Thread-Safe Initialization of a Singleton

30 August 2016

There are a lot of issues with the singleton pattern. I'm totally aware of that. But the singleton pattern is an ideal use case for a variable, which has only to be initialized in a thread-safe way. From that point on you can use it without synchronization. So in this post, I discuss different ways to initialize a singleton in a multithreading environment. You get the performance numbers and can reason about your uses cases for the thread-safe initialization of a variable.

There are a lot of different ways to initialize a singleton in C++11 in a thread-safe way. From a birds-eye, you can have guarantees from the C++ runtime, locks or atomics. I'm totally curious about the performance implications.

My strategy

I use as a reference point for my performance measurement a singleton object which I sequential access 40 million times. The first access will initialize the object. In contrast, the access from the multithreading program will be done by 4 threads. Here I'm only interested in the performance. The program will run on two real PCs. My Linux PC has four, my Windows PC has two cores. I compile the program with maximum and without optimization. For the translation of the program with maximum optimization, I have to use a `volatile` variable in the static method `getInstance`. If not the compiler will optimize away my access to the singleton and my program becomes too fast.

I have three questions in my mind:

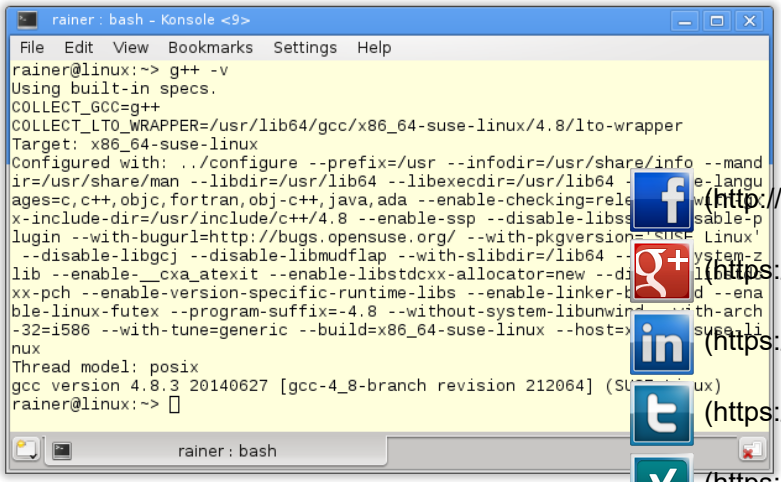
1. How is the relative performance of the different singleton implementations?
2. Is there a significant difference between Linux (gcc) and Windwos (cl.exe)?
3. What's the difference between the optimized and non-optimized versions?

Finally, I collect all numbers in a table. The numbers are in seconds.

The reference values

The both compilers

The command line gives you the details of the compiler Here are the gcc and the cl.exe.



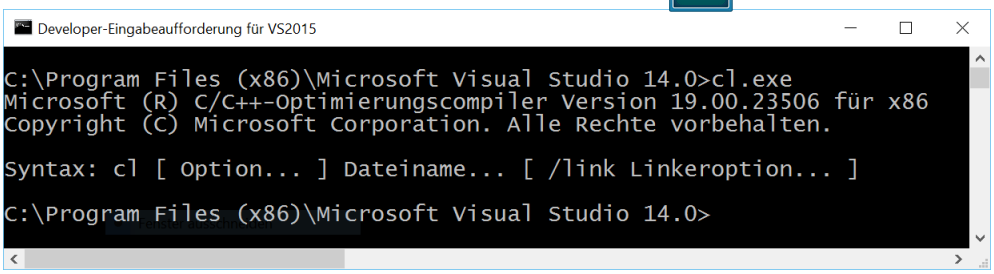
(http://www.facebook.com/rainer.grimm.31)

(https://plus.google.com/109576694736160795401?pr

(https://www.linkedin.com/in/rainergrimm)

(https://twitter.com/rainer_grimm)


(https://www.xing.com/profile/Rainer_Grimm12)



The reference code

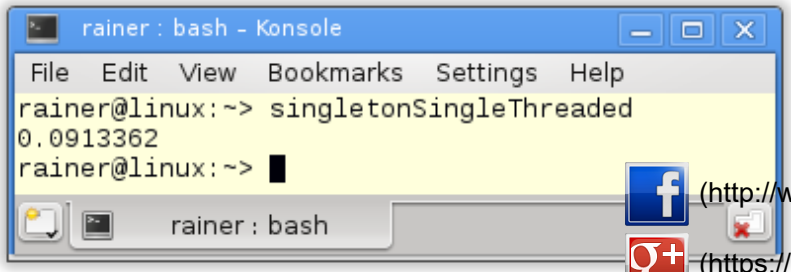
At first, the single-threaded case. Of course without synchronization.

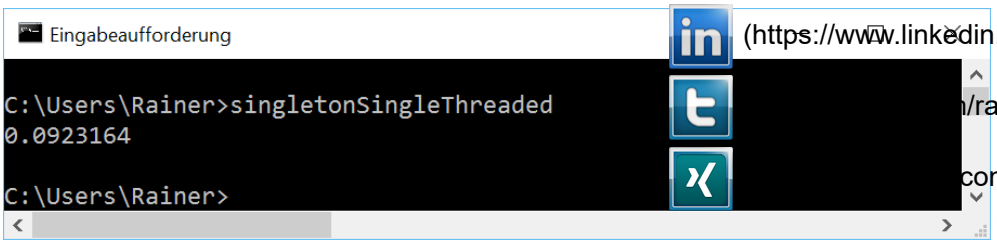
```
1  // singletonSingleThreaded.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  constexpr auto tenMill= 10000000;
7
8  class MySingleton{
9  public:
10   static MySingleton& getInstance(){
11     static MySingleton instance;
12     // volatile int dummy{};
13     return instance;
14   }
15 private:
16   MySingleton()= default;
17   ~MySingleton()= default;
18   MySingleton(const MySingleton&)= delete;
19   MySingleton& operator=(const MySingleton&)= delete;
20
21 };
22
23 int main(){
24
25   constexpr auto fourtyMill= 4* tenMill;
26
27   auto begin= std::chrono::system_clock::now();
28
29   for ( size_t i= 0; i <= fourtyMill; ++i){
30        MySingleton::getInstance();
31   }
32
33   auto end= std::chrono::system_clock::now() - begin;
34
35   std::cout << std::chrono::duration<double>(end).count() << std::endl;
36
37 }
```


I use in the reference implementation the so-called Meyers Singleton. The elegance of this implementation is that the singleton object `instance` in line 11 is a static variable with a block scope. Therefore, `instance` will exactly be initialized, when the static method `getInstance` (line 10 - 14) will be executed the first time. In line 14 the `volatile` variable `dummy` is commented out. When I translate the program with maximum optimization that has to change. So the call `MySingleton::getInstance()` will not be optimized away.

