**Singleton Design Pattern**

# Intent

Ensure a class has only one instance, and provide a global point of access to it.

# Motivation

It is important for some classes to have exactly one instance.

How do we ensure that a class has only one instance and that the instance is easily accessible?

A global variable makes an object accessible, but it does not keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance.

The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

# 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.

# Structure



# Participants

**Singleton**

Defines an Instance operation that lets clients access its unique instance. Instance is a class operation (a static member function in C++).

# Collaborations

Clients access a Singleton instance solely through Singleton's Instance operation.

# Consequences

## Controlled access to sole instance

The Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

## Reduced namespace

It avoids polluting the name space with global variables that store sole instances.

## 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++).

However, this makes 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 cannot override them polymorphically.

## 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.

## 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.

# Implementation

The Singleton must not support copy semantics.

## Ensuring a unique instance

The class is written in such a way 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.

In C++

The class operation is a static member function Instance()

The unique instance is static member variable m\_instance, that contains a pointer to Singleton class

Declaration and implementation of Singleton class is:

class Singleton {

public:

static Singleton\* Instance();

protected:

Singleton();

Singleton(const Singleton&); // do not define this

Singleton& operator=(const Singleton&); // do not define this

private:

static Singleton\* m\_instance;

};

Singleton\* Singleton::m\_instance = nullptr;

Singleton::Singleton() { }

Singleton\* Singleton::Instance () {

if (nullptr == m\_instance) {

m\_instance = new Singleton;

}

return m\_instance;

}

Instance uses lazy initialization; the value it returns isn't created and stored until its first accessed.

The constructor is protected. A client that tries to instantiate Singleton directly will get an error at compile-time. This ensures that only one instance can ever get created.

**Drawback of singleton as a global or static object** and then rely on automatic initialization

* We can't guarantee that only one instance of a static object will ever be declared.
* 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.
* C++ does not define the order in which constructors for global objects are called across translation units. This means that no dependencies can exist between singletons; if any do, then errors are inevitable.
* It forces all singletons to be created whether they are used or not.

Using a static member function avoids all of these problems.

## 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.

1. The simplest technique is to determine which singleton you want to use in the Singleton's Instance operation. This can be achieved using environment variables.
2. 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. Using conditional statements to determine the subclass is more flexible, but it hard-wires the set of possible Singleton classes.
3. A more flexible approach uses a registry of singletons.

Singleton \* Singleton::instance() {

if(nullptr == instance) {

const char \* instance\_type = getenv("SINGLETON\_TYPE");

if(0 == strcmp(instance\_type, "SingletonType\_1")) {

instance = new SingletonType\_1;

}

else if(0 == strcmp(instance\_type, "SingletonType\_2")) {

instance = new SingletonType\_2;

}

else if(0 == strcmp(instance\_type, "SingletonType\_3")) {

instance = new SingletonType\_3;

}

else {

instance = new Singleton;

}

return instance;

}

}

# Known Uses

# Related Patterns

Patterns that can be implemented using the Singleton pattern:

Abstract factory, Builder, Prototype

# A few Important Points

Reference (Head First: Design Patterns)

Traditional implementations of getInstance() method could be problematic in case of multithreaded application, because it is difficult to synchronize the getInstance() method

Remedy to this multithreaded problem:

1. Do nothing if the performance of getInstance() isn’t critical to your application
2. Move to an eagerly created instance rather than a lazily created one
3. Use “double-checked locking” to reduce the use of () synchronization in getInstance()

(NOTE: It is not thread-safe in versions before Java 1.5)

If you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances. One way around this problem is to specify the classloader yourself.

If you are using a JVM earlier than 1.2, you’ll need to create a registry of Singletons to defeat the garbage collector.

**Static Initialization Order Fiasco**

Static variables in one translation unit are initialized according to their definition order. In contrast, the initialization of static variables between translation units has a severe issue. When one static variable staticA is defined in one translation unit and another static variable staticB is defined in another translation unit, and staticB needs staticA to initialize itself, you end up with the static initialization order fiasco. The program is ill-formed because you have no guarantee which static variable is initialized first at run time.

