**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.

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

std::once\_flag flag1, flag2;

void simple\_do\_once()

{

std::call\_once(flag1, []() { std::cout << "Simple example: called once\n"; });

}

void may\_throw\_function(bool do\_throw)

{

if (do\_throw) {

std::cout << "throw: call\_once will retry\n"; // this may appear more than once

throw std::exception();

}

std::cout << "Didn't throw, call\_once will not attempt again\n"; // guaranteed once

}

void do\_once(bool do\_throw)

{

try {

std::call\_once(flag2, may\_throw\_function, do\_throw);

}

catch (...) {

}

}

int main()

{

std::thread st1(simple\_do\_once);

std::thread st2(simple\_do\_once);

std::thread st3(simple\_do\_once);

std::thread st4(simple\_do\_once);

st1.join();

st2.join();

st3.join();

st4.join();

std::thread t1(do\_once, true);

std::thread t2(do\_once, true);

std::thread t3(do\_once, false);

std::thread t4(do\_once, true);

t1.join();

t2.join();

t3.join();

t4.join();

}

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.

// singletonCallOnce.cpp

#include <iostream>

#include <mutex>

class MySingleton{

private:

static std::once\_flag initInstanceFlag;

static MySingleton\* instance;

MySingleton()= default;

~MySingleton()= default;

public:

MySingleton(const MySingleton&)= delete;

MySingleton& operator=(const MySingleton&)= delete;

static MySingleton\* getInstance(){

std::call\_once(initInstanceFlag,MySingleton::initSingleton);

return instance;

}

static void initSingleton(){

instance= new MySingleton();

}

};

MySingleton\* MySingleton::instance= nullptr;

std::once\_flag MySingleton::initInstanceFlag;

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;

}

## 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();

}

## 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.

mutex myMutex;

class MySingleton{

public:

static MySingleton& getInstance(){

lock\_guard<mutex> myLock(myMutex);

if( !instance ) instance= new MySingleton();

return \*instance;

}

private:

MySingleton();

~MySingleton();

MySingleton(const MySingleton&)= delete;

MySingleton& operator=(const MySingleton&)= delete;

static MySingleton\* instance;

};

MySingleton::MySingleton()= default;

MySingleton::~MySingleton()= default;

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.

static MySingleton& getInstance(){

if ( !instance ){

lock\_guard<mutex> myLock(myMutex);

if( !instance ) instance= new MySingleton();

}  
  return \*instance;

}

I use inexpensive pointer comparison  in the line 2 instead of an expensive lock a. Only if 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 singleton between 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. Two 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.