- Solution

In this lesson, we'll discuss the solution to the previous exercise.

WE'LL COVER THE FOLLOWING

Solution

```
// 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;

}
```

Explanation

First, let’s consider `static std::once_flag`. This is declared in line 9 and initialized in line 29. The static method `getInstance` (lines 18 - 21) uses the flag in order to ensure that the static method `initSingleton` (lines 23 - 25) is executed exactly once. The singleton is created in the body of the method.

> **i `default` and `delete`**
>
> We can request special methods from the compiler by using the keyword `default`. These methods are special because the compiler can create them for us. The result of annotating a method with `delete` is that the compiler-generated methods will not be available and, therefore, cannot be called. If we try to use them, we’ll get a compile-time error. Here are the details for the keywords default and delete.

The `MySingleton::getInstance()` method displays the address of the singleton.

In the next lesson, we will look at how variables can be initialized in a thread-safe way using static variables.