Lazy initialization of a C++ singleton in declaration or in implementation

Asked 3 years, 10 months ago Active 3 years, 10 months ago Viewed 2k times

2

I know that the singleton pattern is usually considered a bad design and hence discouraged, but this question concerns the implementation aspects, not the appropriateness of the singleton pattern.

Consider the following three implementations of a singleton in C++ using lazy initialization:

1: Using pointer, split between declaration and implementation

Singleton.hpp:

```
class Singleton {
public:
   static Singleton* instance();
private:
   Singleton() {}
   static Singleton* singleton;
};
```

Singleton.cpp:

```
Singleton* Singleton::singleton = nullptr;

Singleton* Singleton::instance() {
    if( nullptr == singleton ) {
        singleton = new Singleton();
    }
    return singleton;
}
```

2: Using reference, split between declaration and implementation

Singleton.hpp:

```
class Singleton {
public:
   static Singleton& instance();
private:
   Singleton() {}
};
```

Singleton.cpp:

3: Using reference, inline in declaration

Singleton.hpp:

```
class Singleton {
public:
    static Singleton& instance() {
         static Singleton singleton;
         return singleton;
    }
private:
    Singleton() {}
}
```

I personally like and use the third version. But is there any good reason to prefer the first or the second version instead?

It is my understanding that in the third version there is an instance of the object for each translation unit that includes `Singleton.hpp` and then the linker picks a single one. Does this causes any side effect?

And are there any side effects using the third one in a shared library?

Bonus questions: which implementation is actually the "Meyer's singleton"?

c++ design-patterns singleton

Share Improve this question Follow

asked Jul 12 '17 at 9:25

lornova
5,917 8 39 68

Well references cannot be null, so the second and third methods cannot be used for lazy instantiation – meowgoesthedog Jul 12 '17 at 9:26

3 @spug Local static variables with initialization will be initialized the first time the function is called. – Some programmer dude Jul 12 '17 at 9:27

3 The second option (Meyers Singleton) is guaranteed to be thread safe in C++11, just so you know. – StoryTeller - Unslander Monica Jul 12 '17 at 9:28

@StoryTeller The second version should also be thread safe. The only difference between them is that the in the second alternative the `instance` function can't be inlined. The initialization of the instance is still thread-safe. – Some programmer dude Jul 12 '17 at 9:31

1 @VTT - "*No explicit control*", well that's the point of lazy evaluation... – StoryTeller - Unslander Monica Jul 12 '17 at 9:35

1 Answer

3

```
if( nullptr == singleton ) {
   singleton = new Singleton();
}
```

It could happen that multiple threads could execute the allocation statement and create a *memory leak*.

The second and the third ones are thread-safe *since C++11*, because:

> If multiple threads attempt to initialize the same static local variable concurrently, the initialization occurs exactly once (similar behavior can be obtained for arbitrary functions with std::call_once).

from here.

I'd prefer the third one because the inline optimisations are more likely.

Share Improve this answer Follow edited Jul 12 '17 at 14:51

answered Jul 12 '17 at 9:36

BiagioF
8,545 2 21 45

It is correct to define that an "undefined behavior"? To me that is just a plain thread unsafe code. – lornova Jul 12 '17 at 12:01

1 @Wizard79 You're probably right, I've edited the answer. – BiagioF Jul 12 '17 at 14:53