Thread-Safe Initialization of Data

15 May 2016

In case the data is not modified when shared between threads, the story is simple. The data has only to be initialized in the thread safe way. **It is not necessary to use an expensive lock for each access.**

There are three ways in C++ to initialize variables in a thread safe way.

1. Constant expressions
2. The function `std::call_once` in combination with the flag `std::once_flag`
3. Static variables with block scope

Constant expressions

Constant expressions are expressions which the compiler can initialize during compile time. So, they are implicit thread safe. By using the keyword `constexpr` in front of the expression type makes it constant expression.

constexpr double pi=3.14;

In addition, user defined types can also be constant expressions. For those types, there are a few restrictions in order to initialize them at compile time.

- They must not have virtual methods or a virtual base class.
- Their constructor must be empty and itself be a constant expression.
- Their methods, which can be callable at compile time, must be constant expressions.

My struct `MyDouble` satisfies all these requirements. So it's possible to instantiate objects of `MyDouble` at compile time. This instantiation is thread safe.

```
struct MyDouble{
  constexpr MyDouble(double v): val(v){}
  constexpr double getValue(){ return val; }
private:
  double val
};

constexpr MyDouble myDouble(10.5);
std::cout << myDouble.getValue() << std::endl;
```

The function `call_once` in combination with the `once_flag`

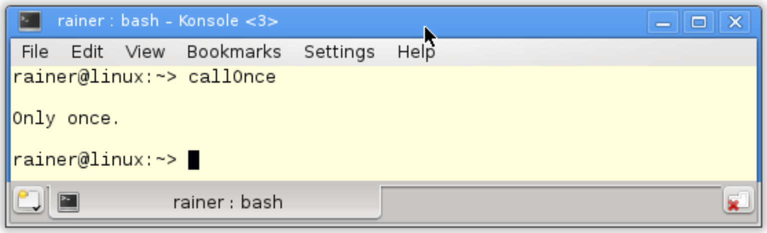
By using the `std::call_once` function, you can register all callable. The `std::once_flag` ensures, that only one registered function will be invoked. So, you can register more different functions via the `once_flag`. Only one function is called.

The short example shows the application of `std::call_once` and `std::once_flag`.

```
 1  // callOnce.cpp
 2
 3  #include <iostream>
 4  #include <thread>
 5  #include <mutex>
 6
 7  std::once_flag onceFlag;
 8
 9  void do_once(){
10    std::call_once(onceFlag, [](){ std::cout << "Only once." << std::endl; });
11  }
12
13  int main(){
14
15    std::cout << std::endl;
16
17    std::thread t1(do_once);
18    std::thread t2(do_once);
19    std::thread t3(do_once);
20    std::thread t4(do_once);
21
22    t1.join();
23    t2.join();
24    t3.join();
25    t4.join();
26
27    std::cout << std::endl;
28
29  }
```

(http://www.facebook.com/rainer.gri
(https://plus.google.com//u/0/109576694736160795401
(https://www.linkedin.com/in/raine
(https://twitter.com/rainer_g
(https://www.xing.com/profile/Rainer_Gr
Hunting (http://www.huntersspeak.com/

The program starts four threads (lines 17 - 20). Each of them should invoke the function `do_once`. The expected result is that the string "only once" is displayed only once.


rainer : bash - Konsole <3>
File Edit View Bookmarks Settings Help
rainer : bash

```
rainer@linux:~> callOnce

Only once.

rainer@linux:~>
```

