std::shared_mutex

Defined in header <shared_mutex>	
`class shared_mutex;`	(since C++17)

The `shared_mutex` class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a shared_mutex has two levels of access:

- *shared* - several threads can share ownership of the same mutex.
- *exclusive* - only one thread can own the mutex.

If one thread has acquired the *exclusive* lock (through `lock`, `try_lock`), no other threads can acquire the lock (including the *shared*).

If one thread has acquired the *shared* lock (through `lock_shared`, `try_lock_shared`), no other thread can acquire the *exclusive* lock, but can acquire the *shared* lock.

Only when the *exclusive* lock has not been acquired by any thread, the *shared* lock can be acquired by multiple threads.

Within one thread, only one lock (*shared* or *exclusive*) can be acquired at the same time.

Shared mutexes are especially useful when shared data can be safely read by any number of threads simultaneously, but a thread may only write the same data when no other thread is reading or writing at the same time.

The `shared_mutex` class satisfies all requirements of *SharedMutex* and *StandardLayoutType*.

Member types

Member type	Definition
`native_handle_type`(optional)	*implementation-defined*

Member functions

(constructor)	constructs the mutex (public member function)
(destructor)	destroys the mutex (public member function)
operator= [deleted]	not copy-assignable (public member function)

Exclusive locking

lock	locks the mutex, blocks if the mutex is not available (public member function)
try_lock	tries to lock the mutex, returns if the mutex is not available (public member function)
unlock	unlocks the mutex (public member function)

Shared locking

lock_shared	locks the mutex for shared ownership, blocks if the mutex is not available (public member function)
try_lock_shared	tries to lock the mutex for shared ownership, returns if the mutex is not available (public member function)
unlock_shared	unlocks the mutex (shared ownership) (public member function)

Native handle

native_handle	returns the underlying implementation-defined native handle object (public member function)

Example

Run this code

```
#include <iostream>
#include <mutex>  // For std::unique_lock
#include <shared_mutex>
#include <thread>

class ThreadSafeCounter {
 public:
  ThreadSafeCounter() = default;

  // Multiple threads/readers can read the counter's value at the same time.
  unsigned int get() const {
    std::shared_lock lock(mutex_);
    return value_;
  }

  // Only one thread/writer can increment/write the counter's value.
  void increment() {
    std::unique_lock lock(mutex_);
    value_++;
  }

  // Only one thread/writer can reset/write the counter's value.
  void reset() {
    std::unique_lock lock(mutex_);
    value_ = 0;
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};

int main() {
  ThreadSafeCounter counter;

  auto increment_and_print = [&counter]() {
    for (int i = 0; i < 3; i++) {
      counter.increment();
      std::cout << std::this_thread::get_id() << ' ' << counter.get() << '\n';

      // Note: Writing to std::cout actually needs to be synchronized as well
      // by another std::mutex. This has been omitted to keep the example small.
    }
  };

  std::thread thread1(increment_and_print);
  std::thread thread2(increment_and_print);

  thread1.join();
  thread2.join();
}

// Explanation: The output below was generated on a single-core machine. When
// thread1 starts, it enters the loop for the first time and calls increment()
// followed by get(). However, before it can print the returned value to
// std::cout, the scheduler puts thread1 to sleep and wakes up thread2, which
// obviously has time enough to run all three loop iterations at once. Back to
// thread1, still in the first loop iteration, it finally prints its local copy
// of the counter's value, which is 1, to std::cout and then runs the remaining
// two loop iterations. On a multi-core machine, none of the threads is put to
// sleep and the output is more likely to be in ascending order.
```

Possible output:

```
123084176803584 2
123084176803584 3
123084176803584 4
```

```
123084185655040 1
123084185655040 5
123084185655040 6
```

See also

`shared_timed_mutex` (C++14)	provides shared mutual exclusion facility and implements locking with a timeout (class)

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/thread/shared_mutex&oldid=118917"