Shared Variables

Now that we've learnt about threads, let's discuss information sharing between them.

If more than one thread is sharing a variable, you have to coordinate the access. That's the job for mutexes and locks in C++.

Data race

A data race is a state in which at least two threads access a shared data at the same time, and at least one of the threads is a writer. Therefore the program has undefined behavior.

You can observe very well the interleaving of threads if a few threads write to `std::cout`. The output stream `std::cout` is, in this case, the shared variable.

```
//...
#include <thread>
//...
using namespace std;
```

```
using namespace std;
struct Worker{
  Worker(string n):name(n){};
  void operator() (){
    for (int i= 1; i <= 3; ++i){
      this_thread::sleep_for(chrono::milliseconds(200));
      cout << name << ": " << "Work " << i << endl;
    }
  }
private:
  string name;
};

// main
thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker(" Andrei"));
thread scott= thread(Worker (" Scott"));
thread bjarne= thread(Worker(" Bjarne"));
```

The output on `std::cout` is uncoordinated.

> 🔍 **The streams are thread-safe**
> The C++11 standard guarantees that the characters are written atomically. Therefore you don't need to protect them. You only have to protect the interleaving of the threads on the stream if the characters should be written or read in the right sequence. That guarantee holds for the input and output streams.

`std::cout` is in the example the shared variable, which should have exclusive access to the stream.

Mutexes

Mutex (mutual exclusion) `m` guarantees that only one thread can access the critical region at one time. They need the header . A mutex `m` locks the critical section by the call `m.lock()` and unlocks it by `m.unlock().`

```
//...
#include <mutex>
#include <thread>
//...
using namespace std;
std::mutex mutexCout;
struct Worker{
  Worker(string n):name(n){};
  void operator() (){
    for (int i= 1; i <= 3; ++i){
      this_thread::sleep_for(chrono::milliseconds(200));
      mutexCout.lock();
      cout << name << ": " << "Work " << i << endl;
```

```
      cout << name << ": " << "Work " << i << endl;
      mutexCout.unlock();
    }

  }
private:
  string name;
};

// main
thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker(" Andrei"));
thread scott= thread(Worker (" Scott"));
thread bjarne= thread(Worker(" Bjarne"));
```

Now each thread after each other writes coordinated to `std::cout` because it uses the same mutex `mutexCout`.

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

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

The `std::shared_time_mutex` enables it to implement reader-writer locks. The method `m.try_-lock_for(relTime)` needs a relative time duration; the method `m.try_lock_until(absTime)` a absolute time point.

Deadlocks

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

You can get a deadlock very quickly if you forget to call `m.unlock()`. That happens for example in case of an exception in the function `getVar()`.

```
m.lock();
sharedVar= getVar();
m.unlock()
```

> **i Don't call an unknown function while holding a lock**
> If the function getVar tries to get the same lock by calling `m.lock()` you will get a deadlock, because it will not be successful and the call will block forever.

Locking two mutexes in the wrong order is another typical reason for a deadlock.

```
// ...
#include <mutex>
#include <mutex>
 // ...
struct CriticalData{
  std::mutex mut;
};
void deadLock(CriticalData& a, CriticalData& b){ a.mut.lock();
std::cout << "get the first mutex\n"; std::this_thread::sleep_for(std::chrono::milliseconds(1
std::cout << "get the second mutex\n";
  a.mut.unlock(), b.mut.unlock();
}

// main
CriticalData c1;
CriticalData c2;
std::thread t1([&]{ deadLock(c1, c2); });
std::thread t2([&]{ deadLock(c2, c1); });
t1.join();
t2.join();
```

The short time window of one millisecond `(std::this_thread::sleep_for(std::chrono::milliseconds(1)))` is enough to produce with high probability a deadlock because each thread is waiting forever on the other mutex. The result is a standstill.