## Meyers Singleton

static MeyersSingleton& getInstance(){

static MeyersSingleton instance; // (1)

return instance;

}

Static variables with local scope are created when they are used the first time. This lazy initialization is a guarantee that C++98 provides.

Instead of a static instance of type Singleton, it has a local static of type Singleton.

Since C++11, static variables with local scope are also initialized in a thread-safe way. This means that the Meyers Singleton does not only solve the static initialization order fiasco, but also guarantees that the Singleton is initialized in a thread-safe way.

## Hidden Dependency

A Singleton introduces a hidden dependency and breaks, therefore, testability.

void func() {

...

DataBase::getInstance().update("something");

...

}

The caller of the function func has no idea that a database is called internally. What are the consequences? The code is no unit anymore and, therefore, not unit-testable. You cannot test this code in isolation.

Solution, restructure the code.

func(DataBaseSingleton::getInstance());

...

void func(DataBase& db) {

...

db.update("something");

...

}

Just make the DataBase part of the interface of the function. Now, there is no hidden dependency anymore. The function can be fast and without side effects.

# Example Singleton in C++: Before and after

## Before

A global variable is default initialized - when it is declared - but it is not initialized in earnest until its first use. This requires that the initialization code be replicated throughout the application.

#include <iostream>

using namespace std;

class GlobalClass {

          int m\_value;

public:

          GlobalClass(int v = 0) {        m\_value = v;          }

int get\_value() {        return m\_value;      }

void set\_value(int v) {        m\_value = v;          }

};

// Default initialization

GlobalClass \*global\_ptr = 0;

void foo(void) {

          // Initialization on first use

          if (!global\_ptr)

                   global\_ptr = new GlobalClass;

          global\_ptr->set\_value(1);

          cout << "foo: global\_ptr is " << global\_ptr->get\_value() << endl;

}

void bar(void) {

          if (!global\_ptr)

                   global\_ptr = new GlobalClass;

          global\_ptr->set\_value(2);

          cout << "bar: global\_ptr is " << global\_ptr->get\_value() << endl;

}

int main() {

          if (!global\_ptr)

                   global\_ptr = new GlobalClass;

          cout << "main: global\_ptr is " << global\_ptr->get\_value() << endl;

          foo();

          bar();

}

Output:

main: global\_ptr is 0

foo: global\_ptr is 1

bar: global\_ptr is 2

## After

Make the class responsible for its own global pointer and "initialization on first use" (by using a private static pointer and a public static accessor method). The client uses only the public accessor method.

#include <iostream>

using namespace std;

class GlobalClass {

int m\_value;

static GlobalClass \*s\_instance;

GlobalClass(int v = 0) { m\_value = v; }

public:

int get\_value() { return m\_value; }

void set\_value(int v) { m\_value = v; }

static GlobalClass \*instance() {

if (!s\_instance)

s\_instance = new GlobalClass;

return s\_instance;

}

};

// Allocating and initializing GlobalClass's

// static data member.  The pointer is being allocated - not the object itself.

GlobalClass \*GlobalClass::s\_instance = nullptr;

void foo(void) {

          GlobalClass::instance()->set\_value(1);

          cout << "foo: global\_ptr is " << GlobalClass::instance()->get\_value() << endl;

}

void bar(void) {

          GlobalClass::instance()->set\_value(2);

          cout << "bar: global\_ptr is " << GlobalClass::instance()->get\_value() << endl;

}

int main() {

          cout << "main: global\_ptr is " << GlobalClass::instance()->get\_value() << endl;

          foo();

          bar();

          return 0;

}

Output:

main: global\_ptr is 0

foo: global\_ptr is 1

bar: global\_ptr is 2

# References

Book: Design Patterns Elements of Reusable Object-Oriented Software

Book: Head First: Design Patterns

<https://sourcemaking.com/design_patterns/singleton>

<https://www.modernescpp.com/index.php/creational-patterns-singleton>

<http://www.modernescpp.com/index.php/singleton-pros-and-cons>

# END