Thread-Safe Initialization: call_once and once_flag

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

By using the `std::call_once` function you can register a callable. The `std::once_flag` ensures that only one registered function will be invoked, but you can register additional functions via the same `std::once_flag`. That being said, only one function from that group is called.

`std::call_once` obeys the following rules:

- Exactly one execution of exactly one of the functions is performed. It is undefined which function will be selected for execution. The selected function runs in the same thread as the `std::call_once` invocation it was passed to.
- No invocation in the group returns before the above-mentioned execution of the selected function completes successfully.
- If the selected function exits via an exception, it is propagated to the caller. Another function is then selected and executed.

The short example demonstrates the application of `std::call_once` and the `std::once_flag`. Both of them are declared in the header `<mutex>`.

```
// callOnce.cpp

#include <iostream>
#include <thread>
#include <mutex>

std::once_flag onceFlag;

void do_once(){
  std::call_once(onceFlag, [](){ std::cout << "Only once." << std::endl; });
}

int main(){

  std::cout << std::endl;
  std::thread t1(do_once);
  std::thread t2(do_once);
  std::thread t3(do_once);
  std::thread t4(do_once);

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  std::cout << std::endl;

}
```

The program starts four threads (lines 17 - 20); each of them invokes `do_once`. The expected result is that the string “only once” is displayed only once.

The famous singleton pattern guarantees that only one instance of an object will be created. This is a challenging task in multithreading environments, but `std::call_once` and `std::once_flag` make the job a piece of cake. Now the singleton is initialized in a thread-safe way.

```
// singletonCallOnce.cpp

#include <iostream>
#include <mutex>

using namespace std;

class MySingleton{

  private:
    static once_flag initInstanceFlag;
    static MySingleton* instance;
    MySingleton() = default;
    ~MySingleton() = default;

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

    static MySingleton* getInstance(){
      call_once(initInstanceFlag,MySingleton::initSingleton);
      return instance;
    }

    static void initSingleton(){
      instance= new MySingleton();
    }
};
```

```
MySingleton* MySingleton::instance = nullptr;
once_flag MySingleton::initInstanceFlag;

int main(){

  cout << endl;

  cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << endl;
  cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << endl;

  cout << endl;

}
```

Let's first review the static `std::once_flag`. It is declared in line 11 and initialized in line 31. The static method `getInstance` (lines 20 - 23) uses the flag `initInstanceFlag` to ensure that the static method `initSingleton` (line 25 - 27) is executed exactly once. The singleton is created in the body of the method.

> **`default` and `delete`**
>
> You 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 you try to use them, you'll get a compile-time error. Here are the details for the keywords default and delete.

The `MySingleton::getIstance()` 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