

Morning Musings

I'm not ready to wake up yet...

- [RSS](#)

- [Blog](#)
- [Archives](#)
- [License](#)
- [About](#)
- [Contact](#)

Singletons

Aug 2nd, 2015 | [Comments](#)

The Singleton pattern is a design pattern that restricts the instantiation of a class to one object. It is typically used for solving the problem of resource contention, such that you need to manage a single instance of a resource. Singletons are misunderstood and difficult to implement correctly, hopefully this post can clear things up.

Table of Contents

- [Creating a Singleton](#)
- [The Double-Checked Locking Pattern is Unsafe](#)
 - [Fixing the Double-Checked Locking Pattern](#)
- [Enhancing the Singleton](#)
- [Boost Pool Method](#)
- [Meyer's Method \(C++11\)](#)
- [Boost Call Once Method](#)

- [My Method](#)

It is known as an anti-pattern in that it is often misused or abused to the point where it can have a negative effect on a code base. Opponents to Singletons compare them to global variables, but that is not entirely true; As an object-oriented pattern, they can implement interfaces and be subclassed, which gives them more useful properties than a normal global. That said, Singletons are considered harmful¹ for a variety of reasons:

- Singletons make code hard to follow
- Singletons make code hard to test
- Singletons make code hard to maintain
- Singletons make code hard to secure

The reason for this is because the use of a Singleton hides the dependency from the interface, giving program flow a path that is not easily visible.

Singletons are nothing more than global state. Global state makes it so your objects can secretly get hold of things which are not declared in their APIs, and, as a result, Singletons make your APIs into pathological liars. If you are the person who built the code originally, you know the true dependencies, but anyone who comes after you is baffled.²

Note that there is a difference between having a single object and having a Singleton. Passing around a single instance to share among your classes via constructor references is known as dependency injection, and is a cleaner solution for most problems that a Singleton can solve. Take the following for example:

```
// Singleton approach
Constructor(void)
{
    this->single_instance = Singleton::instance()
}

// Dependency Injection approach
Constructor(SingleInstance& instance) :
    single_instance(instance)
{

}
```

With the second approach, it becomes very clear from the interface that the object depends on SingleInstance. This dependency is hidden with the first approach. The second approach can also be tested easier by subclassing the SingleInstance to make a testable version.

Still, there are cases where Singletons are very useful:

- Debug Logging
- Memory Pools
- Factories
 - [Abstract Factory](#)
 - [Builder](#)
 - [Prototype](#)

In these types of situations, there is one object that has a single responsibility and must be accessible from anywhere. In particular, a memory pool using a Singleton is not very different from allocating memory from the heap using the global `::operator new`.

Configuration is sometimes done with Singletons, however this isn't considered a good practice. Dependency injection is typically a better way to handle passing configuration through your program.

Creating a Singleton

Singletons conform to the following conditions³:

- Private static attribute in the class.
- Public static accessor function in the class.
- Perform creation on first use in the accessor function.
- All constructors are protected or private.
- Clients may only use the accessor function to manipulate the Singleton.

Creating a Singleton is much harder than it appears, and there are very subtle considerations that are easy to miss or implement incorrectly. A few examples are:

- Lazy instantiation
- Proper destruction
- Thread safety
- Instruction reordering

A first cut at a Singleton may look something like this:

```
1 #include <iostream>
2
3 template<typename T>
4 class Singleton
```

```
5 {
6 public:
7
8     static T& instance(void)
9     {
10         if(!pointer)
11         {
12             pointer = new T();
13         }
14
15         return *pointer;
16     }
17
18 private:
19
20     Singleton(void){}
21
22     static T* pointer;
23 };
24
25 template<typename T> T* Singleton<T>::pointer;
26
27 struct Hello
28 {
29     Hello(void){std::cout << "Hello!" << std::endl;}
30     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
31 };
32
33 int main(int argc, char* argv[])
34 {
35     std::cout << "Beginning of main" << std::endl;
36     Hello& hello = Singleton<Hello>::instance();
37     Hello& hello2 = Singleton<Hello>::instance();
38     std::cout << &hello << " : " << &hello2 << std::endl;
39 }
```

Running this produces the following output:

```
Beginning of main
Hello!
0x1e2a370 : 0x1e2a370
```

Notice the following:

- The constructor was only called once
- The addresses are the same, meaning there is only one object
- The destructor was never called
- The object was created lazily (on first request)

Return a pointer or reference?

