Creational Design Pattern

Hung Tran

Fpt software

October 21, 2021

Outline

Creational Pattern Overview

Construction process of an object.

- **Singleton**: Ensure only one instance.
- **Factory Method**: Create instance without depending on its concrete type.
- **Object pool**: Reuse existing instances.
- **Abstract factory**: Create instances from a specific family.
- **Prototype**: Clone existing objects from a prototype.
- **Builder**: Construct a complex object step by step.

Why we need Singleton Design Pattern?

- A component manages the underlying resources such as database connection, application configuration.
- The class should have only one instance.
- Multiple instances will store its own state. When one instance modifies resource, the other instances will not know about it.
- The state of underlying resource may get corrupted or may fail to provide the service.

That class should have only one instance.

The Intent of Singleton Design Pattern

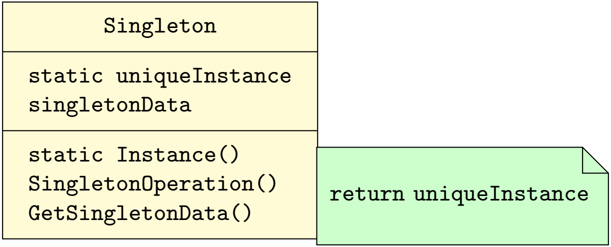
Ensure a class only has one instance, and provide a global point of access to it.

Singleton instance behaves like global variable? Yes!

How to implement Singleton Design Pattern?

- The class is made responsible for its own instance.
- It intercepts the call for construction and returns a single instance.
- Same instance is returned every time.
- A direct construction of object is disabled.
- The class creates its own instance which is provided to the clients.

Structure of Singleton Pattern



Basic implementation

Singleton.h

```cpp
#ifndef SINGLETON_H
#define SINGLETON_H

class Singleton
{
  // First: disable the construction by
      making constructor private.
  Singleton() = default; // faster than
      user-define
  // Second: create the static instance
      of class.
  static Singleton m_Instance;
public:
  // Third: Using instance method
      provides m_Instance to the clients.
  static Singleton & Instance() ;
  void MethodA() ;
  void MethodB() ;
};
#endif
```

Singleton.cpp

```cpp
#include <iostream>
#include "Singleton.h"

Singleton Singleton::m_Instance ;
Singleton& Singleton::Instance() {
  std::cout << "Static instance was
      created!\n";
  return m_Instance ;
}

void Singleton::MethodA() {
}

void Singleton::MethodB() {
}
```

main.cpp

```cpp
#include "Singleton.h"

int main() {
  Singleton &s = Singleton::Instance() ;
  s.MethodA() ;

  //Singleton s2 ;
}
```

Logger1 class: Creating two instance

Logger.h

```
#ifndef LOGGER_H
#define LOGGER_H
#include <cstdio>
#include <string>
class Logger {
  FILE *m_pStream;
  std::string m_Tag;
public:
  Logger();
  ~Logger();
  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```
#include "Logger.h"
#include <iostream>
Logger::Logger() {
  m_pStream = fopen("applog.txt", "w");
  std::cout << "Logger::Logger()" << std
      ::endl;
}

Logger::~Logger() {
  fclose(m_pStream);
  std::cout << "Logger::~Logger()" <<
      std::endl;
}

void Logger::WriteLog(const char*
    pMessage) {
  fprintf(m_pStream, "[%s] %s\n", m_Tag.
      c_str(), pMessage);
  fflush(m_pStream);// to ensure that
      the messages are always returned to
       the log file
}

void Logger::SetTag(const char* pTag) {
  m_Tag = pTag;
}
```

Logger1 class: Creating two instance

main.cpp

```cpp
#include "Logger.h"
#include <iostream>
// Create an other instance
void OpenConnection() {
  Logger lg;
  lg.WriteLog("Attempting to open a connection");
}
int main() {
  std::cout << "main() invoked" << std::endl;
  Logger lg;
  lg.SetTag("192.168.1.101");
  lg.WriteLog("Application has started");
  OpenConnection();
  lg.WriteLog("Application is shutting down");
  return 0;
}
```

Logger1 class: Problem

- **Problem**: Two instances are created and constructor of each instance attempts to open the file in write mode. The stream is already open. When another instance tries to open it, it may either fail or succeed. We do not know, **the behavior is undefined**.
- **Solution**:
- Need to ensure that there is only one instance of the logger.
- Need to prevent the user from creating instances of this class.

