

Introduction to Mutexes

This lesson gives an introduction to mutexes which are used in C++ for concurrency.

Mutex stands for **mutual exclusion**. It ensures that only one thread can access a **critical section** at any one time. By using a mutex, the mess of the workflow turns into a harmony.

```
// coutSynchronised.cpp

#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex coutMutex;

class Worker{
public:
    Worker(std::string n):name(n){};

    void operator() (){
        for (int i = 1; i <= 3; ++i){
            // begin work
            std::this_thread::sleep_for(std::chrono::milliseconds(200));
            // end work
            coutMutex.lock();
            std::cout << name << ": " << "Work " << i << " done !!!" << std::endl;
            coutMutex.unlock();
        }
    }
private:
    std::string name;
};

int main(){

    std::cout << std::endl;

    std::cout << "Boss: Let's start working." << "\n\n";

    std::thread herb= std::thread(Worker("Herb"));
    std::thread andrei= std::thread(Worker(" Andrei"));
    std::thread scott= std::thread(Worker(" Scott"));
    std::thread bjarne= std::thread(Worker(" Bjarne"));
    std::thread bart= std::thread(Worker(" Bart"));
    std::thread jenne= std::thread(Worker(" Jenne"));
```

```

herb.join();
andrei.join();
scott.join();

bjarne.join();
bart.join();
jenne.join();

std::cout << "\n" << "Boss: Let's go home." << std::endl;

std::cout << std::endl;

}

```



Essentially, when the lock is set on a mutex, no other thread can access the locked region of code. In other words, lines between `lock()` and `unlock()` can only be accessed by one thread at a time. `std::cout` is protected by the `coutMutex` in line 8. A simple `lock()` in line 19 and the corresponding `unlock()` call in line 21 ensure that the workers won't scream all at once.



`std::cout` is thread-safe

The C++11 standard guarantees that you must not protect `std::cout`. Each character will be written atomically. It is possible that more output statements like those in the example will interleave. This is only a visual issue; the program is well-defined. This remark is valid for all global stream objects. Insertion to and extraction from global stream objects (`std::cout`, `std::cin`, `std::cerr`, and, `std::clog``) is thread-safe.

Let's put it more formally: writing to `std::cout` is not a **data race**, but it's a **race condition** which means that the result depends on the interleaving of threads.

Different locking methods will be discussed in the next lesson.