The advantage to returning a reference instead of a pointer is:

- You can be assured that there won't be a null pointer
- The user will be unable to delete the pointer
- No need to dereference to use overloaded operators

This falls apart when multi-threading is introduced. Consider the following events:

1. Thread 1 may get through line 8
2. Thread 2 could pass all the way through to line 14
3. Thread 1 then performs line 10 again

This situation causes two `T`'s plus a memory leak. Furthermore, both threads are now pointing at different objects.

To make it thread safe, a mutex can be used:

```
1  #include <iostream>
2  #include <boost/thread/mutex.hpp>
3
4  template<typename T>
5  class Singleton
6  {
7  public:
8
9      static T& instance(void)
10     {
11         boost::mutex::scoped_lock lock(mutex);
12
13         if(!pointer)
14         {
15             pointer = new T();
16         }
17
18         return *pointer;
19     }
20
21 private:
22
23     Singleton(void){}
24
25     static T* pointer;
26     static boost::mutex mutex;
27 };
28
```

```
29 template<typename T> T* Singleton<T>::pointer;
30 template<typename T> boost::mutex Singleton<T>::mutex;
31
32 struct Hello
33 {
34     Hello(void){std::cout << "Hello!" << std::endl;}
35     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
36 };
37
38 int main(int argc, char* argv[])
39 {
40     std::cout << "Beginning of main" << std::endl;
41     Hello& hello = Singleton<Hello>::instance();
42     Hello& hello2 = Singleton<Hello>::instance();
43     std::cout << &hello << " : " << &hello2 << std::endl;
44 }
```

Now, only one thread is able to pass line 10 at a time, making the logic that follows thread safe. However, mutexes are expensive to acquire, and every call to `instance` requires locking the mutex, even though it really only needs to be locked when the object needs to be created the very first time. So a common thing to do is check whether the instance needs to be created before locking the mutex:

```
9 static T& instance(void)
10 {
11     if(!pointer)
12     {
13         boost::mutex::scoped_lock lock(mutex);
14
15         if(!pointer)
16         {
17             pointer = new T();
18         }
19     }
20
21     return *pointer;
22 }
```

This approach is called the Double-Checked Locking Pattern, or DCLP.

The Double-Checked Locking Pattern is Unsafe

Scott Meyers and Andrei Alexandrescu have discussed⁴ the subtle issues with DCLP. Their paper as an interesting read; the problems boil down to the fact that the compiler may

reorder instructions or the hardware may reorder memory accesses, and there was no way to express these constraints using the C++ language at the time (C++11 fixes these issues, as we will see later).

For example, the compiler may do the following:

```
9 static T& instance(void)
10 {
11     if(!pointer)
12     {
13         boost::mutex::scoped_lock lock(mutex);
14
15         if(!pointer)
16         {
17             pointer =          // 3
18                 ::operator new(sizeof(T)); // 1
19             new (pointer) T();      // 2
20         }
21     }
22     return *pointer;
23 }
```

Here, the actual construction is broken down into three steps:

1. Allocating memory for the object
2. Constructing the object
3. Assigning the pointer to the memory

The paper claims that DCLP can only work if steps 1 and 2 are completed before step 3, however the compiler's optimizer is free to perform step 3 after step 1. This means there could be a brief moment where the pointer is pointing to allocated memory, but an unconstructed object. Imagine the following scenario:

1. Thread 1 may get through line 10 (completing steps 1 and 3, but not 2)
2. Thread 2 gets to line 3, passes the check, and skips to line 15, returning an unconstructed object

This type of bug is very subtle and not obvious to the developer. Furthermore, these types of crashes are virtually impossible to track down.

Fixing the Double-Checked Locking Pattern

One way to get around the multi-threaded issues is to have one object for each thread,