Solution for Logger1 class

- Make constructor private
- Create the static Logger instance
- Create the static method

Logger2 class: solve the Logger1 class's problem

Logger.h

```cpp
#ifndef LOGGER_H
#define LOGGER_H
#include <cstdio>
#include <string>
class Logger
{
  FILE *m_pStream;
  std::string m_Tag;
  Logger();
  static Logger m_Instance;
public:
  static Logger & Instance();
  ~Logger();

  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```cpp
#include "Logger.h"
#include <iostream>
Logger Logger::m_Instance;
Logger::Logger() {
  m_pStream = fopen("applog.txt", "w");
  std::cout << "Logger::Logger()" << std::endl;
}
Logger& Logger::Instance() {
  return m_Instance;
}
Logger::~Logger() {
  fclose(m_pStream);
  std::cout << "Logger::~Logger()" << std::endl;
}
void Logger::WriteLog(const char* pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.c_str(), pMessage);
  fflush(m_pStream);
}
void Logger::SetTag(const char* pTag) {
  m_Tag = pTag;
}
```

Logger2 class: solve the Logger1 class's problem

main.cpp

```cpp
#include "Logger.h"
#include <iostream>
void OpenConnection() {
  Logger lg = Logger::Instance() ;
  //Logger &lg = Logger::Instance();
  lg.WriteLog("Attempting to open a connection");
}

int main() {
  std::cout << "main() invoked" << std::endl;
  //Logger lg = Logger::Instance();
  Logger &lg = Logger::Instance();
  lg.SetTag("192.168.1.101");
  lg.WriteLog("Application has started");

  OpenConnection();
  lg.WriteLog("Application is shutting down");
}
```

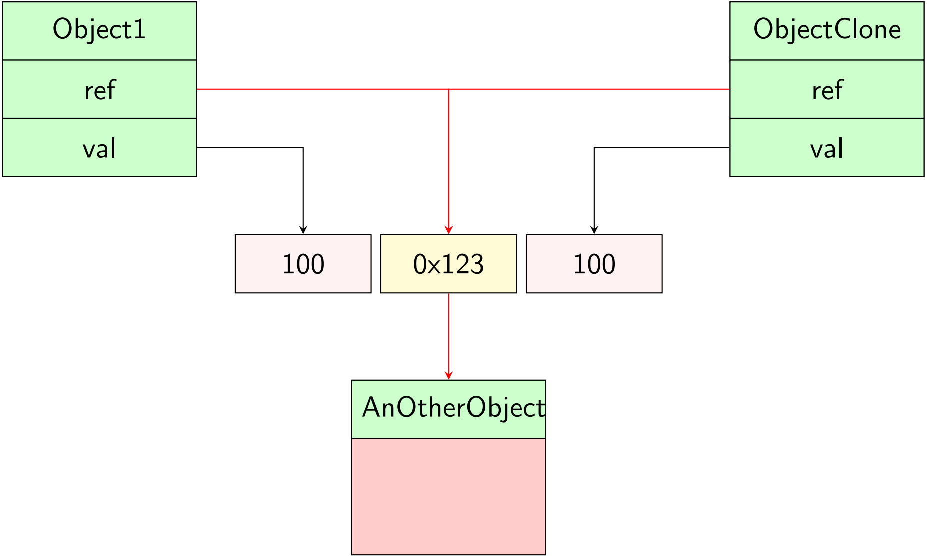
Logger2 class: Problem

- What if the users **do not use the reference** to get the instance of class?
- *lg* is a concrete object so it is initialized through **the copy constructor**.
- The compiler will synthesize its own copy constructor that will perform a **shallow copy**.
- Therefore the m_pStream pointer will be copied into the *lg* object.
- There are now two objects that are sharing **the same stream pointer**.
- At the end of scope of OpenConnection() the local object *lg* was destroyed and the destructor will close the stream and the stream pointer in the local object is left dangling.