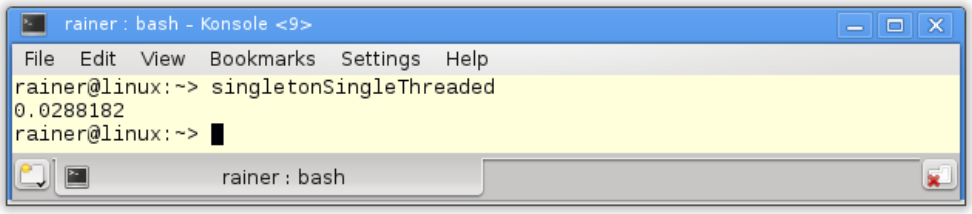
Now the raw numbers on Linux and Windows.

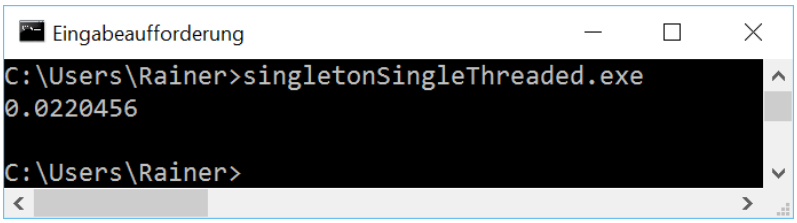
Without optimization





Maximum Optimization





Guarantees of the C++ runtime

I already presented the details to the thread-safe initialization of variables in the post Thread-safe initialization of data. (https://www.modernescpp.com/index.php/thread-safe-initialization-of-data)

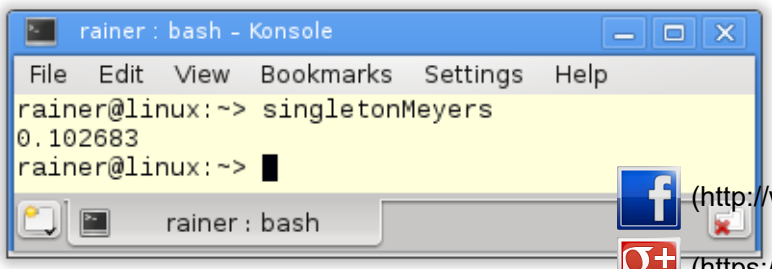
Meyers Singleton

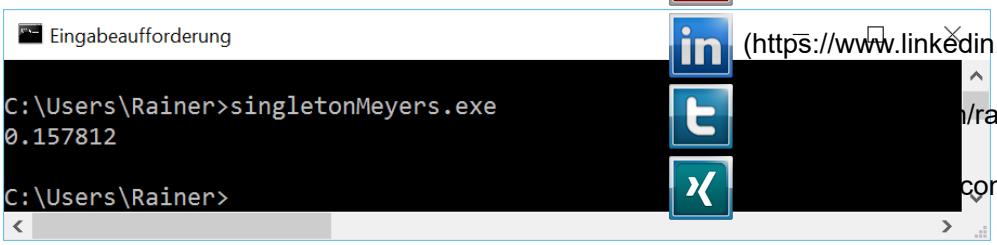
The beauty of the Meyers Singleton in C++11 is that it's automatically thread-safe. That is guaranteed by the standard: Static variables with block scope. (https://www.modernescpp.com/index.php/thread-safe-initialization-of-data)The Meyers Singleton is a static variable with block scope, so we are done. It's still left to rewrite the program for four threads.

```
1  // singletonMeyers.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <future>
6
7  constexpr auto tenMill= 10000000;
8
9  class MySingleton{
10 public:
11   static MySingleton& getInstance(){
12     static MySingleton instance;
13     // volatile int dummy{};
14     return instance;
15   }
16 private:
17   MySingleton()= default;
18   ~MySingleton()= default;
19   MySingleton(const MySingleton&)= delete;
20   MySingleton& operator=(const MySingleton&)= delete;
21
22 };
23
24 std::chrono::duration<double> getTime(){
25
26   auto begin= std::chrono::system_clock::now();
27   for ( size_t i= 0; i <= tenMill; ++i){
28       MySingleton::getInstance();
29   }
30   return std::chrono::system_clock::now() - begin;
31
32 };
33
34 int main(){
35
36     auto fut1= std::async(std::launch::async,getTime);
37     auto fut2= std::async(std::launch::async,getTime);
38     auto fut3= std::async(std::launch::async,getTime);
39     auto fut4= std::async(std::launch::async,getTime);
40
41     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
42
43     std::cout << total.count() << std::endl;
44
45 }
```

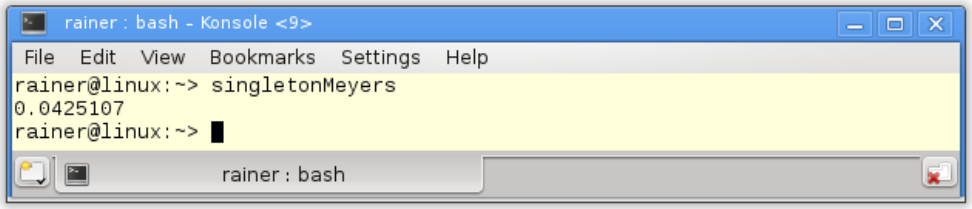

I use the singleton object in the function `getTime` (line 24 - 32). The function is executed by the four `promise` (https://www.modernescpp.com/index.php/asynchronous-function-calls) in line 36 - 39. The results of the associate `futures` (https://www.modernescpp.com/index.php/asynchronous-function-calls) are summed up in line 41. That's all. Only the execution time is missing.

