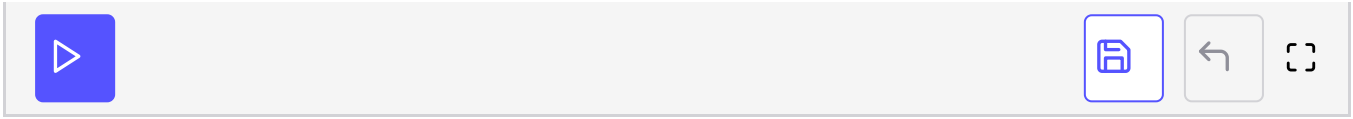
# Introduction to Thread-Safe Meyers Singleton

This lesson gives an introduction to a thread-safe version of Meyers singleton.

The C++11 standard guarantees that static variables with block scope will be initialized in a thread-safe way. The Meyers Singleton uses a static variable with block scope, so we are done. The only work that is left to do is to rewrite the previously used classical Meyers Singleton for the multithreading use-case.

```
// singletonMeyers.cpp

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

constexpr auto tenMill = 10000000;

class MySingleton{
public:
  static MySingleton& getInstance(){
    static MySingleton instance;
    volatile int dummy{};
    return instance;
  }
private:
  MySingleton() = default;
  ~MySingleton() = default;
  MySingleton(const MySingleton&) = delete;
  MySingleton& operator=(const MySingleton&) = delete;

};

std::chrono::duration<double> getTime(){

  auto begin = std::chrono::system_clock::now();
  for (size_t i = 0; i <= tenMill; ++i){
      MySingleton::getInstance();
  }
  return std::chrono::system_clock::now() - begin;

};

int main(){

    auto fut1= std::async(std::launch::async, getTime);
    auto fut2= std::async(std::launch::async, getTime);
    auto fut3= std::async(std::launch::async, getTime);
```

```
    auto fut4= std::async(std::launch::async, getTime);

    const auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();

    std::cout << total.count() << std::endl;

}
```



I use the singleton object in the function `getTime` (lines 24 - 32). The function is executed by the four promises in lines 36 - 39. The results of the associated futures are summed up in line 41. That's all.

> **I reduce the examples to the singleton implementation**
>
> The function `get_Time` for calculating the execution time and the `main` function will be identical; therefore, I will skip them in the remaining examples of this subsection.

Let's go for the most obvious one, and use a lock in the next lesson.