**Applicability:**

Use the Singleton pattern when

* there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
* when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

(Now, first point is clear, but second point needs example, which will be provided in future)

**Benefits:**

The Singleton pattern has several benefits:

1. Controlled access to sole instance. Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

2. Reduced name space. The Singleton pattern is an improvement over global

variables. It avoids polluting the name space with global variables that

store sole instances. (So, first impression, suppose, you have a set of const global variables, we might define them in a singleton class and use the sole instance provided by singleton class. Is that so?)

**Also, we have a good reason to use singleton design pattern over having global variables, except the fact that it reduces the name space pollution**

if you assign an object to a global variable, then that object might be created

when your application begins. Right? What if this object is resource intensive and your application never ends up using it? As you will see, with the Singleton Pattern, we can create our objects only when they are needed.

3. Permits refinement of operations and representation. The Singleton class

may be subclassed, and it's easy to configure an application with an instance

of this extended class. You can configure the application with an instance

of the class you need at run-time.

4. Permits a variable number of instances. The pattern makes it easy to change

your mind and allow more than one instance of the Singleton class. Moreover,

you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.

5. More flexible than class operations. Another way to package a singleton's

functionality is to use class operations (that is, static member functions

in C++ or class methods in Smalltalk). But both of these language techniques

make it hard to change a design to allow more than one instance of a class.

Moreover, static member functions in C++ are never virtual, so subclasses

can't override them polymorphically.

**This is a stackoverflow answer regarding why even in java a singleton class is preferred over a class with static members in some cases:**

1.Use a singleton to better control when initialization occurs. With a static class, any initialization must be at class load-time, which you have little control over. For example, a simple reference to a static final MEMBER will trigger class loading. With a singleton, initialization can trivially be deferred till much later - typically, till first time of use.

**Reasons to delay initialization may be:**

it's expensive and you don't always need it for that class

you can't initialize till some other resource is initialized (say, a database connection). In this case, a lazily-instantiated singleton often provides correct order of operations without any explicit control - if it's not referenced till after the other resource is initialized, everything happens for free.

2.Use a singleton to improve testability. If you need to make some kind of mock object (in the broad sense) of the singleton in order to test its clients, one way to do it is to put an interface on its use, and supply a test singleton that's of a different class but implements the same interface.

3.Using a singleton makes initialization testing easier as well.

1. Use a singleton when you might need to debug initialization. Stack traces from static initialization can be puzzling. Debugging can be puzzling too. If the class is loaded early, it may break before a break point on the first line in main() is even hit.

class Singleton

{

public:

static Singleton\* Instance();

protected:

Singleton();

private:

static Singleton\* \_instance;

};

**The corresponding implementation in c++, is**

Singleton\* Singleton::\_instance = 0;

//Now, It could be instantiated with null, too

Singleton\* Singleton::Instance ()

{

if (\_instance == 0)

{

\_instance = new Singleton;

}

return \_instance;

}

**//you remember that fact that static keyword is not used during definition of Singleton function**

**Now, can you can tell me why is the constructor is protected instead of being private, here?**

Because, of the second point of singleton’s applicability:

when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

So, here, the code is written in such a way that the singleton class’s sole instance should be extensible by subclassing.

However, if that is never needed then we can just make the constructor private.

**A sample implementation in java:**

**public class Singleton**

**{**

**private static Singleton uniqueInstance;**

**// other useful instance variables here**

**private Singleton() {}**

**public static Singleton getInstance()**

**{**

**if (uniqueInstance == null)**

**{**

**uniqueInstance = new Singleton();**

**}**

**return uniqueInstance;**

**}**

**// other useful methods here**

**}**

**Implementation And Related Issues:**

Here are implementation issues to consider when using the Singleton pattern:

1. Ensuring a unique instance. The Singleton pattern makes the sole instance

a normal instance of a class, but that class is written so that only one

instance can ever be created. A common way to do this is to hide the operation

that creates the instance behind a class operation (that is, either a static

member function or a class method) that guarantees only one instance is

created. This operation has access to the variable that holds the unique

instance, and it ensures the variable is initialized with the unique

instance before returning its value. **This approach ensures that a singleton**

**is created and initialized before its first use.**

You can define the class operation in C++ with a static member function

Instance of the Singleton class. Singleton also defines a static member

variable \_instance that contains a pointer to its unique instance.

