Thread-Safe Initialization - Static Variables with Block Scope

This lesson gives an overview of thread-safe initialization with static variables from the perspective of concurrency in C++.

WE'LL COVER THE FOLLOWING

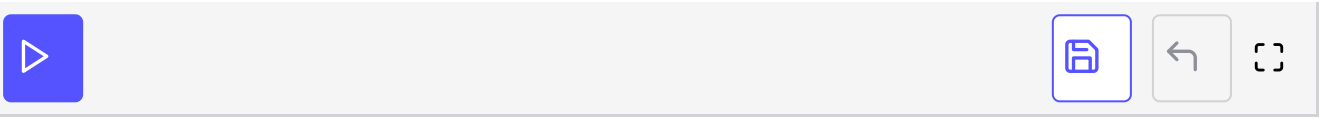
Static variables with block scope will be created exactly once and lazily (i.e. created just at the moment of the usage). This characteristic is the basis of the so-called Meyers Singleton, named after Scott Meyers. This is by far the most elegant implementation of the singleton pattern. With C++11, static variables with block scope have an additional guarantee; they will be initialized in a thread-safe way.

Thread-Safe example:

Here is the thread-safe Meyers Singleton pattern.

```
// meyersSingleton.cpp

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();

}
```



> **Know the Compiler's support for static**
>
> If we use the Meyers Singleton pattern in a concurrent environment, be sure that our compiler implements static variables with the C++11 thread-safe semantic. It happens quite often that programmers rely on the C++11 semantics of static variables, but their compiler does not support it. The result may be that more than one instance of a singleton is created.

In this chapter, we have studied how data sharing is done with the use of threads and locks. In the next chapter, we'll discuss tasks which consist of promises and futures, and are an important part of multithreading in C++.