instead of one object globally. A per-thread Singleton⁵ can be implemented like this:

```
1  #include <boost/thread.hpp>
2
3  template<typename T>
4  class ThreadSingleton
5  {
6  public:
7      static T& instance(void)
8      {
9          static boost::thread_specific_ptr<T> pointer;
10
11         if(!pointer.get())
12         {
13             pointer.reset(new T());
14         }
15
16         return *pointer;
17     }
18
19 protected:
20
21     ThreadSingleton(void);
22 };
23
24 struct Hello
25 {
26     Hello(void){std::cout << "Hello!" << std::endl;}
27     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
28 };
29
30 void run(void)
31 {
32     Hello& hello = ThreadSingleton<Hello>::instance();
33     Hello& hello2 = ThreadSingleton<Hello>::instance();
34     std::cout << "Thread 2: " << &hello << " : " << &hello2 << std::endl;
35 }
36
37 int main(int argc, char* argv[])
38 {
39     std::cout << "Beginning of main" << std::endl;
40     boost::thread thread = boost::thread(&run);
41     Hello& hello = ThreadSingleton<Hello>::instance();
42     Hello& hello2 = ThreadSingleton<Hello>::instance();
43     std::cout << "Thread 1: " << &hello << " : " << &hello2 << std::endl;
44     thread.join();
45 }
```


Notice how this method is closer to what we initially started with: A single check with no mutex. Because the thread specific pointer will never be accessed concurrently by multiple threads, this solution is inherently thread safe. Here are the results of running this code:

```
Beginning of main
Hello!
Thread 1: 0x17d96e0 : 0x17d96e0
Hello!
Thread 2: 0x7f7c500008c0 : 0x7f7c500008c0
Goodbye!
Goodbye!
```

As you can see, each thread creates (and destroys) its per-thread Singleton. You can see that each Singleton has a different address as well.

Warning

Thread specific pointers should always be static. They use their address as the key to establish uniqueness. A program storing their thread specific pointer on the stack will have issues when two threads reach the same point in the program, because their addresses on the stack will match (remember that each thread has its own stack). This will cause the two pointers to believe they are pointing at the same⁶.

In some cases, this solution is acceptable. Most times, a true global Singleton is desired.

The paper claims that the DCLP can be made safe with the addition of memory fences. These provide guarantees that reads will not be reordered before writes. The C++11 standard includes this capability, and Boost has made it available to C++03. In fact, Boost provides a solution in their documentation⁷ (modified to match the templated format used earlier):

```
1  #include <iostream>
2  #include <boost/atomic.hpp>
3  #include <boost/thread/mutex.hpp>
4
5  template<typename T>
6  class Singleton
7  {
8  public:
9
10     static T& instance(void)
11     {
12         T* pointer = storage.load(boost::memory_order_consume);
13     }
```

```
14  if(!pointer)
15  {
16      boost::mutex::scoped_lock lock(mutex);
17
18      pointer = storage.load(boost::memory_order_consume);
19
20      if(!pointer)
21      {
22          pointer = new T();
23
24          storage.store(pointer, boost::memory_order_release);
25      }
26  }
27
28  return *pointer;
29  }
30
31 private:
32
33  Singleton(void){}
34
35  static boost::atomic<T*> storage;
36  static boost::mutex mutex;
37  };
38
39 template<typename T> boost::atomic<T*> Singleton<T>::storage;
40 template<typename T> boost::mutex Singleton<T>::mutex;
41
42 struct Hello
43 {
44     Hello(void){std::cout << "Hello!" << std::endl;}
45     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
46 };
47
48 int main(int argc, char* argv[])
49 {
50     std::cout << "Beginning of main" << std::endl;
51     Hello& hello = Singleton<Hello>::instance();
52     Hello& hello2 = Singleton<Hello>::instance();
53     std::cout << &hello << " : " << &hello2 << std::endl;
54 }
```

Enhancing the Singleton

We still have the problem that the destructor is never called. The solution to this is to convert the raw pointer into a `scoped_ptr`. However, the atomics can only be used with primitive types. We need to apply some⁸ transformations⁹ to the code to decouple the

atomic from the pointer:

```
1  #include <iostream>
2  #include <boost/atomic.hpp>
3  #include <boost/thread/mutex.hpp>
4
5  template<typename T>
6  class Singleton
7  {
8  public:
9
10     static T& instance(void)
11     {
12         bool created = storage.load(boost::memory_order_consume);
13
14         if(!created)
15         {
16             boost::mutex::scoped_lock lock(mutex);
17
18             created = storage.load(boost::memory_order_consume);
19
20             if(!created)
21             {
22                 pointer = new T();
23
24                 storage.store(true, boost::memory_order_release);
25             }
26         }
27
28         return *pointer;
29     }
30
31 private:
32     Singleton(void){}
33
34     static T* pointer;
35     static boost::atomic<bool> storage;
36     static boost::mutex mutex;
37 };
38
39
40 template<typename T> T* Singleton<T>::pointer;
41 template<typename T> boost::atomic<bool> Singleton<T>::storage;
42 template<typename T> boost::mutex Singleton<T>::mutex;
43
44 struct Hello
45 {
46     Hello(void){std::cout << "Hello!" << std::endl;}
47     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
```

```
48 };
49
50 int main(int argc, char* argv[])
51 {
52     std::cout << "Beginning of main" << std::endl;
53     Hello& hello = Singleton<Hello>::instance();
54     Hello& hello2 = Singleton<Hello>::instance();
55     std::cout << &hello << " : " << &hello2 << std::endl;
56 }
```