(you can check the code given before this)

Clients access the singleton exclusively through the Instance member function. The variable \_instance is initialized to 0, and the static member function Instance returns its value, initializing it with the unique instance if it is 0. Instance uses lazy initialization; the value it returns isn't created and stored until it's first accessed.

**(Now, obviously, this approach is not thread safe. But, we will discuss it later)**

There's another thing to note about the C++ implementation. It isn't enough

to define the singleton as a global or static object and then rely on

automatic initialization. There are three reasons for this:

1. We can't guarantee that only one instance of a static object will

ever be declared.

1. We might not have enough information to instantiate every singleton

at static initialization time. A singleton might require values that

are computed later in the program's execution.

1. C++ doesn't define the order in which constructors for global objects

are called across translation units [ES90]. This means that no

dependencies can exist between singletons; if any do, then errors

are inevitable.

But, what that that mean?

**class Singleton**

**{**

**public:**

**Singleton();**

**};**

**Singleton single\_instance;**

I think, this particular section and 3 points are discussing about this design. Because, this design matches the flaws (Now, the instance is like created forever instead of being created when we need it. Also, c++ cannot define the constructor calling order for global objects across translational units. Also, the singleton instantiation might require some information which will be computed later. But, here, since, it is declared and instantiated globally, (before the execution of first line of main even), there will be a problem.

1. **Subclassing the Singleton class:** The main issue is not so much defining

the subclass but installing its unique instance so that clients will be

able to use it. In essence, the variable that refers to the singleton

instance must get initialized with an instance of the subclass. The simplest

technique is to determine which singleton you want to use in the Singleton's

Instance operation. An example in the Sample Code shows how to implement

this technique with environment variables.

Another way to choose the subclass of Singleton is to take the implementation

of Instance out of the parent class (e.g., MazeFactory) and put it in the

subclass. That lets a C++ programmer decide the class of singleton at

link-time (e.g., by linking in an object file containing a different

implementation) but keeps it hidden from the clients of the singleton.

The link approach fixes the choice of singleton class at link-time, which

makes it hard to choose the singleton class at run-time. Using conditional

statements to determine the subclass is more flexible, but it hard-wires

the set of possible Singleton classes. Neither approach is flexible enough

in all cases.

A more flexible approach uses a registry of singletons. Instead of having

Instance define the set of possible Singleton classes, the Singleton classes

can register their singleton instance by name in a well-known registry.

The registry maps between string names and singletons. When Instance needs

a singleton, it consults the registry, asking for the singleton by name.

The registry looks up the corresponding singleton (if it exists) and returns

it. This approach frees Instance from knowing all possible Singleton classes

or instances. All it requires is a common interface for all Singleton classes

that includes operations for the registry:

**(this provides the flexibility of choosing the parent singleton class, right?)**

**class Singleton**

**{**

**public:**

**static void Register(const char\* name, Singleton\*);**

**//register function registers the Singleton instance under the given name.**

**//**

**static Singleton\* Instance();**

**protected:**

**static Singleton\* Lookup(const char\* name);**

**private:**

**static Singleton\* \_instance;**

**//a pointer to the class**

**static List<NameSingletonPair>\* \_registry;**

**};**

**Singleton\* Singleton::Instance ()**

**{**

**if (\_instance == 0)**

**{**

**const char\* singletonName = getenv("SINGLETON");**

**// user or environment supplies this at startup**

**\_instance = Lookup(singletonName);**

**// Lookup returns 0 if there's no such singleton**

**}**

**return \_instance;**

**}**

Now, consider this class.

Now, see that this class does not have a constructor. Because, this is a registry class. It need to be instantiated on its own. Instead other single classes register themselves to it. It returns a single instance of any of the registered singleton instance.

(Now, why only lookup function is protected? Because, it will be inherited and can be used by only derived class, right? Is this for the same reason for which we keep the constructor protected in original implementation?when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.)

Register registers the Singleton instance under the given name. To keep

the registry simple, we'll have it store a list of NameSingletonPair objects.

Each NameSingletonPair maps a name to a singleton. The Lookup operation

finds a singleton given its name. We'll assume that an environment variable

specifies the name of the singleton desired.

Where do Singleton classes register themselves? One possibility is in their

constructor. For example, a MySingleton subclass could do the following:

MySingleton::MySingleton()

{

// ...

