**Singleton vs. 'Static Class'**

**Pages: 1**[**2**](https://www.cplusplus.com/forum/general/43943/2/)

[](https://www.cplusplus.com/forum/general/43943/#msg237794) May 30, 2011 at 2:10pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

For various reasons I have decided to make my log class a singleton.  
  
I could just go back through and change all the class members to static. And yet I see people discussing other methods for implementing singletons in C++. Are there any advantages to the 'proper' singleton methods as a opposed to the 'static class' method?

[](https://www.cplusplus.com/forum/general/43943/#msg237795) May 30, 2011 at 2:24pm

[**coder777** (8215)](https://www.cplusplus.com/user/coder777/)

when all members are static you're loosing OOP

[](https://www.cplusplus.com/forum/general/43943/#msg237801) May 30, 2011 at 3:17pm

[**kev82** (323)](https://www.cplusplus.com/user/kev82/)

Just to expand on what the previous poster has said,  
  
I would define a singleton to be an object where at most one instance may exist at a time.  
  
By having a class of static functions/vars (essentially a namespace) you don't have a singleton. You don't have full control of construction/destruction, you don't have an object reference to pass around, and on top of this you can't destroy it and create another one with different arguments.  
  
That's not to say a namespace of functions/vars is bad, because it is a good solution in a variety situations. It is not however, a singleton.  
  
There is nothing wrong with your statement  
  
``For various reasons I have decided to make my log class a singleton.''  
  
But I think you really need to sit down and replace the word singleton by the attributes that are important to you. Once you realise what you really mean, you will be able to decide the best way to implement it.

[](https://www.cplusplus.com/forum/general/43943/#msg237803) May 30, 2011 at 3:28pm

[**jsmith** (5804)](https://www.cplusplus.com/user/jsmith/)

A class with nothing but static methods and members can be replaced by a set of free functions declared in a header file and implemented in a .cpp file where all of the static data members become file-scope variables within the .cpp file. The advantage to this transformation is that it is less typing on the part of the user -- they don't have to prefix every function call with a class name.  
  
(One pitfall C++ programmers tend to fall into is that everything has to be a class. There can't be any free functions. This is simply not true. A program can contain many free functions and still be fully OO.)  
  
But, the main drawback I see from the static method approach or its replacement is that I'd expect a logger interface to work similar to std::cout ... that is, I'd expect to be able to use the stream insertion operator (<<) to write data into the log. For this to work, you need a real instance of the class on the left-hand side.

[](https://www.cplusplus.com/forum/general/43943/#msg237815) May 30, 2011 at 4:26pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

Thanks for the advice everyone - I think I shall be making it as a proper singleton now, to implement the insertion operator (I *had* had this planned, but it got sidelined somewhere along the way...).  
  
So the obvious next question is how to implement the singleton. I have seen this one, which superficially seems alright:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | class singleton {  singleton() {}  singleton(const singleton&);  singleton& operator=(const singleton&);  public:  singleton& Get()  {  static singleton sing;  return sing;  }  }; |  |

However, I read somewhere (<http://www.infernodevelopment.com/singleton-c> ) that this is not thread safe.  
  
Assuming the site is correct, we must do this (omitting the unchanged parts):

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 | class singleton {  static singleton\* sing; // initalize to 0 in some cpp file  public:  singleton& Get()  {  if (!sing) sing = new singleton;  return \*sing;  }  }; |  |

So how do I delete this singleton now? Could I add the following inside the singleton:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | static class destructor {  public:  ~destructor() { if (sing) delete sing; }  } dtor; |  |

Or would that have exactly the same thread problem as the static variable in the first solution.  
  
Thanks for any advice you can give me about this.

*Last edited on May 30, 2011 at 4:27pm*

[](https://www.cplusplus.com/forum/general/43943/#msg237827) May 30, 2011 at 4:41pm

[**Disch** (13742)](https://www.cplusplus.com/user/Disch/)

Don't trust that article.  
  
I don't see why the new'd version would be any more thread safe than the non-newed version.  
  