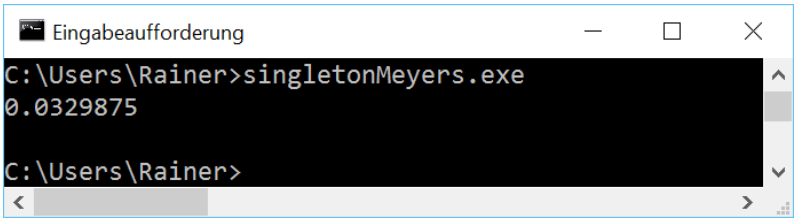
Without optimization





Maximum optimization





The next step is the function `std::call_once` in combination with the flag `std::once_flag`.

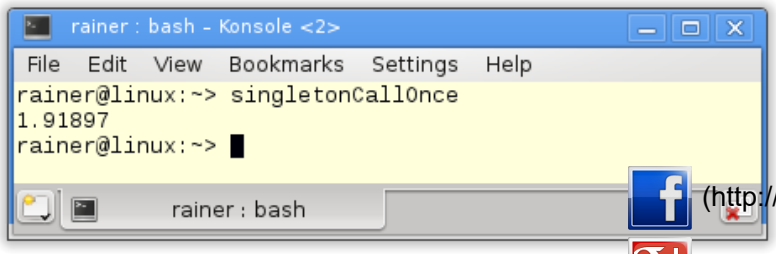
The function std::call_once and the flag std::once_flag

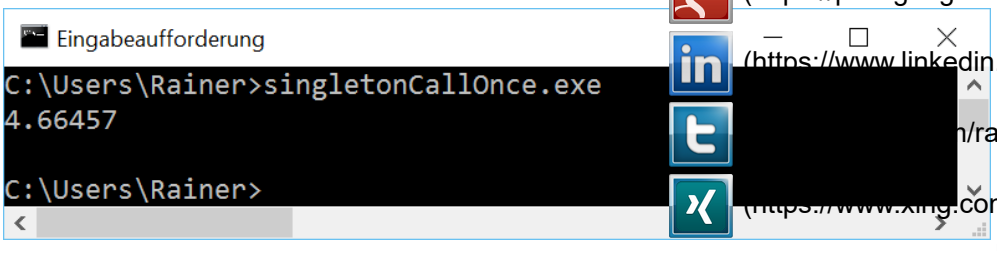
You can use the function `std::call_once (https://www.modernescpp.com/index.php/thread-safe-initialization-of-data)` to register a callable which will be executed exactly once. The flag `std::call_once` in the following implementation guarantees that the singleton will be thread-safe initialized.

```
// 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;

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);

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

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

}
```


Here are the numbers.

Without optimization





(http://www.facebook.com/rainer.grimm.31)

(https://plus.google.com/109576694736160795401?pr

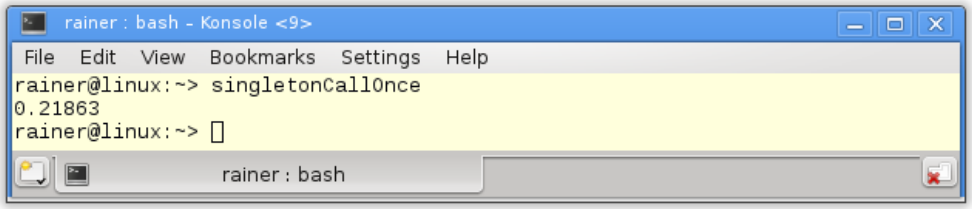
(https://www.linkedin.com/in/rainergrimm)

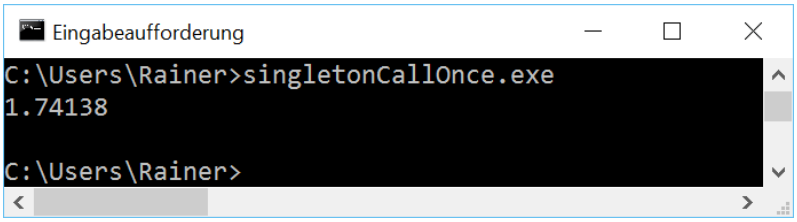
n/rainer_grimm)

(https://www.xing.com/profile/Rainer_Grimm12)

Hunting (http://www.hunterspeak.com/)

Maximum optimization





Of course, the most obvious way is it protects the singleton with a lock.