Now that the pointer is separated from the atomic, we can replace the pointer with the scoped pointer to get the following:

```
1 // dclp.cpp
2 // Copyright (C) 2015 Joe Ruether jrruethe@gmail.com
3 //
4 // This program is free software: you can redistribute it and/or modify
5 // it under the terms of the GNU General Public License as published by
6 // the Free Software Foundation, either version 3 of the License, or
7 // (at your option) any later version.
8 //
9 // This program is distributed in the hope that it will be useful,
10 // but WITHOUT ANY WARRANTY; without even the implied warranty of
11 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 // GNU General Public License for more details.
13 //
14 // You should have received a copy of the GNU General Public License
15 // along with this program. If not, see <http://www.gnu.org/licenses/>.
16
17 #include <iostream>
18 #include <boost/atomic.hpp>
19 #include <boost/thread/mutex.hpp>
20 #include <boost/scoped_ptr.hpp>
21
22 template<typename T>
23 class Singleton
24 {
25 public:
26
27     static T& instance(void)
28     {
29         bool created = storage.load(boost::memory_order_consume);
30
31         if(!created)
32         {
33             boost::mutex::scoped_lock lock(mutex);
34
35             created = storage.load(boost::memory_order_consume);
```

```

36
37     if(!created)
38     {
39         pointer.reset(new T());
40
41         storage.store(true, boost::memory_order_release);
42     }
43 }
44
45 return *pointer;
46 }
47
48 private:
49
50 Singleton(void){}
51
52 static boost::scoped_ptr<T> pointer;
53 static boost::atomic<bool> storage;
54 static boost::mutex mutex;
55 };
56
57 template<typename T> boost::scoped_ptr<T> Singleton<T>::pointer;
58 template<typename T> boost::atomic<bool> Singleton<T>::storage;
59 template<typename T> boost::mutex Singleton<T>::mutex;
60
61 struct Hello
62 {
63     Hello(void){std::cout << "Hello!" << std::endl;}
64     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
65 };
66
67 int main(int argc, char* argv[])
68 {
69     std::cout << "Beginning of main" << std::endl;
70     Hello& hello = Singleton<Hello>::instance();
71     Hello& hello2 = Singleton<Hello>::instance();
72     std::cout << &hello << " : " << &hello2 << std::endl;
73 }

```

At this point we have a flexible and safe implementation of the DCLP. The output is:

```

Beginning of main
Hello!
0x19626e0 : 0x19626e0
Goodbye!

```

Boost Pool Method

As I mentioned, I needed a Singleton for a memory pool. Naturally, I decided to see how Boost handles this for their own pool. I searched for all the boost include files with Singleton in their name:

```
$ find /usr/include/boost/ -name '*singleton*' -type f

/usr/include/boost/log/detail/singleton.hpp
/usr/include/boost/test/utis/trivial_singleton.hpp
/usr/include/boost/accumulators/numeric/detail/pod_singleton.hpp
/usr/include/boost/thread/detail/singleton.hpp
/usr/include/boost/pool/singleton_pool.hpp
/usr/include/boost/serialization/singleton.hpp
/usr/include/boost/interprocess/detail/windows_intermodule_singleton.hpp
/usr/include/boost/interprocess/detail/intermodule_singleton.hpp
/usr/include/boost/interprocess/detail/intermodule_singleton_common.hpp
/usr/include/boost/interprocess/detail/portable_intermodule_singleton.hpp
```

Then I used my IDE to dive into each one and see how it was implemented:

```
#include <boost/log/detail/singleton.hpp>           // Call once method
#include <boost/test/utis/trivial_singleton.hpp>     // Meyer's method
#include <boost/accumulators/numeric/detail/pod_singleton.hpp> // Static method
#include <boost/thread/detail/singleton.hpp>         // Meyer's method
#include <boost/pool/singleton_pool.hpp>             // Pool method
#include <boost/serialization/singleton.hpp>         // Meyer's method
```

