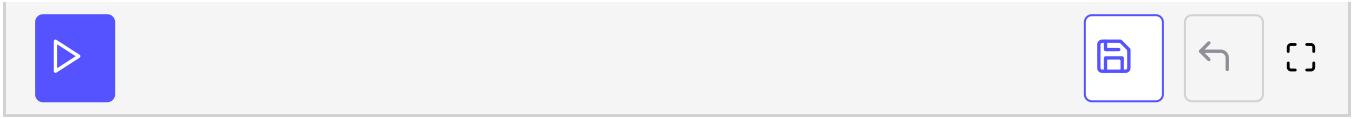
Thread-Safe Singleton: std::lock_guard

This lesson explains the solution for thread-safe initialization of a singleton problem using std::lock_guard in C++.

The mutex wrapped in an `std::lock_guard` guarantees that the singleton will be initialized in a thread-safe way.

```
// singletonLock.cpp

#include <chrono>
#include <iostream>
#include <future>
#include <mutex>

constexpr auto tenMill = 10000000;

std::mutex myMutex;

class MySingleton{
public:
  static MySingleton& getInstance(){
    std::lock_guard<std::mutex> myLock(myMutex);
    if (!instance){
        instance= new MySingleton();
    }
    volatile int dummy{};
    return *instance;
  }
private:
  MySingleton() = default;
  ~MySingleton() = default;
  MySingleton(const MySingleton&) = delete;
  MySingleton& operator=(const MySingleton&) = delete;

  static MySingleton* instance;
};


MySingleton* MySingleton::instance = nullptr;

int main(){

  constexpr auto fourtyMill = 4 * tenMill;

  const auto begin= std::chrono::system_clock::now();

  for ( size_t i = 0; i <= fourtyMill; ++i){
    MySingleton::getInstance();
  }
```

```
  const auto end = std::chrono::system_clock::now() - begin;

  std::cout << std::chrono::duration<double>(end).count() << std::endl;

}
```



You may have already guessed that this approach is the slowest one.

The next version of the thread-safe singleton pattern is also based on the multithreading library. It uses `std::call_once` in combination with the `std::once_flag`.