C++RAII机制

什么是RAII?

RAII是Resource Acquisition Is Initialization（wiki上面翻译成“资源获取就是初始化”）的简称，是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。

为什么要使用RAII?

上面说到RAII是用来管理资源、避免资源泄漏的方法。那么，用了这么久了，也写了这么多程序了，口头上经常会说资源，那么资源是如何定义的？在计算机系统中，资源是数量有限且对系统正常运行具有一定作用的元素。比如：网络套接字、互斥锁、文件句柄和内存等等，它们属于系统资源。由于系统的资源是有限的，就好比自然界的石油，铁矿一样，不是取之不尽，用之不竭的，所以，我们在编程使用系统资源时，都必须遵循一个步骤：
1 申请资源；
2 使用资源；
3 释放资源。
第一步和第三步缺一不可，因为资源必须要申请才能使用的，使用完成以后，必须要释放，如果不释放的话，就会造成资源泄漏。

一个最简单的例子：

```
#include <iostream>

using namespace std;

int main()

{
    int *testArray = new int [10];
    // Here, you can use the array
    delete [] testArray;
    testArray = NULL ;
    return 0;
}

```

小结：
但是如果程序很复杂的时候，需要为所有的new 分配的内存delete掉，导致极度臃肿，效率下降，更可怕的是，程序的可理解性和可维护性明显降低了，当操作增多时，处理资源释放的代码就会越来越多，越来越乱。如果某一个操作发生了异常而导致释放资源的语句没有被调用，怎么办？这个时候，RAII机制就可以派上用场了。

如何使用RAII?

当我们在一个函数内部使用局部变量，当退出了这个局部变量的作用域时，这个变量也就别销毁了；当这个变量是类对象时，这个时候，就会自动调用这个类的析构函数，而这一切都是自动发生的，不要程序员显示的去调用完成。这个也太好了，RAII就是这样去完成的。

由于系统的资源不具有自动释放的功能，而C++中的类具有自动调用析构函数的功能。**如果把资源用类进行封装起来，对资源操作都封装在类的内部，在析构函数中进行释放资源。当定义的局部变量的生命结束时，它的析构函数就会自动的被调用，如此，就不用程序员显示的去调用释放资源的操作了。**

使用RAII 机制的代码：

```
#include <iostream>

using namespace std;

class ArrayOperation
{
public :
    ArrayOperation()
    {
        m_Array = new int [10];
    }
void InitArray()
{
    for (int i = 0; i < 10; ++i)
    {
        *(m_Array + i) = i;
    }
}

void ShowArray()
{
    for (int i = 0; i <10; ++i)
    {
        cout<<m_Array[i]<<endl;
    }
}

~ArrayOperation()
{
    cout<< "~ArrayOperation is called" <<endl;
    if (m_Array != NULL )
    {
        delete[] m_Array;
        m_Array = NULL ;
    }
}
private :
    int *m_Array;
};

bool OperationA();
bool OperationB();

int main()
{
    ArrayOperation arrayOp;
    arrayOp.InitArray();
    arrayOp.ShowArray();
    return 0;
}
```

上面这个例子没有多大的实际意义，只是为了说明RAII的机制问题。下面说一个具有实际意义的例子：

```
template<class... _Mutexes>
    class lock_guard
    {   // class with destructor that unlocks mutexes
public:
    explicit lock_guard(_Mutexes&... _Mtxes)
        : _MyMutexes(_Mtxes...)
        {   // construct and lock
        _STD lock(_Mtxes...);
        }

    lock_guard(_Mutexes&... _Mtxes, adopt_lock_t)
        : _MyMutexes(_Mtxes...)
        {   // construct but don't lock
        }

    ~lock_guard() _NOEXCEPT
        {   // unlock all
        _For_each_tuple_element(
            _MyMutexes,
            [](auto& _Mutex) _NOEXCEPT { _Mutex.unlock(); });
        }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;
private:
    tuple<_Mutexes&...> _MyMutexes;
};
```

在使用多线程时，经常会涉及到共享数据的问题，C++中通过实例化std::mutex创建互斥量，通过调用成员函数lock()进行上锁，unlock()进行解锁。不过这意味着必须记住在每个函数出口都要去调用unlock()，也包括异常的情况，这非常麻烦，而且不易管理。C++标准库为互斥量提供了一个RAII语法的模板类std::lock_guard，其会在构造函数的时候提供已锁的互斥量，并在析构的时候进行解锁，从而保证了一个已锁的互斥量总是会被正确的解锁。上面的代码正式头文件中的源码，其中还使用到很多C++11的特性，比如delete/noexcept等，有兴趣的同学可以查一下。