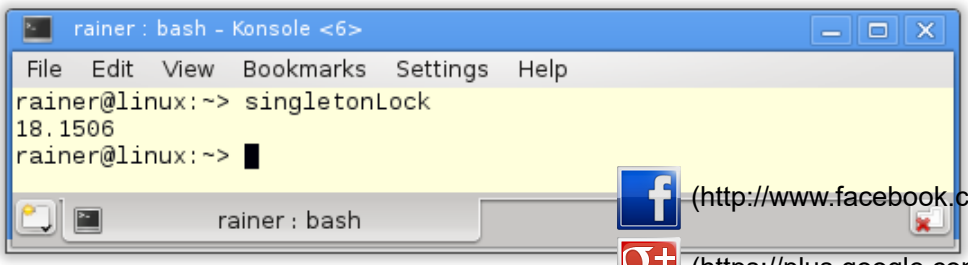
Lock

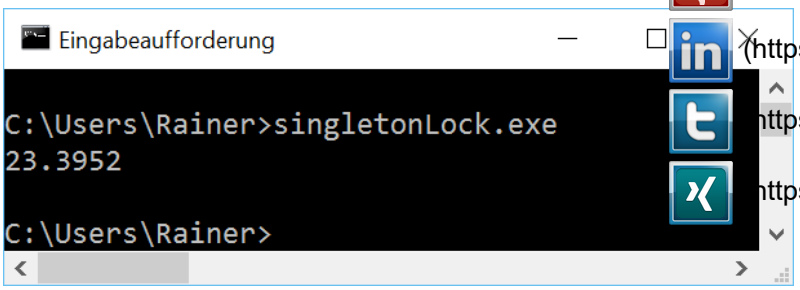
The mutex wrapped in a lock (https://www.modernescpp.com/index.php/prefer-locks-to-mutexes) guarantees that the singleton will be thread-safe initialized.

```
// 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;

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);

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

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

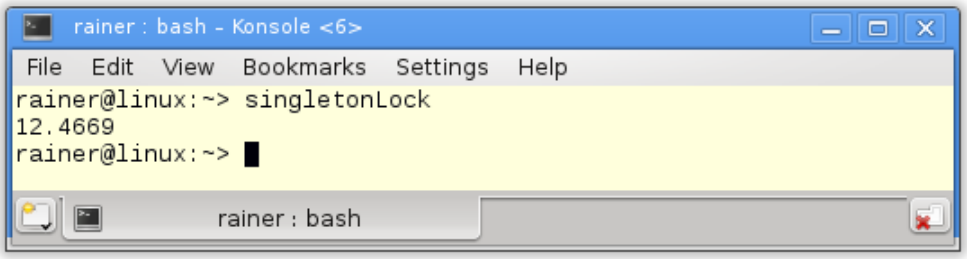

How fast is the classical thread-safe implementation of the singleton pattern?

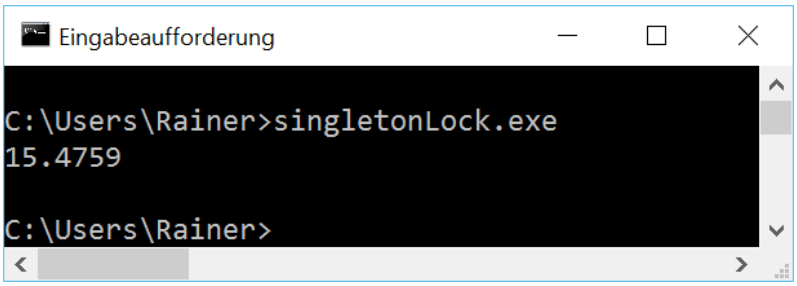
Without optimization





Maximum optimization





Not so fast. Atomics should make the difference.

Atomic variables

With atomic variables, my job becomes extremely challenging. Now I have to use the C++ memory model (https://www.modernescpp.com/index.php/c-memory-model). I base my implementation on the well-known double-checked locking pattern. (https://www.modernescpp.com/index.php/thread-safe-initialization-of-data)

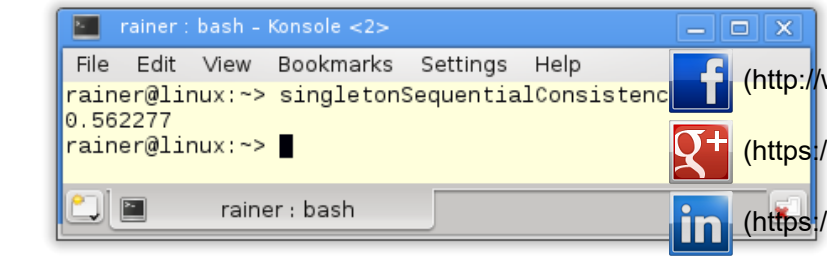
Sequential consistency

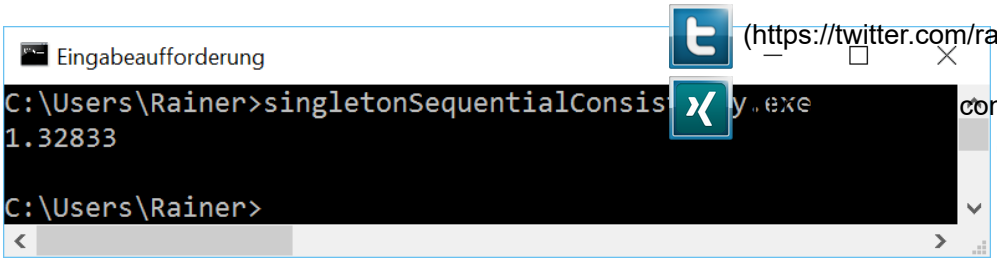
The handle to the singleton is atomic. Because I didn't specify the C++ memory model the default applies: Sequential consistency. (https://www.modernescpp.com/index.php/sequential-consistency)