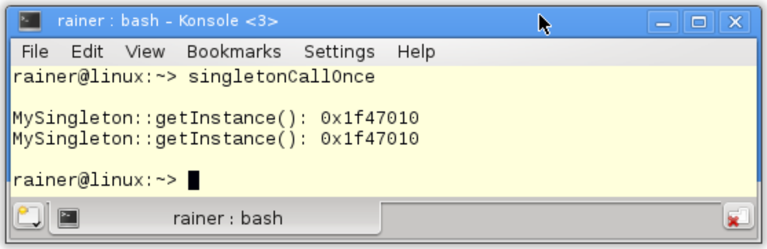
The famous singleton pattern (https://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster))guarantees only one instance of an object will be created. That is a challenging task in multithreaded environment. But, thanks to `std:.call_once` and `std:once_flag` the job is a piece of cake. Now the singleton is initialized in a thread safe way.

```
 1  // singletonCallOnce.cpp
 2
 3  #include <iostream>
 4  #include <mutex>
 5
 6  class MySingleton{
 7
 8    private:
 9      static std::once_flag initInstanceFlag;
10      static MySingleton* instance;
11      MySingleton()= default;
12      ~MySingleton()= default;
13
14    public:
15      MySingleton(const MySingleton&)= delete;
16      MySingleton& operator=(const MySingleton&)= delete;
17
18      static MySingleton* getInstance(){
19        std::call_once(initInstanceFlag,MySingleton::initSingleton);
20        return instance;
21      }
22
23      static void initSingleton(){
24        instance= new MySingleton();
25      }
26  };
27
28  MySingleton* MySingleton::instance= nullptr;
29  std::once_flag MySingleton::initInstanceFlag;
30
31
32  int main(){
33
34    std::cout << std::endl;
35
36    std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;
37    std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;
38
39    std::cout << std::endl;
40
41  }
```

(http://www.facebook.com/rainer.gri)
(https://plus.google.com//u/0/109576694736160795401
(https://www.linkedin.com/in/raine)
(https://twitter.com/rainer_g)
(https://www.xing.com/profile/Rainer_Gri
Hunting(http://www.huntersspeak.com/)

At first the static `std::once_flag`. This is in the line 9 declared and initialized in the line 29. The static method `getInstance` (line 28 - 21) uses the flag in order to ensures, that the static method `initSingleton` (line 23 - 25) is executed exactly once. In the body of the method, the singleton is created.

The output of the program is not so thrilling. The `MySingleton::getIstance()` method displays the address of the singleton.


rainer : bash - Konsole <3>
File Edit View Bookmarks Settings Help
rainer : bash

```
rainer@linux:~> singletonCallOnce

MySingleton::getInstance(): 0x1f47010
MySingleton::getInstance(): 0x1f47010

rainer@linux:~> 
```

The static story goes on.

Static variables with block scope

Static variables with block scope will be created exactly once. This characteristic is the base of the so called Meyers Singleton, named after Scott Meyers. (https://en.wikipedia.org/wiki/Scott_Meyers) This is by far the most elegant implementation of the singleton pattern.

```
#include <thread>

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

};

MySingleton::MySingleton()= default;
MySingleton::~MySingleton()= default;

int main(){

  MySingleton::getInstance();

}
```

(http://www.facebook.com/rainer.gri
(https://plus.google.com//u/0/109576694736160795401
(https://www.linkedin.com/in/raine
(https://twitter.com/rainer_g
(https://www.xing.com/profile/Rainer_Gr
Hunting(http://www.huntersspeak.com/)

By using the keyword `default`, you can request special methods from the compiler. They are Special because only compiler can create them. With `delete`, the result is, that the automatically generated methods (constructor, for example) from the compiler will not be created and, therefore, can not be called. If you try to use them you'll get an compile time error. What's the point of the Meyers Singleton in multithreading programs? The Meyers Singleton is thread safe.

A side note: Double-checked locking pattern

Wrong beliefe exists, that an additional way for the thread safe initialization of a singleton in a multithreading environment is the double-checked locking pattern. The double-checked locking pattern is - in general - an unsafe way to initialize a singleton. It assumes guarantees in the classical implementation, which aren't given by the Java, C# or C++ memory model. The assumption is, that the access of the singleton is atomic.