In fact I'm certain the new'd version isn't thread safe at all. The only way to make either of them threadsafe would be to put creation behind a mutex:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | singleton& Get()  {  criticalsection locker( mymutex ); // RAII mutex locker  static singleton sing;  return sing;  }  // ... or ...  singleton& Get()  {  criticalsection locker( mymutex );  static singleton\* sing = new singleton;  return \*sing;  } |  |

I typically don't bother with the new'd approach. Although I use Singletons very sparingly as they're more often than not the incorrect approach to a problem.

*Last edited on May 30, 2011 at 4:42pm*

[](https://www.cplusplus.com/forum/general/43943/#msg237831) May 30, 2011 at 4:47pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

Okay then.  
  
[edit]Firstly, you have mentioned critical sections and mutexes in your code. I am not very familiar with this stuff, and to be honest don't know the difference. In short, would SFML's sf::Mutex be suitable for the job?  
  
However, at that stage we have the problem that every call to Get involves a mutex, which is presumably bad for performance.  
  
I know that when we have a pointer, one can use that 'double checked' method, but clearly that won't work here as there is no null pointer to check for. I suppose that leaves me with the new'd approach.

|  |
| --- |
| I use Singletons very sparingly |

I wasn't too sure about it either. Ultimately I decided that if I didn't make it a singleton then I would have to make undue effort to ensure different log instances weren't writing over each others' files. This wouldn't have been difficult but it seemed somewhat perverse since there isn't likely to be too pressing a need to have multiple logs anyway ;)  
  
However, when I made my resource manager a singleton I did this:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | class ResourceManagerB { /\* a non singleton resource manager \*/ };  class ResourceManagerC : public resourceManagerB {  // singleton stuff  }; |  |

Thus giving the user of the library a choice as to whether to use a singleton version.

*Last edited on May 30, 2011 at 4:48pm*

[](https://www.cplusplus.com/forum/general/43943/#msg237839) May 30, 2011 at 5:06pm

[**Disch** (13742)](https://www.cplusplus.com/user/Disch/)

|  |
| --- |
| Okay then. However, at that stage we have the problem that every call to Get involves a mutex, which is presumably bad for performance. |

If you want to multithread, you need mutexes. There's no way around it. Besides, locking a mutex is only really a hinderance if multiple threads are trying to lock it at the same time (which of course is the whole point of mutexes anyway). If only one thread is doing it, then there's no serious penalty.  
  
But in addition to that, this only makes the *creation* of the object thread safe. In order to make *using* the object threadsafe, you'd have to put acceses behind another mutex.  
  
  
One option for this would be to put accesses inside a host class:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 | // Note this assumes that the same mutex can be locked multiple times from the same thread,  // and that locks are accumulating.  // That is, if you lock a mutex 3 times from the same thread, it won't unlock until it gets unlocked  // 3 times.  //  // If this is not how the mutex class works, you'd have to either wrap it inside another class, or  // you'd have to redesign this class to accomidate  class SingletonAccess  {  private:  friend class Singleton;  Mutex& mutex;  Singleton\* obj;  SingletonAccess(Mutex& m) : mutex(m), obj(0)  {  mutex.Lock();  }  ~SingletonAccess()  {  mutex.Unlock();  }  void operator = (const SingletonAccess&);  public:  SingletonAccess(const SingletonAccess& r) : mutex(r.mutex), obj(r.obj)  {  mutex.Lock();  }  Singleton\* operator -> ()  {  return obj;  }  };  //===========================  class Singleton  {  public:  //...  static SingletonAccess Get()  {  SingletonAccess acc( singletonmutex ); // locks the singleton mutex  static Singleton theobj;  acc.obj = &theobj;  return acc;  }  }; |  |

The trick here is that you don't get a Singleton\* directly, instead you get a SingletonAccess object, which ensures that all accesses to the Singleton are behind a mutex lock, and are therefore threadsafe:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | {  SingletonAccess foo = Singleton::Get(); // locks the singleton mutex  foo->DoSomething(); // thread safe call to a singleton function  } // mutex unlocked here |  |

EDIT:

|  |
| --- |
| Firstly, you have mentioned critical sections and mutexes in your code. I am not very familiar with this stuff, and to be honest don't know the difference. In short, would SFML's sf::Mutex be suitable for the job? |

sf::Mutex is suitable, yes. Although I don't know if it allows for multiple locks from the same thread. That's something you'd have to look up.  
  