```
1  // singletonAcquireRelease.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <future>
6  #include <mutex>
7  #include <thread>
8
9  constexpr auto tenMill= 10000000;
10
11 class MySingleton{
12 public:
13   static MySingleton* getInstance(){
14     MySingleton* sin= instance.load();
15     if ( !sin ){
16       std::lock_guard<std::mutex> myLock(myMutex);
17       sin= instance.load();
18       if( !sin ){
19         sin= new MySingleton();
20         instance.store(sin);
21       }
22     }
23     // volatile int dummy{};
24     return sin;
25   }
26 private:
27   MySingleton()= default;
28   ~MySingleton()= default;
29   MySingleton(const MySingleton&)= delete;
30   MySingleton& operator=(const MySingleton&)= delete;
31
32   static std::atomic<MySingleton*> instance;
33   static std::mutex myMutex;
34 };
35
36
37 std::atomic<MySingleton*> MySingleton::instance;
38 std::mutex MySingleton::myMutex;
39
40 std::chrono::duration<double> getTime(){
41
42   auto begin= std::chrono::system_clock::now();
43   for ( size_t i= 0; i <= tenMill; ++i){
44        MySingleton::getInstance();
45   }
46   return std::chrono::system_clock::now() - begin;
47
48 };
49
50
51 int main(){
52
53     auto fut1= std::async(std::launch::async,getTime);
54     auto fut2= std::async(std::launch::async,getTime);
55     auto fut3= std::async(std::launch::async,getTime);
56     auto fut4= std::async(std::launch::async,getTime);
57
58     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
59
60     std::cout << total.count() << std::endl;
61
62 }
```

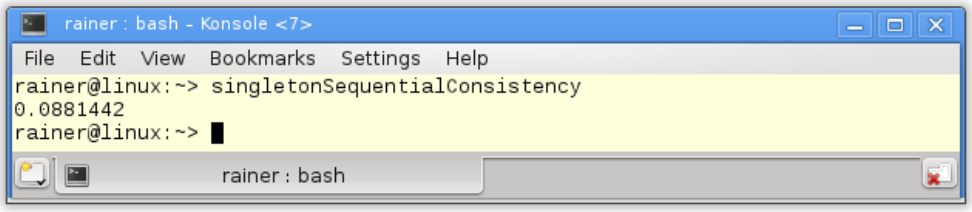
Now I'm curious.

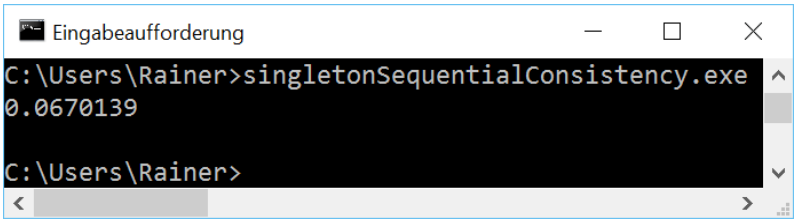
Without optimization





Maximum optimization





But we can do better. There is an additional optimization possibility.

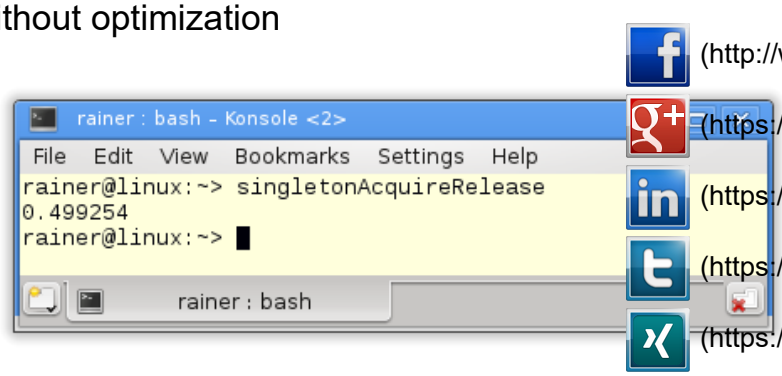
Acquire-release Semantic

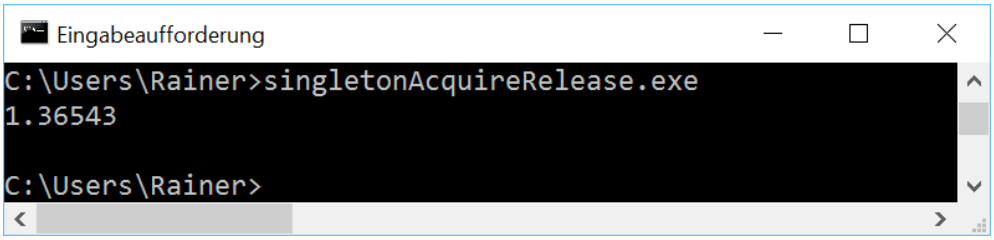
The reading of the singleton (line 14) is an acquire operation, the writing a release operation (line 20). Because both operations take place on the same atomic I don't need sequential consistency. The C++ standard guarantees that an acquire operation synchronizes with a release operation on the same atomic. These conditions hold in this case therefore I can weaken the C++ memory model in line 14 and 20. Acquire-release semantic (https://www.modernescpp.com/index.php/acquire-release-semantic)is sufficient.