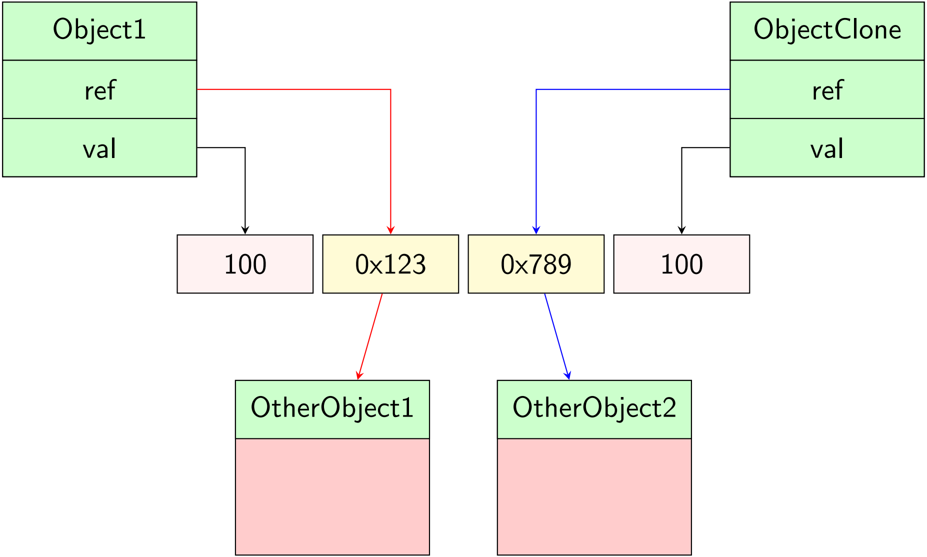
Shadow copy

- A copy is created by copying the state of the object.
- Programming languages support this feature through cloning/copy constructor.
- The default implementation of these methods will copy the references in the object instead of copying the actual data.
- This called shadow copy

Shadow copy


Object1
ref
val
ObjectClone
ref
val
100
0x123
100
AnOtherObject

Deep copy


Object1
ref
val
ObjectClone
ref
val
100
0x123
0x789
100
OtherObject1
OtherObject2

Solution for Logger2 class

- We need to **prevent** the users from creating copied of the object.
- Declare the constructor with **delete** modifier.
- Similarly to assignment operator because using assignment can create a copy of existing object.
- Or, you can do this by declaration of these functions in the private section.

Logger3 class: Prevent user from creating a copy instance

Logger.h

```cpp
#ifndef LOGGER_H
#define LOGGER_H

#include <cstdio>
#include <string>

class Logger
{
  FILE *m_pStream; // file IO
  std::string m_Tag;

  Logger();
  static Logger m_Instance;
public:
  //Logger(const Logger&) = delete;
  //Logger & operator =(const Logger &)
     = delete;
  static Logger & Instance();
  ~Logger();

  void WriteLog(const char *pMessage);
     // write on the log file
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```cpp
#include "Logger.h"

Logger Logger::m_Instance;

Logger::Logger() {
  m_pStream = fopen("applog.txt", "w") ;
}

Logger& Logger::Instance() {
  return m_Instance;
}

Logger::~Logger() {
  fclose(m_pStream);
}

void Logger::WriteLog(const char*
     pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.
     c_str(), pMessage);
  fflush(m_pStream);
}

void Logger::SetTag(const char* pTag) {
  m_Tag = pTag;
}
```

Logger3 class: Prevent user from creating a copy instance

main.cpp

```cpp
#include "Logger.h"

void OpenConnection() {
  Logger lg = Logger::Instance();
  //Logger &lg = Logger::Instance();
  lg.WriteLog("Attempting to open a connection");
}


int main() {
  Logger lg = Logger::Instance();
  //Logger &lg = Logger::Instance();

  lg.SetTag("192.168.1.101");
  lg.WriteLog("Application has started");

  OpenConnection();
  lg.WriteLog("Application is shutting down");
}
```

Logger3 class: Problem

- If the user try to create a copy of instance, error prone due to copy constructor
- How to avoid that problem?
- Returning a pointer?

Problem of returning a pointer

- The user can make a copy by dereference operator
- Implement "if" condition for check "null" pointer
- Rule of Three: copy constructor, assignment operator, destructor
- Rule of Five: move constructor, move assignment but why we do not need?

Lazy Singleton

- The above implementations, the instance is created before main invoked.
- The instances called eager.
- However, we need instance created when we want, after the instance method invoked?
- Using lazy instance

How to implement lazy instance?

- Need a pointer variable
- In Instance() method, implement return a pointer
- To avoid multiple instances, should put if condition for null check in Instance() method.

Lazy Singleton (lazy1 class)

Logger.h

```
#ifndef LOGGER_H
#define LOGGER_H
#include <cstdio>
#include <string>

class Logger
{
  FILE *m_pStream;
  std::string m_Tag;
  Logger();
  static Logger *m_pInstance;
public:
  Logger(const Logger&) = delete;
  ~Logger();
  Logger & operator =(const Logger &) =
      delete;
  static Logger & Instance();
  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```
#include "Logger.h"
#include <iostream>
Logger *Logger::m_pInstance;

Logger::Logger() {
  std::cout << "Logger::Logger()" << std
      ::endl;
  m_pStream = fopen("applog.txt", "w");
}

Logger& Logger::Instance() {
  if (m_pInstance == nullptr)
    m_pInstance = new Logger{};
  return *m_pInstance;
}

Logger::~Logger() {
  std::cout << "Logger::~Logger()" <<
      std::endl;
  fclose(m_pStream);
}

void Logger::WriteLog(const char*
      pMessage) {
  fprintf(m_pStream, "[%s] %s\n", m_Tag.
      c_str(), pMessage);
  fflush(m_pStream);
}
```