A mutex is a device that prevents multiple threads from running sensitive code at the same time. When a mutex is locked in one thread and another thread tries to lock that same mutex, the other thread will have to wait until the first thread unlocks it before it can proceed.  
  
A critical section is just an RAII implementation of a mutex that makes it easier to use and exception safe. The idea is a critical section locks the mutex in its ctor and unlocks it in its dtor, so you don't have to worry about remembering to unlock it when you're done.

*Last edited on May 30, 2011 at 5:12pm*

[](https://www.cplusplus.com/forum/general/43943/#msg237852) May 30, 2011 at 5:45pm

[**kfmfe04** (788)](https://www.cplusplus.com/user/kfmfe04/)

|  |
| --- |
| ***Disch****wrote:* |
| Although I use Singletons very sparingly as they're more often than not the incorrect approach to a problem. |

Agreed 100%  
  
I used to use Singletons until I started multithreading - I looked at a bunch of alternatives including using mutexes (have to worry about deadlocks and context switches), and using alternative design patterns.  
  
In the end, the easiest thing to get rid of all statics (exception: reentrant/stateless static methods are ok), turning it into a plain old class (what the OP did), and simply instantiate at the highest level of the app, passing a handle down to classes that need it.

[](https://www.cplusplus.com/forum/general/43943/#msg238485) Jun 1, 2011 at 9:10am

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

Sorry it's taken me a while to get back to you...

|  |
| --- |
| ***kfmfe****wrote:* |
| what the OP did |

Did I?  
  
@Disch  
What is the need for copying all the mutexes about. Can't you just make the mutex a static member of the access class? (I only ask because sf::Mutex is NonCopyable and I don't really want another library...) Then the singleton doesn't need it's own mutex, as the user would always be accessing through the access class anyway.  
  
This is what I have now come up with. If you have a chance, could you highlight any problems to me. :)  
  
It a template, so if you want to 'singletonize' class Foo to create a singleton called Bar, I guess you would just do typedef Singleton<Foo> Bar;.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | template <class T> class SingletonAccess {  template <class T> friend class Singleton;  static sf::Mutex mutex;  T\* t;  sf::Lock lock; // RAII mutex helper  SingletonAccess(T\* t) : t(t), lock(mutex) {}  public:  T\* operator->()  {  return t;  }  };  template <class T> class Singleton {  public:  static SingletonAccess<T> Get()  {  static T t;  return SingletonAccess<T>(&t);  }  };  template <class T> sf::Mutex Icanos::System::SingletonAccess<T>::mutex; |  |

PS: As you saw in the code, turns out SFML has a RAII thingy to go with the mutex...  
  
EDIT: PPS: I am actually no longer using a singleton for my log (I made a variable static which fixed the problem I mentioned somewhere above about overwritten files). However, I thought I'd still try and get a working singleton template just in case I wanted it in future.  
  
EDIT EDIT: And should I make the SingletonAccess non copyable? Then the user would have to use references to it only, but that wouldn't matter.  
  
EDIT EDIT EDIT: I just looked at the SFML mutex source, and on Windows it just uses critical sections. Thus on Windows at least it obeys the 'nesting' properties that you mentioned.

*Last edited on Jun 1, 2011 at 9:37am*

[](https://www.cplusplus.com/forum/general/43943/#msg238494) Jun 1, 2011 at 10:39am

[**onur** (128)](https://www.cplusplus.com/user/onur/)