```
1  // singletonAcquireRelease.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <future>
6  #include <mutex>
7  #include <thread>
8
9  constexpr auto tenMill= 10000000;
10
11 class MySingleton{
12 public:
13   static MySingleton* getInstance(){
14     MySingleton* sin= instance.load(std::memory_order_acquire);
15     if ( !sin ){
16       std::lock_guard<std::mutex> myLock(myMutex);
17       sin= instance.load(std::memory_order_relaxed);
18       if( !sin ){
19         sin= new MySingleton();
20         instance.store(sin,std::memory_order_release);
21       }
22     }
23     // volatile int dummy{};
24     return sin;
25   }
26 private:
27   MySingleton()= default;
28   ~MySingleton()= default;
29   MySingleton(const MySingleton&)= delete;
30   MySingleton& operator=(const MySingleton&)= delete;
31
32   static std::atomic<MySingleton*> instance;
33   static std::mutex myMutex;
34 };
35
36
37 std::atomic<MySingleton*> MySingleton::instance;
38 std::mutex MySingleton::myMutex;
39
40 std::chrono::duration<double> getTime(){
41
42   auto begin= std::chrono::system_clock::now();
43   for ( size_t i= 0; i <= tenMill; ++i){
44        MySingleton::getInstance();
45   }
46   return std::chrono::system_clock::now() - begin;
47
48 };
49
50
51 int main(){
52
53     auto fut1= std::async(std::launch::async,getTime);
54     auto fut2= std::async(std::launch::async,getTime);
55     auto fut3= std::async(std::launch::async,getTime);
56     auto fut4= std::async(std::launch::async,getTime);
57
58     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
59
60     std::cout << total.count() << std::endl;
61
62 }
```

The acquire-release semantic has a similar performance as the sequential consistency. That's not surprising, because on x86 both memory models are very similar. We would get totally different numbers on an ARMv7 or PowerPC architecture. You can read the details on Jeff Preshings blog Preshing on Programming (http://preshing.com/).

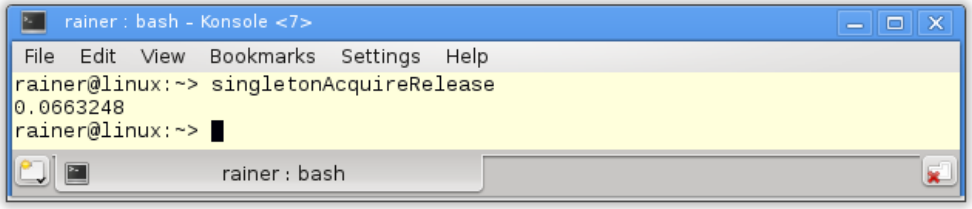
Without optimization

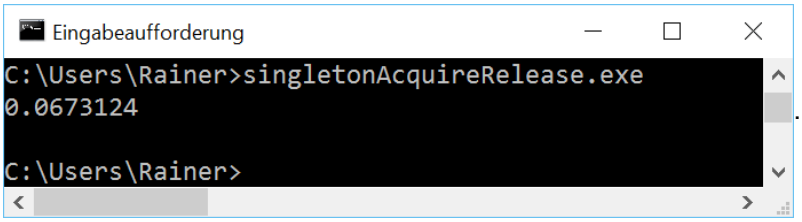




Hunting (http://www.huntersspeak.com/)

Maximum optimization





If I forget an import variant of the thread-safe singleton pattern, please let me know and send me the code. I will measure it and add the numbers to the comparison.

All numbers at one glance

Don't take the numbers too seriously. I executed each program only once and the executable is optimized for four cores on my two core windows PC. But the numbers give a clear indication. The Meyers Singleton is the easiest to get and the fastest one. In particular, the lock-based implementation is by far the slowest one. The numbers are independent of the used platform.

But the numbers show more. Optimization counts. This statement holds not totally true for the `std::lock_guard` based implementation of the singleton pattern.

Operating system (Compiler)	Optimization	Single threaded	Meyers Singleton	std::call_once	std::lock_guard	Sequential consistency	Acquire-release semantic
Linux (GCC)	no	0.09	0.10	1.92	18.15	0.56	0.50
Linux (GCC)	yes	0.03	0.04	0.22	12.47	0.09	0.07
Window (cl.exe)	no	0.09	0.16	4.66	23.40	1.33	1.37
Windows (cl.exe)	yes	0.02	0.03	1.78	15.48	0.07	0.07

What's next?

I'm not so sure. This post is a translation of a german post I wrote half a year ago. My German post gets a lot of reaction. I'm not sure, what will happen this time. A few days letter I'm sure. The next post (https://www.modernescpp.com/index.php/single-threaded-sum-of-the-elements-of-a-vector) will be about the addition of the elements of a vector. First, it takes in one thread.

Thanks a lot to my Patreon Supporters (https://www.patreon.com/rainer_grimm): Matt Braun, Roman Postanciuc, Tobias Zindl, Marko, G Prvulovic, Reinhold Dröge, Abernitzke, Frank Grimm, Sakib, Broeserl, António Pina, Sergey Agafyin, Андрей Бурмистров, Jake, GS, Lawton Shoemake, Animus24, Jozo Leko, John Breland, espkk, Wolfgang Gärtner, Louis St-Amour, Venkat Nandam, Jose Francisco, Douglas Tinkham, Kuchlong Kuchlong, Robert Blanch, Truels Wissneth, Kris Kafka, Mario Luoni, Neil Wang, Friedrich Huber, lennonli, Pramod Tikare Muralidhara, Peter Ware, Tobi Heideman, Daniel Hufschläger, Red Trip, Alexander Schwarz, Tornike Porchxidze, Alessandro Pezzato, Evangelos Denaxas, Bob Perry, and Satish Vangipuram.

Thanks in particular to Jon Hess, Lakshman, Christian Wittenhorst, Sherhy Pyton, Dendi Suhubdy, Sudhakar Belagurusamy, Richard Sargeant, and Rusty Fleming.

My special thanks to Embarcadero (https://www.embarcadero.com/de/products/cbuilder)

(https://www.embarcadero.com/products/cbuilder)

Seminars

I'm happy to give online-seminars or face-to-face seminars world-wide. Please call me if you have any questions.

Bookable (Online)

German

- Clean Code mit modernem C++: (https://www.modernescpp.de/index.php/c/2-c/31-clean-code-mit-modernem-c)22.06.2021 - 24.06.2021
- C++20: (https://www.modernescpp.de/index.php/c/2-c/32-c-20) 10.08.2021 - 12.08.2021
- Embedded Programmierung mit modernem C++: (https://www.modernescpp.de/index.php/c/2-c/33-embedded-programmierung-mit-modernem-c20) (https://www.facebook.com/rainer.grimm.31)

Standard Seminars (English/German)

Here is a compilation of my standard seminars. These seminars are only meant to give you a first orientation.

- C++ - The Core Language (https://www.modernescpp.net/index.php/c/plan/2-c/22)
- C++ - The Standard Library (https://www.modernescpp.net/index.php/c/plan/2-c/23)
- C++ - Compact (https://www.modernescpp.net/index.php/c/plan/2-c/23)
- C++11 and C++14 (https://www.modernescpp.net/index.php/c/plan/2-c/18)
- Concurrency with Modern C++ (https://www.modernescpp.net/index.php/c/plan/2-c/19)
- Design Patterns and Architecture Patterns with C++ (https://www.modernescpp.net/index.php/c/plan/2-c/21)
- Embedded Programming with Modern C++ (https://www.modernescpp.net/index.php/c/plan/2-c/17)
- Generic Programming (Templates) with C++ (https://www.modernescpp.net/index.php/c/plan/2-c/17)

New

- Clean Code with Modern C++ (https://www.modernescpp.net/index.php/c/plan/2-c/16)