The Meyer's method and the call once method will be discussed later. The static method is simply a wrapper to make something static and isn't particularly interesting:

```
template<typename T>
struct pod_singleton
{
    static T instance;
};

template<typename T>
T pod_singleton<T>::instance;
```

The real interesting method, that I haven't seen anywhere else yet, is the Boost Pool method. The code included in the later versions of Boost (~1.55) is more difficult to read, however a version from 1.39^{[10](#)} is very clear and well commented. The code is repeated below with some minor adjustments:

```
1 // Copyright (C) 2000 Stephen Cleary
2 //
3 // Distributed under the Boost Software License, Version 1.0. (See
4 // accompanying file LICENSE_1_0.txt or copy at
```

```
5 // http://www.boost.org/LICENSE_1_0.txt)
6 //
7 // See http://www.boost.org for updates, documentation, and revision history.
8
9 // The following helper classes are placeholders for a generic "singleton"
10 // class. The classes below support usage of singletons, including use in
11 // program startup/shutdown code, AS LONG AS there is only one thread
12 // running before main() begins, and only one thread running after main()
13 // exits.
14 //
15 // This class is also limited in that it can only provide singleton usage for
16 // classes with default constructors.
17 //
18
19 // The design of this class is somewhat twisted, but can be followed by the
20 // calling inheritance. Let us assume that there is some user code that
21 // calls "singleton_default<T>::instance()". The following (convoluted)
22 // sequence ensures that the same function will be called before main():
23 // instance() contains a call to create_object.do_nothing()
24 // Thus, object_creator is implicitly instantiated, and create_object
25 // must exist.
26 // Since create_object is a static member, its constructor must be
27 // called before main().
28 // The constructor contains a call to instance(), thus ensuring that
29 // instance() will be called before main().
30 // The first time instance() is called (i.e., before main()) is the
31 // latest point in program execution where the object of type T
32 // can be created.
33 // Thus, any call to instance() will auto-magically result in a call to
34 // instance() before main(), unless already present.
35 // Furthermore, since the instance() function contains the object, instead
36 // of the singleton_default class containing a static instance of the
37 // object, that object is guaranteed to be constructed (at the latest) in
38 // the first call to instance(). This permits calls to instance() from
39 // static code, even if that code is called before the file-scope objects
40 // in this file have been initialized.
41
42 // T must be: no-throw default constructible and no-throw destructible
43 template <typename T>
44 struct Singleton
45 {
46     private:
47
48     struct object_creator
49     {
50         // This constructor does nothing more than ensure that instance()
51         // is called before main() begins, thus creating the static
52         // T object before multithreading race issues can come up.
53         object_creator(void)
54     {
```

```
55     Singleton<T>::instance();
56 }
57
58 inline void do_nothing(void) const {}
59 };
60 static object_creator create_object;
61
62 Singleton(void);
63
64 public:
65
66 typedef T object_type;
67
68 // If, at any point (in user code), singleton_default<T>::instance()
69 // is called, then the following function is instantiated.
70 static object_type& instance(void)
71 {
72     // This is the object that we return a reference to.
73     // It is guaranteed to be created before main() begins because of
74     // the next line.
75     static object_type obj;
76
77     // The following line does nothing else than force the instantiation
78     // of singleton_default<T>::create_object, whose constructor is
79     // called before main() begins.
80     create_object.do_nothing();
81
82     return obj;
83 }
84 };
85
86 template <typename T>
87 typename Singleton<T>::object_creator Singleton<T>::create_object;
88
89 #include <iostream>
90
91 struct Hello
92 {
93     Hello(void){std::cout << "Hello!" << std::endl;}
94     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
95 };
96
97 int main(int argc, char* argv[])
98 {
99     std::cout << "Beginning of main" << std::endl;
100     Hello& hello = Singleton<Hello>::instance();
101     Hello& hello2 = Singleton<Hello>::instance();
102     std::cout << &hello << " : " << &hello2 << std::endl;
103 }
```


And the output:

```
Hello!  
Beginning of main  
0x64dc88 : 0x64dc88  
Goodbye!
```

Notice that this method does not use lazy instantiation, but rather it uses *eager instantiation*, meaning the Singleton is created before `main` is called. In addition, it properly calls the destructor and is thread safe without any locks or atomics. Despite the author describing this method as “twisted”, I think it is rather elegant. Unfortunately, it lacks the ability to do per-thread singletons, and its eager instantiation is a drawback.