Lazy Singleton (lazy1 class)

main.cpp

```cpp
#include "Logger.h"
#include <iostream>
void OpenConnection() {
  Logger &lg = Logger::Instance() ;
  lg.WriteLog("Attempting to open a connection") ;
}

int main() {
  std::cout << "main() called" << std::endl;
  Logger &lg = Logger::Instance() ;
  lg.SetTag("192.168.1.101") ;
  lg.WriteLog("Application has started") ;

  OpenConnection() ;
  lg.WriteLog("Application is shutting down");
  return 0;
}
```

lazy1 class: Problem

- We did not see the destructor called.
- We can not delete the **m_pInstance** in Instance() method at the end of the main function.
- We do not have a pointer to the instance
- We can get a pointer to lg object in main but it is bad idea to call delete on none pointer.
- How to ensure that the instance will be deleted after the main returned.

Destruction Policies

- There are two ways:
- First, using **the smart pointer**, if the user create a pointer to lg and delete it, the behavior is undefined. Instead we can write our own deleter.
- Second, we can use **atexit()** function.

Using smart pointer

Logger.h

```cpp
#ifndef LOGGER_H
#define LOGGER_H

#include <cstdio>
#include <string>
#include <memory>

class Logger {
  struct Deleter {
    void operator()(Logger *p) {
      delete p;
    }
  };
  FILE *m_pStream;
  std::string m_Tag;
  Logger();
  inline static std::unique_ptr<Logger, Deleter> m_pInstance {};
  ~Logger();
public:
  Logger(const Logger&) = delete;
  Logger & operator=(const Logger &) = delete;
  static Logger & Instance();
  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```cpp
#include "Logger.h"
#include <iostream>
Logger::Logger() {
  std::cout << "Logger::Logger() invoked " << std::endl;
  m_pStream = fopen("applog.txt", "w");
}
Logger& Logger::Instance() {
  if(m_pInstance == nullptr)
    m_pInstance.reset(new Logger{});
  return *m_pInstance;
}
Logger::~Logger() {
  std::cout << "Logger::~Logger() invoked" << std::endl;
  fclose(m_pStream);
}
void Logger::WriteLog(const char* pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.c_str(), pMessage);
  fflush(m_pStream);
}
void Logger::SetTag(const char* pTag) {
  m_Tag = pTag;
}
```

Using smart pointer

main.cpp

```
#include "Logger.h"

void OpenConnection() {
  Logger &lg = Logger::Instance();
  lg.WriteLog("Attempting to open a connection");
}

int main() {
  Logger &lg = Logger::Instance();
  lg.SetTag("192.168.1.101");
  lg.WriteLog("Application has started");
  OpenConnection();
  lg.WriteLog("Application is shutting down");
  //auto *p = &lg ;
  //delete p ;
}
```

Using atexit()

Logger.h

```cpp
#ifndef LOGGER_H
#define LOGGER_H

#include <cstdio>
#include <string>
#include <mutex>

class Logger
{
  FILE *m_pStream;
  std::string m_Tag;
  Logger();
  static Logger *m_pInstance;
  ~Logger();
public:
  Logger(const Logger&) = delete;
  Logger & operator =(const Logger &) =
      delete;

  static Logger & Instance();
  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```cpp
#include "Logger.h"
#include <iostream>
Logger *Logger::m_pInstance;
Logger::Logger() {
  std::cout << "Logger::Logger() invoked
      " << std::endl;
  m_pStream = fopen("applog.txt", "w");
  // atexit() is c runtime function
  atexit([]() {
  delete m_pInstance;
  }) ;
}
Logger& Logger::Instance() {
  if(m_pInstance == nullptr)
    m_pInstance = new Logger{};
  return *m_pInstance ;
}
Logger::~Logger() {
  std::cout << "Logger::~Logger()
      invoked" << std::endl;
  fclose(m_pStream);
}
void Logger::WriteLog(const char*
      pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.
      c_str(), pMessage);
  fflush(m_pStream);
}
```

Using atexit()

main.cpp

```
#include "Logger.h"

void OpenConnection() {
  Logger &lg = Logger::Instance();
  lg.WriteLog("Attempting to open a connection");
}
int main() {
  Logger &lg = Logger::Instance();
  lg.SetTag("192.168.1.101");
  lg.WriteLog("Application has started");
  OpenConnection();
  lg.WriteLog("Application is shutting down");
  //auto *p = &lg ;
  //delete p ;
}
```

Static initialization fiasco?

