Double-Checked Locking Pattern

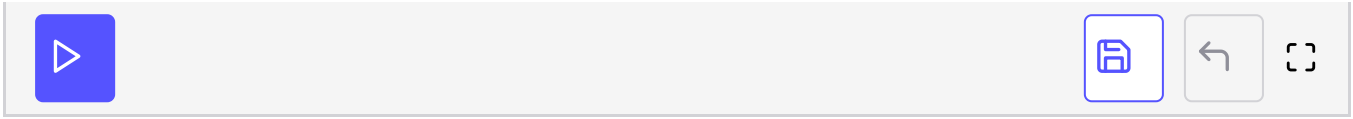
This lesson gives an overview of the double-checked locking pattern for the problem of the thread-safe initialization of a singleton in C++.

The double-checked locking pattern is the classic way to initialize a singleton in a thread-safe way. What sounds like established best practice - or a pattern - is more a kind of an anti-pattern. It assumes guarantees in the classical implementation, which aren't given by the Java, C#, or C++ memory model. The wrong assumption is that the creation of a singleton is an atomic operation; therefore, a solution that seems to be thread-safe is not thread-safe.

What is the double-checked locking pattern? The first idea for implementing a thread-safe singleton is to protect the initialization of the singleton with a lock.

```
#include <iostream>
#include <mutex>
#include <thread>

std::mutex myMutex;

class MySingleton{
public:
  static MySingleton* getInstance(){
    std::lock_guard<std::mutex> myLock(myMutex);
    if(!instance) instance = new MySingleton();
    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(){

  std::cout << std::endl;

  std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;
  std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;
```

```
  std::cout << std::endl;


}
```



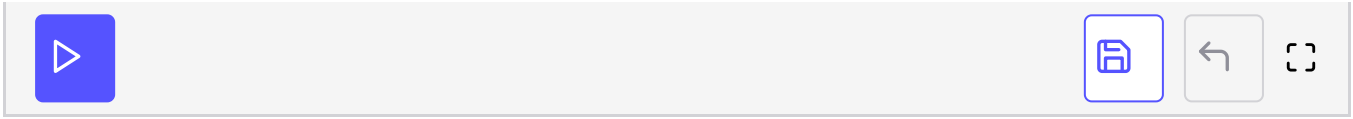
Any issues? Yes and no. Yes, because there is a large performance penalty; No, because the implementation is thread-safe. Each access to the singleton in **line 7** is protected by a heavyweight lock. This also applies to the read access, which is not necessary after the initial construction of `MySingleton`. With that, here comes the double-checked locking pattern to our rescue. Let's have a look at the `getInstance` function.

```
static MySingleton& getInstance(){
  if (!instance){                                  // check
    lock_guard<mutex> myLock(myMutex);             // lock
    if(!instance) instance = new MySingleton();    // check
  }
  return *instance;
}
```

Here's a running example of this extended code:

```
#include <iostream>
#include <mutex>
#include <thread>

std::mutex myMutex;

class MySingleton{
public:
  static MySingleton* getInstance(){
    if (!instance){                                          // check
      std::lock_guard<std::mutex> myLock(myMutex);           // lock
      if(!instance) instance = new MySingleton();            // check
    }
    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(){
```

```
  std::cout << std::endl;

  std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;
  std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;

  std::cout << std::endl;

}
```



Instead of the heavyweight lock, I use a lightweight pointer comparison in line 10. If I get a null pointer, I apply the heavyweight lock on the singleton (line 11). Because there is the possibility that another thread will initialize the singleton between the pointer comparison in line 10 and the lock call in line 11, I have to perform an additional pointer comparison in line 12. So the name is obvious: two times a check and one time a lock.

Smart? Yes. Thread-safe? No.

What is the issue? The call `instance= new MySingleton()` in line 12 consists of at least three steps.

1. Allocate memory for `MySingleton`
2. Initialise the `MySingleton` object
3. Let `instance` refer to the fully initialized `MySingleton` object

The issue is that the C++ runtime provides no guarantee that the steps will be performed in that sequence. For example, it is possible that the processor may reorder the steps to the sequence 1,3, and 2. So in the first step, the memory will be allocated, and in the second step `instance` refers to a non-initialised singleton. If just at that moment another thread `t2` tries to access the singleton and makes the pointer comparison, the comparison will succeed. The consequence is that thread `t2` refers to a non-initialised singleton and the program behavior is undefined.