But, what is the double-checked locking pattern? The first idea to implement the singleton pattern in a thread safe way, is to protected the initialization of the singleton by a lock.

```
 1  mutex myMutex;
 2
 3  class MySingleton{
 4  public:
 5    static MySingleton& getInstance(){
 6      lock_guard<mutex> myLock(myMutex);
 7      if( !instance ) instance= new MySingleton();
 8      return *instance;
 9    }
10  private:
11    MySingleton();
12    ~MySingleton();
13    MySingleton(const MySingleton&)= delete;
14    MySingleton& operator=(const MySingleton&)= delete;
15    static MySingleton* instance;
16  };
17  MySingleton::MySingleton()= default;
18  MySingleton::~MySingleton()= default;
19  MySingleton* MySingleton::instance= nullptr;
```

Any issues? Yes and no. The implementation is thread safe. But there is a great performance penalty. Each access of the singleton in line 6 is protected by an expansive lock. That applies also for the reading access. Most time it's not necessary. Here comes the double-checked locking pattern to our rescue.

```
1  static MySingleton& getInstance(){
2    if ( !instance ){
3      lock_guard<mutex> myLock(myMutex);
4      if( !instance ) instance= new MySingleton();
5    }
6    return *instance;
7  }
```

(http://www.facebook.com/rainer.gri)
(https://plus.google.com//u/0/109576694736160795401
(https://www.linkedin.com/in/rainer)
(https://twitter.com/rainer_g
(https://www.xing.com/profile/Rainer_Gri
Hunting (http://www.huntersspeak.com/)

I use inexpensive pointer comparison in the line 2 instead of an expensive lock a. O[illegible] I get a null pointer, I apply the expensive lock on the singleton (line 3). Because there is the possibility that another thread will initialize the singleto[illegible]ween the pointer comparison (line 2) and the lock (line3), I have to perform an additional pointer comparison the on line 4. So the name is clear: [illegible] times a check and one time a lock.

Smart? Yes. Thread safe? No.

What is the problem? The call instance= new MySingleton() in line 4 consists of at least three steps.

1. Allocate memory for `MySingleton`
2. Create the `MySingleton` object in the memory
3. Let `instance` refer to the `MySingleton` object

The problem: there is no guarantee about the sequence of these steps. For example, out of optimization reasons, the processor can 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 an incomplete singleton. If at that time another thread tries to access the singleton, it compares the pointer and gets the answer `true`. So, the other thread has the illusion that it's dealing with a complete singleton.

The consequence is simple: program behaviour is undefined.

What's next?

At first, I thought, I should continue in the next post (/index.php/thread-local-data) with the singleton pattern. But to write about the singleton pattern, you should have a basic knowledge of the memory model. So I continue in the sequence of my German blog. The next post will be about-thread local storage. In case we are done with the high end API of multithreading in C++, I'll go further with the low end API. **(Proofreader Alexey Elymanov)**

Go to (https://leanpub.com/cpplibrary)Leanpub/cpplibrary (https://leanpub.com/cpplibrary)

(https://leanpub.com/cpplibrary)**"What every professional C++ programmer should know about the C++ standard library".** (https://leanpub.com/cpplibrary) Get your e-book. Support my blog.

Tweet G+ Share

Tags: static (/index.php/tag/static), constexpr (/index.php/tag/constexpr), singleton (/index.php/tag/singleton)

Comments (/index.php/component/jcomments/feed/com_jaggyblog/162)

#1 (/index.php/component/jaggyblog/thread-safe-initialization-of-data#comment-133) **visit** 2016-06-28 04:32 **0**
I genuinely appreciate your work, Great post.

