Locks

In this lesson, we will learn about different locks and their usage in embedded programming.

WE'LL COVER THE FOLLOWING

- RAII-Idiom (Resource Acquisition Is Initialization)
- `std::lock_guard`
- `std::unique_lock`
- `std::shared_lock`

Locks are available in **three** different forms:

1. `std::lock_guard` for simple use-cases
2. `std::unique-lock` for advanced use-cases
3. `std::shared_lock` is available with C++14 and can be used to implement reader-writer locks.

> 💡 Locks need the header `<mutex>`.

RAII-Idiom (Resource Acquisition Is Initialization)

Locks take care of their resource following the ***RAII*** idiom. A lock automatically binds its mutex in the constructor and releases it in the destructor. This reduces the risk of a deadlock because the runtime handles the mutex.

> If the lock goes out of scope, the resource will be immediately released.

`std::lock_guard`

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. In this special case sequential means that each thread gains access to the critical section after the other. This maintains a 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 forgets to unlock the mutex. With `std::lock_guard`, this can be handled in a more elegant way:

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

What's the story with the opening and closing brackets? The lifetime of `std::lock_guard` is limited by the curly brackets, meaning that its lifetime ends when it passes the closing curly brackets. At that moment, the `std::lock_guard` destructor is called, and the mutex is released. This happens automatically if `getVar()` throws an exception in `sharedVariable = getVar()`. The function scope and loop scope also limit the lifetime of an object.

`std::unique_lock`

In addition to what's offered by a `std::lock_guard`, a `std::unique_lock` enables you 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.

- lock the mutex.
- delay the lock on the associated mutex.

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

Method	Description
`lk.lock()`	Locks the associated mutex.
`std::lock(lk1, lk2, ... )`	Locks atomically 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.

`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 resources immediately. On success, it returns `true`. If it fails, it returns `false`. In contrast, the methods `lk.try_lock_for` and `lk.try_lock_until` block until the specified timeout occurs or the lock is acquired. You should use a ***steady clock*** for your time constraint. Note: a steady clock cannot be adjusted.

The method `lk.release()` returns the mutex; therefore, you must unlock it manually.

Due to `std::unique_lock`, it is easy to lock multiple mutexes in one atomic step, meaning that you can overcome deadlocks by locking mutexes in a different order.

`std::shared_lock`

A `std::shared_lock` has the same interface as a `std::unique_lock` but it behaves differently when used with a `std::shared_timed_mutex`. Many threads can share one `std::shared_timed_mutex` and, therefore, implement a reader-writer lock. The idea of reader-writer locks is straightforward and extremely useful. An arbitrary number of threads executing read operations can access the critical region simultaneously, but only one thread is allowed to write.

Reader-writer locks do not solve the fundamental problem. Threads compete for access to a critical region, but they do help to minimize the bottleneck.

A telephone book is a common example that uses a reader-writer lock. Usually, many people want to look up a telephone number, but only a few want to change them.

This topic will be further explained in the next lesson with the help of a few examples.