Locks

- `std::lock_guard` and `std::unique_lock` manage the lifetime of the their mutex according to the RAII-Idiom.
  - Needs the header `<mutex>`.

- RAII-Idiom (Resource Acquisition Is Initialization)
  - The lifetime of a resource is bound to an automatic object.
  - The resource will be initialized in the constructor of the object; released in the destructor of the object.
  - The RAII-Idiom is often used in C++: Smart pointer.

In case the lock goes out of scope, the resource will be immediately released.

`std::lock_guard`

`std::lock_guard` is for the simple use case.

- std::lock_guard
  - Automatically locks the mutex in its constructor and releases it in its destructor.
  - Is cheaper to use than its more powerful brother `std::unique_lock`.

```
std::mutex myMutex;
auto res = getVar();
{
    std::lock_guard<std::mutex> myLock(myMutex);
    sharedVariable = res;
}
```

std::unique_lock

Function	Description
`lk.lock()`	Locks the associated mutex.
`lk.unlock()`	Releases the associated mutex.

`lk.try_lock(),`
`lk.try_lock_for(rel_time),`
`lk.try_lock_until(abs_time)`
	lk tries to lock the mutex.

`lk.release()`	Releases the mutex without releasing it.
`lk.swap(lk2), std::swap(lk,lk2)`	Swaps the locks.
`lk.mutex()`	Returns a pointer to the associated mutex.
`lk.owns_lock()`	Tests if the lock has a mutex.
`std::lock( ... )`	Locks an arbitrary number of mutex atomically.

In C++14 there is a `std::shared_timed_mutex`.

➡ You can implement reader-writer locks in combination with `std::shared_lock`.

`uniqueLock.cpp`

`readerWriterLock.cpp`