Quote

#2 (/index.php/component/jaggyblog/thread-safe-initialization-of-data#comment-134) **goo.gl** 2016-06-28 04:41 **0**
I got this web page from my pall who informed me regarding
this websitfe and now this time I am visioting this site and
reading very informative posts here.
Quote

(http://www.facebook.com/rainer.gri
(https://plus.google.com//u/0/109576694736160795401
(https://www.linkedin.com/in/raine
(https://twitter.com/rainer_g
(https://www.xing.com/profile/Rainer_Gri

#3 (/index.php/component/jaggyblog/thread-safe-initialization-of-data#comment **visit** 2016-07-02 14:37 **0**
I really treasure your pikece of work, Great post.
Quote

#4 (/index.php/component/jaggyblog/thread-safe-initialization-of-data#comment **Dianne** 2016-07-06 13:03 **0**
I do not even know how I finished up here, but I believed this publish was
once good. I don't understand whho you're but definitely you are going to a
famous bloggewr iff you are not already. Cheers!
Hunting (http://www.huntersspeak.com/)
Quote

#5 (/index.php/component/jaggyblog/thread-safe-initialization-of-data#comment-187) **twitter.com** 2016-07-11 05:05 **0**
I'm still learning from you, ass I'm improving myself.

I definitely love reading everything that is posted on your site.Keep
the posts coming. I liked it!
Quote

#6 (/index.php/component/jaggyblog/thread-safe-initialization-of-data#comment-5777) **Thomas** 2017-07-04 06:20 **0**
last closing bracket in double-checked locking pattern must be written before instance is returned. apart from that a nice article. thanks!
Quote

Refresh comments list
RSS feed for comments to this post (/index.php/component/jcomments/feed/com_jaggyblog/162)

Add comment

Name (required)

E-mail (required, but will not display)

Website

Notify me of follow-up comments

Send

JComments (http://www.joomlatune.com)

(http://www.modernescpp.de/)

Open C++ Seminars

Embedded Programmierung mit modernem C++: 16.01 - 18.01 (http://www.modernescpp.de/index.php/c/2-c/16-embedded-programmierung-mit-modernem-c)

C++11 und C++14: 13.03 - 15.03 (http://www.modernescpp.de/index.php/c/2-c/3-c-11-und-c-14)

Multithreading mit modernem C++: 08.05 - 09.05 (http://www.modernescpp.de/index.php/c/2-c/13-multithreading-mit-c)

Contact

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

Impressum (/index.php/impressum)

My Newest E-Books

(https://leanpub.com/cpplibrary)

Subscribe to the newsletter (+ pdf bundle)

E-mail Subscribe

(http://www.facebook.com/rainer.gri

Become a patron (https://www.patreon.com

(https://plus.google.com/u/0/109576694736160795401

(https://www.linkedin.com/in/rainer/rainer_grimm)

(https://twitter.com/rainer_g

(https://www.xing.com/profile/Rainer_Gri

>> Start here << (/index.php/der-einstieg-in-modernes-c)

Categories

- Embedded (/index.php/category/embedded)
- Functional (/index.php/category/functional)
- Modern C++ (/index.php/category/modern-c)
- Multithreading (/index.php/category/multithreading)
- Multithreading - Application (/index.php/category/multithreading-application)
- 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/ueberblick)
- C++ 17 (/index.php/category/c-17)
- Pdf bundles (/index.php/category/pdf-bundle)

