**Singleton Design Pattern**

# Intent

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

Encapsulated "just-in-time initialization" or "initialization on first use".

# Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

# Discussion

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

1.   Ownership of the single instance cannot be reasonably assigned

2.   Lazy initialization is desirable

3.   Global access is not otherwise provided for

If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

# Structure

Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.

|  |
| --- |
| **GlobalResource** |
| -theInstance : GlobalResource |
| +getInstance() : GlobalResource |

# Check list

1.   Define a private static attribute in the "single instance" class.

2.   Define a public static accessor function in the class.

3.   Do "lazy initialization" (creation on first use) in the accessor function.

4.   Define all constructors to be protected or private.

5.   Clients may only use the accessor function to manipulate the Singleton.

# 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 = 0;

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

# Example Logger Class In C++

It is a best example for a singleton design pattern.

// Logger.h

#ifndef CUSTOM\_CLogger\_H

#define CUSTOM\_CLogger\_H

#include <fstream>

#include <iostream>

#include <cstdarg>

#include <string>

using namespace std;

#define LOGGER CLogger::GetLogger()

class CLogger {

          static const std::string m\_sFileName;       // Log file name

          static CLogger\* m\_pThis; // Singleton logger class object pointer

          static ofstream m\_Logfile; // Log file stream object

          CLogger(); //  Default constructor

          CLogger(const CLogger&){}; // copy constructor

          CLogger& operator=(const CLogger&){ return \*this; };   // assignment operator

public:

          void Log(const std::string& sMessage); // Logs a message

          void Log(const char \* format, ...); // overloaded function to Logs a message

          CLogger& operator<<(const string& sMessage);   // overloaded function to Logs a message

          static CLogger\* GetLogger();         // Funtion to create the instance of logger class

};

#endif

//Logger.cpp

#include "Logger.h"

#include "Logger.h"

#include"Utilities.h"

const string CLogger::m\_sFileName = "Log.txt";

CLogger\* CLogger::m\_pThis = NULL;

ofstream CLogger::m\_Logfile;

CLogger::CLogger()  {        }

CLogger\* CLogger::GetLogger() {

          if (m\_pThis == NULL){

                   m\_pThis = new CLogger();

                   m\_Logfile.open(m\_sFileName.c\_str(), ios::out | ios::app);

    }

    return m\_pThis;

}

void CLogger::Log(const char \* format, ...) {

          char\* sMessage = NULL;

          int nLength = 0;

          va\_list args;

          va\_start(args, format);

//  Return the number of characters in the string referenced the list of arguments.

// \_vscprintf doesn't count terminating '\0' (that's why +1)

          nLength = \_vscprintf(format, args) + 1;

sMessage = new char[nLength];

vsprintf\_s(sMessage, nLength, format, args);

          //vsprintf(sMessage, format, args);

m\_Logfile << Util::CurrentDateTime() << ":\t";

m\_Logfile << sMessage << "\n";

          va\_end(args);

delete [] sMessage;

}

void CLogger::Log(const string& sMessage) {

          m\_Logfile <<  Util::CurrentDateTime() << ":\t";

          m\_Logfile << sMessage << "\n";

}

CLogger& CLogger::operator<<(const string& sMessage) {

          m\_Logfile << "\n" << Util::CurrentDateTime() << ":\t";

          m\_Logfile << sMessage << "\n";

          return \*this;

}

// Utilities.h

#ifndef UTILITIES\_H

#define UTILITIES\_H

#include <iostream>

#include <string>

#include <time.h>

namespace Util {

          // Get current date/time, format is YYYY-MM-DD.HH:mm:ss

    const std::string CurrentDateTime() {

                   time\_t     now = time(NULL);

                   struct tm  tstruct;

                   char       buf[80];

                   localtime\_s(&tstruct, &now);

                   strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);

                   return buf;

    }

}

#endif

// Usage.c

#include <string>

#include "Logger.h"

using namespace std;

int main(int argc, char \*argv[]) {

          string message1 = "logg message 1 ...";

          string message2 = "logg message 2 ...";

          int    nNum = 10;

          CLogger::GetLogger()->Log("message to be logged");

          CLogger::GetLogger()->Log(message1);

          LOGGER->Log(" Message is:%s Number is:%d", message2.c\_str(), nNum);

          return 0;

}

# END