- C++20 (https://www.modernescpp.net/index.php/c/plan/2-c/25)

Contact Me

- Phone: +49 152 31965939
- Mail: schulung@ModernesCpp.de (mailto:schulung@ModernesCpp.de) (https://www.modernescpp.com/<a href=)
- German Seminar Page: www.ModernesCpp.de (https://www.modernescpp.de/)
- English Seminar Page: www.ModernesCpp.net (http://www.ModernesCpp.net)

Modernes C++,

Tweet (http://twitter.com/share) Like 4 Share

Tags: atomics (/index.php/tag/atomics), sequential consistency (/index.php/tag/sequential-consistency), lock (/index.php/tag/lock), mutex (/index.php/tag/mutex), static (/index.php/tag/static), acquire-release semantic (/index.php/tag/acquire-release-semantik), singleton (/index.php/tag/singleton)

Comments (/index.php/component/jcomments/feed/com_jaggyblog/197)

#1 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-372) **0**
Anton 2016-08-30 15:04
why you not measured this method
http://ideone.com/8wePDz

i know what writing and reading of volatile, can be not atomic, but for aligned register-size data it looks like atomic.

Quote

#2 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-374) **0**
Rainer Grimm 2016-08-30 18:37

I hope I get your point.
You proposing using volatile as a kind of atomi?. If so volatile has no multithreading semantic. Maybe the compiler guarantees it. With volatile we are in the area of the broken double-checked locking pattern. To measure undefined behaviours makes no sense from my perspective.

Quote

#3 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-376) **0**
Anton 2016-08-31 11:35
Thank for your answer. I just checked asm listing of your Acquire-release Semantic, and it doesn't contain lock prefixes or fenses (like in version with volatile).

Quote

#4 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-648) **0**
nikitablack 2016-10-11 07:16
Hi. Can you please explain Acquire-release Semantic in a more detail?

Why do you need a mutex on line 16? I know the problem with double check locking but I thought with atomics it's not the case so no need for the mutex.

Is it possible that relaxed read (line 17) move above mutex lock (line 16)? In that case this approach is also broken. Or maybe a mutex lock guarantees a happens-before relationship?

Quote

#5 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-651) **0**
Rainer Grimm 2016-10-11 16:20

Quoting nikitablack:

> Hi. Can you please explain Acquire-release Semantic in a more detail?
>
> Why do you need a mutex on line 16? I know the problem with double check locking but I thought with atomics it's not the case so no need for the mutex.
>
> Is it possible that relaxed read (line 17) move above mutex lock (line 16)? In that case this approach is also broken. Or maybe a mutex lock guarantees a happens-before relationship?

You have no guarantee that the line 19 (sin= new MySingleton()) is an atomic operation. Therefore you can have a half initialized singleton.
The lock establishes a kind of a barrier. Nothing from inside can cross the border. => See the details in the post: Acquire-release semantic

Quote

#6 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-4723) **+2**
centos_linux 2017-04-25 13:20
Your style is very unique in comparison to other people I've read stuff from.
I appreciate you for posting when you have the opportunity, Guess I will just book mark this blog.

Quote

#7 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-11850) **0**
Julio 2018-02-21 04:10
Just desire to say your article iss as astounding. Thhe clarity to your submit
is just nice aand that i ccan assume you're knowledgeable in this subject.
Well with your permission let me tto take
hod off your RSS feed too stay up to datee with
imminent post. Thankk yoou one million and please keep up the gratifying work.

Quote

#8 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-38414) **0**
Vivek Lala 2019-03-16 05:46
please share ebook for c++

Quote

#9 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-77833) **0**
Opher 2020-05-10 08:17
Thanks so much for the article!
I'd like to second @Julio - You've obviously invested a lot into it.

Quote

#10 (/index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-77952) **Karel** 2021-02-05 07:38 **0**

Thanks for the arcticle. If you write a output message of the MySingleton constructor, it will be more descriptive. It will show thread-safe clearly.

Quote

Refresh comments list

RSS feed for comments to this post (/index.php/component/jcomments/feed/com_jaggyblog/197)

Add comment

(http://www.facebook.com/rainer.grimm.31)

(https://plus.google.com/109576694736160795401?prs

(https://www.linkedin.com/in/rainergrimm)

(https://twitter.com/rainer_grimm)

(https://www.xing.com/profile/Rainer_Grimm12)

(https://www.patreon.com/rainer_grimm)

(https://www.modernescpp.net/)

Subscribe to the newsletter (+ pdf bundle)

Name

Email

Subscribe

>> Start here <<

(/index.php/der-einstieg-in-modernes-c)

>> Visual Tour <<

(https://youtu.be/hrXoVSi0O28)

Contact

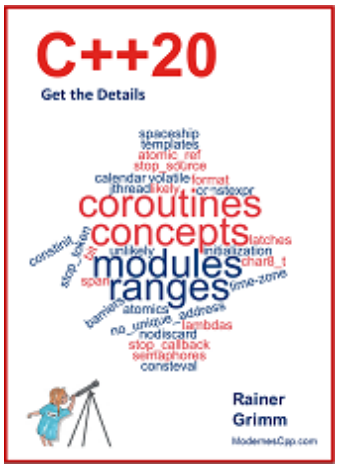
rainer@grimm-jaud.de (/index.php/rainer-grimm)

Impressum (/index.php/impressum)

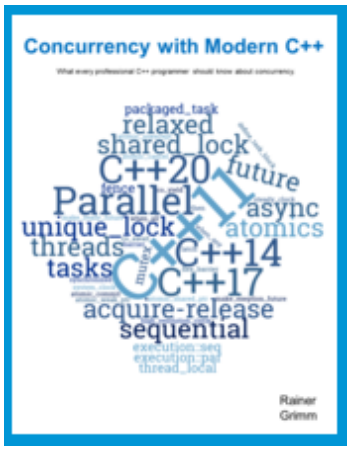
My Newest E-Books

Categories

- C++17 (/index.php/category/c-17)
- C++20 (/index.php/category/c-20)
- C++ Core Guidelines (/index.php/category/modern-c)
- C++ Insights (/index.php/category/cppinsight)
- Embedded (/index.php/category/embedded)
- Functional (/index.php/category/functional)
- Multithreading (/index.php/category/multithreading)
- Multithreading - Application (/index.php/category/multithreading-application)



(https://leanpub.com/c20)



(https://leanpub.com/concurrencywithmodernc)

(https://leanpub.com/cpplibrary)

Course: Modern C++ Concurrency in Practice

(https://www.educative.io/courses/modern-cpp-

- Multithreading - C++17 and C++20 (/index.php/category/multithreading-c-17-and-c-20)
- Multithreading - Memory Model (/index.php/category/multithreading-memory-model)
- News (/index.php/category/news)
- Overview (/index.php/category/uberblick)
- Pdf bundles (/index.php/category/pdf-bundle)
- Review (/index.php/category/review)
- Templates (/index.php/category/templates)

