Classical Meyers Singleton

This lesson gives an overview of a classical Meyers Singleton in C++.

Here is the sequential program. The `getInstance` method is not thread-safe with the C++03 standard.

```
// singletonSingleThreaded.cpp

#include <chrono>
#include <iostream>

constexpr auto tenMill = 10000000;

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

};

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;

}
```

As the reference implementation, I use the so-called Meyers Singleton, named after Scott Meyers. The elegance of this implementation is that the singleton object in line 11 is a static variable with a block scope; therefore, `instance` will be initialized only once. This initialization happens when the static method `getInstance` (lines 10 - 14) will be executed the first time.

> **The volatile Variable dummy**
>
> When I compiled the program with maximum optimisation, the compiler removed the call `MySingleton::getInstance()` in line 30 because the call has no effect; therefore, I got very fast execution, but wrong performance numbers. By using the `volatile` variable `dummy` (line 12), the compiler is not allowed to optimise away the `MySingleton::getInstance()` call in line 30.

The beauty of the Meyers Singleton is that it becomes thread-safe with C++11, so let's see how in the next lesson.