Thread-Safe Singleton: Atomics

This lesson explains the solution for the thread-safe initialization of singleton problem using atomics in C++.

WE'LL COVER THE FOLLOWING

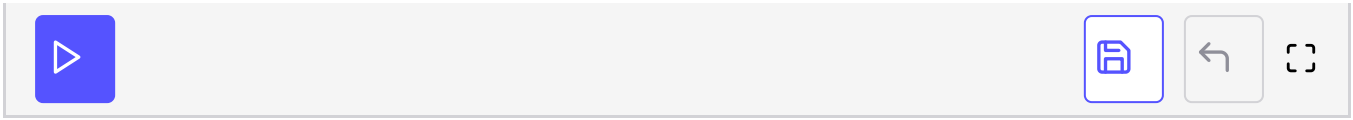
With atomic variables, my implementation becomes a lot more challenging; I can even specify the memory model for my atomic operations. The following two implementations of the thread-safe singletons are based on the previously mentioned double-checked locking pattern.

Sequential Consistency

In my first implementation, I use atomic operations without specifying the memory model; therefore, sequential consistency applies.

```
// singletonSequentialConsistency.cpp

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

constexpr auto tenMill = 10000000;

class MySingleton{
public:
  static MySingleton* getInstance(){
    MySingleton* sin = instance.load();
    if (!sin){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin = instance.load(std::memory_order_relaxed);
      if(!sin){
        sin= new MySingleton();
        instance.store(sin);
```

```
      }
    }
    volatile int dummy{};

    return sin;
  }
private:
  MySingleton() = default;
  ~MySingleton() = default;
  MySingleton(const MySingleton&) = delete;
  MySingleton& operator=(const MySingleton&) = delete;

  static std::atomic<MySingleton*> instance;
  static std::mutex myMutex;
};


std::atomic<MySingleton*> MySingleton::instance;
std::mutex MySingleton::myMutex;

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;

}
```



In contrast to the double-checked locking pattern, I now have the guarantee that the expression `sin= new MySingleton()` in line 19 will happen before the store expression `instance.store(sin)` in line 20. This is due to using sequential consistency as the default memory model for atomic operations. Have a look at line 17:

`sin = instance.load(std::memory_order_relaxed)`.

This additional load is necessary because, between the first load in line 14 and the usage of the lock in line 16, another thread may kick in and change the value of `instance`. That being said, we can optimize the program even more.

Acquire-Release Semantic

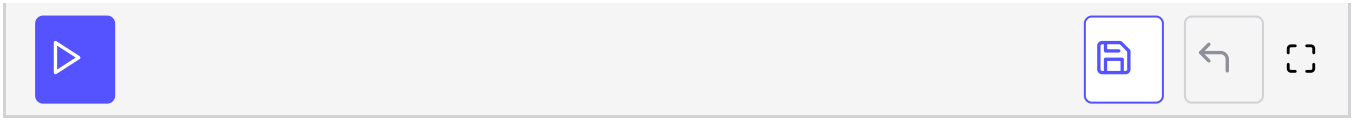
Acquire-Release Semantic

Let's have a closer look at the previous thread-safe implementation of the singleton pattern using atomics. The loading (or reading) of the singleton in line 14 is an acquire operation, the storing (or writing) in line 20 is a release operation. Both operations take place on the same atomic, therefore sequential consistency is overkill. The C++11 standard guarantees that an acquire operation synchronizes with a release operation on the same atomic and establishes an ordering constraint. This means that all subsequent read and write operations cannot be moved before an acquire operation, and all read and write operations cannot be moved after a release operation.

These are the minimal guarantees required to implement a thread-safe singleton.

```
// singletonAcquireRelease.cpp

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

constexpr auto tenMill = 10000000;

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

  static std::atomic<MySingleton*> instance;
  static std::mutex myMutex;
};


std::atomic<MySingleton*> MySingleton::instance;
```

```
std::mutex MySingleton::myMutex;

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;

}
```



The acquire-release semantic has a similar performance to the sequential consistency. This is not surprising because both memory models are very similar in the x86 architecture. We would probably get a greater difference in the performance numbers on the ARMv7 or PowerPC architecture. You can read the details on *Jeff Preshings* blog Preshing on Programming.