(http://www.facebook.com/rainer.grimm.31)
(https://plus.google.com/109357669473616079540?prs
(https://www.linkedin.com/in/rainergrimm)
(https://twitter.com/rainer_grimm)
(https://www.xing.com/profile/Rainer_Grimm12)

All tags

Hunting (http://www.hunterspeak.com/)

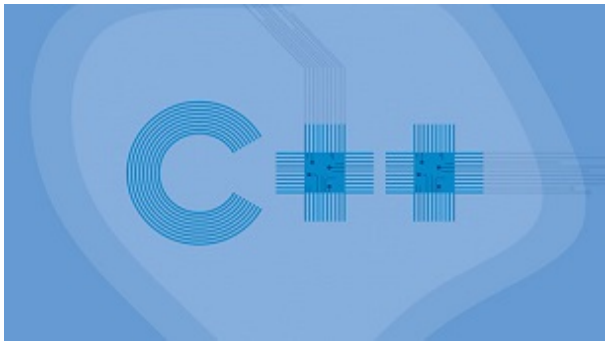
acquire-release semantic (/index.php/tag/acquire-release-semantik) arithmetic (/index.php/tag/arithmetic) associative containers (/index.php/tag/hashtabellen) async (/index.php/tag/async) atomics (/index.php/tag/atomics) atomic_thread_fence (/index.php/tag/atomic-thread-fence) auto (/index.php/tag/auto) bit manipulation (/index.php/tag/bit-manipulation) C (/index.php/tag/c) C++17 (/index.php/tag/c-17) C++20 (/index.php/tag/c-20) class hierarchies (/index.php/tag/class-hierarchies) classes (/index.php/tag/classes) concepts (/index.php/tag/concepts) condition variables (/index.php/tag/condition-variable) constexpr (/index.php/tag/constexpr) contracts (/index.php/tag/contracts) conversions (/index.php/tag/conversions) coroutines (/index.php/tag/coroutines) CppMem (/index.php/tag/cppmem) declarations (/index.php/tag/declarations) decltype (/index.php/tag/decltype) enum (/index.php/tag/enum) error handling (/index.php/tag/error-handling) exceptions (/index.php/tag/exceptions) expressions (/index.php/tag/expressions) final (/index.php/tag/final) finally (/index.php/tag/finally) functions (/index.php/tag/functions) Guideline Support Library (/index.php/tag/guideline-support-library) if (/index.php/tag/if) initialisations (/index.php/tag/initialisations) inline (/index.php/tag/inline) interfaces (/index.php/tag/interfaces) lambdas (/index.php/tag/lambdas) lock (/index.php/tag/lock) lock-free (/index.php/tag/lock-free) memory (/index.php/tag/memory) memory_order_consume (/index.php/tag/memory-order-consume) modules (/index.php/tag/modules) move (/index.php/tag/move) mutex (/index.php/tag/mutex) new/delete (/index.php/tag/new-delete) noexcept (/index.php/tag/noexcept) nullptr (/index.php/tag/nullptr) Ongoing Optimization (/index.php/tag/ongoingoptimization) overloading (/index.php/tag/overloading) override (/index.php/tag/override) performance (/index.php/tag/performance) pointers (/index.php/tag/pointers) ranges library (/index.php/tag/ranges-

concurrency-in-practice-get-the-most-out-of-any-machine?authorName=Rainer+Grimm)

Course: C++ Standard Library including C++14 & C++17

(https://www.educative.io/collection/10370001/5712008708096000?authorName=Rainer%20Grimm)

Course: Embedded Programming with Modern C++



(https://www.educative.io/courses/embedded-programming-with-cpp)

Course: Generic Programming (Templates)

(https://www.educative.io/courses/generic-templates-in-cpp)

Course: C++ Fundamentals for Professionals

library) relaxed semantic (/index.php/tag/relaxed-semantik) semaphores (/index.php/tag/semaphores) sequential consistency (/index.php/tag/sequential-consistency) shared_ptr (/index.php/tag/shared-ptr) singleton (/index.php/tag/singleton) smart pointers (/index.php/tag/smart-pointers) source files (/index.php/tag/source-files) spaceship (/index.php/tag/spaceship) [illegible] (/index.php/tag/static) static_assert (/index.php/tag/static-assert) switch (/index.php/tag/switch) tasks (/index.php/tag/tasks) template metaprogramming (/index.php/tag/template-metaprogramming) templates (/index.php/tag/templates) ThreadSanitizer (/index.php/tag/threadsanitizer) thread_local [illegible] (/index.php/tag/threadlokal) time (/index.php/tag/time) type erasure (/index.php/tag/type-erasure) type-traits (/index.php/tag/type-traits) unique_ptr (/index.php/tag/unique-ptr) user-defined literals (/index.php/tag/benutzerdefinierte-literale) vector (/index.php/tag/vector) volatile (/index.php/tag/volatile) weak_ptr (/index.php/tag/weak-ptr)

Blog archive

Source Code

GitHub (https://github.com/RainerGrimm/ModernesCppSource

Most Popular Posts

(https://www.educative.io/courses/cpp-fundamentals-for-professionals)

More Profiles

Training, coaching, and technology consulting (https://www.modernescpp.net/)

guidelines-be-aware-of-the-traps-of-condition-variables)

- C++20: Structure Modules (200888 hits) (https://www.modernescpp.com/index.php/c-20-divide-modules)

(http://www.facebook.com/rainer.grimm.31)

(https://plus.google.com/109576694736160795401?prs

(https://www.linkedin.com/in/rainergrimm)

(https://twitter.com/rainer_grimm)

(https://www.xing.com/profile/Rainer_Grimm12)

Visitors

Today	1452
Yesterday	7228
Week	37773
Month	194938
All	6423410

Hunting (http://www.huntersspeak.com/)

Currently are 397 guests and no members online

Kubik-Rubik Joomla! Extensions (https://kubik-rubik.de/)

Latest comments

Performance Comparison of Condition Variables and Atomics in C++20 (/index.php/component/jaggyblog/perfoı of-condition-variables-and-atomics-in-c-20)

Alex

I just installed gcc 11.1 to try some new c++20 features. There is bug in pong method , it should be ...

Read more... (/index.php/component/jaggyblog/performancecor of-condition-variables-and-atomics-in-c-20#comment-77980)

C++ is Lazy: CRTP (/index.php/component/jaggyblog/c-is-still-lazy)

Daniel T

Lines 18 and 19 are unused

Read more... (/index.php/component/jaggyblog/c-is-still-lazy#comment-77979)

No New New: Raw Pointers Removed from C++ (/index.php/component/jaggyblog/no-new-new)

Joon

Oh my god this was such a scare stumbling upon on definitely not April 1st

Read more...
(/index.php/component/jaggyblog/no-new-new#comment-77978)

Templates - First Steps (/index.php/component/jaggyblog/template-get-insight)

Pramod Tikare

Nice article. I have got 2 feedbacks. 1. It makes sense to add line numbers to your code snippet

Read more...
(/index.php/component/jaggyblog/template-get-insight#comment-77977)

Function Templates (/index.php/component/jaggyblog/function-templates)

Kiran

Hi Rainner, Thanks for writing such awesome content. Two description lines I found, might need ...

Read more...
(/index.php/component/jaggyblog/function-templates#comment-77976)

You are here: Home (/index.php) / Thread-Safe Initialization of a Singleton