Lesson09---特殊类设计

[本节目标]

- **掌握常见特殊类的设计方式**

1. 请设计一个类，只能在堆上创建对象

> 实现方式：
>
> 1. 将类的构造函数私有，拷贝构造声明成私有。防止别人调用拷贝在栈上生成对象。
> 2. 提供一个静态的成员函数，在该静态成员函数中完成堆对象的创建

```
class HeapOnly
{
public:
    static HeapOnly* CreateObject()
    {
        return new HeapOnly;
    }
private:
    HeapOnly() {}

    // C++98
    // 1.只声明,不实现。因为实现可能会很麻烦，而你本身不需要
    // 2.声明成私有
    HeapOnly(const HeapOnly&);

    // or

    // C++11
    HeapOnly(const HeapOnly&) = delete;
};
```

2. 请设计一个类，只能在栈上创建对象

> - 方法一：同上将构造函数私有化，然后设计静态方法创建对象返回即可。

```
class StackOnly
{
public:
    static StackOnly CreateObject()
    {
        return StackOnly();
    }
private:
    StackOnly()  {}
};
```

- 方法二：屏蔽new

  因为**new在底层调用void* operator new(size_t size)函数，只需将该函数屏蔽掉**即可。

  注意：**也要防止定位new**

```
class StackOnly
{
public:
    StackOnly()  {}
private:
    void* operator new(size_t size);
    void operator delete(void* p);
};
```

3. 请设计一个类，不能被拷贝

拷贝只会放生在两个场景中：拷贝构造函数以及赋值运算符重载，因此**想要让一个类禁止拷贝，只需让该类不能调用拷贝构造函数以及赋值运算符重载即可**。

- C++98

  将拷贝构造函数与赋值运算符重载只声明不定义，并且将其访问权限设置为私有即可。

```
class CopyBan
{
    // ...

private:
    CopyBan(const CopyBan&);
    CopyBan& operator=(const CopyBan&);
    //...
};
```

  原因：

  1. 设置成私有：如果只声明没有设置成private，用户自己如果在类外定义了，就可以不能禁止拷贝了
  2. 只声明不定义：不定义是因为该函数根本不会调用，定义了其实也没有什么意义，不写反而还简单，而且如果定义了就不会防止成员函数内部拷贝了。

- C++11

C++11扩展delete的用法，delete除了释放new申请的资源外，如果在默认成员函数后跟上=delete，表示让编译器删除掉该默认成员函数。

```
class CopyBan
{
    // ...
    CopyBan(const CopyBan&)=delete;
    CopyBan& operator=(const CopyBan&)=delete;
    //...
};
```

4. 请设计一个类，不能被继承

- C++98方式

```
// C++98中构造函数私有化，派生类中调不到基类的构造函数。则无法继承
class NonInherit
{
public:
    static NonInherit GetInstance()
    {
        return NonInherit();
    }
private:
    NonInherit()
    {}
};
```

- C++11方法

final关键字，**final修饰类，表示该类不能被继承。**

```
class A  final
{
    // ....
};
```

5. 请设计一个类，只能创建一个对象(单例模式)

设计模式：

设计模式（Design Pattern）是一套**被反复使用、多数人知晓的、经过分类的、代码设计经验的总结**。为什么会产生设计模式这样的东西呢？就像人类历史发展会产生兵法。最开始部落之间打仗时都是人拼人的对砍。后来春秋战国时期，七国之间经常打仗，就发现打仗也是有**套路**的，后来孙子就总结出了《孙子兵法》。孙子兵法也是类似。

使用设计模式的目的：为了代码可重用性、让代码更容易被他人理解、保证代码可靠性。 设计模式使代码编写真正工程化；设计模式是软件工程的基石脉络，如同大厦的结构一样。

单例模式：

一个类只能创建一个对象，即单例模式，该模式可以保证系统中该类只有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息，这种方式简化了在复杂环境下的配置管理。

单例模式有两种实现模式：

- **饿汉模式**

  就是说不管你将来用不用，程序启动时就创建一个唯一的实例对象。

```
// 饿汉模式
// 优点：简单
// 缺点：可能会导致进程启动慢，且如果有多个单例类对象实例启动顺序不确定。
 class Singleton
 {
 public:
     static Singleton* GetInstance()
     {
         return &m_instance;
     }

 private:
    // 构造函数私有
   Singleton(){};

   // C++98 防拷贝
   Singleton(Singleton const&);
   Singleton& operator=(Singleton const&);

   // or

   // C++11
   Singleton(Singleton const&) = delete;
   Singleton& operator=(Singleton const&) = delete;

   static Singleton m_instance;
 };

 Singleton Singleton::m_instance;  // 在程序入口之前就完成单例对象的初始化
```

如果这个单例对象在多线程高并发环境下频繁使用，性能要求较高，那么显然使用饿汉模式来避免资源竞争，提高响应速度更好。

- **懒汉模式**

  如果单例对象构造十分耗时或者占用很多资源，比如加载插件啊， 初始化网络连接啊，读取文件啊等等，而有可能该对象程序运行时不会用到，那么也要在程序一开始就进行初始化，就会导致程序启动时非常的缓慢。 所以这种情况使用懒汉模式（**延迟加载**）更好。

```
// 懒汉
// 优点：第一次使用实例对象时，创建对象。进程启动无负载。多个单例实例启动顺序自由控制。
// 缺点：复杂
```

```cpp

#include <iostream>
#include <mutex>
#include <thread>
using namespace std;

class Singleton
{
public:
    static Singleton* GetInstance() {
        // 注意这里一定要使用Double-Check的方式加锁，才能保证效率和线程安全
        if (nullptr == m_pInstance) {
            m_mtx.lock();
            if (nullptr == m_pInstance) {
                m_pInstance = new Singleton();
            }
            m_mtx.unlock();
        }
        return m_pInstance;
    }

    // 实现一个内嵌垃圾回收类
    class CGarbo {
    public:
        ~CGarbo(){
            if (Singleton::m_pInstance)
                delete Singleton::m_pInstance;
        }
    };

    // 定义一个静态成员变量，程序结束时，系统会自动调用它的析构函数从而释放单例对象
    static CGarbo Garbo;

private:
    // 构造函数私有
    Singleton(){};

    // 防拷贝
    Singleton(Singleton const&);
    Singleton& operator=(Singleton const&);

    static Singleton* m_pInstance; // 单例对象指针
    static mutex m_mtx;            //互斥锁
};

Singleton* Singleton::m_pInstance = nullptr;
Singleton::CGarbo Garbo;
mutex Singleton::m_mtx;

void func(int n)
{
    cout<< Singleton::GetInstance() << endl;

}
```

```
57
58  // 多线程环境下演示上面GetInstance()加锁和不加锁的区别。
59  int main()
60  {
61      thread t1(func, 10);
62      thread t2(func, 10);
63
64      t1.join();
65      t2.join();
66
67      cout << Singleton::GetInstance() << endl;
68      cout << Singleton::GetInstance() << endl;
69  }
```