Multiplethreads issue

Logger.h

```
#ifndef LOGGER_H
#define LOGGER_H
#include <cstdio>
#include <string>

class Logger
{
  FILE *m_pStream;
  std::string m_Tag;
  Logger();
  static Logger *m_pInstance;
  ~Logger();

public:
  Logger(const Logger&) = delete;
  Logger & operator =(const Logger &) = delete;

  static Logger & Instance();
  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```
#include "Logger.h"
#include <iostream>
Logger *Logger::m_pInstance;
Logger::Logger() {
  std::cout << "Logger::Logger()" << std::endl;
  m_pStream = fopen("applog.txt", "w");
  atexit([]() {
    delete m_pInstance;
  });
}
Logger& Logger::Instance() {
  if(m_pInstance == nullptr)
    m_pInstance = new Logger{};
  return *m_pInstance;
}
Logger::~Logger() {
  std::cout << "Logger::~Logger()" << std::endl;
  fclose(m_pStream);
}
void Logger::WriteLog(const char* pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.c_str(), pMessage);
  fflush(m_pStream);
}
void Logger::SetTag(const char* pTag) {
```

Multiplethreads issue

main.cpp

```cpp
#include "Logger.h"
#include <thread>

void OpenConnection() {
  Logger &lg = Logger::Instance();
  lg.WriteLog("Attempting to open a connection");
}

int main() {
  std::thread t1{[](){
    Logger &lg = Logger::Instance();
    lg.WriteLog("Thread 1 has started!");
    }
  };

  std::thread t2{[](){
    Logger &lg = Logger::Instance();
    lg.WriteLog("Thread 2 has started!");
    }
  };

  t1.join();
  t2.join();

  return 0;
}
```

Static initialization fiasco?

Multiplethreads issue

- We can see that two instances is created
- Solution: we can use lock()

Using mutex::lock() to solve multiplethreads issue

Logger.h

```cpp
#ifndef LOGGER_H
#define LOGGER_H
#include <cstdio>
#include <string>
#include <mutex>
class Logger
{
  static std::mutex m_Mtx;
  FILE *m_pStream;
  std::string m_Tag;
  Logger();
  static Logger *m_pInstance;
  ~Logger();
public:
  Logger(const Logger&) = delete;
  Logger & operator =(const Logger &) =
      delete;

  static Logger & Instance();
  void WriteLog(const char *pMessage);
  void SetTag(const char *pTag);
};
#endif
```

Logger.cpp

```cpp
#include "Logger.h"
#include <iostream>
Logger *Logger::m_pInstance;
std::mutex Logger::m_Mtx;
Logger::Logger() {
  std::cout << "Logger::Logger()" << std
      ::endl;
  m_pStream = fopen("applog.txt", "w");
  atexit([]() {
    delete m_pInstance;
  });
}
Logger& Logger::Instance() {
  m_Mtx.lock();
  if(m_pInstance == nullptr)
    m_pInstance = new Logger{};
  m_Mtx.unlock();
  return *m_pInstance;
}
Logger::~Logger() {
  std::cout << "Logger::~Logger()" <<
      std::endl;
  fclose(m_pStream);
}
void Logger::WriteLog(const char*
      pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.
      c_str(), pMessage);
```

Using mutex::lock() to solve multiplethreads issue

main.cpp

```
#include "Logger.h"
#include <thread>

void OpenConnection() {
  Logger &lg = Logger::Instance();
  lg.WriteLog("Attempting to open a connection");
}

int main() {
  std::thread t1{[](){
    Logger &lg = Logger::Instance();
    lg.WriteLog("Thread 1 has started!");
    }
  };

  std::thread t2{[](){
    Logger &lg = Logger::Instance();
    lg.WriteLog("Thread 2 has started!");
    }
  };

  t1.join();
  t2.join();

  return 0;
}
```

Problem of mutex::lock

If m_pInstance is not null

- Thread does not perform creating new instance.
- The others do not have to wait.

Using null check before lock()?

double-check locking pattern to handle the issue

Logger.cpp

```cpp
#include "Logger.h"
#include <iostream>
Logger *Logger::m_pInstance;
std::mutex Logger::m_Mtx;
Logger::Logger() {
  std::cout << "Logger::Logger() invoked" << std::endl;
  m_pStream = fopen("applog.txt", "w");
  atexit([]() {
    delete m_pInstance;
  });
}
Logger& Logger::Instance() {
  if(m_pInstance == nullptr){ // double-check locking pattern
    m_Mtx.lock();
    if(m_pInstance == nullptr)
      m_pInstance = new Logger{};
    m_Mtx.unlock();
  }
  return *m_pInstance;
}
Logger::~Logger() {
  std::cout << "Logger::~Logger() invoked" << std::endl;
  fclose(m_pStream);
}
void Logger::WriteLog(const char* pMessage){
  fprintf(m_pStream, "[%s] %s\n", m_Tag.c_str(), pMessage);
  fflush(m_pStream);
}
void Logger::SetTag(const char* pTag) {
```

double-check locking pattern issue

The statement m_pInstance = new Logger; is performed in three steps:

- Memory is allocated
- It is initialized
- Memory address is assigned

Therefore, the first thread is executing the above statement, the other thread may read from it and get invalid result.