> 🔍 **Encapsulate a mutex in a lock**
> It's very easy to forget to unlock a mutex or lock mutexes in a different order. To overcome most of the problems with a mutex, encapsulate it in a lock.

Locks

You should encapsulate a mutex in a lock to release the mutex automatically. A lock is an implementation of the RAII idiom because the lock binds the lifetime of the mutex to its lifetime. C++11 has `std::lock_guard` for the simple and `std::unique_lock` for the advanced use case, respectively. Both need the header `<mutex>`. With C++14 C++ has a `std::shared_lock` which is in the combination with the mutex `std::shared_time_mutex` the base for reader-writer locks.

std::lock_guard

`std::lock_guard` supports only the simple use case. Therefore it can only bind its mutex in the constructor and release it in the destructor. So the synchronization of the *worker* example is reduced to the call of the constructor.

```
//...
std::mutex coutMutex;
struct Worker{
  Worker(std::string n):name(n){};
  void operator() (){
    for (int i= 1; i <= 3; ++i){
      std::this_thread::sleep_for(std::chrono::milliseconds(200));
      std::lock_guard<std::mutex> myLock(coutMutex);
      std::cout << name << ": " << "Work " << i << std::endl;
    }
  }
private:
  std::string name;
};
```

std::unique_lock

The usage of `std::unique_lock` is more expensive than the usage of `std::lock_guard`. In contrary, a `std::unique_lock` can be created with and without mutex, can explicitly lock or release its mutex or can delay the lock of its 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()`	Release 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.

Deadlocks caused by acquiring locks in different order can easily be solved by `std::atomic`.

```
//...
#include <mutex>
//...
using namespace std;

struct CriticalData{
  mutex mut;
};

void deadLockResolved(CriticalData& a, CriticalData& b){
  unique_lock<mutex>guard1(a.mut, defer_lock);
```

```
  cout << this_thread::get_id() << ": get the first mutex" << endl;
  this_thread::sleep_for(chrono::milliseconds(1));
  unique_lock<mutex>guard2(b.mut, defer_lock);

  cout << this_thread::get_id() << ": get the second mutex" << endl;
  cout << this_thread::get_id() << ": atomic locking";
  lock(guard1, guard2);
}

CriticalData c1;
CriticalData c2;

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

Because of the argument `std::defer_lock` of the `std::unique_lock`, the locking of `a.mut` and `b.mut` is deferred. The locking takes place atomically in the call `std::lock(guard1, guard2)`.

std::shared_lock

`std::shared_lock` has the same interface as `std::unique_lock`. Also, a `std::shared_lock` supports the case where multiple threads share the same locked mutex. For this special use case, you have to use a `std::shared_lock` in combination with a `std::shared_timed_mutex`. However, if multiple threads use the same `std::shared_time_mutex` in a `std::unique_lock` only one thread can possess it.

```
#include <mutex>
//...
std::shared_timed_mutex sharedMutex;
std::unique_lock<std::shared_timed_mutex> writerLock(sharedMutex);
std::shared_lock<std::shared_time_mutex> readerLock(sharedMutex);
std::shared_lock<std::shared_time_mutex> readerLock2(sharedMutex);
```

```
std::shared_lock<std::shared_time_mutex> readerLock2(sharedMutex);
```

The example presents the typical *reader-writer lock scenario*. The writerLock of type `std::unique_- lock <std::shared_timed_mutex>` can only exclusively have the `sharedMutex`. Both of the reader locks `readerLock` and `readerLock2` of type `std::shared_lock <std::shared_time_mutex>` can share the same mutex `sharedMutex`.

Thread-safe Initialization

If you don't modify the data, it's sufficient to initialize them in a thread-safe way. C++ offers various ways to achieve this: using a constant expression, using static variables with block scope and using the function `std::call_once` in combination with the flag `std::once::flag`.

Constant Expressions

A constant expression is initialized at compile time. Therefore they are per se thread-safe. By using the keyword `constexpr` before a variable, the variable becomes a constant expression. Instances of user-defined type can also be a constant expression and therefore be initialised in a thread-safe way if the methods are declared as constant expressions.

