Singleton 패턴

타입

Creational Pattern

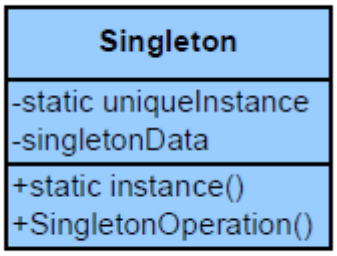
문제

객체를 단 한개만 만들어야 한다.

해결

클래스의 생성자, 파괴자를 private으로 선언하여 객체의 임의 생성, 파괴를 막는다. 그리고 객체를 생성하거나 얻는 메쏘드를 따로 정의한다.

클래스 다이어그램



예제

singleton1.cpp

```
class StringSingleton
{
public:
    // Some accessor functions for the class, itself
    std::string GetString() const
    {return mString;}
    void SetString(const std::string &newStr)
    {mString = newStr;}

    // The magic function, which allows access to the class from
anywhere
```

```
    // To get the value of the instance of the class, call:
    //     StringSingleton::Instance().GetString();
    static StringSingleton &Instance()
    {
        // This line only runs once, thus creating the only instance in
existence
        static StringSingleton *instance = new StringSingleton;
        // dereferencing the variable here, saves the caller from
having to use
        // the arrow operator, and removes temptation to try and delete
the
        // returned instance.
        return *instance; // always returns the same instance
    }

private:
    // We need to make some given functions private to finish the
definition of the singleton
    StringSingleton(){} // default constructor available only to
members or friends of this class

    // Note that the next two functions are not given bodies, thus any
attempt
    // to call them implicitly will return as compiler errors. This
prevents
    // accidental copying of the only instance of the class.
    StringSingleton(const StringSingleton &old); // disallow copy
constructor
    const StringSingleton &operator=(const StringSingleton &old);
//disallow assignment operator

    // Note that although this should be allowed,
    // some compilers may not implement private destructors
    // This prevents others from deleting our one single instance,
which was otherwise created on the heap
    ~StringSingleton(){}
private: // private data for an instance of this class
    std::string mString;
};
```

singleton2.cpp

```
#include <iostream>
 using namespace std;

 /* Place holder for thread synchronization mutex */
 class Mutex
 {   /* placeholder for code to create, use, and free a mutex */
 };
```

```
 /* Place holder for thread synchronization lock */
 class Lock
 {   public:
         Lock(Mutex& m) : mutex(m) { /* placeholder code to acquire the
mutex */ }
         ~Lock() { /* placeholder code to release the mutex */ }
     private:
         Mutex & mutex;
 };

 class Singleton
 {   public:
         static Singleton* GetInstance();
         int a;
         ~Singleton() { cout << "In Destructor" << endl; }

     private:
         Singleton(int _a) : a(_a) { cout << "In Constructor" << endl;
}


         static Mutex mutex;

         // Not defined, to prevent copying
         Singleton(const Singleton& );
         Singleton& operator =(const Singleton& other);
 };

 Mutex Singleton::mutex;

 Singleton* Singleton::GetInstance()
 {
     Lock lock(mutex);

     cout << "Get Instance" << endl;

     // Initialized during first access
     static Singleton inst(1);

     return &inst;
 }

 int main()
 {
     Singleton* singleton = Singleton::GetInstance();
     cout << "The value of the singleton: " << singleton->a << endl;
     return 0;
}
```

참고

http://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns#Singleton