Meyer’s Method (C++11)

Things get much easier in C++11. Scott Meyers introduced a very simple and elegant Singleton back in 1996^{[11](#)}:

```
1  template<typename T>  
2  class Singleton  
3  {  
4  public:  
5  
6      static T& instance(void)  
7      {  
8          static T singleton;  
9          return singleton;  
10     }  
11  
12 private:  
13     Singleton(void);  
14 };  
15  
16 #include <iostream>  
17  
18 struct Hello  
19 {  
20     Hello(void){std::cout << "Hello!" << std::endl;}  
21     ~Hello(void){std::cout << "Goodbye!" << std::endl;}  
22 };  
23  
24 int main(int argc, char* argv[])  
25 {  
26     std::cout << "Beginning of main" << std::endl;  
27     Hello& hello = Singleton<Hello>::instance();  
28     Hello& hello2 = Singleton<Hello>::instance();
```

```
29  std::cout << &hello << " : " << &hello2 << std::endl;  
30 }
```

Outputs:

```
Beginning of main  
Hello!  
0x64dc78 : 0x64dc78  
Goodbye!
```

This method is only thread safe with C++11 and up. It uses lazy instantiation, properly calls the destructor, and doesn't use any mutexes or atomics. With C++11, it is now considered the correct way to implement a Singleton.

Boost Call Once Method

Meyer's Singleton cannot be used in C++03, but we can get pretty close to it using Boost's `call_once` capabilities.

`boost::call_once`

The `call_once` function and `once_flag` type (statically initialized to `BOOST_ONCE_INIT`) can be used to run a routine exactly once. This can be used to initialize data in a thread-safe manner. [12](#)

```
1  #include <boost/scoped_ptr.hpp>  
2  #include <boost/thread/once.hpp>  
3  #include <boost/noncopyable.hpp>  
4  
5  template<typename T>  
6  class Singleton : private boost::noncopyable  
7  {  
8  public:  
9  
10     static T& instance(void)  
11     {  
12         boost::call_once(&Singleton::create, flag);  
13         return *pointer;  
14     }  
15  
16     protected:  
17  
18     Singleton(void){}  
19  
20     static void create(void)  
21     {
```

```
22     pointer.reset(new T());
23 }
24
25 static boost::scoped_ptr<T> pointer;
26 static boost::once_flag flag;
27 };
28
29 template<typename T>
30 boost::scoped_ptr<T> Singleton<T>::pointer;
31
32 template<typename T>
33 boost::once_flag Singleton<T>::flag = BOOST_ONCE_INIT;
34
35 #include <iostream>
36
37 struct Hello
38 {
39     Hello(void){std::cout << "Hello!" << std::endl;}
40     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
41 };
42
43 int main(int argc, char* argv[])
44 {
45     std::cout << "Beginning of main" << std::endl;
46     Hello& hello = Singleton<Hello>::instance();
47     Hello& hello2 = Singleton<Hello>::instance();
48     std::cout << &hello << " : " << &hello2 << std::endl;
49 }
```

Output:

```
Beginning of main
Hello!
0x1d20370 : 0x1d20370
Goodbye!
```

My Method

I'm a pretty big fan of the `call_once` method:

- Avoids the complications of DCLP
- Aligns with the preferred C++11 standard method of achieving a Singleton
- Works with C++03
- Properly handles destruction
- Properly handles multi-threading
- Employs lazy instantiation

With some clever manipulations, we can also get this to work as a per-thread singleton:

1. Template the pointer type
2. Store the once flag inside a pointer
3. Move the code to a base class
4. Specialize an implementation for each supported pointer type
5. Thread specific pointers should check if their flag is null when getting an instance

```

1  // singleton.cpp
2  // Copyright (C) 2015 Joe Ruether jrruethe@gmail.com
3  //
4  // This program is free software: you can redistribute it and/or modify
5  // it under the terms of the GNU General Public License as published by
6  // the Free Software Foundation, either version 3 of the License, or
7  // (at your option) any later version.
8  //
9  // This program is distributed in the hope that it will be useful,
10 // but WITHOUT ANY WARRANTY; without even the implied warranty of
11 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 // GNU General Public License for more details.
13 //
14 // You should have received a copy of the GNU General Public License
15 // along with this program. If not, see <http://www.gnu.org/licenses/>.
16
17 #include <boost/scoped_ptr.hpp>
18 #include <boost/thread/once.hpp>
19 #include <boost/thread/tss.hpp>
20 #include <boost/noncopyable.hpp>
21
22 namespace detail
23 {
24     ///////////////////////////////////////////////////////////////////
25
26     template<typename T, template<typename U> class P>
27     class SingletonImpl;
28
29     ///////////////////////////////////////////////////////////////////
30
31     template<typename T,
32             template<typename U> class pointer_type>
33     class SingletonBase
34     {
35     protected:
36
37         static T& reference(boost::once_flag& flag)
38         {
39             boost::call_once(&SingletonBase::create, flag);
40             return *pointer;