So double-check locking pattern fails.

double-check locking pattern issue

Logger.cpp

```cpp
#include "Logger.h"
#include <iostream>

Logger *Logger::m_pInstance;
std::mutex Logger::m_Mtx;

Logger::Logger() {
  std::cout << "Logger::Logger() invoked" << std::endl;
  m_pStream = fopen("applog.txt", "w");
  atexit([]() {
  delete m_pInstance;
  });
}

// static method can only acess static member => mutex object has to be
// static.
Logger& Logger::Instance() {
  if(m_pInstance == nullptr){ // double-check locking pattern
    m_Mtx.lock();
    if(m_pInstance == nullptr){
      /*
      void *p = operator new (sizeof(Logger));
      new(p)Logger{};
      m_pInstance = static_cast<Logger*>(p);
      */

      void *p = operator new (sizeof(Logger));
      m_pInstance = static_cast<Logger*>(p);
      new(p)Logger{};
```

Meyer singleton

Create static instance inside the Instance() method.

- Thread safe because of static member
- No worry about how to delete instance
- Eager instance

call_one method

- Thread safe
- Meyer or call_one? Meyer is more efficient

Curiously Recurring Template Pattern Singleton

- You may have multiple classes behave like singleton, so you will have implement singular behavior in all classes.
- This pattern will reduce effort to make multiple singleton classes by using inheritance property.

Clock class

clock.h

```cpp
#ifndef CLOCK_H_
#define CLOCK_H_
#include <string>
class Clock {
  int m_Hour;
  int m_Minute;
  int m_Second;
  void CurrentTime();
public:
  Clock();
  int GetHour();
  int GetMinute();
  int GetSecond();
  std::string GetTimeString();
};
#endif // CLOCK_H
```

clock.cpp

```cpp
#include "clock.h"
#include <ctime>
#include <sstream>

void Clock::CurrentTime() {
  time_t raw_time;
  time(&raw_time);
  tm *local_time = localtime(&raw_time);
  m_Hour = local_time->tm_hour;
  m_Minute = local_time->tm_min;
  m_Second = local_time->tm_sec;
}

Clock::Clock() {
  CurrentTime();
}

int Clock::GetHour() {
  CurrentTime();
  return m_Hour;
}

int Clock::GetMinute() {
  CurrentTime();
  return m_Minute;
}

int Clock::GetSecond() {
```

Clock class

main.cpp

```
#include "clock.h"
#include <iostream>

int main() {
  Clock clk;
  std::cout << clk.GetTimeString() <<
      std::endl;
}
```

- The above implementation can create multiple instances (they have their own state) but they are the same state.
- What if we create 1000 instances leading to memory consumption.

Mono state: Singleton-like

clock.h

```
#ifndef CLOCK_H_
#define CLOCK_H_

#include <string>
class Clock
{
  // using inline key word => do not
     have to define them outside class
  inline static int m_Hour;
  inline static int m_Minute;
  inline static int m_Second;
  void CurrentTime();
public:
  Clock();
  int GetHour();
  int GetMinute();
  int GetSecond();
  std::string GetTimeString();
};
#endif // CLOCK_H
/*
 * earlier: if three instances are
     created => three different
     attributes
 * c1 - h m s
 * c2 - h m s
 * c3 - h m s
 *
```

clock.cpp

```
#include "Clock.h"
#include <ctime>
#include <sstream>

void Clock::CurrentTime() {
  time_t raw_time;
  time(&raw_time);
  tm *local_time = localtime(&raw_time);
  m_Hour = local_time->tm_hour;
  m_Minute = local_time->tm_min;
  m_Second = local_time->tm_sec;
}

Clock::Clock() {
  CurrentTime();
}

int Clock::GetHour() {
  CurrentTime();
  return m_Hour;
}

int Clock::GetMinute() {
  CurrentTime();
  return m_Minute;
}

int Clock::GetSecond() {
```

Mono state: Singleton-like

main.cpp

```cpp
#include "Clock.h"
#include <iostream>

int main() {
  Clock clk;
  std::cout << clk.GetTimeString() <<
      std::endl;
  return 0;
}
```

- What if we create 1000 instances leading to multiple instance sharing the same attributes. This making of illusion of multiple instances.
- So make construction is private.
- However, they can not invoke the method. Therefore making all methods static.

