# 一、单例设计模式

单例：整个项目中，有某个或者某些特殊的类，属于该类的对象，我只能创建1个，多了创建不了；

Eg：

1. #include <iostream>
3. **using** **namespace** std;
5. **class** MyCAS
6. {
7. **private**:
8. MyCAS();    //构造函数私有化
9. **static** MyCAS \*m\_instance;
10. **public**:
11. **static** MyCAS \*GetInstance()
12. {
13. **if**(NULL == m\_instance)
14. {
15. m\_instance = **new** MyCAS();
16. **static** CGar c1;
17. }
18. **return** m\_instance;
19. }
21. **class** CGar()
22. {
23. **public**:
24. ~CGar()
25. {
26. **if**(MyCAS::m\_instance)
27. {
28. **delete** MyCAS::m\_instance;
29. MyCAS::m\_instance = NULL:
30. }
31. }
32. }
34. **void** func()
35. {
36. cout << "测试函数" << endl;
37. }
38. }
40. MyCAS \*MyCAS::m\_instance = NULL:

43. **int** main()
44. {
45. MyCAS \*myCas = MyCAS::GetInstance();
46. myCas->func();
47. **return** 0;
48. }

此例中，含有MyCAS::m\_instance成员的delete函数，即调用其他类的析构函数来delete成员变量。

# 二、单例设计模式共享数据问题分析、解决

如果上述的代码出现在多线程函数中的话，便可能出现以下问题，在

1. **static** MyCAS \*GetInstance()
2. {
3. **if**(NULL == m\_instance)
4. {
5. m\_instance = **new** MyCAS();
6. **static** CGar c1;
7. }
8. **return** m\_instance;
9. }

这个函数中，可能有多个线程同时进入，使得new MyCAS多次运行，产生错误，这是我们便要添加互斥锁类避免这种问题，如下

1. #include <iostream>
3. **using** **namespace** std;
5. std::mutex resource\_mutex;
7. **class** MyCAS
8. {
9. **private**:
10. MyCAS();    //构造函数私有化
11. **static** MyCAS \*m\_instance;
12. **public**:
13. **static** MyCAS \*GetInstance()
14. {
15. std::unique\_lock<std::mutex> mymutex(resource\_mutex);
16. **if**(NULL == m\_instance)
17. {
18. m\_instance = **new** MyCAS();
19. **static** CGar c1;
20. }
21. **return** m\_instance;
22. }
24. **class** CGar()
25. {
26. **public**:
27. ~CGar()
28. {
29. **if**(MyCAS::m\_instance)
30. {
31. **delete** MyCAS::m\_instance;
32. MyCAS::m\_instance = NULL:
33. }
34. }
35. }
37. **void** func()
38. {
39. cout << "测试函数" << endl;
40. }
41. }
43. MyCAS \*MyCAS::m\_instance = NULL:

46. **int** main()
47. {
48. MyCAS \*myCas = MyCAS::GetInstance();
49. myCas->func();
50. **return** 0;
51. }

此时，虽然实现了我们想要的，但是程序的运行速率不是很高，每次程序运行到

1. **static** MyCAS \*GetInstance()

这个函数时，都需要加锁判断成员变量是否为空，然后解锁，这样便会拉低程序的运行速率，为了解决该问题，我们使用双重锁定（双重检查），即：

1. **static** MyCAS \*GetInstance()
2. {
3. **if**(NULL == m\_instance)  //双重锁定（双重检查）
4. {
5. std::unique\_lock<std::mutex> mymutex(resource\_mutex);
6. **if**(NULL == m\_instance)
7. {
8. m\_instance = **new** MyCAS();
9. **static** CGar c1;
10. }
11. }
12. **return** m\_instance;
13. }

这样的话，程序运行速率便有了显著的提升。

# 三、std::call\_once()

C++11引入的函数；

参数：

第二个参数是一个函数名a()；

功能：

能够保证函数a()只被调用一次，call\_once具备互斥量这个能力，而且效率上，比互斥量消耗的资源更少；

call\_once()需要与一个标记结合使用，这个标记 std::once\_flag；其实once\_flag是一个结构。call\_once()就是通过这个标记决定对应的函数a()是否执行，调用call\_once()成功后，call\_once()就把这个标记设置为“已调用”状态，后续再次调用call\_once（），只要once\_flag被设置为了“已调用”状态，那么对应的函数a()就不会再被执行了；