Singleton in C++

Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

Singleton has almost the same pros and cons as global variables. Although they're super-handy, they break the modularity of your code.

You can't just use a class that depends on a Singleton in some other context, without carrying over the Singleton to the other context. Most of the time, this limitation comes up during the creation of unit tests.

Learn more about Singleton

Complexity:

Popularity:

Usage examples: A lot of developers consider the Singleton pattern an antipattern. That's why its usage is on the decline in C++ code.

Identification: Singleton can be recognized by a static creation method, which returns the same cached object.

Naïve Singleton | Thread-safe Singleton

Naïve Singleton

It's pretty easy to implement a sloppy Singleton. You just need to hide the constructor and implement a static creation method.

The same class behaves incorrectly in a multithreaded environment. Multiple threads can call the creation method simultaneously and get several instances of Singleton class.

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 should always be private to prevent direct
     * construction calls with the `new` operator.
     */

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

    static Singleton* 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_;
```

```
    }
};

Singleton* Singleton::singleton_= nullptr;;

/**
 * Static methods should be defined outside the class.
 */
Singleton *Singleton::GetInstance(const std::string& value)
{
    /**
     * This is a safer way to create an instance. instance = new Singleton is
     * dangeruous in case two instance threads wants to access at the same time
     */
    if(singleton_==nullptr){
        singleton_ = new Singleton(value);
    }
    return singleton_;
}

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:
BAR
FOO
```