Mono state: Singleton-like

clock.h

```cpp
#ifndef CLOCK_H
#define CLOCK_H

#include <string>
//Monostate
class Clock
{
  // using inline key word => do not
     have to define them outside class
  inline static int m_Hour;
  inline static int m_Minute;
  inline static int m_Second;
  void CurrentTime();
  Clock();
public:
  static int GetHour();
  static int GetMinute();
  static int GetSecond();
  static std::string GetTimeString();
};
#endif // CLOCK_H
/*
 * earlier: if three instances are
     created => three different
     attributes
 * c1 - h m s
 * c2 - h m s
 * c3 - h m s
```

clock.cpp

```cpp
#include "Clock.h"
#include <ctime>
#include <sstream>

void Clock::CurrentTime() {
  time_t raw_time;
  time(&raw_time);
  tm *local_time = localtime(&raw_time);
  m_Hour = local_time->tm_hour;
  m_Minute = local_time->tm_min;
  m_Second = local_time->tm_sec;
}

Clock::Clock() {
  CurrentTime();
}

int Clock::GetHour() {
  CurrentTime();
  return m_Hour;
}

int Clock::GetMinute() {
  CurrentTime();
  return m_Minute;
}

int Clock::GetSecond() {
```

Mono state: Singleton-like

main.cpp

```cpp
#include "Clock.h"
#include <iostream>

int main() {
  Clock clk;
  std::cout << clk.GetTimeString() <<
      std::endl;
  return 0;
}
```

- Mono state achieves the singularity through behavior.
- Singleton achieves the singularity through structure.
- There is no instance method in mono state.

Singleton and Mono state

Singleton

- Enforces singular instance through structure
- Only one instance can exist
- Support for lazy instantiation
- Requires static instance method
- Can support inheritance and polymorphism
- Existing classes can be made singleton
- Flexible

Mono state

- Enforces singular instance through behavior
- Class may or may not be instantiated
- No support for lay instantiation
- Making all attributes static (method may be static)
- Static methods can not be overridden
- Difficult to change existing classes to monostate
- Inflexible

Singleton issue

Testing issue

- Singleton: The name of class using directly
- Violate the dependency inversion principle
- Program an interface is not implementation
- Can not replace singleton class with mock object

Testing issue

LocalPrinter.h

```
#ifndef LOCAL_PRINTER_H
#define LOCAL_PRINTER_H
#include <string>
class LocalPrinter {
  static LocalPrinter m_Instance;
  LocalPrinter() = default;
public:
  LocalPrinter(const LocalPrinter&) =
      delete;
  LocalPrinter& operator=(const
      LocalPrinter&) = delete;
  static  LocalPrinter & GetInstance();
  void Print(const std::string &data);
};
#endif
```

LocalPrinter.cpp

```
#include "LocalPrinter.h"
#include <iostream>

LocalPrinter LocalPrinter::m_Instance;
LocalPrinter& LocalPrinter::GetInstance
     () {
  return m_Instance;
}

void LocalPrinter::Print(const std::
     string& data) {
  std::cout << "[LOCALPRINTER]" << data
     << '\n';
}
```

Testing issue

main.cpp

```cpp
#include "LocalPrinter.h"

void PrintSales() {
  LocalPrinter::GetInstance().Print("Sales data");
}

int main() {
  auto &p = LocalPrinter::GetInstance();
  p.Print("Printing data to local printer");
  PrintSales();
}
```

- Using the name of class in different part of the code
- For example: can not replace LocalPrinter with NetworkPrinter
- Making unit test: replace LocalPrinter with mock object?
- There are a way around the issue: inherit the singleton class from other class

Solution: testing issue

Printer.h

```
#ifndef PRINTER_H
#define PRINTER_H
#include <string>

class Printer
{
protected:
  Printer() = default;
public:
  Printer(const Printer &) = delete;
  Printer & operator=(const Printer &) =
      delete;
  virtual ~Printer() = default;
  virtual void Print(const std::string &
     data) = 0;
  static  Printer& GetInstance(const std
     ::string & key);
};
#endif
```

Printer.cpp

```
#include "Printer.h"
#include "LocalPrinter.h"

Printer& Printer::GetInstance(const std
     ::string& key) {
  if(key == "local") {
    return LocalPrinter::GetInstance();
  }
}
```

Solution: testing issue

LocalPrinter.h

```cpp
#ifndef LOCAL_PRINTER_H
#define LOCAL_PRINTER_H

#include <string>
#include "Printer.h"

class LocalPrinter : public Printer
{
  static LocalPrinter m_Instance;
  LocalPrinter() = default;
public:
  static  LocalPrinter & GetInstance();
  void Print(const std::string & data);
};
#endif
```

LocalPrinter.cpp

```cpp
#include "LocalPrinter.h"
#include <iostream>

LocalPrinter LocalPrinter::m_Instance;
LocalPrinter& LocalPrinter::GetInstance() {
  return m_Instance;
}

void LocalPrinter::Print(const std::string& data) {
  std::cout << "[LOCALPRINTER]" << data << '\n';
}
```