Here is what I'll do when I need an object like "the log", "the printer" etc.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 | //1.) Define an interface/abc offering the functionality I need.  class Logger  {  public:  virtual ~Logger();  virtual void log(const std::string message) = 0;  // of course you can also offer a more fancier syntax like "logger << message".  };  //2.) Building a concrete class implementing the interface.  class CoutLogger : public Logger  {  public:  void log(const std::string message) {std::cout << message};  };  //3.) each function/class in need of logging specifies this by adding a logger to the argument list/constructor  myFunction(Logger& logger);  class MyClass  {  public:  MyClass(Logger& logger) : logger\_(logger){};    void doSomething(){logger\_.log("doing something");};  private:  Logger& logger\_;  };  //4.) Create an instance of the class in main and pass it to each function/class that needs an logger  int main()  {  std::auto\_ptr<Logger> theLoggerPtr (new CoutLogger());  // could also be any other logger, or we could decorate this logger, ...  Logger& theLogger = \*theLoggerPtr; // for convenience  myFunction(theLogger);  MyClass myClass(theLogger);  return 0;  } | [Edit & Run](https://www.cplusplus.com/forum/general/43943/) |

Why do I do it this way?  
It's more to code and less convenient to use than the "usual" approach but I gain the following advantages:  
  
\* I don't depend upon global state and state which resources I need  
\* I have control over creation and destruction of the logger especially when it comes to singleton interaction (if logger uses printer and v.v.)  
\* I can have a separate "singleton" logger for each thread that logs to a separate file and therefore have no threading related problems  
\* I can easily pass it to my libraries  
\* I can easily test a function/class by using a special "TestLogger"  
\* I can build a logger depending on a config file for instance at application start  
\* If you name the logger reference consistently in all your classes then it feels like a global singleton logger so there is no added complexity for the main part of your code  
\* You have all the options that oo offers: e.g. decorate your logger (e.g. filter some messages, redirect some to another file etc.)

[](https://www.cplusplus.com/forum/general/43943/#msg238549) Jun 1, 2011 at 3:49pm

[**Disch** (13742)](https://www.cplusplus.com/user/Disch/)

|  |
| --- |
| What is the need for copying all the mutexes about. Can't you just make the mutex a static member of the access class? (I only ask because sf::Mutex is NonCopyable and I don't really want another library...) |

I wasn't copying the mutex, I was taking a reference to it.  
  
The reason for this is because I wanted the singleton to own the mutex since that seemed more logical. Although I suppose you're right, the SingletonAccess object could just as easily own it. That might make things simpler overall (and it does, by the looks of your code)  
  
But then you have to instantiate it.... instantiating a static member of a template class... I'm not sure how that works.

|  |
| --- |
| And should I make the SingletonAccess non copyable? |

No, then you won't be able to return it from the Get() function which would defeat the whole point.  
  
You should make it non-assignable though.

*Last edited on Jun 1, 2011 at 3:49pm*

[](https://www.cplusplus.com/forum/general/43943/#msg238578) Jun 1, 2011 at 6:31pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

|  |
| --- |
| I wasn't copying the mutex, I was taking a reference to it. |

Sorry, yes. I didn't notice the ampersands last time... :/

|  |
| --- |
| won't be able to return it from the Get() function |

 OK, it's a bad idea...  
  
@onur  
Seems like a good design. However, because my class is a library component which should be usable both individually and as part of a standalone engine, there would be conflict as to whether to instantiate the class in the user's code or the engine's code. Thus I shall stick to an (optional) thread safe singleton...

*Last edited on Jun 1, 2011 at 7:37pm*

[](https://www.cplusplus.com/forum/general/43943/#msg238601) Jun 1, 2011 at 7:30pm

[**Disch** (13742)](https://www.cplusplus.com/user/Disch/)

|  |
| --- |
| I was thinking about returning references. Or is that a bad idea? |

Probably. If you're returning a reference to a local object, it is destroyed before the function returns (ie, you get a bad reference from the function).  
  
And if the object isn't local, then when will it be destructed?  
  
The whole point of having the "Access" object is that it is automatically destructed (and the mutex is unlocked) when it goes out of scope.  
  
If you return a reference then the time of destruction isn't clear anymore.

[](https://www.cplusplus.com/forum/general/43943/#msg238606) Jun 1, 2011 at 7:37pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

Okay, yeah forget that what I said above. For some reason I was thinking I about returning the singleton, but of course we are talking about the access object.  
  
Sorry about that :/ I have a tendency to say incredibly stupid stuff which I would disagree with myself if I read it a few minutes later...  
  
I'll just need to add the copy constructor then. Later I'll test the static template member and if that's okay, I'll post the final code for future viewers' reference.  
  