```

```
41     }
42
43     static void create(void)
44     {
45         pointer.reset(new T());
46     }
47
48     static boost::scoped_ptr<T> pointer;
49 };
50
51 template<typename T, template<typename U> class P>
52 boost::scoped_ptr<T> SingletonBase<T, P>::pointer;
53
54 ///////////////////////////////////////////////////////////////////
55
56 template<typename T>
57 class SingletonImpl<T, boost::scoped_ptr>
58     : public SingletonBase<T, boost::scoped_ptr>
59 {
60 public:
61
62     static T& instance(void)
63     {
64         return SingletonImpl::reference(*once_flag_ptr);
65     }
66
67 protected:
68     static boost::scoped_ptr<boost::once_flag> once_flag_ptr;
69 };
70
71 template<typename T>
72 boost::scoped_ptr<boost::once_flag>
73 SingletonImpl<T, boost::scoped_ptr>::once_flag_ptr(new boost::once_flag());
74
75 ///////////////////////////////////////////////////////////////////
76
77 template<typename T>
78 class SingletonImpl<T, boost::thread_specific_ptr>
79     : public SingletonBase<T, boost::thread_specific_ptr>
80 {
81 public:
82
83     static T& instance(void)
84     {
85         if(!once_flag_ptr.get())
86             once_flag_ptr.reset(new boost::once_flag());
87
88         return SingletonImpl::reference(*once_flag_ptr);
89     }
90 }
```

```

91     protected:
92         static boost::thread_specific_ptr<boost::once_flag> once_flag_ptr;
93     };
94
95     template<typename T>
96     boost::thread_specific_ptr<boost::once_flag>
97     SingletonImpl<T, boost::thread_specific_ptr::once_flag_ptr>
98
99     //////////////////////////////////////
100 }
101
102 template<typename T>
103 class Singleton : public detail::SingletonImpl<T, boost::scoped_ptr>{};
104
105 template<typename T>
106 class ThreadSingleton : public detail::SingletonImpl<T, boost::thread_specific_ptr>{};
107
108 #include <iostream>
109 #include <boost/thread.hpp>
110
111 struct Hello
112 {
113     Hello(void){std::cout << "Hello!" << std::endl;}
114     ~Hello(void){std::cout << "Goodbye!" << std::endl;}
115 };
116
117 void run(void)
118 {
119     Hello& hello = ThreadSingleton<Hello>::instance();
120     Hello& hello2 = ThreadSingleton<Hello>::instance();
121     std::cout << "Thread 2: " << &hello << " : " << &hello2 << std::endl;
122 }
123
124 int main(int argc, char* argv[])
125 {
126     std::cout << "Beginning of main" << std::endl;
127     boost::thread thread = boost::thread(&run);
128     Hello& hello = Singleton<Hello>::instance();
129     Hello& hello2 = Singleton<Hello>::instance();
130     std::cout << "Thread 1: " << &hello << " : " << &hello2 << std::endl;
131     thread.join();
132 }

```

Outputs:

Beginning of main

Hello!

Thread 1: 0x109a700 : 0x109a700

Hello!

Thread 2: 0x7f9528000930 : 0x7f9528000930

Goodbye!

Goodbye!

And there you have it. Sometimes Singletons are good, sometimes they are evil, and sometimes they are difficult to implement correctly. Hopefully now you understand a little more about this pattern.