Solution: testing issue

main.cpp

```
#include "LocalPrinter.h"

void PrintSales() {
  Printer::GetInstance("local").Print("Sales data");
}

int main() {
  auto &p = Printer::GetInstance("local");
  p.Print("Printing data to local printer");
  PrintSales();
  return 0;
}
```

- Using printer without depending on concrete type
- Making unit test easy
- Can create mock object

Open close principle issue

- The above implementation introduces new issue: adding more printer type, modifying code
- Violate open close principle

Registry Singleton

Printer.h

```cpp
#ifndef PRINTER_H
#define PRINTER_H

#include <string>

class Printer
{
protected:
  Printer() =default ;
public:
  Printer(const Printer &) = delete ;
  Printer & operator=(const Printer &) =
      delete ;
  virtual ~Printer()=default ;
  virtual void Print(const std::string &
      data)=0 ;
};
#endif
```

main.cpp

```cpp
#include "LocalPrinter.h"
#include "PrinterProvider.h"

void PrintSales() {
  auto p = PrinterProvider::
      GetPrinterPtr("local");
  if(p)
    p->Print("Sales data");
}

int main() {
  auto p = PrinterProvider::
      GetPrinterPtr("pdf");
  if(p)
    p->Print("Printing data to the
      printer");
  PrintSales();
}
```

Registry Singleton

LocalPrinter.h

```cpp
#ifndef LOCAL_PRINTER_H
#define LOCAL_PRINTER_H

#include <string>

#include "Printer.h"

class LocalPrinter : public Printer {
  static LocalPrinter m_Instance;
  LocalPrinter();
public:
  static  LocalPrinter & GetInstance();
  void Print(const std::string & data);
};
#endif
```

LocalPrinter.cpp

```cpp
#include "LocalPrinter.h"
#include <iostream>

#include "PrinterProvider.h"

LocalPrinter LocalPrinter::m_Instance;
LocalPrinter& LocalPrinter::GetInstance() {
  return m_Instance;
}

void LocalPrinter::Print(const std::string& data) {
  std::cout << "[LOCALPRINTER]" << data << '\n';
}
LocalPrinter::LocalPrinter() {
  PrinterProvider::RegisterPrinter("local", this);
}
```

Registry Singleton

PDFPrinter.h

```cpp
#ifndef PDF_PRINTER_H
#define PDF_PRINTER_H

#include "Printer.h"
class PDFPrinter : public Printer
{
  static PDFPrinter m_Instance ;
  PDFPrinter() ;
public:
  void Print(const std::string& data)
      override;
};
#endif
```

PDFPrinter.cpp

```cpp
#include "PDFPrinter.h"

#include <iostream>


#include "PrinterProvider.h"
PDFPrinter PDFPrinter::m_Instance ;
PDFPrinter::PDFPrinter() {
  PrinterProvider::RegisterPrinter("pdf"
      , this) ;
}

void PDFPrinter::Print(const std::string
      & data) {
  std::cout << "[PDFPRINTER]" << data<<
      '\n' ;
}
```

Registry Singleton

PrinterProvider.cpp

```cpp
#ifndef PRINTER_PROVIDER_H
#define PRINTER_PROVIDER_H

#include <mutex>
#include <unordered_map>

#include "Printer.h"

//Registry of singletons
//Multiton

class PrinterProvider
{
  inline static std::mutex m_mtx;
  inline static std::unordered_map<std::string, Printer*> m_Printers{};
  PrinterProvider() = default;
public:
  static void RegisterPrinter(const std::string &key, Printer *p);
  static Printer* GetPrinterPtr(const std::string &key);
  static Printer& GetPrinterRef(const std::string &key);
};
#endif
```

PDFPrinter.cpp

```cpp
#include "PrinterProvider.h"

#include <iostream>

void PrinterProvider::RegisterPrinter(const std::string& key, Printer* p) {
  std::lock_guard lock{m_mtx} ;
  if(auto it = m_Printers.find(key); it == end(m_Printers)) {
    m_Printers[key] = p;
  }else {
    std::cout << "Already registered\n";
  }
}

Printer* PrinterProvider::GetPrinterPtr(const std::string& key) {
  std::lock_guard lock{m_mtx};
  if(auto it = m_Printers.find(key); it != end(m_Printers)) {
    return m_Printers[key];
  }
  return nullptr;
}

Printer& PrinterProvider::GetPrinterRef(const std::string& key) {
```

Pros and Cons

Pros

- Class itself control the instantiation process.
- Can allow multiple instances.
- Better than global variable.
- Can be subclassed.

Cons

- Testing is difficult
- DCLP is defective
- Lazy destruction is complex

Where to use?

When only one instance should be use because:

- multiple instances cause data corruption.
- managing global state or shared state.
- multiple instances are not required.

Thank You!