EDIT:

|  |
| --- |
| instantiating a static member of a template class |

The following test code worked fine, so I suppose it's alright.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | #include <iostream>  #include <string>  template <class T> class S {  public:  static int i;  };  template <class T> int S<T>::i = 0;  int main()  {  S<int>::i = 4;  S<char\*>::i = 6;  S<std::string>::i = 8;  std::cout << S<int>::i << ' ' << S<char\*>::i << ' ' << S<std::string>::i << ' ' << S<double>::i << std::endl;  std::cin.get();  } | [Edit & Run](https://www.cplusplus.com/forum/general/43943/) |

|  |
| --- |
| 4 6 8 0 |

*Last edited on Jun 1, 2011 at 7:45pm*

[](https://www.cplusplus.com/forum/general/43943/#msg238622) Jun 1, 2011 at 8:11pm

[**Disch** (13742)](https://www.cplusplus.com/user/Disch/)

|  |
| --- |
| The following test code worked fine, so I suppose it's alright. |

What if you try to use S<int>::i from a different cpp file, though?  
  
You can probably put the instantiation line in the header since it's a template, I'm just not sure if that would work because I can't test it now.

[](https://www.cplusplus.com/forum/general/43943/#msg238626) Jun 1, 2011 at 8:17pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

I set S<int>::i in one file and outputted it in another. Still worked fine.

[](https://www.cplusplus.com/forum/general/43943/#msg238630) Jun 1, 2011 at 8:26pm

[**anonymous23323124** (1383)](https://www.cplusplus.com/user/anonymous23323124/)

Here is my final template singleton design. Thanks to everyone on this thread for their help in its creation, especially Disch :)

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 | // forward declaration of singleton class  template <class T> class Singleton;  // the singleton access class  // to control access to the singleton object and  // protect against multithreading problems with a  // mutual exclusion lock  template <class T> class SingletonAccess {  friend class Singleton<T>;  static sf::Mutex mutex;  T\* t;  // A RAII helper for the mutex (locks on construction and unlocks on destruction)  sf::Lock lock;  SingletonAccess(T\* t) : t(t), lock(mutex) {} // give the access class a pointer to the singleton  SingletonAccess operator=(const SingletonAccess&); // no assignment operator  public:  SingletonAccess(const SingletonAccess& sa) : t(sa.t) {} // default copy constructor (@Disch do I need this? or is it implied)  T\* operator->()  {  return t;  }  };  // the singleton itself  template <class T> class Singleton {  public:  // get a singleton access object to allow thread safe access  // to the singleton object  static SingletonAccess<T> Get()  {  static T t;  return SingletonAccess<T>(&t);  }  };  template <class T> sf::Mutex Icanos::System::SingletonAccess<T>::mutex; |  |

Please do share your opinions, suggestions and improvements. :)  
  
(@**Disch**There is a question for you on line 19 ;) )

*Last edited on Jun 1, 2011 at 8:37pm*

[](https://www.cplusplus.com/forum/general/43943/#msg238646) Jun 1, 2011 at 9:08pm

[**exiledAussie** (131)](https://www.cplusplus.com/user/exiledAussie/)

Dishc wrote:

|  |
| --- |
| A critical section is just an RAII implementation of a mutex that makes it easier to use and exception safe. The idea is a critical section locks the mutex in its ctor and unlocks it in its dtor, so you don't have to worry about remembering to unlock it when you're done. |

Really?  
  
What I have read indicates that Mutexes are executed in kernal mode while critical sections are executed in user mode, with the result that the latter are tens to hundreds of times faster.

[](https://www.cplusplus.com/forum/general/43943/#msg238666) Jun 1, 2011 at 9:42pm

[**Disch** (13742)](https://www.cplusplus.com/user/Disch/)

@exiledAussie:  
  
? Maybe I'm wrong then? That's what I always thought. Could be I've just been thinking the wrong thing. =x  
  
@Xander:  
  
The reason you need the copy ctor is to lock the mutex. If you don't lock the mutex in the copy ctor but unlock it in the dtor you'll have mismatching locks.  
  
The question here is... is sf::Lock copyable? And if so, does it do the nested lock like you'd expect?