

82 SINGLETONS in C++

本节讨论的是C++的一种**设计模式**，即 *Singleton*（单例模式）。

1. 什么是单例

单例是一个类的单一实例，如果我们不打算实例化一个特定类的多个实例，那有什么必要为它创建一个类呢？

当我们想要拥有应用于某种全局数据集的功能，且我们只是想要重复使用时，单例是非常有用的。

比如一个随机数生成器类，我们只希望能够查询它并得到一个随机数，并不需要多次实例化它；又比如说渲染器，这是通常是一个非常**全局**的东西，通常不会有一个渲染器的多个实例，只会向一个渲染器提交所有的渲染命令。

我们可能想要的是能够通过将类用作命名空间，来调用某些函数，这也是单例有点乱的地方。因为根本来说，单例类可以像命名空间一样工作，并没有要求单例类像普通类一样工作。

所以C++中的单例只是一种组织一堆全局变量和静态函数的方式。

2. 使用单例

```
class Singleton
{
public:
    // 删除复制构造函数
    Singleton(const Singleton&) = delete;

    // 获取单例实例的公共静态方法
    static Singleton& Get()
    {
        return s_Instance;
    }

    // 一个示例方法
    void Function() {};

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

    // 单例实例的静态成员
    static Singleton s_Instance;
};

// 初始化单例实例
Singleton Singleton::s_Instance;

int main()
{
    Singleton::Get(); // 获取单例实例
    Singleton::Get().Function(); // 通过单例实例调用方法
}
```

这只是一个懒汉式单例模式，不适合多线程使用，下面来看看更实际的例子，实现一个随机数生成器单例：

```

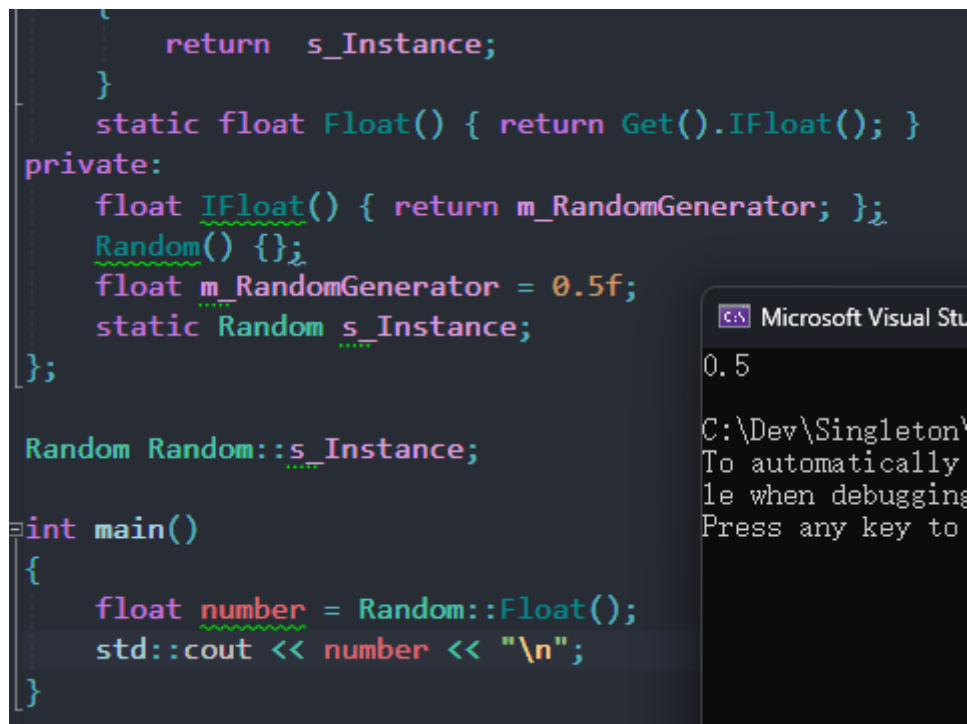
class Random
{
public:
    Random(const Random&) = delete;
    static Random& Get()
    {
        return s_Instance;
    }
    float Float() { return m_RandomGenerator; };
private:
    Random() {};
    float m_RandomGenerator = 0.5f; // 只是体现构造, 没实现“随机数”
    static Random s_Instance;
};

Random Random::s_Instance;

int main()
{
    float number = Random::Get().Float();
    std::cout << number << "\n"; // 0.5
}

```

我们最后调用这个单例需要像命名空间一样访问，那怎么能去掉中间的“Get()”呢：



```

        return s_Instance;
    }
    static float Float() { return Get().IFloat(); }
private:
    float IFloat() { return m_RandomGenerator; };
    Random() {};
    float m_RandomGenerator = 0.5f;
    static Random s_Instance;
};

Random Random::s_Instance;

int main()
{
    float number = Random::Float();
    std::cout << number << "\n";
}

```

0.5

C:\Dev\Singleton\To automatically le when debugging Press any key to

可以使用这种间接的方式，一旦这些都被编译器内联后，从所有这些函数调用中不会有什么性能损失

最后想说的是现在代码中的Get函数，我们现在写它的方式，意味着我们需要在类成员中有一个静态实例，然后需要在某个翻译单元中来定义它，这样有点麻烦，更好的方法是把这个静态声明移动到静态函数中：

C++

```

class Random
{
public:
    Random(const Random&) = delete;
    static Random& Get()
    {
        static Random s_Instance; // 静态内存中的静态变量
        return s_Instance;
    }
    static float Float() { return Get().IFloat(); }
private:
    float IFloat() { return m_RandomGenerator; };
    Random() {};
    float m_RandomGenerator = 0.5f;
    // static Random s_Instance;
};
// Random Random::s_Instance;
int main()
{
    float number = Random::Float();
    std::cout << number << "\n";
}

```

这样的效果是，一旦Get函数第一次被调用，它将实例化，接下来的时间中只是在静态内存中被引用，代码更干净了

你可能还会想要删掉复制操作符等进一步的清理操作：

C++

```

Random& operator=(const Random&) = delete; // 删除赋值操作符

```

相关知识可参考[41 OPERATORS and OPERATOR OVERLOADING in C++](#)

但课程这里简化了操作以更好理解。

3. 为什么不写成namespace

C++

```

namespace RandomClass
{
    static float s_RandomGenerator = 0.5f;
    static float Float() { return s_RandomGenerator; }
}

```

这样的效果基本一样，只不过无法直接使用public、private等类特有功能，虽然也能通过头文件中只包含Float函数的声明，然后用另一个从cpp文件实现这个函数和静态变量来实现有效地隐藏。

所以，使用类来实现单例是没有什么真正的缺点的，会让你的代码更加有条理，更不用说用 **namespace** 就丧失了将其赋值给一个变量这样的能力，而类就可以。

