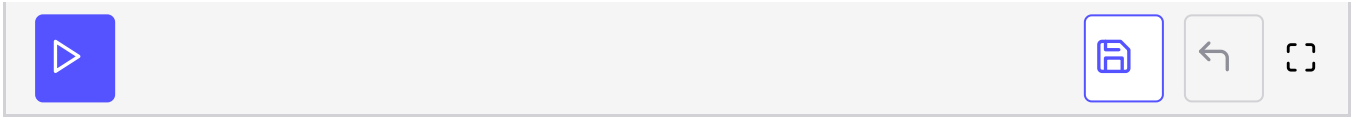
Thread-Safe Singleton: std::call_once with std::once_flag

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

You can use the function std::call_once together with the std::once_flag to register callables that will be executed exactly once in a thread-safe way.

```
// singletonCallOnce.cpp

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

constexpr auto tenMill = 10000000;

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

  static MySingleton* instance;
  static std::once_flag initInstanceFlag;

  static void initSingleton(){
    instance= new MySingleton;
  }
};

MySingleton* MySingleton::instance = nullptr;
std::once_flag MySingleton::initInstanceFlag;

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;

}
```



Let’s continue our thread-safe singleton implementation using atomics.