



C++ Island

Doing More With Less

C++'s Simple Singletons

The singleton is one of the most controversial design patterns. Yet, it is omnipresent in many real-world code bases. With C++ 11, creating a thread safe, lazy initialized singleton is cheap, painless and straight forward process, by either using local statics or `std::call_once`:

Local statics

Creating a singleton with a local statics boils down to:

```
Singleton& Singleton::Instance()
{
    static Singleton instance;
    return instance;
}
```

The standard guarantees the *Singleton* instance is created once, even if more than one thread attempts to call `Instance` at the same time (ISO/IEC 14882:2011 6.7 footnote #4)

call_once

`std::call_once` wraps a callable object and ensure it is called only once. Even if multiple threads try to call it at the same time.

The wrapper is using an object of type `std::once_flag` to keep track whether the code was already called:

```
#include <iostream>
#include <mutex>

class Singleton
{
private:
    Singleton(const Singleton&) = delete;
    Singleton & operator=(const Singleton&) = delete;

    static std::unique_ptr<Singleton> instance;
    static std::once_flag onceFlag;
public:
    Singleton() = default;

    static void NofityInit()
    {
        std::cout << "Initializing Singleton" << '\n';
    }
    static Singleton& Singleton::Instance()
    {
        std::call_once(Singleton::onceFlag, [] (){
            NofityInit();
            instance.reset(new Singleton);
        });
    }
};
```

```
        std::cout << "Getting Singleton instance" << '\n';
        return *(instance.get());
    }
};

std::unique_ptr<Singleton> Singleton::instance;
std::once_flag Singleton::onceFlag;


int main()
{
    Singleton& s1 = Singleton::Instance();
    Singleton& s2 = Singleton::Instance();

    return 0;
}
```

The output for this program:

```
Initializing Singleton
Getting Singleton instance
Getting Singleton instance
```

For more details, including performance study, check out Rainer Grimm's in-depth [post](#).

 Alon / January 15, 2017 / Multithreading, Standard Library

5 thoughts on “C++’s Simple Singletons”



Michael Cook

January 16, 2017 at 3:29 pm

The problem with `static Singleton instance;` is that it’s impractical in general to know what’s going to happen at shutdown. Static objects will be destroyed in the reverse order they were constructed, but what happens if a static object’s destructor tries to use another statically constructed object, for example? This is known as the “static initialization fiasco”.

This page:

<https://isocpp.org/wiki/faq/ctors#static-init-order>

recommends instead to use `static auto instance = new Singleton;` and then let the static object leak on shutdown (i.e., never destroy it).



Gaetano

January 16, 2017 at 8:42 pm

Leak at exit it’s fine if acquired resources are managed by OS otherwise you are going to have troubles.

**Michael Cook**

January 16, 2017 at 8:54 pm

Agreed. If the destructor removes temporary files, for example, then choosing not to run the destructor would probably not be acceptable.

**Alon**

January 17, 2017 at 1:26 am

Thank you for your comments Michael and Gaetano

You are making a very good point, C++ 11 has made Singletons creation very simple. Getting rid of them – not so much. It's the global scope of the singleton object that makes it the black sheep of design patterns. Unlike local variables whose lifetime can be finely bound by a simple use of curly braces, Singletons all live in the same scope where a subtle factor such as initialization order is a deciding factor on the object lifetime compared to other objects.

Unfortunately, there is no one-size fits all solution, there are options to mitigate the problem each suited to the particular setup (e.g. resources need to be released), here are a few ideas

1. As Michael Cook suggested above, do not use static object, instead use a static pointer to an object which will get destroyed (no clean up)
2. Call an Instance Only from a static object's destructor, ensuring the singleton will always be constructed before any other static – the singleton destructor may do any necessary cleanup.
3. Do not use the singleton object in other statics destructors (the singleton destructor can do any needed cleanup) – I would not use this option as there is no way to enforce it through the compiler only with coding guideline which is a weak defense indeed.

4. Consider another approach other than a singleton



Girish

April 4, 2017 at 7:30 pm

Now that Singleton class has a public destructor (provided by the compiler), what if someone does `delete &s2` (in main?)

Ideally we don't want users to call destructor directly for a singleton class – isn't it?

C++ Island / Proudly powered by WordPress