Creational Design Pattern

Hung Tran

Fpt software

August 8, 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

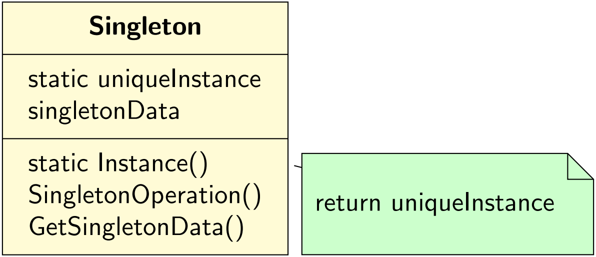
Ensure a class only has one instance, and provide a global point of access to it.

Singleton instance behaves like global variable? Yes!

How to Implement of 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);
}

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

```
#include "Logger.h"

// Create an other instance
void OpenConnection() {
  Logger lg;
  lg.WriteLog("Attempting to open a connection");
}
int main() {
  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);
}

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

```
#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()" << 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");
}
```

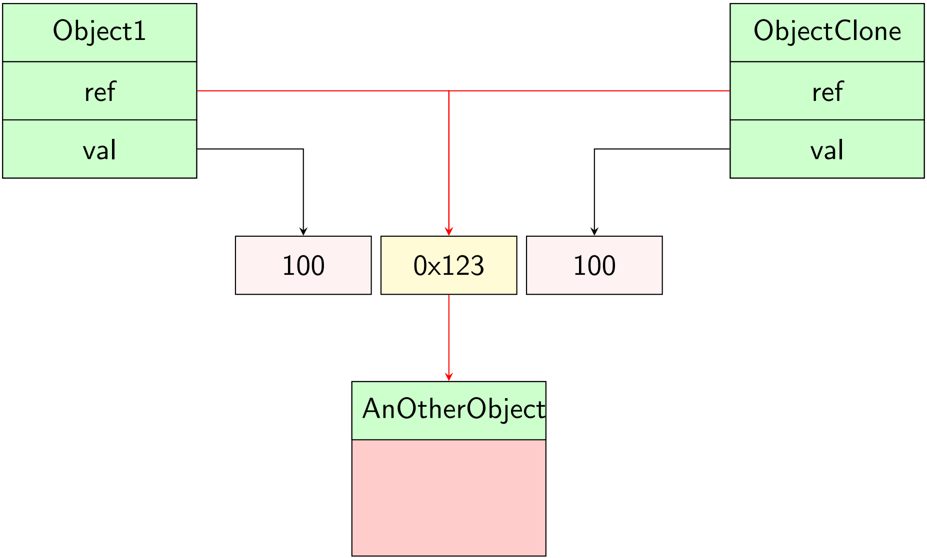
Logger2 class: Problem

- 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 copy constructor.
- The compiler will synthesize its own copy constructor that will perform a shallow copy.
- Therefore the mpStream 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 (point to stream had closed.)

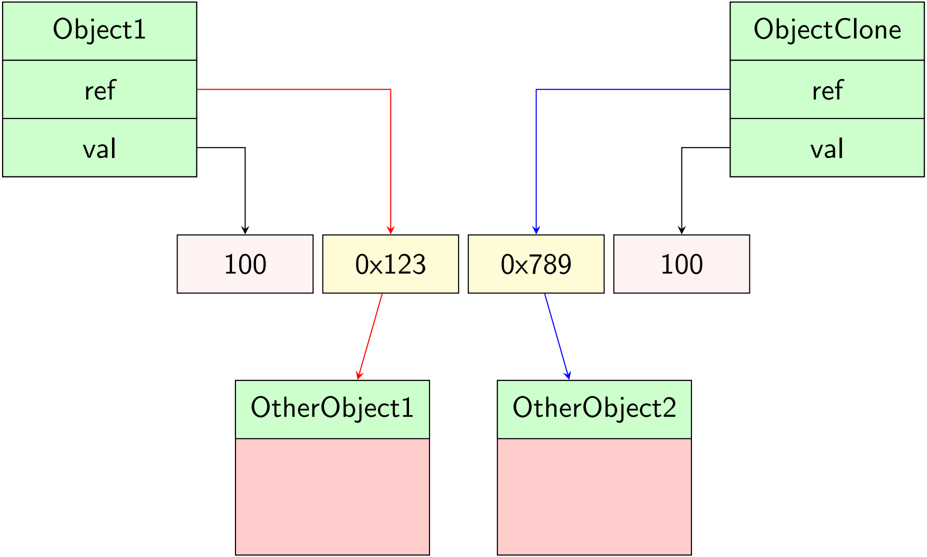
Shadow copy

- 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

So, We need to prevent the users from creating copied of the object. =¿ declare the constructor with delete modifier. =¿ similary 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

```
#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

```
#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 we do not need?

Lazy Singleton

- The above implementations, the instance is created before main invoked.
- The above instances called eager instances.
- 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.
- Using lazy instance

Lazy1 class

- 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.
- Using lazy instance

Lazy1 class

- 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.
- Using lazy instance

Lazy class

- 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.
- Using lazy instance

lazy class

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 ;
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"

Logger *Logger::m_pInstance;

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

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

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;
}
```

lazy class

main.cpp

```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");
  Logger* ptrLogger;
  ptrLogger = &lg;
  delete ptrLogger;
  return 0;
}
```

Lazy 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

```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

```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

```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;
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?

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.