Thread-safe Singleton

To fix the problem, you have to synchronize threads during the first creation of the Singleton object.

main.cc: Conceptual example

```
/**
 * The Singleton class defines the `GetInstance` method that serves as an
 * alternative to constructor and lets clients access the same instance of this
 * class over and over.
 */
class Singleton
{

    /**
     * The Singleton's constructor/destructor should always be private to
     * prevent direct construction/desctruction calls with the `new`/`delete`
     * operator.
     */
private:
    static Singleton * pinstance_;
    static std::mutex mutex_;

protected:
    Singleton(const std::string value): value_(value)
    {
    }
    ~Singleton() {}
    std::string value_;

public:
    /**
     * Singletons should not be cloneable.
     */
    Singleton(Singleton &other) = delete;
    /**
     * Singletons should not be assignable.
     */
    void operator=(const Singleton &) = delete;
    /**
     * This is the static method that controls the access to the singleton
     * instance. On the first run, it creates a singleton object and places it
```

```
     * into the static field. On subsequent runs, it returns the client existing
     * object stored in the static field.
     */

    static Singleton *GetInstance(const std::string& value);
    /**
     * Finally, any singleton should define some business logic, which can be
     * executed on its instance.
     */
    void SomeBusinessLogic()
    {
        // ...
    }

    std::string value() const{
        return value_;
    }
};

/**
 * Static methods should be defined outside the class.
 */

Singleton* Singleton::pinstance_{nullptr};
std::mutex Singleton::mutex_;

/**
 * The first time we call GetInstance we will lock the storage location
 *      and then we make sure again that the variable is null and then we
 *      set the value. RU:
 */
Singleton *Singleton::GetInstance(const std::string& value)
{
    std::lock_guard<std::mutex> lock(mutex_);
    if (pinstance_ == nullptr)
    {
        pinstance_ = new Singleton(value);
    }
    return pinstance_;
}

void ThreadFoo(){
    // Following code emulates slow initialization.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("FOO");
    std::cout << singleton->value() << "\n";
}

void ThreadBar(){
    // Following code emulates slow initialization.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("BAR");
```

```
    std::cout << singleton->value() << "\n";
}

int main()
{
    std::cout <<"If you see the same value, then singleton was reused (yay!\n" <<
                "If you see different values, then 2 singletons were created (booo!!
                "RESULT:\n";
    std::thread t1(ThreadFoo);
    std::thread t2(ThreadBar);
    t1.join();
    t2.join();

    return 0;
}
```

Output.txt: Execution result

```
If you see the same value, then singleton was reused (yay!
If you see different values, then 2 singletons were created (booo!!)

RESULT:
FOO
FOO
```