

- `std::lock_guard`

Locks take care of their resources following the RAII idiom. A lock automatically binds its mutex in the constructor and releases it in the destructor; this considerably reduces the risk of a deadlock because the runtime takes care of the mutex.

Locks are available in three different flavors: `std::lock_guard` for the simple use-cases; `std::unique-lock` for the advanced use-cases; `std::shared_lock` is available (with C++14) and can be used to implement reader-writer locks.

First, the simple use-case:

```
std::mutex m;
m.lock();
sharedVariable = getVar();
m.unlock();
```

The mutex `m` ensures that access to the critical section `sharedVariable= getVar()` is sequential. Sequential, in this special case, means that each thread gains access to the critical section one after the other. This maintains a kind of total order in the system. The code is simple but prone to deadlocks. A deadlock appears if the critical section throws an exception or if the programmer simply forgets to unlock the mutex.

`std::lock_guard`

With `std::lock_guard` we can do this in a more elegant way:

With `std::lock_guard` we can do this in a more elegant way:

```
{
  std::mutex m,
  std::lock_guard<std::mutex> lockGuard(m);
  sharedVariable= getVar();
}
```

That was easy, but what's the story with the opening and closing brackets? The lifetime of `std::lock_guard` is limited by the curly bracket, which means that its lifetime ends when it passes the closing curly brackets. At exactly that point, the `std::lock_guard` destructor is called and - as we may have guessed - the mutex is released automatically. It is also released if `getVar()` in `sharedVariable = getVar()` throws an exception. The function scope and loop scope also limit the lifetime of an object.

> **i `std::scoped_lock` with C++17**
>
> With C++17, we get an `std::scoped_lock`. It's very similar to `std::unique_lock`, but `std::scoped_lock` can lock an arbitrary number of mutexes atomically. That being said, we have to keep two facts in mind:
>
> 1. If one of the current threads already owns the corresponding mutex and the mutex is not recursive, the behaviour is undefined.
> 2. We can only take ownership of the mutexes without locking them. In this case, we have to provide the `std::adopt_lock_t` flag to the constructor: `std::scoped_lock(std::adopt_lock_t, MutexTypes& ... m)`.
>
> We can quite elegantly solve the previous deadlock by using an `std::scoped_lock`. We will discuss the resolution to the deadlock in the following section.

In the next lesson, we will see how `std::unique_lock` is stronger but more expensive than its counterpart `std::lock_guard`.