All tags

acquire-release semantic (/index.php/tag/acquire-release-semantik) async (/index.php/tag/async) atomics (/index.php/tag/atomics) atomic_thread_fence (/index.php/tag/atomic-thread-fence) auto (/index.php/tag/auto) C++17 (/index.php/tag/c-17) C++20 (/index.php/tag/c-20) concepts (/index.php/tag/concepts) condition variable (/index.php/tag/condition-variable) constexpr (/index.php/tag/constexpr) CppMem (/index.php/tag/cppmem) enum (/index.php/tag/enum) final (/index.php/tag/final) hash tables (/index.php/tag/hashtabellen) inline (/index.php/tag/inline) lock (/index.php/tag/lock) memory (/index.php/tag/memory) memory_order_consume (/index.php/tag/memory-order-consume) mutex (/index.php/tag/mutex) nullptr (/index.php/tag/nullptr) Ongoing Optimization (/index.php/tag/ongoingoptimization) override (/index.php/tag/override) relaxed semantic (/index.php/tag/relaxed-semantik) 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) static (/index.php/tag/static) static_assert (/index.php/tag/static-assert) tasks (/index.php/tag/tasks) templates (/index.php/tag/templates) thread_local (/index.php/tag/threadlokal) time (/index.php/tag/time) type-traits (/index.php/tag/type-traits) unique_ptr (/index.php/tag/unique-ptr) user-defined literals (/index.php/tag/benutzerdefinierte-literale) volatile (/index.php/tag/volatile) weak_ptr (/index.php/tag/weak-ptr)

Blog archive

► 2017 (89)
► 2016 (97)

(https://leanpub.com/concurrencywithmodernc)

More Profiles

Training, coaching, and technology consulting (http://www.ModernesCpp.de)

My books, articles and presentations (http://www.grimm-jaud.de/)

Latest comments

Multithreading in Modern C++ (/index.php/component/jaggyblog/multithreading-in-modern-c)

Violet

Well I really enjoyed reading it. This subject provided by you is very useful for good planning.
Read more... (/index.php/component/jaggyblog/multithreading-in-modern-c#comment-8838)

Multithreading in Modern C++ (/index.php/component/jaggyblog/multithreading-in-modern-c)

Trena

Great article, totally what I needed.
Read more... (/index.php/component/jaggyblog/multithreading-in-modern-c#comment-8608)

The new pdf bundle is available: Functional Programming with C++17 and C++20 (/index.php/component/jaggyblog/the-new-pdf-bundle-is-available-functional-programming-with-c-17-and-c-20)

Maxim

Send me a pdf bundle, please.
Read more... (/index.php/component/jaggyblog/the-new-pdf-bundle-is-available-functional-programming-with-c-17-and-c-20#comment-8553)

C++ Core Guidelines: Function Definitions (/index.php/component/jaggyblog/c-core-guidelines-functions)

Jeremiah

Very nice blog post. I certainly appreciate this website. Keep writing!
Read more... (/index.php/component/jaggyblog/c-core-guidelines-functions#comment-8526)

Strongly-Typed Enums (/index.php/component/jaggyblog/strongly-typed-enums)

Horacio

Excellent post. I was checking constantly this blog and I amm

Source Code

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

(http://www.facebook.com/rainer.gri
(https://plus.google.com/u/0/109576694736160795401
(https://www.linkedin.com/in/raine)
(https://twitter.com/rainer_g
(https://www.xing.com/profile/Rainer_Gr
Hunting (http://www.huntersspeak.com/)

Most Popular Posts

- Multithreading in Modern C++ (http://www.modernescpp.com/index.php/multithreading-in-modern-c)
- What is Modern C++? (http://www.modernescpp.com/index.php/what-is-modern-c)
- Memory and Performance Overhead of Smart Pointers (http://www.modernescpp.com/index.php/memory-and-performance-overhead-of-smart-pointer)
- Multithreading with C++17 and C++20 (http://www.modernescpp.com/index.php/multithreading-in-c-17-and-c-20)
- C++17 - What's New in the Core Language? (http://www.modernescpp.com/index.php/cpp17-core)

Visitors

Today	266
All	471695

Currently are 160 guests and no members online

impressed! Extremely useful info ...
Read more... (/index.php/component/jaggyblog/strongly-typed-enums#comment-8509)

(http://www.facebook.com/rainer.gr)
(https://plus.google.com//u/0/10957669473616079540
(https://www.linkedin.com/in/raine)
(https://twitter.com/rainer_g)
(https://www.xing.com/profile/Rainer_Gr)
Hunting (http://www.hunterspeak.com/)

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