Singleton::Register("MySingleton", this);

}

Now, could this registration definition is suitable with the original definition

class Singleton

{

public:

static Singleton\* Instance();

protected:

Singleton();

private:

static Singleton\* \_instance;

};

Singleton\* Singleton::Instance ()

{

if (\_instance == 0)

{

\_instance = new Singleton;

}

return \_instance;

}

**Because, in this definition, if we make the registration process of the Singleton to the Registry class, the registration process will be waited upto the first time when a Singleton class’s instance is created.**

We can get around this problem in C++ by defining a static instance of

MySingleton. For example, we can define

**static MySingleton theSingleton;**

in the file that contains MySingleton's implementation.

No longer is the Singleton class responsible for creating the singleton. Instead, its primary responsibility is to make the singleton object of choice accessible in the system. The static object approach still has a potential drawback—namely that instances of all possible Singleton subclasses must be created, or else they won't get registered.

**Thread Safe Singleton:**Consider the following approach of singleton in java:

**public class ASingleton**

**{**

**private static ASingleton instance = null;**

**private ASingleton()**

**{**

**}**

**public static ASingleton getInstance()**

**{**

**if (instance == null)**

**{**

**instance = new ASingleton();**

**}**

**return instance;**

**}**

**}**

Now, this approach is not not thread safe. Multiple threads can access it at the same time and for the first few threads when the instance variable is not initialized, multiple threads can enters the if loop and create multiple instances and break our singleton implementation.

There are three ways through which we can achieve thread safety.

**First:**public class ASingleton

{

private static ASingleton instance =new ASingleton();

private ASingleton()

{

}

public static ASingleton getInstance()

{

return instance;

}

}

**Pros:**

* Thread safety without synchronization
* Easy to implement

**Cons:**

* Early creation of resource that might not be used in the application. (That getInstance function might never be called. That means we might never require the instance of Singleton class. However, here, resource allocation is done for it)
* The client application can’t pass any argument, so we can’t reuse it. For example, having a generic singleton class for database connection where client application supplies database server properties.

**Second:**

Synchronize the getInstance() method  
  
public class ASingleton

{

private static ASingleton instance =null;

private ASingleton()

{

}

public synchronized static ASingleton getInstance()

{

instance=new ASingleton();

return instance;

}

}

**Pros:**

* Thread safety is guaranteed.
* Client application can pass parameters
* Lazy initialization achieved

**Cons:**

* Slow performance because of locking overhead.
* Unnecessary synchronization that is not required once the instance variable is initialized.

**Third: Use synchronized block inside the if block and volatile variable**

**Pros:**

Thread safety is guaranteed

Client application can pass arguments

Lazy initialization achieved

Synchronization overhead is minimal and applicable only for first few threads when the variable is null.

**Cons:**

Extra if condition

**A little modified version of third approach:**

**(which is best)**

**public class ASingleton {**

**private static volatile ASingleton instance;**

**private static Object mutex = new Object();**

**private ASingleton() {**

**}**

**public static ASingleton getInstance() {**

**ASingleton result = instance;**

**if (result == null) {**

**synchronized (mutex) {**

**result = instance;**

**if (result == null)**

**instance = result = new ASingleton();**

**}**

**}**

**return result;**

**}**

**}**

Now, there are some factors which you should check:

**private static volatile ASingleton instance;**

Now, static is a storage class specifier and volatile is a qualifier. So, both can be mixed. Now, why volatile? So, in multi core multithreading environment the threads don’t have a local cached version of this variable.

Local variable result seems unnecessary. But it’s there to improve performance of our code. In cases where instance is already initialized (most of the time), the volatile field is only accessed once (due to “return result;” instead of “return instance;”). This can improve the method’s overall performance by as much as 25 percent.

(because, local variable can be cached to improve performance probably. But, volatile variable cannot be optimized)

Also, note this optimization:

**instance = result = new ASingleton();**

result=instance=new ASingleton() this would make instance accessed twice. And we should reduce the access to volatile.

**Problems With Singleton:**

They are generally used as a global instance, why is that so bad? Because you hide the dependencies of your application in your code, instead of exposing them through the interfaces. Making something global to avoid passing it around is a code smell.

They violate the single responsibility principle: by virtue of the fact that they control their own creation and lifecycle.

They inherently cause code to be tightly coupled. This makes faking them out under test rather difficult in many cases.

They carry state around for the lifetime of the application. Another hit to testing since you can end up with a situation where tests need to be ordered which is a big no no for unit tests. Why? Because each unit test should be independent from the other.