```
struct MyDouble{
  constexpr MyDouble(double v):val(v){};
  constexpr double getValue(){
    return val;
  }
private:
  double val
};
constexpr MyDouble myDouble(10.5);
std::cout << myDouble.getValue();
```

Static Variables Inside a Block

If you define a static variable in a block, the C++11 runtime guarantees that the variable is initialized in a thread-safe way.

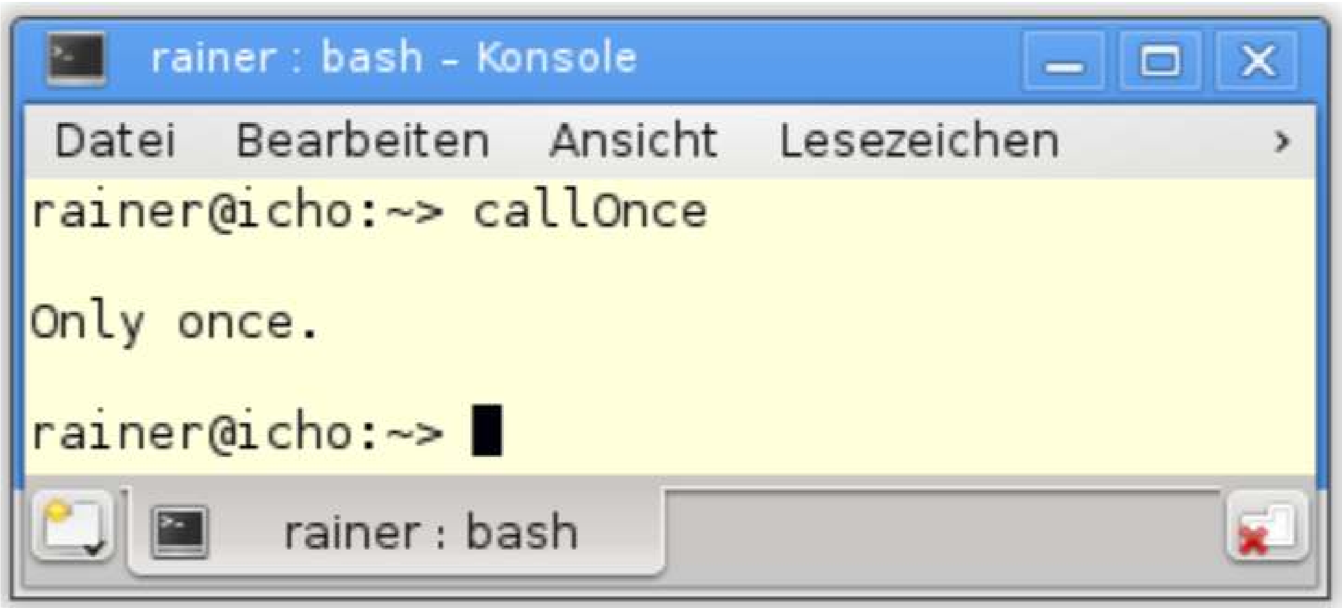
```
void blockScope(){
  static int MySharedDataInt= 2011;
}
```

std::call_once and std::once_flag

`std::call_once` takes two arguments: the flag `std::once_flag` and a callable. The C++ runtime guarantees with the help of the flag `std::once_flag` that the callable is executed exactly once.

```
//...
#include <mutex>
//...
using namespace std;
once_flag onceFlag;
void do_once(){
  call_once(onceFlag, []{ cout << "Only once." << endl; });
}
thread t1(do_once);
thread t2(do_once);
```

Although both threads executed the function `do_once` only one of them is successful, and the `lambda function []{cout << "Only once." << endl;}` is executed exactly once.



You can further use the same `std::once_flag` to register different callables and only one of this callables is called.