1. [Singletons Considered Harmful](#) – Kenton Varda↵
2. [Singletons are Pathological Liars](#) – Misko Hevery↵
3. [Singleton Design Pattern](#)↵
4. [C++ and the Perils of Double-Checked Locking](#) – Scott Meyers and Andrei Alexandrescu↵
5. [Two Useful Singleton Templates](#) – Lachlan Orr↵
6. [A problem with boost::thread specific ptr](#) – Edd Dawson↵
7. [Boost Atomic Usage Examples](#)↵
8. [Acquire and Release Fences](#) – Jeff Preshing↵
9. [Is this implementation of DCLP in C++11 correct?](#) – User TCS↵
10. [Boost Pool Singleton](#) – Stephen Cleary↵
11. [The Meyers Singleton](#) – Scott Meyers↵
12. [Boost Call Once](#)↵

Posted by Joe Ruether Aug 2nd, 2015 [C++](#)



About The Author

Joe Ruether is the lead software engineer for the Multisensor Aircraft Tracking system at SRC Inc. He is an expert in C++ template metaprogramming and is interested in cryptography and open source software.

Follow @jrruethe

[« Object Counter](#) [Yaml De/Serialization with Boost Fusion](#) »

Comments

13 Comments Morning Musings

[1 Login](#) ▾[♥ Recommend](#)[🔗 Share](#)[Sort by Best](#) ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS [?](#)**Rob thomasser** • 5 months ago

Do you have a std only version of this? or do I have to use boost. I have an application where I cannot use boost. OR should I just fall back to the Myers implementation?

[^](#) | [v](#) • [Reply](#) • [Share](#) >**Mahesh Attarde** • 2 years ago

Awesome description about singletons. I would wonder, would it make any difference if this static [singleton] data happens to be part of library, so how would initialization and destruction of objects takes place? If you have worked on this corner, i would love to hear about it?

[^](#) | [v](#) • [Reply](#) • [Share](#) >**jrruethe** **Mod** ➔ **Mahesh Attarde** • 2 years ago

As I understand it, the initialization is taken care of by the first call to `::instance()`, and the destruction is handled by the `scoped_ptr` automatically. I haven't yet had this code in its own library (since it is a template, I use it as a header-library instead of a static-library).

[^](#) | [v](#) • [Reply](#) • [Share](#) >**Stefan Reinalter** • 2 years ago

I really would like to know why debug logging and memory pools are a good use case for Singletons? Why can't you have more than one memory pool? Why can't debug logging be simple functions in a namespace? Why does everything always have to be an object?

[^](#) | [v](#) • [Reply](#) • [Share](#) >**jrruethe** **Mod** ➔ **Stefan Reinalter** • 2 years ago

Being a good use case does not imply it is the only way to do it. I'll be posting about memory pools in the future, but one reason having a single memory pool for a type is advantageous is because it keeps memory contiguous. As for objects, my brain thinks in a very object-oriented manner, and it is only natural that my blog posts will reflect that. No where do I say this is the only way to do things.

Contact

jrruethe@gmail.com

keybase.io/jrruethe

Public Key:

[4F40 99F8 276B DBA5 475A](#)

[8446 4630 BEDC 40B9 35FE](#)

Follow @jrruethe { 155 followers }

Link to this page




Related Posts

- [Placement New, Memory Dumps, and Alignment](#)
- [Yaml De/Serialization with Boost Fusion](#)
- [Boost Fusion Json Serializer](#)
- [Cryptography Primer](#)
- [Parameter Forwarding](#)

Recent Posts

- [Cryptography Using OpenSSL](#)
- [Running Docker in Qubes](#)
- [Bitcoin Donation Proofs](#)
- [Bitcoin Donations](#)
- [Object Pool](#)

GitHub Repos



Joe Ruether
@jrruethe

Follow

76
REPOS

4
GISTS

9
FOLLOWERS

Not available for hire.

- [dockerfile](#)

Generates Dockerfiles

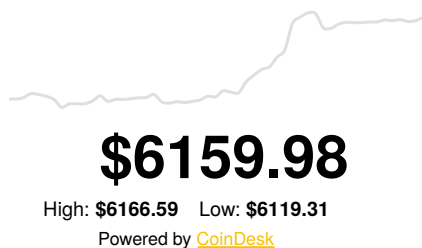
- [interlock-dev](#)

Development environment for Interlock using Docker

Bitcoin Donations



[187gRAhdyD2KaAhMN](#)
[D92RQMqQQMQtW678m](#)
[Proof](#)



Powered by [Octopress](#) - Copyright © 2016 - Joe Ruether - All rights reserved with the following exceptions:

- All embedded code on this page licensed under [GPLv3](#) or a later version unless otherwise noted
- All non-code text/image content on this page licensed under [CC BY-NC-SA 4.0](#) or a later version unless otherwise noted