Mutexes

In this lesson, let's look at how to tackle data races with mutexes.

WE'LL COVER THE FOLLOWING

Shared Data

We only need to think about synchronization if we have shared mutable data since this kind of data is prone to **data races**.

As previously stated, when the program has a data race, it has undefined behavior. Undefined behavior affects the program before and after the undefined behavior, causing many problems for the programmer and the program. When our program has undefined behavior, the C++ community says:

> *When the program has a catch-fire semantic, even the computer can catch fire* 🔥

The easiest way to visualize concurrent, unsynchronized read and write operations is to write something to `std::cout`.

> A mutex (**mut**ual **ex**clusion) guarantees that at most one thread has

> A mutex (**mutual exclusion**) guarantees that at most one thread has access to the critical region.

Mutexes

Mutex stands for mutual exclusion. It ensures that only one thread can access a critical section once at any time.

By using a mutex, the mess of the workflow turns into harmony.

> The mutex variations need the header `<mutex>`.

C++ has five different mutexes that can lock recursively, tentative with and without time constraints.

Method	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
`m.lock`	yes	yes	yes	yes
`m.unlock`	yes	yes	yes	yes
`m.try_lock`	yes	yes	yes	yes
`m.try_lock_for`			yes	yes
`m.try_lock_until`			yes	yes

With C++14, we have a `std::shared_timed_mutex` that is the base for ***reader-writer locks***.The `std::shared_timed_mutex` enables us to implement ***reader-writer locks***. This means that we can use `std::shared_timed_mutex` for exclusive or for shared locking. We will get an exclusive lock if we put the `std::shared_timed_mutex` into a ***`std::lock_guard`***. We will get a shared lock if we put the `std::shared_timed_mutex` into a `std::unique_lock`.

We should not use mutexes directly; rather, we should put mutexes into locks. Let's examine the reasons in the next section.

Deadlocks

The issues with mutexes boil down to one main concern: deadlocks.

Deadlock: A deadlock is a state where two or more threads are blocked because each thread waits for the release of a resource before it releases its own resource.

The result of a deadlock is a total standstill. The thread that tries to acquire the resource is blocked forever. Let's examine the concept of deadlocks further.

Exceptions and Unknown Code

The small code snippet has a lot of issues.

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

The five main issues:

1. When the function `getVar()` throws an exception, the mutex `m` will not be released.

2. When the `m.unlock()` is not called, the mutex `m` will not be released.

3. **Do not** call an unknown function while holding a lock. If the function `getVar` tries to lock the mutex `m`, the program has undefined behavior since `m` is not a recursive mutex. Most of the time, undefined behavior will result in a deadlock.

4. Avoid calling a function while holding a lock. The function might be from a library, and we get a new version of the library or the function will be rewritten. That is one of the dangers of a deadlock.

5. Make our critical regions as short as possible. Remove the code that does not need to be protected.

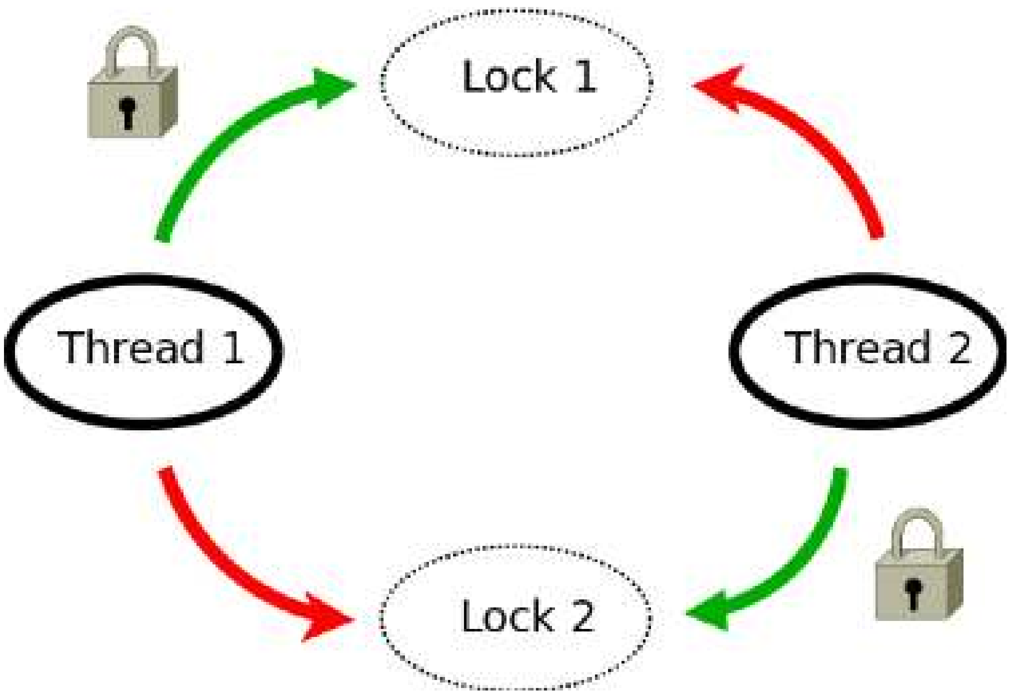
The following changed code snippet solves most of the issues of the original one.

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

The dependency is very non-linear: the more locks the program needs, the more challenging it becomes.

Lock Mutexes in Different Order

Here is a typical scenario of a deadlock resulting from locking in a different order.



Thread 1 and thread 2 need access to two resources in order to finish their work. The problem arises when the requested resources are protected by two separate mutexes and are requested in different orders (Thread 1: Lock 1 and Lock 2; Thread 2: Lock 2 and Lock 1). In this case, the thread executions will interleave in such a way that thread 1 gets mutex 1, and thread 2 gets mutex 2, resulting in a standstill. Each thread wants to get the other's mutex, but to do so, the other thread must first release it. The expression "deadly embrace" best describes this kind of deadlock.

The example in the next lesson will build on your understanding of this topic.